Contents

Credits

Purpose

This website contains all the resources to learn the principles of computer programming using C#. It is used in the delivery of CSCI 1301 - Principles of Computer Programming I and CSCI 1302 - Principles of Computer Programming II in the Bachelor of Science in Computer Science[1] at Augusta University[2], and contains practical guides and additional resources for students and instructors.

Authors

At the time of writing, this resource is actively maintained Clément Aubert[3]. Additional contributions, by (under)graduate course assistants[4] and other contributors, are tracked by version control[5].

Some of the material originated from discussion, handouts and contribu-

[1]https://www.augusta.edu/ccs/bs-cs.php
[2]https://www.augusta.edu
[3]https://spots.augusta.edu/caubert/#contact
[4]/docs/academic_life/ca
[5]https://github.com/princomp/princomp.github.io/graphs/contributors

tions by Clément Aubert[6], Aubrey Bryant[7], Michael Dowell[8], Richard DeFrancisco[9], Onyeka Ezenwoye[10], Leszek Gasieniec[11], Reza Rahaeimehr[12], Neea Rusch[13], Edward Tremel[14] and Paul York[15].

Additionally, the School of Computer & Cyber Sciences[16]'s past and present academic advisors[17], Laura Austin, Denise Coleman, Markus Bacha, and Wennie Squires, and communications & marketing specialist, Haley Bourne, improved the Academic Life[18] notes through their suggestions and references.

Supports

The first source of support is the constant stream of feedback we receive from students and users: thank *you*.

This project has been monetarily supported by an Affordable Learning Georgia[19] Transformation Grants[20] (Proposal 571[21]) and a Continuous Improvement Grant (M260).

Affordable Learning Georgia

[22]

This project also received the support of Augusta University[23]'s School

[6]http://spots.augusta.edu/caubert/
[7]https://www.linkedin.com/in/aubrey-bryant-61898176
[8]https://spots.augusta.edu/mdowell/
[9]https://www.augusta.edu/faculty/directory/view.php?id=RDEFRANCISCO
[10]https://www.augusta.edu/faculty/directory/view.php?id=oezenwoye
[11]https://cgi.csc.liv.ac.uk/~leszek/
[12]https://www.augusta.edu/faculty/directory/view.php?id=RRAHAEIMEHR
[13]https://nkrusch.github.io/
[14]https://edwardtremel.com/
[15]https://www.augusta.edu/faculty/directory/view.php?id=pyork1
[16]https://www.augusta.edu/ccs/
[17]https://www.augusta.edu/ccs/faculty.php#administration
[18]/docs/academic_life
[19]https://www.affordablelearninggeorgia.org/
[20]https://www.affordablelearninggeorgia.org/grants/overview/
[21]https://www.affordablelearninggeorgia.org/assets/documents/571-proposal.docx
[22]https://www.affordablelearninggeorgia.org/
[23]https://www.augusta.edu/

of Computer and Cyber Sciences[24] and Center for Instructional Innovation[25].

Tools

We strive to prioritize open-source software when possible, and occasionally contribute to them.

Software

This website uses different technologies.

- The markdown source code is converted to (a slightly different) `md`, `pdf`, `odt` and `docx` formats thanks to pandoc[26] and pandoc-include[27] (among other `lua` filters).
- The `pdf` format is compiled using XeLaTeX[28].
- The source code is highlighted thanks to Pygments[29].
- The website is powered by quartz[30].

More details on the tools we use and how this resource is made can be found in dev. guide[31].

Fonts

We use the URW Gothic[32] and Hack[33] (inspired by the DejaVu[34] font) fonts. Those fonts have been specially selected for their legibility and lower impact on environment[35].

Services

The source code and the website are graciously hosted and built by github[36].

[24] https://www.augusta.edu/ccs/
[25] https://www.augusta.edu/innovation/
[26] https://pandoc.org/
[27] https://github.com/DCsunset/pandoc-include
[28] https://tug.org/xetex/
[29] https://pygments.org/
[30] https://quartz.jzhao.xyz/
[31] dev_guide
[32] https://fontesk.com/gothic-typeface/
[33] https://sourcefoundry.org/hack/
[34] https://sourcefoundry.org/hack/
[35] https://en.wikipedia.org/wiki/Century_Gothic#Printer_ink_usage
[36] https://github.com/

Licence

This work is under Creative Commons Attribution 4.0 International[37]. Concretely, this means that you are free to:

- Save, print, copy and redistribute the entirety of the resources presented here,
- Modify them as you see fit,

as long as you give proper credit and keep the same licence.

Please refer to our licence file[38] for the detail of this licence.

Contributing

How can I contribute?

If you are a student

We would like to hear your thoughts on this resource to understand how to make it better for you and your fellow students. If you encounter a mistake, run into an issue while using the resource, or find it missing something important, you can contribute by providing feedback in one of the following ways:

- talk to your instructor about the issue
- talk to your section's UCA about the issue
- leave feedback on this website on the page where you notice the issue
- Open an issue[39]
- print the resource and identify the issue, then hand it to your instructor or UCA

If you have suggestions on how to make it better, we encourage you to share those ideas too.

If you are an instructor

You will need to have a Github account[40]. Next contact any of the authors[41] of this resource over email, provide your Github username, and request an invitation to be added to the instructors team.

[37] https://creativecommons.org/licenses/by/4.0/

[38] https://github.com/princomp/princomp.github.io/blob/main/license.md

[39] https://github.com/princomp/princomp.github.io/issues/new/choose

[40] https://github.com/join

[41] credits#authors

If you are a UCA

You will need to have a Github account[42]. Next ask your course section instructor to invite you to the 1301 UCAs team. Your instructor needs your Github username to send you the invitation.

If you are an outside collaborator

When you have identified a mistake in this resource and want to notify the authors, leave feedback on this website on the page where you notice the issue or open an issue[43] explaining the issue.

If you want to make edits yourself, you can fork[44] the source code, make edits, then open a pull request[45] for us to review.

Next steps for editors

If you are looking to edit this resource and making your first contribution, read through the dev. guide[46]. It explains:

- how to locate different resources
- how to edit the resources
- how to label content

Following the dev. guide[47] will help to ensure your edits meet the expected quality guidelines and can be integrated into the existing resource with ease.

Dev. Guide

This guide explains how this resource is organized, how it is built and deployed, and how to maintain this resource. It is intended to be comprehensive, but should most likely be read only after having read our contributing[48] and UCA[49] guides.

[42]https://github.com/join
[43]https://github.com/princomp/princomp.github.io/issues/new/choose
[44]https://github.com/princomp/princomp.github.io/fork
[45]https://github.com/princomp/princomp.github.io/pulls
[46]dev_guide
[47]dev_guide
[48]https:/princomp.github.io/docs/about/contributing
[49]https:/princomp.github.io/docs/academic_life/uca_guide#editing-the-resources

Resources Organization Overview

Folders and Files

The source code repository[50]'s main branch is organized as follows:

path	description
`.github/`	github templates and configuration for github actions
`misc/`	resources that need to be either integrated into the resource, or discarded
`source/`	source for the material
`licence.md`	license file
`readme.md`	presentation of the repository

The `source/` folder contains the following:

path	description
`code/`	code examples (snippets and projects)
`docs/`	additional helpful documentation
`fonts/`	the fonts (redistributed with permission) used by this resource
`img/`	images, sometimes with their LaTeX source code
`labs/`	lab exercises
`lectures/`	lecture notes
`slides/`	slides
`templates/`	templates and filters used for building this resource
`vid/`	video files
`Makefile`	makefile used to compile this resource
`index.md`	website index page
`order`	file used to specify the order on the website's menu and the book

[50]https://github.com/princomp/princomp.github.io

Building and Deploying

The content is built and deployed in two phases:

- Running `make all` in the `source/` folder will create a `content/` folder at root level containing:
  - one `.md` file per `.md` file in the `source/` folder (in the same location: `source/labs/If.md` is compiled to `content/labs/If.md`), resulting from pandoc[51]'s conversion,
  - one `.pdf`, `.odt` and `.docx` file per `.md` file (with the exception of the `index.md` files) in the `source/` folder (in the same location: `source/labs/If.md` is compiled to `content/labs/If.pdf`), resulting from pandoc[52]'s conversion,
  - some files from the `img/`, `slides/` and `vid/` folders, copied selectively (for example, only the `.jpeg`, `.png`, `.pdf`, `.svg` and `.gif` files are copied from the `img/` folder),
  - the `.woff` and `.woff2` files copied from the `fonts/` folders,
  - a `code/projects/` folder containing, for each `Program.cs` file contained in a `source/code/projects/x/y`, a `x.zip` archive containing a C# project including `Program.cs` along with some (optional) class file,
  - a `web-order.ts` file, compiled from the `source/order` file, that fixes the order used by the website in the menu,
  - a `book.html`, a `book.pdf`, a `book.html` and a `book.docx` file resulting from pandoc[53]'s conversion of the `.md` files contained in the `SOURCE_BOOK`'s makefile variable (containing all the `.md` files in the `source/docs/` and `source/lectures/`, in the order fixed by the `order` file).
- Then, using the files in the generated `content/` folder, a website is built using quartz[54] and deployed to https://princomp.github.io/. This is achieved mainly thanks to the `.github/workflows/build_and_deploy.yaml` file and github's actions[55].

Tools, Briefly

This resource is mainly developed and powered using

- git[56]

[51] https://pandoc.org/
[52] https://pandoc.org/
[53] https://pandoc.org/
[54] https://quartz.jzhao.xyz/
[55] https://docs.github.com/en/actions
[56] https://git-scm.com/

- pandoc[57]
- make[58]
- python[59]
- quartz[60],
- github's actions[61].

But note that knowing git and markdown are enough to contribute online through the github repository[62].

While most of those tools are standard (with the exception of quartz, but it relies itself on the standard Node[63] and `npm` technologies), we acknowledge that

1. It is challenging to understand that many different technologies,
2. We should strive to welcome contributions from collaborators not familiar with them,
3. Our set-up is unique in some respects.

This guide tries to alleviate some challenges resulting from this overall unique and diverse resource organization. For more details about our tools, please refer to the Installing dependencies and Repository Maintenance sections.

Locating Resources

To obtain the latest version of this resource, you can either

- visit the accompanying website princomp.github.io[64],
- download the latest version of the built resource[65],
- clone our repository[66].

This resource is an extension of csci-1301.github.io/[67], please refer to their user guide[68] for more information about it.

[57] https://pandoc.org/installing.html
[58] https://www.gnu.org/software/make/
[59] https://www.python.org
[60] https://quartz.jzhao.xyz/
[61] https://docs.github.com/en/actions
[62] https://github.com/princomp/princomp.github.io
[63] https://nodejs.org/
[64] https://princomp.github.io
[65] https://github.com/princomp/princomp.github.io/releases/download/latest/release.zip
[66] https://github.com/princomp/princomp.github.io/
[67] https://csci-1301.github.io/
[68] https://csci-1301.github.io/user_guide.html#locating-course-resources

Editing Resources

If you are new to this project, first read through Contributing Guidelines[69] to learn how you can contribute to the improvement of this resource, and if applicable, how to join a contributing team.

Best practices for all forms of content

Inclusivity Follow the IT Inclusive Language Guide[70] from the University of Washington:

> use gender-neutral terms; avoid ableist language; focus on people not disabilities or circumstances; avoid generalizations about people, regions, cultures and countries; and avoid slang, idioms, metaphors and other words with layers of meaning and a negative history.

Typically, we recommend using

- "unethical hacker" instead of "black hat",
- "main" instead of "master",
- "blank space" instead of "white space",
- "display on the screen" instead of "printing", -etc.

In doubt, please start by referring to this list of problematic words and phrases[71].

Structure for accessibility

- All resources are titled
  - title each markdown document by having one (and only one) title at top level (that is, using #),
  - use subtitles when appropriate,
  - title all images with a descriptive title and add an alt-tag,
  - title all code blocks in labs and lecture notes.
- All resources are labelled when applicable, see content labelling for more details

Resources to assess accessibility:

- Affordable Learning Georgia's guide[72]

[69] /contributing

[70] https://itconnect.uw.edu/guides-by-topic/identity-diversity-inclusion/inclusive-language-guide/

[71] https://itconnect.uw.edu/guides-by-topic/identity-diversity-inclusion/inclusive-language-guide/#list

[72] https://alg.manifoldapp.org/projects/oer-accessibility-series-and-rubric

- Specific Review Standards from the QM Higher Education Rubric[73]
- UWG Accessibility Services's guide[74]
- Penn State's recommendations for alternative text and complex images.[75]

- WebAim Color Contrast Checker[76]
- WebAIM (Web Accessibility In Mind)[77]

Markdown Text documents are written using standard markdown syntax[78]. More precisely,

- in the `markdown+emoji` format, that is, in pandoc's markdown[79], using the emoji[80] extension[81]),
- using the pandoc-include[82] filter,
- and a custom[83] filter that sets all the code blocks[84], or all the code block and inline code[85]'s syntax highlighting to C# by default.

Because of the way the markdown is processed, please refrain from using the " and " characters: pandoc will automatically convert `"` into language-appropriate quotes for us.

Images

- Images belong in `source/img/` directory.
- Explain the image in written form.
- Title each image, this will create a URL for the image and enables linking to it.
- Always include a descriptive alt tag for accessibility.
- Do not rely on everyone seeing colors the same way[86].

[73] https://www.qualitymatters.org/sites/default/files/PDFs/StandardsfromtheQMHigherEducationRubric.pdf

[74] https://docs.google.com/document/d/16Ri1XgaXiGx28ooO-zRvYPraV3Aq3F5ZNJYbVDGVnEA/edit?ts=57b4c82d#

[75] https://accessibility.psu.edu/images/

[76] https://webaim.org/resources/contrastchecker/

[77] https://webaim.org/

[78] https://commonmark.org/

[79] https://pandoc.org/MANUAL.html#pandocs-markdown

[80] https://pandoc.org/MANUAL.html#extension-emoji

[81] https://pandoc.org/MANUAL.html#extensions

[82] https://github.com/DCsunset/pandoc-include

[83] https://github.com/princomp/princomp.github.io/tree/main/source/templates/filters

[84] https://github.com/princomp/princomp.github.io/blob/main/source/templates/filters/default-code-class-block.lua

[85] https://github.com/princomp/princomp.github.io/blob/main/source/templates/filters/default-code-class-block-inline.lua

[86] https://www.wikiwand.com/en/Color_blindness

- Prefer scalable vector images.
- When referring to images in markdown, use path from root, see example below

Syntax example. The quoted text is the alt tag and in parentheses is path to file

```
!["image of visual studio IDE"](./img/vs_ide.jpg){
 ↪  width=80% }
```

The `{ width=80% }` attribute is optional.

Images generated by LaTeX Some images are generated by LaTeX: the `.tex` file is what is used to generate the `.pdf` file, and then pdf2svg converts the `.pdf` into a `.svg` file. The `.svg` files are used in the `.html`, `.odt` and `.docx` documents, while the `.pdf` is used in the `.pdf` documents. The resulting images are added to the repository so that there is no need to re-compile them every time, or to set-up LaTeX and latexmk on each system.

UML class diagrams The UML class diagrams are created using Mermaid[87] and located in `source/uml`. To create a new class diagram, say for a `Documentation` class, follow those steps:

1. Create a `Documentation.txt` file in `source/uml` that follows the syntax for class diagrams[88] (note that there is no need to add `classDiagram` at the beginning, it will be done automatically),
2. Run (from the `source/` folder) `make uml/Documentation.md`,
3. Integrate the resulting drawing, properly captioned and with a link to your `Documentation.txt` file (for visually impaired readers, or to facilitate automatic processing) using `!include uml/Documentation.md`.

Source code

- Source code programs belong in `source/code/` directory.
- The code included in this directory should either be:
  - Placed in the `snippets/` sub-folder, and be a complete program.
  - Placed in the `projects/<solution>/<project>/` sub-folder, and contains a `Program.cs` file:
    - Go to `source/code/projets/`,
    - Create a subdirectory with the name of the solution you would like to use,

[87] https://mermaid.js.org/

[88] https://mermaid.js.org/syntax/classDiagram.html

        * Create a subdirectory with the name of the project you would like to use,
        * Create a file called `Program.cs` in
      `source/code/projects/<solution>/<project>/Program.cs`
        * If you want to add additional classes, add them in `code/projects/<solution>/<project>/<Class>.cs` files.

      Do **not** add solution (`sln`) or project (`csproj`) files: they will be created automatically using the project and solution's name you specified (and a makefile rule similar to this one[89]), if multiple classes are present they will all be linked, and the resulting archive will be hosted at `content/code/projects/<solution>.zip`.
- Source code that is faulty, partial, or does not terminate can be included in markdown as inline code block.

Code snippets can be included in markdown documents using pandoc-include[90] filter:

~~~
```text
!include code/sample.cs
```
~~~

Note that for an unknown reason[91], no special characters (such as _) should be used in the filenames.

- Title each source code block included in markdown, this will create a URL for the code block and enables linking to it.
- code blocks are by default annotated as `csharp`
  - syntax highlighting is applied automatically at build time based on the code block language
  - to use a language other than C#, specify the language locally in the specific code block:
- only include code in text form such that it can be copy-pasted for reuse
- make sure to include blank lines before and after code blocks, since the absence of these can cause the code block to display incorrectly.

Tidying Source code CSharpier[92] is used to tidy the source code and make it uniform. Use

```
make tidy
```

[89]https://github.com/csci-1301/C-Sharp-project-maker
[90]https://github.com/DCsunset/pandoc-include
[91]https://github.com/DCsunset/pandoc-include/issues/45
[92]https://csharpier.com/

to tidy all the source code present in the `source/code/` folders. The configuration file[93] is at `source/code/csharpierrc.yaml`.

Creating new lectures

Lecture notes belong to the `source/lectures/` directory.

To create a new lecture, for instance on exception handling:

1. Create a directory corresponding to the theme if it does not exist already (say, `exceptions`), under `source/lectures/` directory

   - Follow the existing pattern for naming convention which is lowercase and separation by underscores.

   - At the root of this folder, create an `index.md` file (so, at `source/lectures/exceptions/index.md`) containing

     ```
     ---
     title: Desired Title for Theme
     ---
     ```

     so that your theme will be labeled "Desired Title for Theme" on the website's menu (see content labelling on how to further label it).

2. Under the directory corresponding to your theme, create a file named after the lecture's title (e.g., `exception-handling.md`) in lowercase. Write lecture notes in this file using markdown.

3. Edit the `source/order` file and insert where appropriate

   - `./lectures/exception/` (if you created a folder called `exception`),
   - `./lectures/exception/exception-handling.md` (which *must be* between `./lectures/exception/` and the next `./lectures/xyz/` folder).

   This last step will insure that your lecture is 1. included in the book, 2. sorted correctly on the website's menu (the default ordering is alphabetical).

If the lecture does not appear, here are the steps for troubleshooting the issue:

1. Check that after committing changes, the automated build has completed successfully, by checking the workflows[94],

[93]https://csharpier.com/docs/Configuration
[94]https://github.com/princomp/princomp.github.io/actions

2. The newly created lecture is under the subdirectory you picked in the `source/lectures/` directory[95],
3. The `.md` file exists,
4. Hard refresh the browser page if viewing the resources website

Known issues: When concatenating files pandoc may or may not include empty spaces between individual files. This may cause the subsequent lecture title to not appear in the generated book. For this reason, each lecture file should end with a newline.

Creating new labs

The process is very close to the process to create a new lecture, with the following exceptions:

- All lab resources are located under `source/labs/` directory, at root level (there is no "theme" sub-folder).
- You do not need to edit the `source/order` file, since labs are not included in the book nor sorted on the website.

Additionally, remember to:

1. Choose a short and unique name that describes the lab (say, `StringMethods.md`)
    - follow the existing convention for naming,
    - do not number labs or make assumptions about numbering because another instructor may not follow the exact same lab order,
    - make the lab standalone to support alternative ordering (avoid assumptions about what was done "last time"),
    - do not make assumptions about student using specific OS, include instructions for all supported options (Windows, MacOS, Linux),
    - do not make assumptions about student using Visual Studio, refer to IDE instead.
2. (optional) You can add a downloadable project (use a link of the form `[the Rectangle project](./code/projects/Rectangle.zip)`) or include snippets of code by following our instructions to add source code.

Using this established build system generates labs that are cross-platform (Windows, MacOS, Linux) and work on different IDEs (this process is documented in the corresponding repository[96]). Do not attempt to create labs locally as that approach does not have the same cross-platform guarantee.

[95]https://github.com/princomp/princomp.github.io/tree/main/source/lectures

[96]https://github.com/csci-1301/C-Sharp-project-maker

Content Labelling

Technically Quartz[97] support a powerful tagging system[98] which should be leveraged. Markdown files can contain at their very top a YAML metadata block[99] containing, e.g.

```
---
tags:
- Resource
---
```

to "tag" this resource with "Resource" so that it will appears in the tag listing[100]. To include multiple tags, simply make a list:

```
---
tags:
- Resource
- Guide
---
```

Conceptually We will follow the guidance provided on this page[101]:

- Use as Few Tags as Possible
- Limit Yourself to a Self-Defined Set of Tags
- Tags Within Your Set Must Not Overlap
- By Convention, Tags Are in Plural
- Tags Lower Case
- Tags Are Single Words
- Keep Tags on a General Level
- Omit Tags That Are Obvious
- Use One Tag Language
- Explain Your Tags

Styling and Templating

Templating files are under `source/templates/` directory. Templates directory contain layout files that are applied by pandoc when resources are built: note that the website's style uses a completely different mechanism.

For maintainability reasons it is preferable to apply templates during build time. This strategy makes it easy to edit templates later and apply those

[97] https://quartz.jzhao.xyz/authoring-content#syntax
[98] https://quartz.jzhao.xyz/features/folder-and-tag-listings#tag-listings
[99] https://pandoc.org/MANUAL.html#extension-yaml_metadata_block
[100] https://princomp.github.io/tags/Guide
[101] https://karl-voit.at/2022/01/29/How-to-Use-Tags/

changes across all resources. Avoid applying templating to individual resource files whenever possible.

Currently templates directory contains the following:

- `docx/` - contains template used to produce `.docx` files (this template is not used yet, for size issues[102]).
- `filters/` - contains pandoc filters for annotating code blocks, configured to default to C#, which then allows applying syntax highlighting to all code block.
- `html/` - contains template used to produce *only the book.html file* (to edit the style of the website, refer to editing website)
- `latex` - contains templates used to produce `.pdf` files,
- `docx/` - contains template used to produce `.odt` files.

Updating docx template

Note that this template is not used yet, for (among other) size issues[103].

To edit this template, start by obtaining the default template file:

```
pandoc -o custom-reference.docx --print-default-data-file
 ↪  reference.docx
```

Then, open `reference.docx`, and, following loosely this tutorial[104], do:

- Click pretty much anywhere, and right-click on the highlighted style (displayed if you are under "Home", you may need to scroll down the styles),
- Change the font for everything but the source code,
- Click on the "Block code", then right-click on the highlighted style, and select the font for the source code,
- The font for "Verbatim Char" was also changed, but I am not sure if this has an impact,
- Make sure the fonts are embedded[105],
- Save and close the document.

This was inspired by this post[106] but does not seem to work properly.

Updating odt template

First, output the default template file:

[102] https://github.com/csci-1301/csci-1301.github.io/issues/156

[103] https://github.com/csci-1301/csci-1301.github.io/issues/156

[104] https://support.microsoft.com/en-us/office/customize-or-create-new-styles-d38d6e47-f6fc-48eb-a607-1eb120dec563?ui=en-us&rs=en-us&ad=us

[105] https://support.microsoft.com/en-us/office/benefits-of-embedding-custom-fonts-cb3982aa-ea76-4323-b008-86670f222dbc

[106] https://stackoverflow.com/a/70513063

```
pandoc -o custom-reference.odt --print-default-data-file
 ↪  reference.odt
```

Then, open `reference.odt`, and, following loosely this tutorial[107], do:

- Click on View, then Styles.
- Right-click on "Preformatted Text", click on "Modify…", and then select the desired font family for source code.
- In the dialog or sidebar which opens make sure the button in the top panel marked with ¶ is highlighted (it is very subtle).
- In the menu at the bottom of the dialog/sidebar choose Applied Styles. Only "Default Paragraph Style" and "Footer" should appear.
- Right-click on "Default Paragraph Style", click on "Modify…", and then select the desired font family for the rest of the text.
- Then, highlight the A next to ¶.
- Right-click on "Source_Text", click on "Modify…", and then select the desired font family for source code.
- Click on File, then Properties, then on the Font tab, click on "Embed fonts in the document".
- Save and close the document.

Building locally

It is generally not necessary to build this resource locally unless the intent is to preview templating changes or to make changes to build scripts. For the purposes of editing content, it is sufficient to make edits to markdown files and commit those changes.

Installing dependencies To find the current list of dependencies needed to build this resource, refer to the build and deploy script install section[108]. The exact installation steps vary depending on your local operating system.

In general the following dependencies are needed:

- pandoc[109]
- texlive[110]
- make[111] and other standard unix utilities (such as sed or wget, all included in the Windows Subsystem for Linux[112]),

[107]https://github.com/jgm/pandoc/wiki/Defining-custom-DOCX-styles-in-LibreOffice-(and-Word)#libreoffice
[108]https://github.com/princomp/princomp.github.io/blob/main/.github/workflows/build_and_deploy.yaml
[109]https://pandoc.org/installing.html
[110]https://www.tug.org/texlive/
[111]https://www.gnu.org/software/make/
[112]https://learn.microsoft.com/en-us/windows/wsl/install

- python 3.+[113]
- packages and filters: Pygments[114], pandoc-include[115], texlive-xetex[116], texlive-latex-extra, lmodern, librsvg2-bin[117]
- symbola font

For this later, note that starting with version 11[118], the licence is too restrictive for non-personal use. As a consequence, users are asked to make sure they do not use a version greater than v.10.24, which is "free for any use" and archived on-line[119] (curious users can also refer to the related webpage[120]). Note that installing this dependency using a unix-like package manager will result in installing a version of the font that is free to use in any context[121].

You can make sure you are currently using the latest version of panflute by running

```
pip install -U panflute
```

This is needed if running a recent version of pandoc (as of pandoc 3.1.6.1 at least).

Running the build

⚠ Warning

Running `make all` can be *very resource-incentive* and may render your system unstable. Read this section entirely before running any command.

Testing the installation After installing all dependencies, from the `source/` folder, run:

```
make
```

[113]https://www.python.org/
[114]https://pygments.org/download/
[115]https://github.com/DCsunset/pandoc-include#installation
[116]https://tug.org/xetex/
[117]https://askubuntu.com/a/31446
[118]http://web.archive.org/web/20181228102842/http://users.teilar.gr/%7Eg1951d/Symbola.pdf
[119]http://web.archive.org/web/20180307012615/http://users.teilar.gr/~g1951d/Symbola.zip
[120]http://web.archive.org/web/20180307012615/http://users.teilar.gr/~g1951d/
[121]https://metadata.ftp-master.debian.org/changelogs//main/t/ttf-ancient-fonts/ttf-ancient-fonts_2.60-1.1_copyright

to display a list of useful rules.

It is recommended to first run a command building simple documents or copying files to test your installation, such as

```
make ../content/docs/about/credits.md
make ../content/docs/about/credits.pdf
make ../content/docs/about/credits.odt
make ../content/docs/about/credits.docx
make ../content/code/projects/Rectangle.zip
make ../content/web-order.ts
make ../content/img/create_project_monodevelop.png
make ../content/fonts/hack/hack-italic-subset.woff
```

If this was successful, you can compile the resources needed for the website using

```
make build-light
```

Building all resources You can run

```
make -l 2.5 -j$(nproc --ignore=2) all
```

to create and populate the `content/` folder at root level with all the resources compiled. Note that this command limits the number of jobs in parallel and the number of CPU used (using this trick[122]), but that tweaking those values[123] may be needed to find the sweet spot on your own machine.

If you want to speed-up the compilation time, you can run

```
make fetch
```

which will fetch the latest build output, extract it and populate the `content/` folder using its content. Due to make's unique feature[124] only the files whose source was edited will be re-created when executing `make all` the next time, hence saving *a lot* of time. However, please not that files moved or deleted will still be present in the build.

[122]https://stackoverflow.com/a/56607839
[123]https://stackoverflow.com/a/32487943
[124]https://makefiletutorial.com/

Website

Editing the website

The website https://princomp.github.io/ is built from the `.md` files contained in the `content/` folder using a dedicated branch[125] of quartz[126]. To edit the layout, style, or other features such as the footer, please *start by checking out the quartz branch* (using `git checkout quartz`), and then

- Refer to quartz's website[127], repository[128] and general community,
- Knowing that multiple edits already tweaked its style.

A couple of indications about the edits made to quartz:

- The favicon at `quartz/static/`, and have been generated using https://realfavicongenerator.net/.
- The order in the menu is constructed using the `content/web-order.ts` file, itself generated from the `source/order` file in the main branch: refer to the makefile (again, in the main branch) for explanations on how this file is created, to the quartz documentation[129] for the main inspiration, and to the `quartz.layout.ts` and `sortFn.ts` files for the concrete implementation. If you change the order, setting

```
useSavedState: true, // To debug the explorer, change to
↪  "false" (this way, the menu is not cached /
↪  permanent),
```

to `false` in the `quartz/components/Explorer.tsx` file *may* help in refreshing the menu more easily.

- Other files edited or created include:
  - The files
    `quartz/components/AlternativeFormats.tsx`
    `quartz/components/styles/alternativeFormats.scss`
    list alternative formats at the top of the page,
  - The files
    `quartz/components/Comments.tsx`
    `quartz/components/scripts/darkmode.inline.ts`
    `quartz/components/Footer.tsx`
    `quartz/components/styles/listPage.scss`

[125] https://github.com/princomp/princomp.github.io/tree/quartz
[126] https://quartz.jzhao.xyz/
[127] https://quartz.jzhao.xyz/
[128] https://github.com/jackyzha0/quartz
[129] https://quartz.jzhao.xyz/features/explorer#use-sort-with-pre-defined-sort-order

customize the footer and add a link to our repository feedback (while following the selected style[130]),
- `quartz/styles/base.scss` loads a different set of fonts,
- The files
  `quartz/components/Explorer.tsx`
  `quartz.layout.ts`
  tweak the menu and layout,
- `quartz.config.ts` sets meta-data about the website,
- `quartz/components/pages/404.tsx` customizes the 404 error message,
- `quartz/plugins/emitters/assets.ts` emits the `.md` files (they are not available by default),
- `quartz/components/index.ts` ties it all together.

Refer to Generate the git patch for instruction on how to generate a patch containing all the edits performed to our local copy of quartz.

Deploying locally the website

Follow closely those steps:

- Build the resource locally (note that running `make build-light` is enough to deploy the website).
- Move to the `quartz` branch by running

  ```
  git checkout quartz
  ```

  Note that the `content/` folder is still here, but that the source is absent from this branch: only files related to quartz are committed in this branch.
- Rename the `content/index.md` file (this is due to an annoying bug[131]) by running

```
mv content/index.md content/index_b.md
```

- Follow quartz's instructions[132]:
  - If you don't have at least Node v18.14 and npm v9.3.1, install node[133] and npm[134] (npm is probably installed automatically when you install node),
  - Run the following commands *at root level* (do *not* enter the `quartz/` folder):

[130]https://github.com/jackyzha0/quartz/issues/1161
[131]https://github.com/jackyzha0/quartz/issues/1175
[132]https://quartz.jzhao.xyz/#-get-started
[133]https://nodejs.org/en/download/package-manager
[134]https://github.com/npm/cli?tab=readme-ov-file#installation

```
npm i
npx quartz create
```

for this last command, select

> • Empty Quartz

then,

> • Treat links as shortest path ((default))

- If the previous command succeeded, run

```
mv content/index_b.md content/index.md
```

to restore our index file, then

```
npx quartz build --serve
```

to start the server. Then, navigate to `localhost:8080/` to see the website deployed locally.

Updating quartz

Our local copy of quartz, in the `quartz` branch[135], is "frozen" in the sense that it corresponds to the development of quartz at a point of time. It is possible to

1. Save the edits made to our local copy (as a git patch[136]),
2. Pull the current version of quartz in a different branch (called `quartz-update`),
3. Apply our edits to this updated version of quartz,
4. Replace the `quartz` branch with the `quartz-update` branch to deploy the updated version of quartz with our edits.

This process is not without risks and requires to be able to deploy locally the website to test it before deploying it. The following guide was inspired by this discussion[137].

Generate the git patch The first step is to save as a git patch all the edits that have been made on our local copy of quartz since it was last updated.

- Make sure you are
    1. At root level in your repository's copy,
    2. In the `quartz` branch,
    3. That your branch is up-to-date.

[135]https://github.com/princomp/princomp.github.io/tree/quartz
[136]https://git-scm.com/docs/git-apply
[137]https://github.com/jackyzha0/quartz/issues/1145

by running a command such as

```
pwd && git checkout quartz && git pull
```

- Locate the commit (short) `id` of the last commit performed by quartz maintainer. A way of achieving this is to look for "PCP" in the commit messages, using

  ```
  git rev-parse --short :/PCP
  ```

  and then to look for the commit id of the commit that came *before* it. For instance, if the previous command returns `847e3356`, then the command

  ```
  git rev-parse --short 847e3356^1
  ```

  will return information about the commit that came before that last commit: we will assume its (short) id is `3b74453f` in the following.

  Visual inspection using github's interface[138] or a program such as gitk[139] can facilitate this process. Note that removing the `--short` option will give the *long* version of the id, which may be harder to compare.

- Use the (short) id previously obtained to generate a patch containing all the changes made since that commit:

  ```
  git diff-index 3b74453f --binary > pcp_quartz_patch
  ```

  The `--binary` option insures that any file created will be included in the patch: as a result, this file can be heavy.

- Make sure this `pcp_quartz_patch` file is located at the root level in your repository's copy but do not commit it to the repository.

Clone the latest version of quartz Execute the following commands:

```
git remote add quartz
 ↪  https://github.com/jackyzha0/quartz.git
git fetch quartz
git checkout -b quartz-update quartz/v4
```

where `quartz-update` is the name we use for our branch, and `quartz/v4` is the name of the branch in the quartz repository we want to copy.

Apply the git patch There are two ways of applying the patch. First, make sure you are in the `quartz-update` branch by executing

[138] https://github.com/princomp/princomp.github.io/commits/quartz/
[139] https://git-scm.com/docs/gitk

```
git rev-parse --abbrev-ref HEAD
```

Then follows the first method if possible.

Using apply First, check if the `pcp_quartz_patch` patch is applicable, by executing

```
git apply --ignore-space-change --ignore-whitespace
 ↪  --check --reject pcp_quartz_patch
```

Some sections of the patch may be rejected: make sure you take note of which file will need to be merged by hand. Then, apply the patch, using

```
git apply --ignore-space-change --ignore-whitespace
 ↪  --reject pcp_quartz_patch
```

Then look for the `.rej` files: they will contain the edited version of a file that you will need to merge manually with the updated version of the same file from quartz's update.

Using patch If `git apply` gave an error starting with

```
Checking patch quartz.layout.ts...
error: while searching for:
```

then, instead, do

```
patch -p1 < pcp_quartz_patch
```

And look for the `.rej` files as described above. Note that using this technique *requires to copy the binary files by hand*. Indeed, you should receive warning messages like

```
File quartz/static/android-chrome-192x192.png: git binary
 ↪  diffs are not supported.
```

and those files will have to be copied by hand from another branch, and / or re-added to the repository.

Testing Once you are done manually merging, **test** your updated version by deploying locally the website and making sure that quartz does not return any error. If everything looks ok, add all the new files and commit the edits using a message containing the "PCP" string (to facilitate future generation of git patch), and push, using for example:

- First, use `bash    git add --all -n .` to list all the files you are about to add: make sure you are not adding files from the `content/` folder, for instance. If everything looks fine, proceed to the next step.

- Then, actually add the files, commit, and push, using:

  ```
  git add --all
  git commit -a -m "Applying previous PCP patch."
  git push origin quartz-update
  ```

Update the branch **If you were able to fix all the conflicts and to check that the website could still be deployed locally**, then overwrite the `quartz` branch with the `quartz-update` branch, by executing[140]:

```
# Make sure your working tree is in a clean state
git status

# Check out the branch you want to change, e.g.
↪ some-branch
git checkout quartz

# Reset that branch to some other branch/commit, e.g.
↪ target-branch
git reset --hard quartz-update
```

If the deployment was successful and everything seems to be working, you can delete the quartz-update branch, locally then remotely, by executing

```
git branch -D quartz-update
git push -d origin quartz-update
```

Repository Maintenance

This repository uses following tools and technologies:

- git - version control
- Github - to make source code available on the web
- markdown, LaTeX - for writing the resources
- pandoc - for converting documents to various output formats
- make - for specifying how to build this resource
- github actions - to automatically build the resource
- github pages - to serve the accompanying website
- additional packages for specific tasks: texlive, Pygments, pandoc filters, lua filter[141], etc.
- fonts-symbola - to produce the emoji and other symbols in the pdf document.
- utteranc.es[142] - for feedback through website

[140] https://www.reddit.com/r/git/comments/bqx85v/comment/eo8j4zh
[141] https://github.com/jgm/pandoc/issues/2104
[142] https://utteranc.es/

- csharpier[143] - to tidy the C# source code

Build outputs

The resource material is organized into specific directories inside the `source/` folder. These resources are then compiled into templated documents in various formats using pandoc[144]. The makefile explains the exact steps applied to each type of resource.

Github actions

This resource is built automatically every time changes concerning files in the `source/` folder are committed to the main branch of the repository. This is configured to run on Github actions[145]. The workflow[146] that is automatically triggered has two jobs: one to build the resource, and one to deploy it.

Currently Github actions offers unlimited free build minutes for public repositories (and 2000 min/mo. for *private* repositories, should we ever need them), which hopefully continues in perpetuity (if it does not there are other alternative services). Going with one specific CI service over another is simply a matter of preference.

Following a successful build, the build script will automatically deploy the generated resources to an accompanying website hosted on github pages[147].

Fetch and No Fetch Versions There is a second workflow[148] that is identical to the first one with one important exception: to speed up compilation, `build_and_deploy.yaml` uses `make fetch` to speed up compilation time by re-downloading the latest build output, and then compiling only the required files. This can sometimes complicate the propagation of changes, typically if a template is modified (as this does not triggers a re-compilation of the files using it currently) or if a file is renamed (as the previous version will not be deleted).

[143] https://github.com/belav/csharpier

[144] https://pandoc.org/MANUAL.html

[145] https://github.com/features/actions

[146] https://github.com/princomp/princomp.github.io/blob/main/.github/workflows/build_and_deploy.yaml

[147] https://pages.github.com/

[148] https://github.com/princomp/princomp.github.io/blob/main/.github/workflows/build_and_deploy_no_fetch.yaml

The build_and_deploy_no_fetch.yaml[149] can be triggered manually[150] to force a "fresh" remote compilation.

Creating releases

Currently a github action is setup to do the following: whenever a new commit is made to the main branch, the action will build the resource and add the generated resources as a pre-release[151] and tag them as "latest"[152]. If a subsequent commit occurs it will overwrite the previous latest files and become the new latest version. This cycle continues until maintainers are ready to make a versioned release (or "package").

Making a versioned release is done as follows:

1. Go to repository releases[153]
2. Choose latest, which contains the files of the latest build
3. Edit this release, giving it a semantic name and a version, such as v1.0.0. Name and version can be the same. (cf. semantic versioning[154])
4. Enter release notes to explain what changed since last release
5. Uncheck "This is a pre-release"
6. Check "Set as the latest release"
7. Update release

Following these steps will generate a new, versioned release. The versioned releases will be manually uploaded to and archived on galileo.

Once this is done, remember to create the next pre-release:

1. Go to the repository releases[155].
2. Click on "Draft a new release".
3. Pick the tag "Latest".
4. Click on "Generate release notes"
5. Check "This is a pre-release"
6. Click on "Publish release"

[149]https://github.com/princomp/princomp.github.io/blob/main/.github/workflows/build_and_deploy_no_fetch.yaml

[150]https://github.com/princomp/princomp.github.io/actions/workflows/build_and_deploy_no_fetch.yaml

[151]https://github.com/princomp/princomp.github.io/releases

[152]https://github.com/princomp/princomp.github.io/releases/tag/latest

[153]https://github.com/princomp/princomp.github.io/releases

[154]https://semver.org/

[155]https://github.com/princomp/princomp.github.io/releases

Maintaining repository feedback

Resource users can submit feedback about the resource through various means, one of which is leaving comments on the website. This feature is enabled by utteranc.es[156], using repositories hosted by the `princomp` github organization[157].

To manage user feedback over time, a semester-specific repository is created for issues only. This must be a public repository and located under the same organization as the resources repository. utteranc.es widget is configured to point to this repository. After a semester is over, this feedback repository will be archived, and a new one created for the next semester. This will simultaneously archive all older issues and reset the feedback across website pages.

Migrating feedback repository The steps for migrating feedback target repository are as follows:

1. Create a new **public** repository under `princomp` github organization[158]. Follow the established naming convention (`feedback-<fall|spring|summer>-<YYYY>`), and leave all the options except for visibility (which needs to be set to public) by default.
2. Go to repository Issues (make sure issues is enabled in repository settings).
3. Create a new label whose *label name* is `comment` (to match widget configuration as indicated in `quartz/components/Footer.tsx`, in the `quartz` branch).
4. Go to `Organization Settings > Installed GitHub Apps`[159].
5. Choose "utterances" > "configure"
6. Under "Repository access" > "Only select repositories"
   - Select the repository created in step 1.
   - Remove the previous semester feedback repository.
   - Save.
7. In `princomp/princomp.github.io/` repository, in the `quartz` branch[160], open `quartz/components/Footer.tsx`

[156] https://utteranc.es/
[157] https://github.com/princomp
[158] https://github.com/organizations/princomp/repositories/new
[159] https://github.com/organizations/princomp/settings/installations
[160] https://github.com/princomp/princomp.github.io/blob/quartz/quartz/components/Footer.tsx

8. Update utteranc.es widget code to point to the new feedback repository created in step 1.

   ```
   <script data-external="1"
           src="https://utteranc.es/client.js"
           repo="princomp/{REPOSITORY_NAME}"
           label="comment" …>
   </script>
   ```

9. Commit change to `quartz/components/Footer.tsx`
10. Make sure the feedback works after migration. If it does not, retrace your steps.
11. Archive the earlier feedback repository in its settings.

Maintaining Instructors / G/UCA rights

This is handled by the `csci-1301` github organization[161] and documented at https://csci-1301.github.io/user_guide.html#maintaining-instructors-guca-rights.

How to get Help

ⓘ Info

This page is primarily targeted for Augusta University students.

This page lists resources for Augusta University students to receive help with their course of studies, in general, for students of the School of Computer and Cyber Sciences, and for this course in particular.

In General

Many resources are available to help you be a successful student:

- If you are food insecure, you are not alone[162], and the Open Paws Food Pantry[163] will help you.
- For tutoring resources, consult the Academic Success Center[164] (or "ASC"). It can help you, among other things, in the areas of time management, test preparation and study strategies.

[161] https://github.com/csci-1301

[162] https://www.wjbf.com/csra-news/nearly-36-percent-of-college-students-are-hungry/

[163] https://www.augusta.edu/student-affairs/open-paws.php

[164] https://www.augusta.edu/academicsuccess/

- Student Counseling & Psychological Services[165] (or "SCAPS") is here to assist students with a variety of personal, developmental, and mental health concerns.
- The Writing Center[166] can help you with any written, oral, or multimedia project.
- To get help with technologies, refer to our Instructional Technology Support[167] correspondent Sienna Sewell[168].
- The Department of Multicultural Student Engagement (MSE)[169] aims to provide education, training, and programming to foster awareness of diversity and inclusion among Augusta University students. Their Multicultural Mentorship Program and African American Male Initiative[170] are excellent resources to receive additional help.

For Students of the School of Computer and Cyber Sciences

School of Computer and Cyber Sciences Tutoring Center

The School has a tutoring center that can be reached:

- On discord[171],
- During their tutoring hours (hours posted on the door and on discord), in University Hall[172] 129.

ACM Club

The Augusta University chapter[173] of the A.C.M[174] is one of the university's best resources for Computer Science, Information Technology and Cyber Security students. It provides a platform to network with other students in similar majors, presenting countless opportunities to not only expand the people you know, but also a fantastic place to learn and ask questions. To learn more, you can sign up for the newsletter, or attend one of the subgroup meetings (meeting times and locations are listed on the website[175]).

[165] https://www.augusta.edu/counseling/
[166] https://www.augusta.edu/cwe/
[167] https://www.augusta.edu/continuity/index.php
[168] https://spots.augusta.edu/sSewell/
[169] https://www.augusta.edu/multicultural/
[170] https://www.augusta.edu/multicultural/programming.php
[171] https://discord.gg/AYSw3UNKEh
[172] https://map.concept3d.com/?id=824#!m/268018
[173] https://spots.augusta.edu/cyberdefense
[174] https://www.acm.org/
[175] https://spots.augusta.edu/cyberdefense

Other Club Activities

The Augusta University Game Design Club and Girls Who Code College Loop "will be continuing activities in full force this year". Notifications for upcoming activities will be shared in class alongside school-wide emails.

How to Ask a Question?

It may seems silly, but asking a question "the right way" may not always be easy.

1. Once you've identified your issue, try again from scratch to see if you missed a point.
2. Go over the instructions, and look in our resources[176] for some meaningful keywords.
3. Think about how you can describe your issue, what is the shortest route to reproduce it.
4. If you are still facing difficulties, be detailed and clear about what you think went wrong: if the question is related to computers, specify which operating system, what you have tried, the exact nature of the error message, etc. Screenshots are not always the right way to convey your question: try to be descriptive, and explain what you tried. If you want to refer to a particular lab or lecture, open the corresponding page, look for the closest title, hover over it, and you should see a "§" symbol appears: click on it, you can now share that link[177] so that your interlocutor knows precisely what you are talking about!

And, remember: your instructor(s) knows that you are a student and here to learn, so you should *never* feel intimidated or assume that *everyone knows better than you*: many students struggle in this class at times, and you could actually do them all a favor by asking your instructor(s) to go over a particular dimension that they may have overlooked or explained poorly!

Commenting Using a Github Account

On this website, if you look below, you will see a box where you can comment. This will require that you create a Github account[178], which is free and may serve multiple purpose if you intend to study, use, or contribute to open-source projects. The comment can use the markdown syntax[179]

[176]https://github.com/princomp/princomp.github.io/search?q=ask+a+question
[177]https://www.wikihow.com/Copy-and-Paste-a-Link
[178]https://github.com/login
[179]https://commonmark.org/

(exactly like this resource!), which is also used on websites like stackoverflow[180] and extremely popular!

Choosing Your Major

ⓘ Info

This page is primarily targeted for Augusta University students.

Which degree is best for you?

Most universities offer both a Computer Science degree and an Information Technology degree, and some universities even offer a Management Information Systems degree. Here at Augusta Unversity[181], we have all three options for you:

- Computer Science[182] (CS / CSCI),
- Information Technology[183] (IT / AIST),
- and Cybersecurity[184] (CYBR),

along with two unique diploma,

- Cybersecurity Engineering[185],
- and Cyber Operations[186].

While all of these degrees are high-quality and should place students on a fast-track towards a successful career, students always ask the same question, "*Which degree is best for me?*" The answer to this question depends on the student, their career goals, and a variety of other factors.

Students even ask more specific questions:

- Which degree will give me the highest salary?
- Which degree is easiest?
- Which degree is hardest?
- Which degree has the most job opportunities?

[180]https://stackoverflow.com/editing-help
[181]https://www.augusta.edu/ccs/programs.php
[182]https://www.augusta.edu/ccs/bs-cs.php
[183]https://www.augusta.edu/ccs/bs-it.php
[184]https://www.augusta.edu/ccs/bs-it-cybersecurity.php
[185]https://www.augusta.edu/ccs/bs-cybersecurity-engineering.php
[186]https://www.augusta.edu/ccs/bs-cyber-ops.php

These are all great questions! But before answering them, it is more important to have a basic understanding of the degree options.

The following links detail these three degrees and explain the benefits of each:

- Difference Between a Computer Science & Information Technology Degree[187]
- Computer Science vs Information Systems/Technology[188]
- Degrees that Pay You Back (from Wall Street Journal)[189]

Additionally, Augusta University has more information on its advising page[190]. To answer the first question ("*Which degree will give me the highest salary?*"), you can use Georgia Degrees Pay[191]

Summary

Computer scientists design and develop computer programs, software, and applications. *IT and IS professionals* then use, configure, and troubleshoot those programs, software, and applications.

So it really depends on what you want to do. Do you want to be on the front end, designing the software and applications? Do you prefer to use and troubleshoot them? One of the websites gave the analogy of a home: computer scientists build the home, set up home, install the lighting, plumbing, etc., and then the IT/IS professionals come and live in the home to use it, test it, and troubleshoot it.

So which degree is "best"?

Perhaps you can now see how this question is not fair or at least not clear. If we ask which degree is more difficult, the students will immediately exclaim, "Computer Science is the most challenging!" Therefore, one can perhaps argue that the Computer Science degree is the most rigorous (challenging) and will likely provide the student with more opportunities in their career. And the salary statistics support this argument, as CS students, on average, have a higher salary than their IT and IS colleagues.

That said, is Computer Science better? Yes, and no. It depends on **you**! It depends on your goals. It depends on how hard you want to work.

[187]http://online.king.edu/information-technology/difference-between-a-computer-science-information-technology-degree/

[188]https://www.geteducated.com/careers/521-computer-information-systems-vs-computer-science

[189]http://online.wsj.com/public/resources/documents/info-Degrees_that_Pay_you_Back-sort.html

[190]https://www.augusta.edu/advising/

[191]https://www.usg.edu/georgia-degrees-pay

For some, "better" means more money and more career opportunities. For others, "better" means easier studies and less math! So again, which degree is "best"? There is no short answer. As mentioned above, all three degrees provide the tools you need to hopefully have a great career. Perhaps the question is best worded as, "Which Degree is Best for **me**?" And of course, only **you** can answer this question!

Course Assistants

ⓘ Info

This page is primarily targeted for Augusta University students.

What Is an Undergraduate Course Assistant?

In this course, an Undergraduate Course Assistant (UCA) is generally present in addition to your instructor. A UCA is a student, generally in the School of Cyber and Computer Sciences, who successfully passed CSCI 1301 and that is hired by the School to assist other students.

Their duties generally include:

- Helping the students during the labs,
  - To set-up their computers,
  - To find the right resources,
  - To understand their IDE's error messages,
  - To investigate bugs with them,
  - etc.
- Helping the students outside of the lab (through email, teams, or office hours), for similar tasks as in lab, but also to get ready for an exam or a quiz,
- Reporting to the instructors any issue, mistake or confusion they noticed,
- Suggesting improvements to the resources shared with the students.

Their duties *can not* include:

- Understanding for you[192],
- Helping you or even commenting on graded material *before* it was graded,
- Grading students' work,
- Helping you with other classes,

[192] Although that may sound curious, we believe it is important to remind you of the fact that they can only *help* you understanding, but that you have to do your part!

- Helping you becoming a self-regulated learner and work on your schedule[193].

How Do I Become One?

A UCA is hired by the School upon recommendation of instructors, after discussion with our Academic Program Coordinator, and possibly our Director of Undergraduate Studies.

A UCA *must*:

- Be a student, that is, currently enrolled in courses, or, if during the Summer, being enrolled in courses for the next Fall semester,
- Pass our Human Resources background check,
- Have an interest in tutoring,
- Clearly understand the limits and boundaries to the help they can provide to students.

Additionally, if a student wants to help with this particular class, then the student must have successfully passed CSCI 1301 with a grade of B or higher

A UCA will:

- Be able to work up to 25 hours per week (an average of 10 hours per week is typical, but needs to be discussed with the instructor), paid $12.50 per hour, without other benefits,
- Be adequately trained to use our platforms and edit our resources,
- Be able to work on Campus and discuss their schedule with their referent instructor,
- Develop a stronger bond with the instructors, facilitating possible future reference or research projects.

So, in short: talk to any CSCI 1301 instructor if you feel like becoming a UCA.

I Am a UCA, What Should I Do Now?

Congratulations! You should now read more about your position in the UCA starting guide[195]!

What Is the Difference With a GRA?

Graduate Course Assistants (GRA) hold a bachelor and are generally PhD or Master student. Their duties generally overlap with those of the

[193]That's a job really well taken care of by the Academic Success Center[194]!

[195]uca_guide.html

instructors and those of the UCAs, as they are the first point of contact of UCAs, design projects, organize the schedule of the tutoring center and of the labs.

What Is the Difference With a URA?

Undegraduate *Research* Assistants (or "URAs") share many similarities with UCAs:

- They both are students employed by the University,
- They both have a maximum of 25 hours/week,
- Their pay rates are the same,
- They both work under the direction of a Faculty member[196].

However, their focus is on working on *research*[197] instead of being focused on teaching. The difference is sometimes tenuous, but URAs positions are generally given in priority to "advanced" students (that is, close to graduation), to use their gained knowledge to push further the limits of human knowledge!

It is not possible to cumulate an URA *and* an UCA position, but obtaining an UCA position is in general an excellent stepping stone to obtain a URA position, if you wish to do so: by proving that you are reliable, serious, agreeable to work with, you will maximize your chances of having a Faculty member notice you and offer you to work on their research with them.

UCA starting guide

ⓘ Info

This page is primarily targeted for Augusta University students.

Congratulations on your new position! This page briefly explain what is expected from you as an Undergraduate Course Assistant (UCA).

The Three Rules

There are three important rules for you:

1. **This is a job.**

[196]https://www.augusta.edu/ccs/faculty.php
[197]https://www.augusta.edu/ccs/research.php

Meaning that you have a contract that you should have read and understood, and that you need to carefully clock in and out to receive the pay you deserve. Briefly reviewing the information listed here[198], and in particular those slides[199] can help you in making sure that you understand all aspects of your position. Do not forget that you are first and foremost a *student*, and that your main goal here is to *graduate*.

2. **You are here to help students, not to solve their problems.**

   Please, review what you should and should not do on this section[200]. It is difficult to strike the right balance when helping a student, but a good rule of thumb is that you should not do anything yourself, just explain and give hints so that they can solve the problem they are facing. You are here to help students understand how to solve a problem, not to solve it for them.

3. **Don't hesitate to ask.**

   That's it. You are not alone to deal with difficult situations (cheating, rude behavior, student abusing your time, etc.), and it is normal if you are sometimes unsure of the best course of action. The instructors are happy to train you and help you solve problems that may arise.

In general, UCAs should prioritize giving clear and concise explanations and hints, as to avoid confusion while also helping them better understand the problem-solving process. This means that when you encounter a problem that you are not able to solve, it's important to ask a colleague who is available for help and try to understand their approach. This way, the student can receive assistance more quickly and will be less likely to get confused during the troubleshooting process. By emphasizing the importance of understanding and working through the problem, rather than just providing a solution, tutors can help students develop the skills they need to become more independent problem-solvers.

On top of supporting students and helping the instructor, you are also encouraged to work on the improvement of those resources. Your contribution may range from spell-checking to pointing inconsistencies, from clarifying statements to re-organizing exercises. Thanks to git and pull requests[201], you do not need to worry (too much) about introducing mistakes or blunders: the changes you suggest will always be reviewed by instructors before being merged in our master document. We discuss

[198] https://www.augusta.edu/hr/university/university_benefits/studenthires.php

[199] https://www.augusta.edu/hr/university/university_benefits/documents/department_guide_studenthireprocess_fy_23.pdf

[200] ca.html#what-is-an-undergraduate-course-assistant

[201] https://github.com/princomp/princomp.github.io/pulls

below how you can edit our resources.

Editing the Resources

You need three things to start editing our resources:

- A github account & an invitation,
- Some working knowledge of markdown,
- Some working knowledge of github's interface.

Follow the instructions in our "Contributing Guidelines"[202] for the first step.

For a quick syntax guide in Markdown, the best resource is this website[203] and its 10 minutes tutorial[204]. We list some best practices[205], and would appreciate if you could follow them.

For github's interface, please refer to the following guide (where the screenshots where taken for the csci-1301.github.io[206] website, but remains relevant).

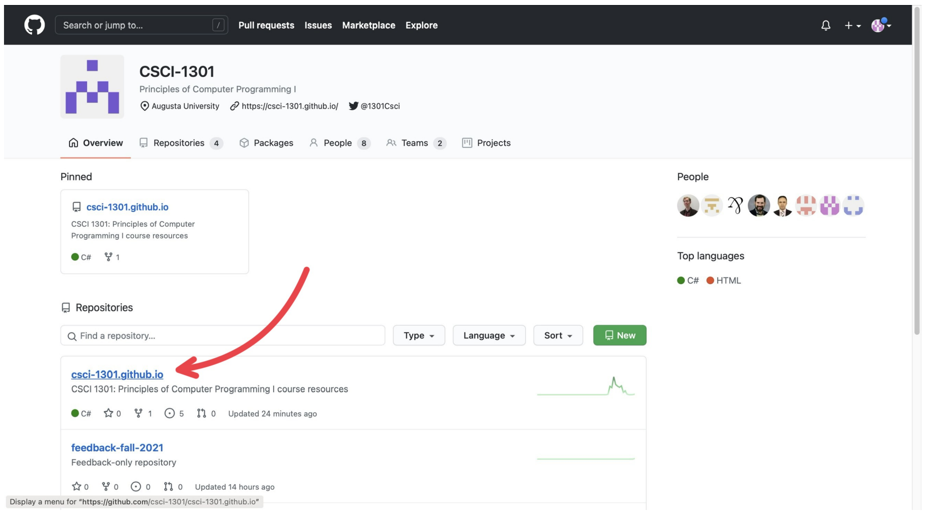


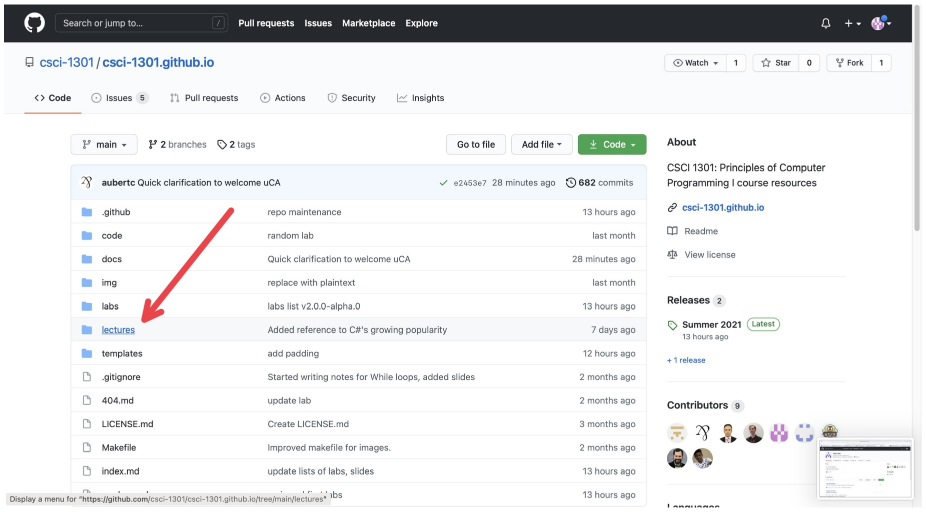
Figure 1: "Navigating repositories"

GitHub is separated into many "repositories":

- The **princomp.github.io** contains most of the resources that will be used (so it will be where you will navigate to the most),

[202] contributing.html#if-you-are-a-uca
[203] https://commonmark.org/help/
[204] https://commonmark.org/help/tutorial/
[205] https:/princomp.github.io/docs/about/dev_guide#editing-resources
[206] https://github.com/csci-1301/csci-1301.github.io

- The **uca-resources–YYYY** is a *private* repository where material useful to UCAs but not accessible to students (such as project solution, listings, etc.) will be shared,
- The **feedback–YYYY** and similarly named repositories contains feedback submitted by students/users.



aubertc Quick clarification to welcome uCA e2453e7 28 minutes ago 682 commits

.github	repo maintenance	13 hours ago
code	random lab	last month
docs	Quick clarification to welcome uCA	28 minutes ago
img	replace with plaintext	last month
labs	labs list v2.0.0-alpha.0	13 hours ago
lectures	Added reference to C#'s growing popularity	7 days ago
templates	add padding	12 hours ago
.gitignore	Started writing notes for While loops, added slides	2 months ago
404.md	update lab	2 months ago
LICENSE.md	Create LICENSE.md	3 months ago
Makefile	Improved makefile for images.	2 months ago
index.md	update lists of labs, slides	13 hours ago
	abs	13 hours ago

Display a menu for "https://github.com/csci-1301/csci-1301.github.io/tree/main/lectures"

About

CSCI 1301: Principles of Computer Programming I course resources

csci-1301.github.io

Readme

View license

Releases 2

Summer 2021 Latest
13 hours ago

+ 1 release

Contributors 9

Figure 2: "Navigating folders"

Under the **Code** section (next to Issues, Pull Requests, Actions, etc.), you will find various folders containing documents for the website. Typically, if there is some error or mistake in the lecture notes, so that will be where you will navigate to the most. The way the resources are organized is explained here[207].

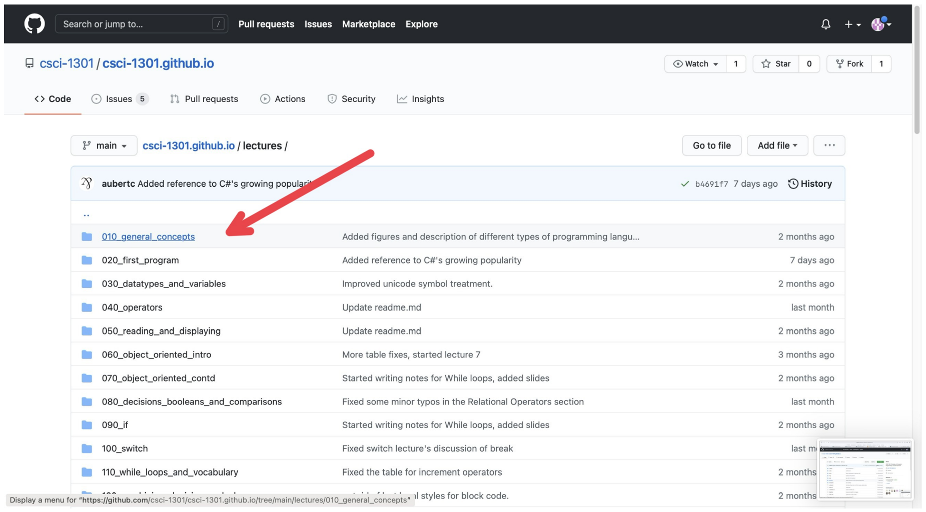
For this example, I just clicked on the first chapter, "General Concepts".

On this page, you can see the edit history of that specific document you clicked on. In the corner above the document and below the edit history, there is a pencil icon that will put you into editing mode for that document.

On this page, you will see the document formatted as markdown with two sections at the top of the document: *Edit file* and *Preview*. If you have *Edit file* selected, then you will see the "code" version of the document whereas if you click on the *Preview* button, you will see the document in its "final" form, or how the website users should see it, without the "code". To edit, make sure you have *Edit file* selected.

Once you have made the edits you wanted, you need to "commit"

[207]https:/princomp.github.io/docs/about/dev_guide#resources-organization-overview

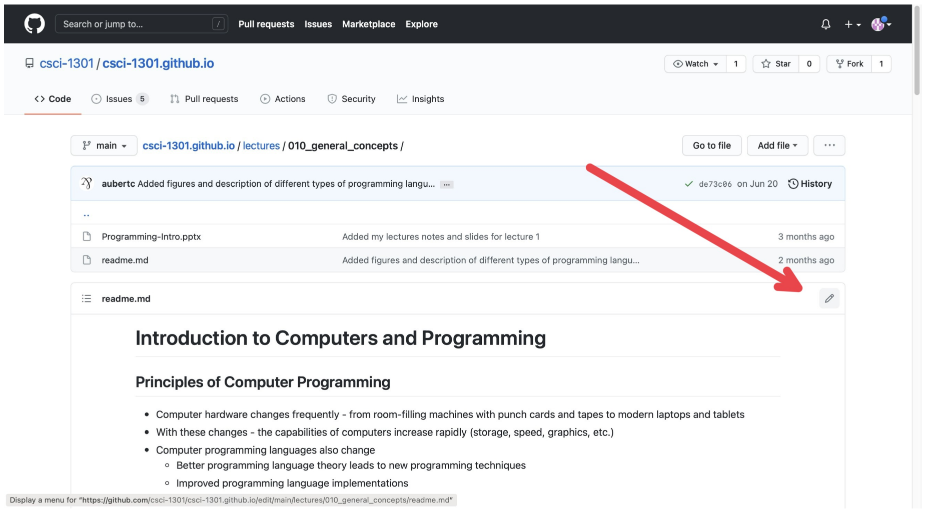


aubertc Added reference to C#'s growing populari... b4691f7 7 days ago

..		
010_general_concepts	Added figures and description of different types of programming langu...	2 months ago
020_first_program	Added reference to C#'s growing popularity	7 days ago
030_datatypes_and_variables	Improved unicode symbol treatment.	2 months ago
040_operators	Update readme.md	last month
050_reading_and_displaying	Update readme.md	2 months ago
060_object_oriented_intro	More table fixes, started lecture 7	3 months ago
070_object_oriented_contd	Started writing notes for While loops, added slides	2 months ago
080_decisions_booleans_and_comparisons	Fixed some minor typos in the Relational Operators section	last month
090_if	Started writing notes for While loops, added slides	2 months ago
100_switch	Fixed switch lecture's discussion of break	last m
110_while_loops_and_vocabulary	Fixed the table for increment operators	2 months
	l styles for block code.	2 months

Display a menu for "https://github.com/csci-1301/csci-1301.github.io/tree/main/lectures/010_general_concepts"

Figure 3: "Navigating documents"



aubertc Added figures and description of different types of programming langu... de73c06 on Jun 20

..		
Programming-Intro.pptx	Added my lectures notes and slides for lecture 1	3 months ago
readme.md	Added figures and description of different types of programming langu...	2 months ago

readme.md

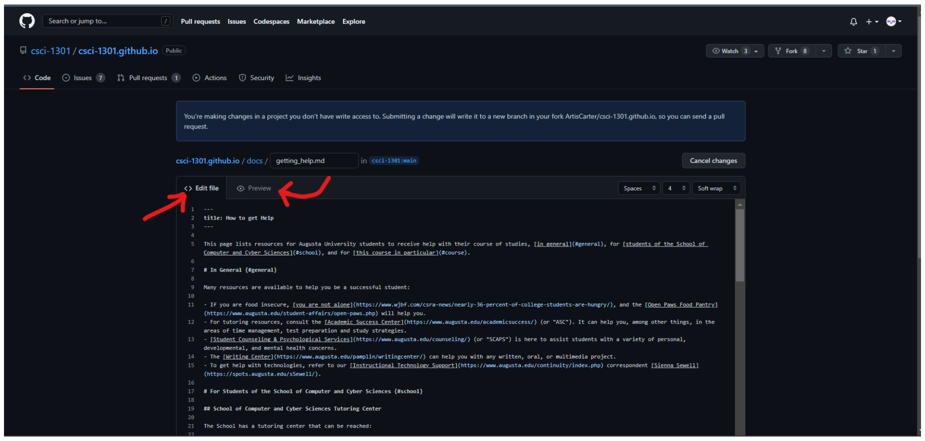
Introduction to Computers and Programming

Principles of Computer Programming

- Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets
- With these changes - the capabilities of computers increase rapidly (storage, speed, graphics, etc.)
- Computer programming languages also change
  - Better programming language theory leads to new programming techniques
  - Improved programming language implementations

Display a menu for "https://github.com/csci-1301/csci-1301.github.io/edit/main/lectures/010_general_concepts/readme.md"

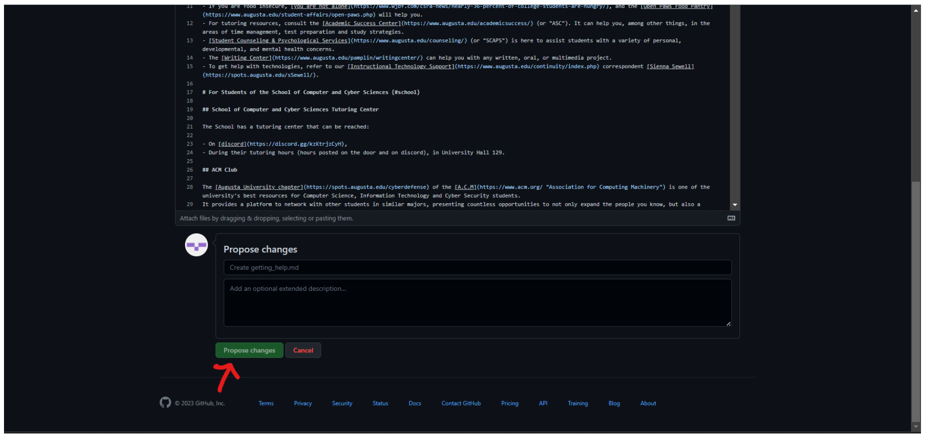
Figure 4: "Editing Mode"



You're making changes in a project you don't have write access to. Submitting a change will write it to a new branch in your fork ArtisCarter/csci-1301.github.io, so you can send a pull request.

```
---
title: How to get Help
---

This page lists resources for Augusta University students to receive help with their course of studies, [in general](#general), for [students of the School of Computer and Cyber Sciences](#school), and for [this course in particular](#course).

# In General {#general}

Many resources are available to help you be a successful student:

- If you are food insecure, [you are not alone](https://www.wjbf.com/csra-news/nearly-36-percent-of-college-students-are-hungry/), and the [Open Paws Food Pantry](https://www.augusta.edu/student-affairs/open-paws.php) will help you.
- For tutoring resources, consult the [Academic Success Center](https://www.augusta.edu/academicsuccess/) (or "ASC"). It can help you, among other things, in the areas of time management, test preparation and study strategies.
- [Student Counseling & Psychological Services](https://www.augusta.edu/counseling/) (or "SCAPS") is here to assist students with a variety of personal, developmental, and mental health concerns.
- The [Writing Center](https://www.augusta.edu/pamplin/writingcenter/) can help you with any written, oral, or multimedia project.
- To get help with technologies, refer to our [Instructional Technology Support](https://www.augusta.edu/continuity/index.php) correspondent [Sienna Sewell](https://spots.augusta.edu/sSewell/).

# For Students of the School of Computer and Cyber Sciences {#school}

## School of Computer and Cyber Sciences Tutoring Center

The School has a tutoring center that can be reached:
```

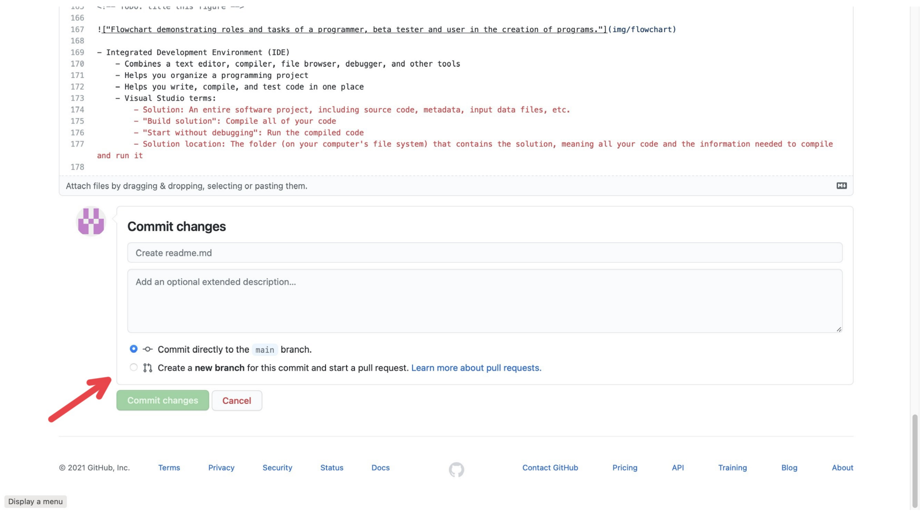
Figure 5: "Editing vs Previewing"



```
- If you are food insecure, [you are not alone](https://www.wjbf.com/csra-news/nearly-36-percent-of-college-students-are-hungry/), and the [Open Paws Food Pantry](https://www.augusta.edu/student-affairs/open-paws.php) will help you.
- For tutoring resources, consult the [Academic Success Center](https://www.augusta.edu/academicsuccess/) (or "ASC"). It can help you, among other things, in the areas of time management, test preparation and study strategies.
- [Student Counseling & Psychological Services](https://www.augusta.edu/counseling/) (or "SCAPS") is here to assist students with a variety of personal, developmental, and mental health concerns.
- The [Writing Center](https://www.augusta.edu/pamplin/writingcenter/) can help you with any written, oral, or multimedia project.
- To get help with technologies, refer to our [Instructional Technology Support](https://www.augusta.edu/continuity/index.php) correspondent [Sienna Sewell](https://spots.augusta.edu/sSewell/).

# For Students of the School of Computer and Cyber Sciences {#school}

## School of Computer and Cyber Sciences Tutoring Center

The School has a tutoring center that can be reached:

- On [discord](https://discord.gg/kzKtrjzCyH),
- During their tutoring hours (hours posted on the door and on discord), in University Hall 129.

## ACM Club

The [Augusta University chapter](https://spots.augusta.edu/cyberdefense) of the [A.C.M](https://www.acm.org/ "Association for Computing Machinery") is one of the university's best resources for Computer Science, Information Technology and Cyber Security students.
It provides a platform to network with other students in similar majors, presenting countless opportunities to not only expand the people you know, but also a
```

Figure 6: "Proposing Changes"

them; just like how you may write a paper, you need to submit it to the professor for them to see it. At the bottom of the page, there is a header box and a description box for you to describe what you did so others will know the changes you did (you do not need to go into *every* detail; just describe it generally, like "I fixed grammatical issues" or "Fixed code error"). As a UCA, you do not have write access to the **princomp.github.io** repository, so submitting a change will write it to a new branch in your fork <`your name`>/princomp.github.io, so you can send a pull request. Given the new protocol by Github, after making the neccessary edits, click the "Propose Changes" button located at the bottom. On this page and the next, there will be a "Create pull request" button, by clicking on this you will start a pull request. After you have successfully created a new branch for your commit and started a pull request, your edits will be checked by others so as to catch any mistake(s) you may have introduced before your pull request is merged into the base branch.



```

!["Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs."](img/flowchart)

- Integrated Development Environment (IDE)
    - Combines a text editor, compiler, file browser, debugger, and other tools
    - Helps you organize a programming project
    - Helps you write, compile, and test code in one place
    - Visual Studio terms:
        - Solution: An entire software project, including source code, metadata, input data files, etc.
        - "Build solution": Compile all of your code
        - "Start without debugging": Run the compiled code
        - Solution location: The folder (on your computer's file system) that contains the solution, meaning all your code and the information needed to compile and run it

```

Attach files by dragging & dropping, selecting or pasting them.

Commit changes

Create readme.md

Add an optional extended description...

[✓ Commit directly to the `main` branch.]

Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

Figure 7: "Committing"

Note that if you are making edits inside the repository for UCAs, `uca-resources-<semester>-YYYY`, you do have write access so there will instead be two buttons: **Commit directly to `main` branch** and **Create a new branch for this commit and start a pull request**

- **Commit directly to `main` branch** submits your edits directly into the document.
- **Create a new branch for this commit and start a pull request** creates a "pull request" (which can be found in the *Pull Requests* tab at the top of the page[208]) which essentially notifies others "you edited this

[208] https://github.com/princomp/princomp.github.io/pulls

document and you want them to check it". Others can check the changes you make, improve them, change them, and can submit them for you.

You can **Create a new branch for this commit and start a pull request** so others can double check your edits: it can act as a safety net, so your colleagues will be able to catch any mistake(s) you may have introduced!

Computer Requirements

ⓘ Info

This page contains some recommendations on students wishing to buy a computer to complete their program in the School of Cyber and Computer Sciences[209]. Note that **possessing a computer is not required to complete CSCI 1301**[210], but recommended.

In Short

Anything less than 5 years old running Microsoft Windows, macOs or a Linux operating system is probably fine. Second hand and custom built are fine, but you will in all likelihood needs a portable computer (as opposed to a desktop computer) to present your work and work on projects.

In Terms of Hardware

Desktop, Laptop, or something else?

A laptop is generally recommended (to take notes in class, make presentations, work on projects at School, …) but technically possessing only a desktop *should* be ok (and will be more comfortable to use, in all likeliness). Tablets and other "small" handled devices (such as Netbooks[211], Chromebooks[212] or Mini PCs[213]) are **not** recommended and will in all likelihood prove challenging to use for some classes.

[209] https://www.augusta.edu/ccs/

[210] https:/princomp.github.io/installing_software.md#accessing-an-ide

[211] https://en.wikipedia.org/wiki/Netbook

[212] https://en.wikipedia.org/wiki/Chromebook

[213] https://en.wikipedia.org/wiki/Mini_PC

Specifications

Component	Minimum	Suggested	Comfortable
CPU[214]	4 cores @ 2.66 GHz	6 cores @ 3.8 GHz	6 cores @ 4.4 GHz
RAM[215]	8GB	16GB	32GB
Hard Drive[216]	100GB	500GB of SSD[217]	1TB of SSD[218]

GPU[219] and other special equipment are not required, but recent USB-C connectors will be useful.

As An Example Dr. Aubert[220] uses a Dell Latitude 5480/5488[221] from **2017** (*but in no way endorses it*) with

- 4 cores @ 2.40 GHz CPU,
- 8GB of ram,
- 238 GB of hard drive,

and of courses wishes that it was a bit more responsive at times, but can conduct otherwise all his professional activities.

In Terms of Operating System

We will briefly consider four "families" of operating systems:

- Microsoft Windows[222] (Windows 10, Windows 11, etc.)
- macOS[223] (macOS Ventura, macOS Sonoma, etc.)
- Linux operating systems[224] (Ubuntu, Debian, Gentoo, etc.)

[214]https://en.wikipedia.org/wiki/Processor_(computing)
[215]https://en.wikipedia.org/wiki/Random-access_memory
[216]https://en.wikipedia.org/wiki/Hard_disk_drive
[217]https://en.wikipedia.org/wiki/Solid-state_drive
[218]https://en.wikipedia.org/wiki/Solid-state_drive
[219]https://en.wikipedia.org/wiki/Graphics_processing_unit
[220]https://spots.augusta.edu/caubert/
[221]https://www.dell.com/support/home/en-us/product-support/product/latitude-14-5480-laptop/docs
[222]https://en.wikipedia.org/wiki/Microsoft_Windows
[223]https://en.wikipedia.org/wiki/MacOS
[224]https://en.wikipedia.org/wiki/Linux

- Operating systems that uses their web browsers as their principal user interface (essentially, ChromeOS[225]).

Note we do not discuss Android[226] or iOS[227] since they are primarily mobile operating systems, and not easily suited for the development workload in our curriculum.

In Short

Anything but ChromeOS is (probably) fine.

Expanded

- If you are (planning on) using Visual Studio[228] as your IDE, then windows is your best choice of operating system.
- If you need to use macOS for whatever reason, then you will probably be able to accommodate all the requirements, but it may require some tweaking at times.
- Using Linux-based operating systems are a great way to learn how to tinker with your computer (you have full control!), but will sometimes require you to be creative to meet courses expectations.
- Virtual machines allow you to simulate (almost) any operating system using (almost) any operating system, and is required for some courses. Using virtual machines means, essentially, that your choice of operating system *does not matter at all*.
- Remember that multi-boots[229] (that is, installing multiple operating systems side-by-side) is an option.

Virtual Machines

Virtual machines allow you to simulate (almost) any operating system using (almost) any operating system: this means that, for instance, you can load the Windows 11 operating system from your computer running Debian 12.5, or the Debian 12.5 operating system from macOS 14.

Note that CSCI 4532 - Hardware and Embedded Systems and CSCI 4531 - Malware Analysis and Reverse Engineering *require* you to run virtual machines. If you are planning on taking one of those classes, make sure your computer can run virtual machines!

You can find on this page[230] some indications on how to run a virtual ma-

[225] https://en.wikipedia.org/wiki/ChromeOS
[226] https://en.wikipedia.org/wiki/Android_(operating_system)
[227] https://en.wikipedia.org/wiki/IOS
[228] https://visualstudio.microsoft.com/
[229] https://en.wikipedia.org/wiki/Multi-booting
[230] https:/princomp.github.io/installing_software.md#installing-anything-anywhere

chine on your computer, and you can check on-line the recommended specifications for Hyper-V[231], VirtualBox[232], kvm[233], vmware[234]. Note that, as a student, you can obtain a free licence for Windows[235].

Where to Buy?

That is really up to you, but remember that, as a student (or employee), you are allowed to

- Some discounts[236],
- A free licence for Windows[237].

Second-hand computers or even custom-built computers are probably fine, but requires more skills (such as how to factory-reset a computer and / or how to (re)install an operating system) and inspections on your end.

Is There Anything Else I Should Know?

- A well taken-care of computer can easily last 5 years, but laptops are harder to upgrade and preserve in good shape than desktops.
- Ergonomics is important: you will most likely spend *many hours* on your computer, so make sure your workstation is well organized[238].
- A programmer is first and foremost a typist: make sure you develop good habits and learn to type correctly[239]. Exploring ergonomics keyboard layouts[240] and ergonomics mice[241] can save you later from carpal tunnel syndrome, arthritis, and other repetitive strain injuries.

[231] https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-requirements

[232] https://www.virtualbox.org/wiki/End-user_documentation

[233] https://www.linux-kvm.org/page/FAQ#What_do_I_need_to_use_KVM?

[234] https://www.vmware.com/products/workstation-player.html

[235] https://portal.azure.com/?Microsoft_Azure_Education_correlationId=696fbf50-4829-476c-bfc8-09974888f850#view/Microsoft_Azure_Education/EducationMenuBlade/~/software

[236] https://my.augusta.edu/discounts/electronics.php

[237] https://portal.azure.com/?Microsoft_Azure_Education_correlationId=696fbf50-4829-476c-bfc8-09974888f850#view/Microsoft_Azure_Education/EducationMenuBlade/~/software

[238] https://www.wikihow.com/Set-Up-an-Ergonomically-Correct-Workstation

[239] https://www.wikihow.com/Type

[240] https://en.wikipedia.org/wiki/Keyboard_layout#Other_Latin-script_keyboard_layouts

[241] https://en.wikipedia.org/wiki/Computer_mouse#Ergonomic_mice

Installing Software

Generalities on Installing Software

You probably already installed software in your life, be it VLC[242], Microsoft Teams[243], or Whatsapp[244]. However, depending on whether you installed it on a phone, a tablet, a computer, and depending on the operating systems (Android, Windows 10, iOS, Ubuntu, etc.) your experience may have varied drastically.

Between the Play store[245], the command-line interface[246], homebrew[247] and the act of downloading software using your browser and then installing it using the navigator, there can be a lot of differences, but in all those circumstances you should keep security in mind. In addition to making sure that you are downloading the software from a trusted source, you should also be vigilant about the information the software will be able to access about e.g., your private life.

As data can be lost or corrupted upon downloading, many platforms now use checksums[248] to verify the integrity of the software you downloaded before installing it. This is an excellent practice that can also be performed "by hand", as explained for instance for the database manager MySQL[249]: the main idea is that the probability of the signature matching a tampered-with file is extremely low, and that as long as you are downloading the signature and the software from two different sources, you are considerably reducing the attack surface[250].

Executing Code Found on-line

As you progress in this class, you will be asked more and more to download and execute code hosted in our repository[251]. How can you tell that you can trust this code?

We have not implemented checksum-matching (yet!), but you can trust this code as it was coded by your instructors, and hosted on a platform

[242] http://www.videolan.org/
[243] https://www.microsoft.com/en-us/microsoft-teams/download-app
[244] https://www.whatsapp.com/
[245] https://www.wikiwand.com/en/Google_Play#Play_Store_on_Android
[246] https://www.wikiwand.com/en/Command-line_interface
[247] https://brew.sh/
[248] https://www.wikiwand.com/en/Checksum
[249] https://dev.mysql.com/doc/refman/8.0/en/verifying-package-integrity.html
[250] https://www.wikiwand.com/en/Attack_surface
[251] https://github.com/princomp/princomp.github.io/

using two-factor authentication[252] where every action is tracked using versioning[253]. Concretely, this means that only somebody who manages to steal your instructor's credentials and their phone, and thwart all the other instructors' vigilance, would be able to host malicious code on our platform: while we certainly imagine that this is theoretically possible, we hope that you will agree that the probability is low enough for you to trust the code on this site.

As often, security is not absolute, but aims at providing reasonable confidence. Executing "blindly" code found on-line, on the other hand, gives you a good chance of facing unpleasant surprises: while there certainly is a lot of useful, good code on websites like stackoverflow[254]your instructor probably uses such websites, by the way!, copying-and-pasting it without understanding its purpose or general structure is almost guaranteed to, at best, not execute properly, at worst, make your system unstable or insecure.

Accessing an IDE

An IDE[255], for "Integrated development environment", is the software or service you will be using to write, compile, execute and debug your code. There are many available IDEs, and some can accommodate multiple different programming languages.

For C#, there are many different possibilities[256]: some are cross-platforms (meaning you can use them on macOS, Windows or Linux), some are provided free of charge, some have not been updated in a long time. Three natural choices are Visual Studio[257], MonoDevelop[258] and Rider[259]. While the last two are accessible on every operating systems, Visual Studio is available only for Windows, and in a slightly different version for macOS.

To access one or the other, you will need either

- a computer with the right to install software on it,
- to access one of the computers in the computer lab[260], or

[252]https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/about-two-factor-authentication

[253]https://www.wikiwand.com/en/Software_versioning

[254]https://stackoverflow.com/

[255]https://www.wikiwand.com/en/Integrated_development_environment

[256]https://www.wikiwand.com/en/Comparison_of_integrated_development_environments#C%23

[257]https://visualstudio.microsoft.com/

[258]https://www.monodevelop.com/

[259]https://www.jetbrains.com/rider/features/

[260]https://my.augusta.edu/it/computers-printing.php#walkinlabs

- a computer with internet access.

The third solution is a backup plan, as instead you will access a very minimal version of an IDE to test small snippets of code. You should not rely on it for the duration of this course.

Installing an IDE On Your Own Computer

This part gathers some references for you to install Visual Studio[261], MonoDevelop[262] and Rider[263] on your own computer, regardless of your operating system. It is strongly encouraged that you do so, especially if you want to continue in a CS/IT/Cyber degree, but is not mandatory[264].

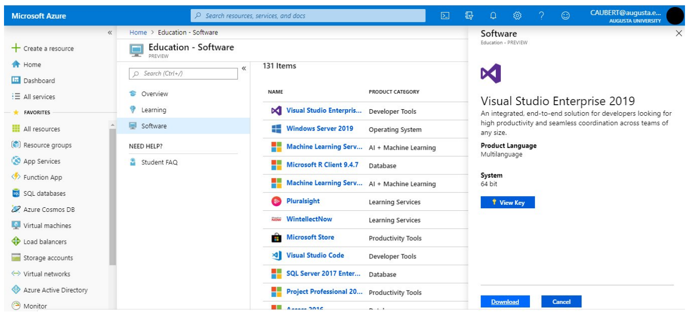
The instructions are detailed, but there are plenty of ways this can go wrong: make sure you have read and followed those instructions carefully before asking for help[265]!

Installing Visual Studio On Your Own Computer Note that we are *not* installing "Visual Studio *Code*", but simply "Visual Studio".

For Windows

1. Visit Azure Dev Tools for Teaching[266].
2. Log in using your Augusta University credentials.
3. Select "Download software".
4. Look for Visual Studio. The path is Education → Software → Visual Studio Enterprise 2019/2022. You can search "Services" for the "Education" group and then click "Software" if the education group is not immediately displayed. It should look like the following:

[261] https://visualstudio.microsoft.com/
[262] https://www.monodevelop.com/
[263] https://www.jetbrains.com/rider/features/
[264] Unless this class is fully online, of course.
[265] https:/princomp.github.io/labs/Introduction#how-to-get-help
[266] https://aka.ms/devtoolsforteaching



Microsoft Azure | Search resources, services, and docs | CAUBERT@augusta.e... AUGUSTA UNIVERSITY

Home > Education - Software

Education - Software
PREVIEW

131 Items

NAME	PRODUCT CATEGORY
Visual Studio Enterpris...	Developer Tools
Windows Server 2019	Operating System
Machine Learning Serv...	AI + Machine Learning
Microsoft R Client 9.4.7	Database
Machine Learning Serv...	AI + Machine Learning
Pluralsight	Learning Services
WintellectNow	Learning Services
Microsoft Store	Productivity Tools
Visual Studio Code	Developer Tools
SQL Server 2017 Enter...	Database
Project Professional 20...	Productivity Tools

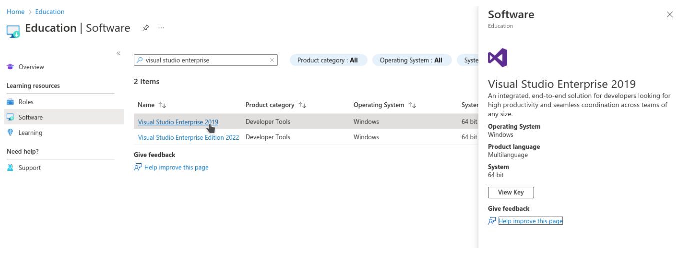
Software
Education - PREVIEW

Visual Studio Enterprise 2019
An integrated, end-to-end solution for developers looking for high productivity and seamless coordination across teams of any size.

Product Language
Multilanguage

System
64 bit

Normally, the following direct link should get you to the right page: https://portal.azure.com/?Microsoft_Azure_Education_correlationId=8ee63052-dc32-46f7-a109-e26793622dbf#view/Microsoft_Azure_Education/EducationMenuBlade/~/software. Type "Visual Studio Enterprise" in the search bar and you should find what you are looking for:



Home > Education

Education | Software

visual studio enterprise | Product category : All | Operating System : All

2 Items

Name	Product category	Operating System	Syster
Visual Studio Enterprise 2019	Developer Tools	Windows	64 bit
Visual Studio Enterprise Edition 2022	Developer Tools	Windows	64 bit

Software
Education

Visual Studio Enterprise 2019
An integrated, end-to-end solution for developers looking for high productivity and seamless coordination across teams of any size.

Operating System
Windows

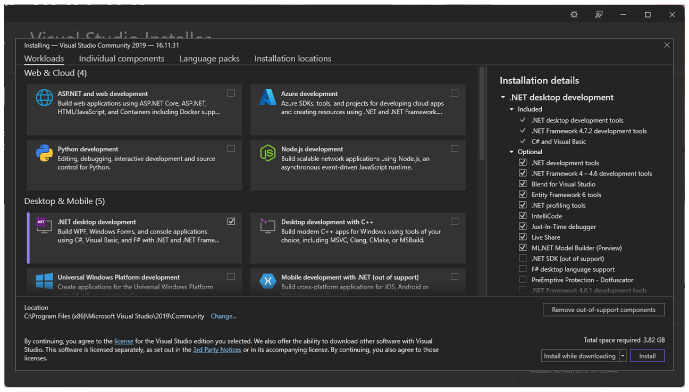
Product language
Multilanguage

System
64 bit

5. Download and install Visual Studio (leave all the options on their default settings).

   Before clicking install, make sure to check ".NET Desktop Development"

   If you are installing Visual Studio 2019, click the dropdown for .NET Desktop Development and check ".NET SDK (out of support)". You do not have to do this for Visual Studio 2022



Installing — Visual Studio Community 2019 — 16.11.31

Workloads | Individual components | Language packs | Installation locations

Web & Cloud (4)

- ASP.NET and web development — Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- Azure development — Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
- Python development — Editing, debugging, interactive development and source control for Python.
- Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

- [✓] .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...
- Desktop development with C++ — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- Universal Windows Platform development — Create applications for the Universal Windows Platform
- Mobile development with .NET (out of support) — Build cross-platform applications for iOS, Android or

Installation details

- .NET desktop development
  - Included
    - ✓ .NET desktop development tools
    - ✓ .NET Framework 4.7.2 development tools
    - ✓ C# and Visual Basic
  - Optional
    - ☑ .NET development tools
    - ☑ .NET Framework 4 – 4.6 development tools
    - ☑ Blend for Visual Studio
    - ☑ Entity Framework 6 tools
    - ☑ .NET profiling tools
    - ☑ IntelliCode
    - ☑ Just-In-Time debugger
    - ☑ Live Share
    - ☑ ML.NET Model Builder (Preview)
    - ☐ .NET SDK (out of support)
    - ☐ F# desktop language support
    - ☐ PreEmptive Protection - Dotfuscator
    - ☐ .NET Framework 4.6.1 development tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community Change...

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 3.82 GB

6. Enter the product key you obtained previously, following the instructions in the documentation[267]. Normally, clicking on "View key" on the screen pictured in the fourth step above should give you access to a key, that you simply need to copy-and-paste in the menu you can access on Visual Studio by clicking on "Select File" → "Account Settings" → "License with a Product Key".

For Mac Download a version of Visual Studio at https://visualstudio.microsoft.com/vs/mac/. It differs a bit from the Windows version, but that should not impact your experience in this class. The only Visual Studio feature we rely on is the ability to create "Console Apps with C#", which is equally available in both the Windows and Mac versions.

Installing MonoDevelop On Your Own Computer Unfortunately, MonoDevelop offers pre-packaged release only for linux distributions

- If you are using linux (e.g. Ubuntu, Debian, etc.), then please head out to MonoDevelop's download page[268].
- If you are using MacOS, you can have a look at the compilation instructions[269], but it is very likely that you will find them impossible to understand.
- If you are using Windows, you can have a look at the compilation instructions[270], but it is very likely that you will find them impossible to understand.

Installing Rider On Your Own Computer You can download Rider from their website[271], for any operating system. Note that, as a student, you can obtain a licence for free[272]: simply fill out this form[273], making sure you use your @augusta.edu email account, and you should receive a free licence instantaneously!

Note that Jetbrains offers to use a SHA-256 checksum (for instance, for the linux version[274]) for you to check that your download has not been tampered with. In any case, you can consult their detailed instructions[275] to install and execute Rider on any operating system.

[267] https://learn.microsoft.com/en-us/visualstudio/ide/how-to-unlock-visual-studio?view=vs-2019

[268] https://www.monodevelop.com/download/#fndtn-download-lin

[269] https://www.monodevelop.com/developers/building-monodevelop/#macos

[270] https://www.monodevelop.com/developers/building-monodevelop/#windows

[271] https://www.jetbrains.com/rider/download/

[272] https://www.jetbrains.com/community/education

[273] https://www.jetbrains.com/shop/eform/students

[274] https://download.jetbrains.com/rider/JetBrains.Rider-2022.2.2.tar.gz.sha256

[275] https://www.jetbrains.com/help/rider/Installation_guide.html#standalone

Installing a Code Editor On Your Own Computer

IDE in general performs the operation of setting up the compiler for you, but if you are willing to try to do it yourself, you can then access a larger offering of editors. Indeed, an alternative to installing an IDE is to install a C# compiler on one hand, and a code editor on the other hand (which is just a text editor with some completion or visualization related to the programming language you are using).

Among other code editors suited for C# code, we can mention:

- RosalynPad[276],
- Geany[277],
- LinqPad[278].

We give below some indications on how to set-up Geany.

Installing Geany On Your Own Computer

Note: *This method will only allow you edit and compile individual .cs files, and will not compile C# Solution Projects. To set-up Geany so that you can compile projects, could start by reading this exchange[279] (which is about projects in Linux, but applies equally well to projects in C#) or this one[280].*

You can download Geany from their website[281], for any operating system. To use Geany as a text editor for C#, we must download the Mono C# compiler from their website[282]. Make sure to download the most recent version to assure your compiler has the most up-to-date version of ".NET".

Once you installed Mono, locate the "csc.bat", "csc.exe" or "csc" file in Mono's "bin" folder and copy the file path. This path can be of the form

```
C:\Program Files (x86)\Mono\bin\csc.bat
```

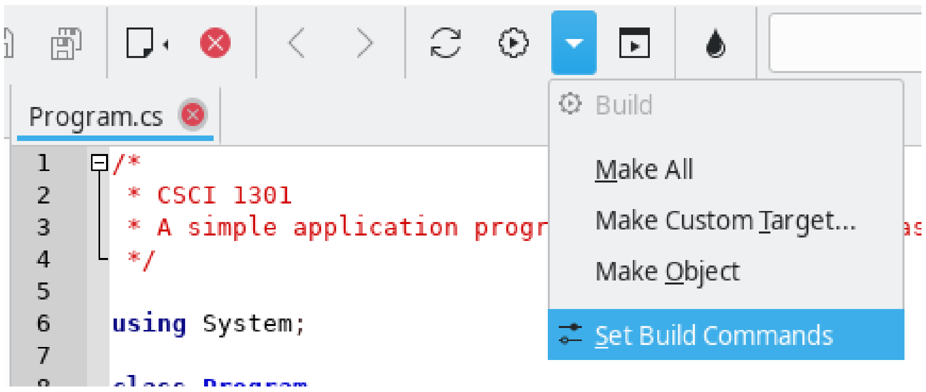
on windows, or

```
/usr/bin/csc
```

on Unix systems.

Now open a .cs file using Geany. Click the arrow next to the "Build" Button and click "Set Build Commands" from the dropdown menu.

[276]https://roslynpad.net/
[277]https://www.geany.org/
[278]https://www.linqpad.net/
[279]https://stackoverflow.com/q/54041013
[280]https://stackoverflow.com/q/8264323
[281]https://www.geany.org/
[282]https://www.mono-project.com/download/stable/



```
/*
 * CSCI 1301
 * A simple application progr
 */

using System;

class Program
```

Figure 8: Accessing the menu to set build commands

In the “Set Build Commands” window, erase the entry next to the “Compile” button and paste the file path to the “csc.bat” in quotation marks. After the file path, create a single space followed by “%f” with the quotaion marks. All in all, you should have something of the form

```
"C:\Program Files (x86)\Mono\bin\csc.bat" "%f"
```

in the “Command” field of the “Compile” line.

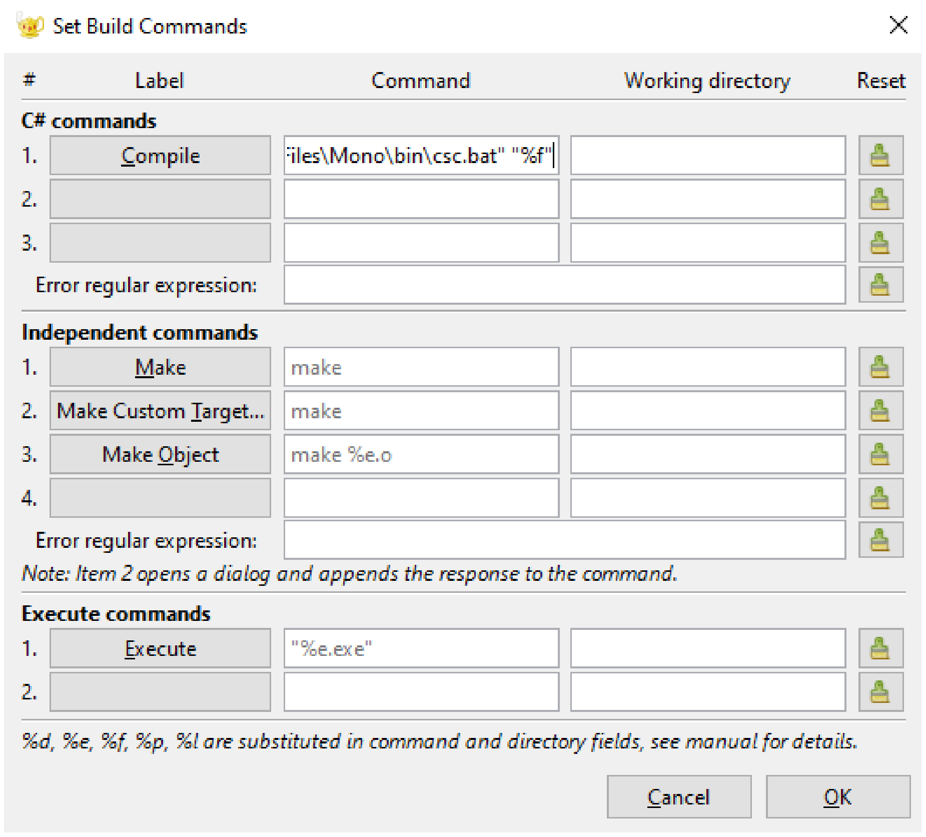
Confirm the change by clicking OK and now you will be able to compile, build, and execute **standalone** .cs files.

Installing Anything Anywhere If the IDE you would like to adopt is not available for your operating system, you can use a Virtual Machine[283] manager to execute a linux-based distribution or a Windows image on top of your operating system.

For this, and regardless of your current operating system, you will need a Virtual Machine[284] manager.

1. There are many (free) options to chose from, let us mention
   (a) Virtual Box[285] (for Windows, Linux and Mac),
   (b) QEMU[286] (for Windows, Linux and Mac),
   (c) Hyper-V[287] (for Windows),

[283] https://www.wikiwand.com/en/Virtual_machine
[284] https://www.wikiwand.com/en/Virtual_machine
[285] https://www.virtualbox.org/
[286] https://www.qemu.org
[287] https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v



Set Build Commands

#	Label	Command	Working directory	Reset
C# commands				
1.	Compile	iles\Mono\bin\csc.bat" "%f"		
2.				
3.				
Error regular expression:				
Independent commands				
1.	Make	make		
2.	Make Custom Target...	make		
3.	Make Object	make %e.o		
4.				
Error regular expression:				

Note: Item 2 opens a dialog and appends the response to the command.

#	Label	Command	Working directory	Reset
Execute commands				
1.	Execute	"%e.exe"		
2.				

%d, %e, %f, %p, %l are substituted in command and directory fields, see manual for details.

Figure 9: Setting the build commands

2. Download a version of "Microsoft Operating Systems" from Azure Dev Tools for Teaching[288], or a linux-based distribution (typically, ubuntu[289] has a good reputation of being accessible and user-friendly).
3. Install and execute your version of Windows or Linux from your virtual machine, and follow the corresponding instructions to install the IDE you are interested in.

Note that it is illegal to execute macOS in a virtual environment that is not hosted on a mac computer[290], which drastically reduces the interest for you to consider this option.

Accessing One of the Computers in a Computer Lab

Please refer to this page from AU's Information Technology[291] to know where the computer labs are located. Visual Studio should be pre-installed on every computer.

Compiling Code On-Line

As a backup or only to test snippets of code, you can compile C# code online. Multiple online platforms exist, such as:

- https://www.browxy.com/
- https://www.tutorialspoint.com/compile_csharp_online.php
- https://www.onlinegdb.com/online_csharp_compiler
- https://www.jdoodle.com/compile-c-sharp-online/
- https://dotnetfiddle.net/
- https://www.w3schools.com/CS/trycs.php?filename=demo_hello world

Note that none of them are endorsed by the school and that they can pose security and privacy challenges: never enter any sensitive information and do not rely on them too heavily. However, they can be a good support if you would like to test a short snippet of code but do not have access at the moment to a computer with an IDE installed.

(Un)Zipping Archives

This short note explains how to

[288] https://aka.ms/devtoolsforteaching
[289] https://ubuntu.com/appliance/vm
[290] https://law.stackexchange.com/q/18282
[291] https://my.augusta.edu/it/computers-printing.php#walkinlabs

- Unzip files,
- Zip folders,
- Locate your project

for the three main operating systems (Windows, Linux and macOS).

Unzipping Files

Windows

Navigate your file explorer and navigate to your Downloads folder (or wherever you downloaded the file). From there, look for the file you downloaded, right-click, and select "Extract All...". When the "Extract Compressed (Zipped) Folder" window opens, click the "Extract" button.

Linux

This guide is assuming you have Zip/Unzip installed on your Linux distribution. If this is not the case, first follow this install guide[292].

Using the graphical interface Normally, a simple right click and choose "Extract" or "Open with Ark"[293] should do it.

Using the Command-Line Navigate to your command-line interface and execute the following command (as a normal user, as indicated by $):

```
$ unzip [FileName].zip
```

where "(FileName).zip" is the name of the zip file.

macOS

Simply double-click on the zip file to unzip it onto your desktop.

Zipping Files

Windows

Navigate to your file explorer and go to where your solution is stored on your system, the default file path being:

```
C:\Users\[UserName]\source\repos
```

[292]https://www.tecmint.com/install-zip-and-unzip-in-linux/
[293]https://www.wikihow.tech/Unzip-Files-in-Linux

where "[UserName]" is your Windows username (on school computers, this should be your AU username). Right click the folder you want to zip, go down the list to the "Send to" option, and then click on the "Compressed (Zipped) Folder" option. This should then create a new zip file.

Linux

Using the graphical interface Normally, a simple right click and choose "Compress"[294] should do it.

Using the Command-Line Navigate to your command-line interface and execute the following command (as a normal user, as indicated by $):

```
$ zip -r [ZipFileName].zip [FileName]
```

where "[ZipFileName].zip" is the name you want for the zip file, and "[FileName]" for the folder you want to zip.

macOS

Navigate to your file explorer and go to where your solution is stored on your system, the default file path being:

```
[UserName]\source\repos
```

where "[UserName]" is your Mac username. Right-click on the folder that you want to zip up and click on the "Compress the Folder" option.

But Where Is My Project?

By default, it *should* be stored in a folder located in

```
C:\Users\[UserName]\source\repos
```

for Windows users,

```
[UserName]\source\repos
```

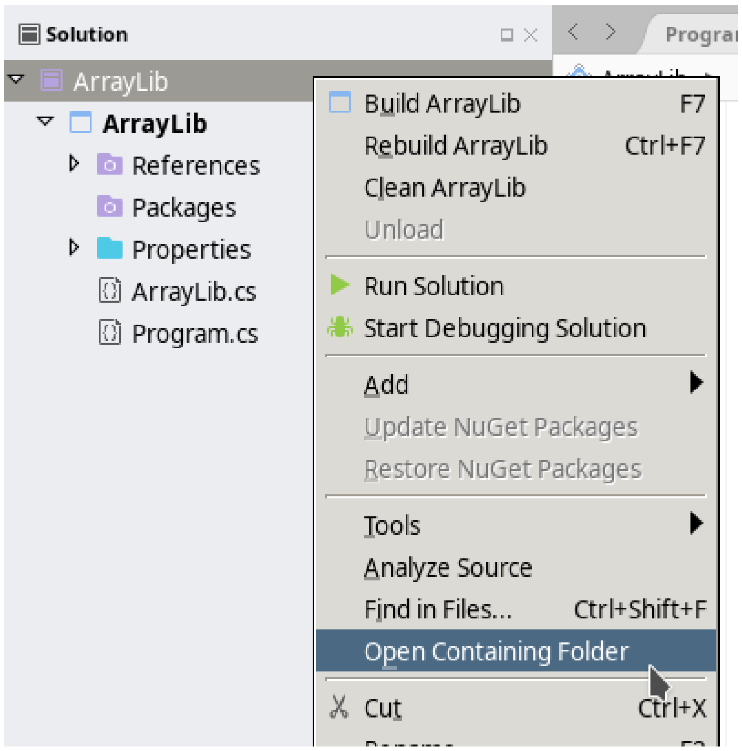
for macOS users,

```
/home/[UserName]/Projects
```

for Linux users.

When in doubt, open your project in the IDE, right-click on the solution, and look for an option called "Open in File Explorer" or "Open Containing Folder":

[294] https://www.wikihow.com/Make-a-Zip-File-in-Linux



Keyboard Shortcuts

Foreword

This document contains useful keyboard shortcuts for different operating systems and IDEs. We use the following symbols:

Symbol	Common Name
⇧	Shift
⌥	Option (or Alt)
⌘	Command (or Cmd)
↵	(Carriage) Return

The sections labeled with the star symbol ("*") work generally everywhere, beyond your IDE.

More advanced shortcuts may be available to your particular IDE:

- For Visual Studio for Windows, refer to the documentation[295],
- For Visual Studio for MacOS, refer to the documentation[296]
- For Rider, refer to the documentation[297],
- For MonoDevelop, you can refer to this cheatsheet[298] or directly access the key binding panel[299].

Useful Shortcuts

Build solution

OS	Keys
Linux	Ctrl + ⇧ + B
MacOS	⌘ + B
Windows	Ctrl + ⇧ + B

Exit any program*

OS	Keys
Linux	Alt + F4 or Ctrl + q
MacOS	⌘ + q
Windows	Alt + F4

Redo*

OS	Keys
Linux	Ctrl + y
MacOS	⌘ + y
Windows	Ctrl + y

[295]https://docs.microsoft.com/en-us/visualstudio/ide/default-keyboard-shortcuts-in-visual-studio?view=vs-2019
[296]https://docs.microsoft.com/en-us/visualstudio/mac/keyboard-shortcuts?view=vsmac-2019
[297]https://www.jetbrains.com/help/rider/mastering_keyboard_shortcuts.html
[298]https://shortcutworld.com/Xamarin-Studio/win/Xamarin-Studio-(MonoDevelop)_Shortcuts
[299]https://mhut.ch/journal/2011/02/05/monodevelop-tips-key-bindings

Run/execute program

OS	Keys
Linux	Ctrl + F5
MacOS	F5 -or- ⌥ + ⌘ + ↵
Windows	Ctrl + F5

Save*

OS	Keys
Linux	Ctrl + s
MacOS	⌘ + s
Windows	Ctrl + s

Save All*

OS	Keys
Linux	Ctrl + ⇧ + s
MacOS	⌘ + ⇧ + s
Windows	Ctrl + ⇧ + s

Undo*

OS	Keys
Linux	Ctrl + z
MacOS	⌘ + z
Windows	Ctrl + z

Comment Code Selection

OS	Keys
Linux	Ctrl + k + c
MacOS	⌘ + k + c
Windows	Ctrl + k + c

Uncomment Code Selection

OS	Keys
Linux	Ctrl + k + u
MacOS	⌘ + k + u
Windows	Ctrl + k + u

Datatypes in C

Value Types

Numeric

Signed Integer

Type	Range	Size
`sbyte`	-128 to 127	Signed 8-bit integer
`short`	-32,768 to 32,767	Signed 16-bit integer
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer

Unsigned Integer

Type	Range	Size
`byte`	0 to 255	Unsigned 8-bit integer
`ushort`	0 to 65,535	Unsigned 16-bit integer
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Floating-point Numbers

Type	Approximate Range	Precision
`float`	±1.5e−45 to ±3.4e38	7 digits
`double`	±5.0e−324 to ±1.7e308	15–16 digits

Type	Approximate Range	Precision
decimal	(-7.9 x 1028 to 7.9 x 1028)/(100 to 1028)	28–29 significant digits

Logical

Type	Possible Values	Size
bool	**true**, **false**	8-bit

Character

Type	Range	Size
char	U+0000 to U+ffff	Unicode 16-bit character

Literals

Name	Corresponding datatype	Examples
Integer Literal	int	40, -39, 291838, 0, …
Float Literal	float	3.5F, -43.5f, 309430.70006F, …
Double Literal	double	28.98, 239.0, -391.089, 0.0, …
Decimal Literal	decimal	8.95m, 3283.9M, -30m, …
Boolean Literal	bool	**true**, **false**
Character Literal	char	'Y', 'a', '0', '\n', '\x0058', '\u0058', …

Compatibility

This table is to be read as

> ✓ means that those values or variables from the datatypes in the row and column can be "operated together" (meaning, you can for instance multiply them), ✘ means that those values or variables from the datatypes in the row and column can*not* be "operated together" (meaning, you can*not* for instance multiply them).

	Integer Literal	Float Literal	Double Literal	Decimal Literal
`int`	✓	✘	✘	✘
`float`	✓	✓	✘	✘
`double`	✓	✓	✓	✘
`decimal`	✓	✘	✘	✓

Result Type of Operations

	`int`	**`float`**	**`double`**	**`decimal`**
`int`	`int`	`float`	`double`	`decimal`
`float`	`float`	`float`	`double`	illegal
`double`	`double`	`double`	`double`	illegal
`decimal`	`decimal`	illegal	illegal	`decimal`

This table is to be read as

> Values or variables from the datatypes in the row and column can be "operated together" and will produce the datatype indicated in the cell, or cannot be "operated together" if the value in the cell is "illegal".

References

- https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/integral-types-table
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/floating-point-types-table
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types-table
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/implicit-numeric-conversions-table
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/explicit-numeric-conversions-table

Computers and Programming

Principles of Computer Programming

- Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets - and will continue doing so.
- With these changes, the capabilities of computers increase rapidly (storage, speed, graphics, etc.)
- Computer programming languages also change
  - Better programming language theory leads to new programming techniques
  - Improved programming language implementations
  - New languages are created, old ones updated
- There are hundreds of programming languages[300], why?
  - Different tools for different jobs
    - Some languages are better suited for certain jobs
    - For example, Python is best for scripting, Javascript is best for web pages, MySQL is best for databases, etc.
  - Personal preference and popularity
- This class is about "principles" of computer programming
  - Common principles behind all languages will not change, even though hardware and languages do
  - How to organize and structure data
  - How to express logical conditions and relations
  - How to solve problems with programs

Programming Language Concepts

We begin by discussing three categories of languages manipulated by computers. We will be studying and writing programs in *high-level languages*, but understanding their differences and relationships to other languages[301] is of importance to become familiar with them.

- Machine language
  - Computers are made of electronic circuits
    - Circuits are components connected by wires
    - Some wires carry data - e.g. numbers
    - Some carry control signals - e.g. do an add or a subtract operation
  - Instructions are settings on these control signals
    - A setting is represented as a 0 or 1

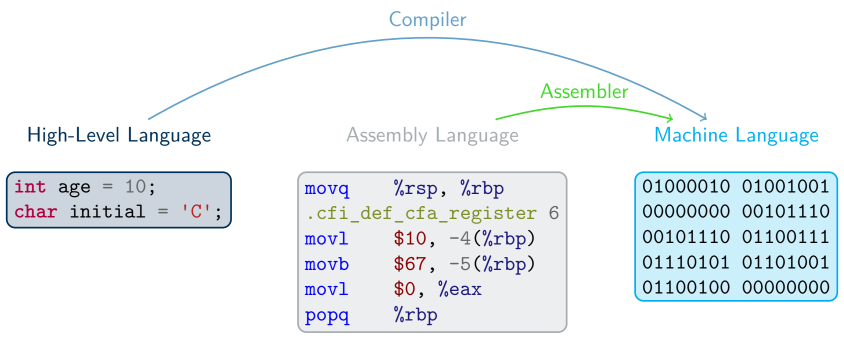
[300] https://www.wikiwand.com/en/List_of_programming_languages
[301] That will be studied in the course of your study if you continue as a CS major.

    - A machine language instruction is a group of settings - For example: `1000100111011000`
  - Most CPUs use one of two languages: x86 or ARM
- Assembly language
  - Easier way for humans to write machine-language instructions
  - Instead of 1s and 0s, it uses letters and "words" to represent an instruction.
    - Example x86 instruction:

    `MOV BX, AX`
    which makes a copy of data stored in a component called AX and places it in one called BX
  - **Assembler**: Translates assembly language instructions to machine language instructions
    - For example: `MOV BX, AX` translates into `1000100111011000`
    - One assembly instruction = one machine-language instruction
    - x86 assembly produces x86 machine code
  - Computers can only execute the machine code
- High-level language
  - Hundreds including C#, C++, Java, Python, etc.
  - Most programs are written in a high-level language since:
    - More human-readable than assembly language
    - High-level concepts such as processing a collection of items are easier to write and understand
    - Takes less code since each statement might be translated into several assembly instructions
  - **Compiler**: Translates high-level language to machine code
    - Finds "spelling" errors but not problem-solving errors
    - Incorporates code libraries – commonly used pieces of code previously written such as Math.Sqrt(9)
    - Optimizes high-level instructions – your code may look very different after it has been optimized
    - Compiler is specific to both the source language and the target computer
  - Compile high-level instructions into machine code then execute since computers can only execute machine code

A more subtle difference exists between high-level languages. Some (like C) are *compiled* (as we discussed above), some (like Python) are *interpreted*, and some (like C#) are in an in-between called *managed*.

- Compiled vs. Interpreted languages
  - Not all high-level languages use a compiler - some use an interpreter
  - **Interpreter**: Lets a computer "execute" high-level code by translating one statement at a time to machine code
  - Advantage: Less waiting time before you can execute the pro-



```
int age = 10;
char initial = 'C';
```

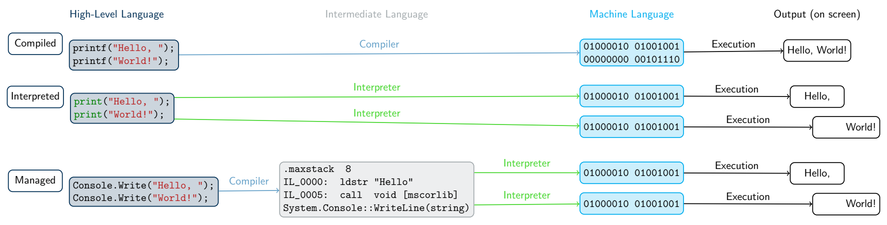
```
movq    %rsp, %rbp
.cfi_def_cfa_register 6
movl    $10, -4(%rbp)
movb    $67, -5(%rbp)
movl    $0, %eax
popq    %rbp
```

```
01000010 01001001
00000000 00101110
00101110 01100111
01110101 01101001
01100100 00000000
```

Figure 10: A Visual Representation of the Relationships Between Languages

gram (no separate "compile" step)
  - Disadvantage: Program executes slower since you wait for the high-level statements to be translated then the program is executed
- Managed high-level languages (like C#)
  - Combine features of compiled and interpreted languages
  - Compiler translates high-level statements to **intermediate language** instructions, not machine code
    - Intermediate language: Looks like assembly language, but not specific to any CPU
  - **run-time** executes by *interpreting* the intermediate language instructions - translates one at a time to machine code
    - Faster since translation is partially done already (by compiler), only a simple "last step" is done when executing the program
  - Advantages of managed languages:
    - In a "non-managed" language, a compiled program only works on one OS + CPU combination (**platform**) because it is machine code
    - Managed-language programs can be reused on a different platform without recompiling - intermediate language is not machine code and not CPU-specific
    - Still need to write an intermediate language interpreter for each platform (so it produces the right machine code), but, for a non-managed language, you must write a compiler for each platform
    - Writing a compiler is more complicated and more work than writing an interpreter thus an interpreter is a quicker

    (and cheaper) way to put your language on different platforms
    * Intermediate-language interpreter is much faster than a high-level language interpreter, so programs execute faster than an "interpreted language" like Python
  - This still executes slower than a non-managed language (due to the interpreter), so performance-minded programmers use non-managed compiled languages (e.g. for video games)



```
printf("Hello, ");
printf("World!");
```

```
01000010 01001001
00000000 00101110
```

```
print("Hello, ");
print("World!");
```

```
01000010 01001001
```

```
01000010 01001001
```

```
Console.Write("Hello, ");
Console.Write("World!");
```

```
.maxstack  8
IL_0000:  ldstr "Hello"
IL_0005:  call  void [mscorlib]
System.Console::WriteLine(string)
```

```
01000010 01001001
```

```
01000010 01001001
```

Figure 11: A Visual Representation of the Differences Between High-Level Languages

Software Concepts

- Flow of execution in a program
  - Program receives input from some source, e.g. keyboard, mouse, data in files
  - Program uses input to make decisions
  - Program produces output for the outside world to see, e.g. by displaying images on screen, writing text to console, or saving data in files
- Program interfaces
  - **GUI** or Graphical User Interface: Input is from clicking mouse in visual elements on screen (buttons, menus, etc.), output is by drawing onto the screen
  - **CLI** or Command Line Interface: Input is from text typed into "command prompt" or "terminal window," output is text printed at same terminal window
  - This class will use CLI because it is simple, portable, easy to work with – no need to learn how to draw images, just read and write text

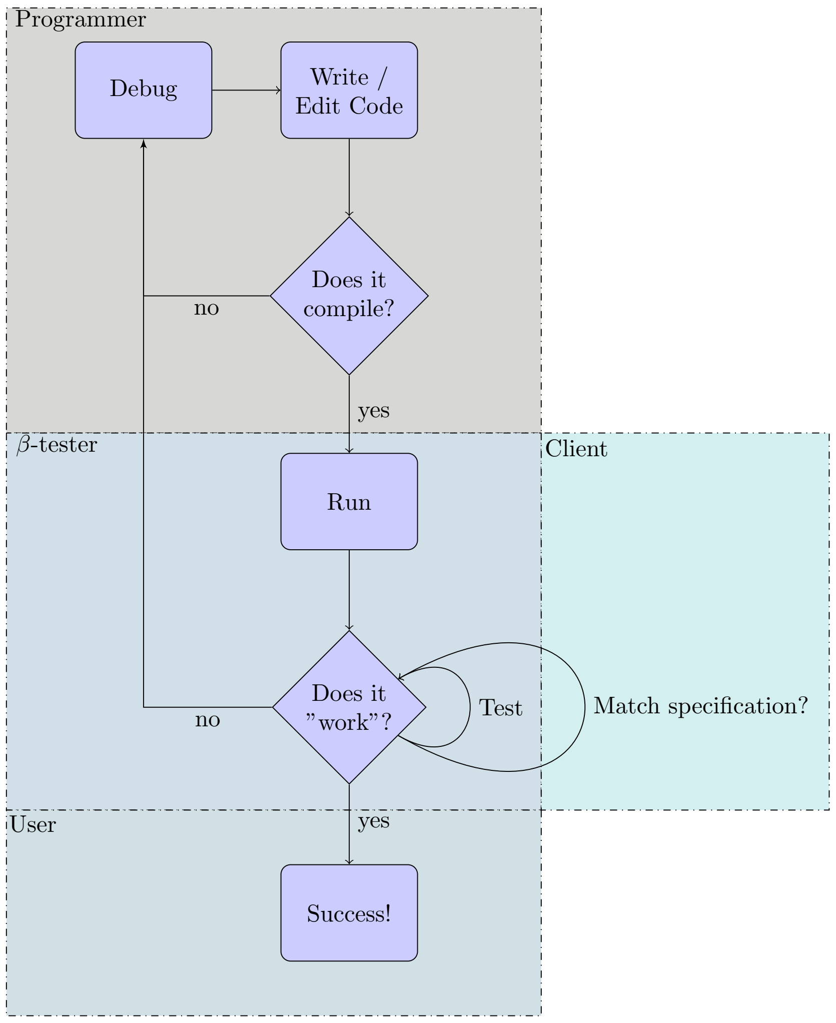


Figure 12: Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs.

Programming Concepts

Programming workflow

The workflow of the programmer will differ a bit depending on if the program is written in a compiled or an intprepreted programming language. From the distance, both looks like what is pictured in the the flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs, but some differences remain:

- Compiled language workflow
  - Writing down specifications
  - Creating the source code
  - Running the compiler
  - Reading the compiler's output, warning and error messages
  - Fixing compile errors, if necessary
  - Executing and testing the program
  - Debugging the program, if necessary
- Interpreted language workflow
  - Writing down specifications
  - Creating the source code
  - Executing the program in the interpreter
  - Reading the interpreter's output, determining if there is a syntax (language) error or the program finished executing
  - Editing the program to fix syntax errors
  - Testing the program (once it can execute with no errors)
  - Debugging the program, if necessary

Interpreted languages have

- **Advantages**: Fewer steps between writing and executing, can be a faster cycle
- **Disadvantages**: All errors happen when you execute the program, no distinction between syntax errors (compile errors) and logic errors (bugs in executing program)

(Integrated) Development Environment

Programmers can either use a collection of tools to write, compile, debug and execute a program, or use an "all-in-one" solution called an Integrated Development Environment (IDE).

- The "Unix philosophy"[302] states that a program should do only one task, and do it properly. For programmers, this means that
  - One program will be needed to edit the source code, a text editor (it can be Geany, notepad, kwrite, emacs, sublime text,

[302] https://www.wikiwand.com/en/Unix_philosophy

vi, etc.),
  - One program will be needed to compile the source code, a compiler (for C#, it will be either mono[303] or Roslyn[304],
  - Other programs may be needed to debug, execute, or organize larger projects, such as makefile or MKBundle[305].

IDE "bundle" all of those functionality into a single interface, to ease the workflow of the programmer. This means sometimes that programmers have fewer control over their tools, but that it is easier to get started.

- Integrated Development Environment (IDE)
  - Combines a text editor, compiler, file browser, debugger, and other tools
  - Helps you organize a programming project
  - Helps you write, compile, and test code in one place

In particular, Visual Studio is an IDE, and it uses its own vocabulary:

- Solution: An entire software project, including source code, metadata, input data files, etc.
- "Build solution": Compile all of your code
- "Start without debugging": Execute the compiled code
- Solution location: The folder (on your computer's file system) that contains the solution, meaning all your code and the information needed to compile and execute it

C# Fundamentals

Introduction to the C# Language

- C# is a managed language (as discussed previously[306])
  - Write in a high-level language, compile to intermediate language, run intermediate language in interpreter
  - Intermediate language is called CIL (Common Intermediate Language)
  - Interpreter is called .NET run-time
  - Standard library is called .NET Framework, comes with the compiler and run-time
- It is widespread and popular

[303] https://www.wikiwand.com/en/Mono_(software)

[304] https://www.wikiwand.com/en/Roslyn_(compiler)

[305] https://www.mono-project.com/docs/tools+libraries/tools/mkbundle/

[306] https:/princomp.github.io/lectures/intro/computers_and_programming#programming-language-concepts

  - It is "programming language of the year 2023"[307] in the very well-respected TIOBE Index[308].
  - It was the first in the list of "3 Future Programming Languages You Should Learn Between 2022 and 2030"[309], because of the growing popularity of Unity[310].
  - 7th most "desired / admired" language on StackOverflow[311]
  - .NET is the first most used "other" library/framework[312]
  - More insights on its evolution can be found in this blog post[313].

The Object-Oriented Paradigm

- C# is called an "object-oriented" language
  - Programming languages have different *paradigms*: philosophies for organizing code, expressing ideas
  - Object-oriented is one such paradigm, C# uses it
  - Meaning of object-oriented: Program mostly consists of *objects*, which are reusable modules of code
  - Each object contains some data (*attributes*) and some functions related to that data (*methods*)
- Object-oriented terms
  - **Class**: A blueprint or template for an object. Code that defines what kind of data the object will contain and what operations (functions) you will be able to do with that data
  - **Object**: A single instance of a class, containing running code with specific values for the data. Each object is a separate "copy" based on the template given by the class.
    Analogy: A *class* is like a floorplan while an *object* is the house build from the floorplan. Plus, you can make as many houses as you would like from a single floorplan.
  - **Attribute**: A piece of data stored in an object.
    Example: A *House* class has a spot for a color property while an house object has a color (e.g. "Green").
  - **Method**: A function that modifies an object. This code is part of the class, but when it is executed, it modifies only a specific object and not the class.

[307] https://www.tiobe.com/tiobe-index/
[308] https://en.wikipedia.org/wiki/TIOBE_index
[309] https://betterprogramming.pub/3-future-programming-languages-you-should-learn-between-2022-and-2030-8a618a15eca6
[310] https://unity.com/
[311] https://survey.stackoverflow.co/2023/#programming-scripting-and-markup-languages
[312] https://survey.stackoverflow.co/2023/#most-popular-technologies-misc-tech
[313] https://dottutorials.net/stats-surveys-about-net-core-future-2020/#stackoverflow-surveys

Example: A *House* class with a method to change the house color. Using this method changes the color a single house object but does not change the *House* class or the color on any other house objects.

- Examples:
  - A Car *Class*
    - Attributes: Color, engine status (on/off), gear position
    - Methods: Press gas or brake pedal, turn key on/off, shift transmission
  - A Car *Object*
    Example: A *Porsche911* object that is Red, Engine On, and in 1st gear
  - An "Audio File" *Class* represents a song being played in a music player
    - Attributes: Sound wave data, current playback position, target speaker device
    - Methods: Play, pause, stop, fast-forward, rewind
  - An Audio File *Object*
    Example: A *NeverGonnaGiveYouUp* object that is "rolled wave data", 0:00, speaker01

First Program

It is customary to start the study of a programming language with a "Hello World" program[314], that simply displays "Hello World". It is a simple way of seeing a first, simple example of the basic structure of a program. Here's a simple "hello world" program in the C# language:

Hello World

```
/* I'm a multi-line comment,
 * I can span over multiple lines!
 */
using System;

class Program
{
  static void Main()
  {
    Console.WriteLine("Hello, world!"); // I'm an in-line
 ↪  comment.
  }
}
```

[314]https://www.wikiwand.com/en/%22Hello,_World!%22_program

Features of this program:

- A multi-line comment: everything between the `/*` and `*/` is considered a *comment*, i.e. text for humans to read. It will be ignored by the C# compiler and has no effect on the program.
- A **`using`** statement: This imports code definitions from the System *namespace*, which is part of the .NET Framework (the standard library).
  - In C#, code is organized into **namespaces**, which group related classes together
  - If you want to use code from a different namespace, you need a **`using`** statement to "import" that namespace
  - All the standard library code is in different namespaces from the code you will be writing, so you'll need **`using`** statements to access it
- A class declaration[315]
  - Syntax:

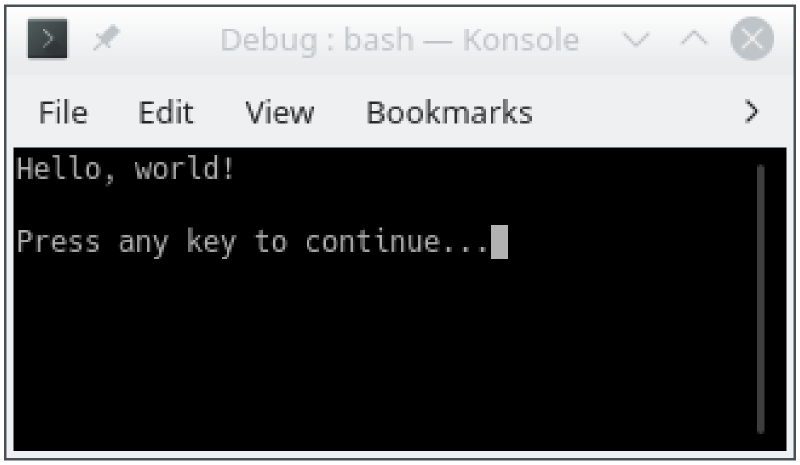
```
class [name of class]
{
        [body of the class]
}
```

  - All code between opening { and closing } is the *body* of the class named by the `class [name of class]` statement
- A method declaration
  - A collection of instructions with a name
  - Can be used by typing its name
  - A method is similar to a paragraph, in that it can contain multiple statements, and a class is similar to a chapter, in that it can have multiple methods within its body.
  - A C# program requires a method called `Main`, and, in our example, is followed by empty parentheses (we will get to those later, but they are required)
  - Just like the class declaration, the body of the method beings with { and ends with }
- A statement inside the body of the method:

```
Console.WriteLine("Hello, world!"); // I'm an in-line
↪  comment.
```

[315]We use the notation [...] to denote what "should" be there, but this is just a place holder: you are not supposed to *actually* have the braces in the code.

  - This is the part of the program that actually "does something": It displays a line of text to the console:



```
Hello, world!

Press any key to continue...
```

  - This statement contains a class name (`Console`), followed by a method name (`WriteLine`). It calls the `WriteLine` method in the `Console` class.
  - The **argument** to the `WriteLine` method is the text "Hello, world!", which is in parentheses after the name of the method. This is the text that gets printed in the console: The `WriteLine` method (which is in the standard library) takes an argument and prints it to the console.
  - Note that the argument to `WriteLine` is inside double-quotes. This means it is a **string**, i.e. textual data, not a piece of C# code. The quotes are required in order to distinguish between text and code.
  - A statement *must* end in a semicolon (the class header and method header are not statements)
- An in-line comment: All the text from the `//` to the end of the line is considered a comment, and is ignored by the C# compiler.

Rules of C# Syntax

- Each statement must end in a semicolon (`;`), except for some statements that we will study in the future that contains opening `{` and closing `}`, that do not end in a `;`.
  - Note that class and method declarations, as well as comments, are not statements[316] and hence do not need to ends with a

[316]https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-

`;`. Typically, a method *contains* some statements, but it is not a statement.

- All words are case-sensitive
  - A class named `Program` is not the same as one named `program`
  - A method named `writeline` is not the same as one named `WriteLine`
- Braces and parentheses must always be matched
  - Once you start a class or method definition with `{`, you must end it with `}`
- Blank space has *almost* no meaning
  - Blank spaces refer to spaces (sometimes denoted “ ”, “␣” or “⊔”), tabs[317] (which consists in 4 spaces), and new lines (sometimes denoted “↵”, “↵”, “←” or “↵”)
  - There must be at least 1 space between words
  - Other than that, spaces and new lines are just to help humans read the code
  - Spaces are counted exactly if they are inside string data, e.g. `"Hello      world!"` is different from `"Hello world!"`
  - Otherwise, entire program could be written on one line[318]; it would have the same meaning
- All C# applications must have a `Main` method
  - Name must match exactly, otherwise .NET run-time will get confused
  - This is the first code to execute when the application starts – any other code (in methods) will only execute when its method is called

Conventions of C# Programs

- Conventions: Not enforced by the compiler/language, but expected by humans
  - Program will still work if you break them, but other programmers will be confused
- Indentation
  - After a class or method declaration (header), put the opening `{` on a new line underneath it
  - Then indent the next line by 4 spaces, and all other lines “inside” the class or method body
  - De-indent by 4 spaces at end of method body, so ending `}` aligns vertically with opening `{`

expressions-operators/statements

[317] https://www.wikiwand.com/en/Tab_key#Tab_characters

[318] Well, if there are no in-line comments in it. Can you figure out why?

    - Method definition inside class definition: Indent body of method by another 4 spaces
    - In general, any code between { and } should be indented by 4 spaces relative to the { and }
- Code files
    - C# code is stored in files that end with the extension ".cs"
    - Each ".cs" file contains exactly one class
    - The name of the file is the same as the name of the class (Program.cs contains **`class`** `Program`)

Note that some of those conventions are actually rules in different programming languages (typically, the last two regarding code files are mandatory rules in java).

Reserved Words and Identifiers

- Reserved words: Keywords in the C# language
    - Note they have a distinct color in the code sample and in your IDE
    - Built-in commands/features of the language
    - Can only be used for one specific purpose; meaning cannot be changed
    - Examples:
        * **`using`**
        * **`class`**
        * **`public`**
        * **`private`**
        * **`namespace`**
        * **`this`**
        * **`if`**
        * **`else`**
        * **`for`**
        * **`while`**
        * **`do`**
        * **`return`**
    - There is no need to memorize the whole list of keywords[319], as we will only introduce the ones we need on a "per need" basis.
- Identifiers: Human-chosen names
    - Names for classes (`Rectangle`, `ClassRoom`, etc.), variables (`age`, `name`, etc.), methods (`ComputeArea`, `GetLength`, etc), namespaces, etc.
    - Some have already been chosen for the standard library (e.g. `system`, `Console`, `WriteLine`, `Main`), but they are still identifiers, not keywords

[319]https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/

- Rules for identifiers:
  * Must not be a reserved word
  * Must contain only letters (lower case, from `a` to `z`, or upper case, from `A` to `Z`), numbers (made of digits from `0` to `9`), and underscore (_). But they cannot contain spaces.
  * Must not begin with a number
  * Are case sensitive
  * Must be unique (you cannot re-use the same identifier twice in the same scope – a concept we will discuss later)
- Conventions for identifiers
  * Should be descriptive, e.g. "`AudioFile`" or "`userInput`" not "`a`" or "`x`"
  * Should be easy for humans to read and type
  * If name is multiple words, use CamelCase[320] (or its variation Pascal case[321]) to distinguish words, e.g. `myHeightInMeters` or `distanceFromEarthToMoon`.
  * Class and method names should start with capitals, e.g. "`class AudioFile`"
  * Variable names should start with lowercase letters, then capitalize subsequent words, e.g. "`myFavoriteNumber`"

Write and WriteLine

- The `WriteLine` method
  - We saw this in the "Hello World" program: `Console.WriteLine("Hello World!");` results in "Hello World!" being displayed in the terminal
  - In general, `Console.WriteLine("text");` will display the text but not the "'s in the terminal, then *start a new line*
  - This means a second `Console.WriteLine` will display its text on the next line of the terminal. For example, this program:

```
using System;

class Welcome
{
  static void Main()
  {
    Console.WriteLine("Hello");
    Console.WriteLine("World!");
  }
}
```

[320] https://www.wikiwand.com/en/Camel_case

[321] https://www.c-sharpcorner.com/UploadFile/8a67c0/C-Sharp-coding-standards-and-naming-conventions/

will display the following output in the terminal:

```
Hello
World!
```

- Methods with multiple statements
  - Note that our two-line example has a `Main` method with multiple statements
  - In C#, each statement must end in a semicolon
  - Class and method declarations are not statements
  - Each line of code in your .cs file is not necessarily a statement
  - A single invocation/call of the `WriteLine` method is a statement
- The `Write` method
  - `Console.WriteLine("text")` prints the text, then starts a new line in the terminal – it effectively "hits enter" after printing the text
  - `Console.Write("text")` just prints the text, without starting a new line. It's like typing the text without hitting "enter" afterwards.
  - Even though two `Console.Write` calls are two statements, and appear on two lines, they will result in the text being printed on just one line. For example, this program:

    ```
    using System;

    class Welcome
    {
      static void Main()
      {
        Console.Write("Hello");
        Console.Write("World!");
      }
    }
    ```

    will display the following output in the terminal:

    ```
    HelloWorld!
    ```

  - Note that there is no space between "Hello" and "World!" because we did not type one in the argument to `Console.Write`
- Combining `Write` and `WriteLine`
  - We can use both `WriteLine` and `Write` in the same program

- After a call to `Write`, the "cursor" is on the same line after the printed text; after a call to `WriteLine` the "cursor" is at the beginning of the next line
- This program:

  ```
  using System;

  class Welcome
  {
    static void Main()
    {
      Console.Write("Hello ");
      Console.WriteLine("World!");
      Console.Write("Welcome to ");
      Console.WriteLine("CSCI 1301!");
    }
  }
  ```

  will display the following output in the terminal:

  ```
  Hello world!
  Welcome to CSCI 1301!
  ```

Escape Sequences

- Explicitly writing a new line
  - So far we've used `WriteLine` when we want to create a new line in the output
  - The **escape sequence** `\n` can also be used to create a new line – it represents the "newline character," which is what gets printed when you type "enter"
  - This program will produce the same output as our two-line "Hello World" example, with each word on its own line:

    ```
    using System;

    class Welcome
    {
      static void Main()
      {
        Console.Write("Hello\nWorld!\n");
      }
    }
    ```
- Escape sequences in detail

- An **escape sequence** uses "normal" letters to represent "special", hard-to-type characters
- `\n` represents the newline character, i.e. the result of pressing "enter"
- `\t` represents the tab character, which is a single extra-wide space (you usually get it by pressing the "tab" key)
- `\"` represents a double-quote character that will get printed on the screen, rather than ending the text string in the C# code.
  - Without this, you couldn't write a sentence with quotation marks in a `Console.WriteLine`, because the C# compiler would assume the quotation marks meant the string was ending
  - This program will not compile because `in quotes` is not valid C# code, and the compiler thinks it is not part of the string:

    ```
    //  Incorrect Code
    class Welcome
    {
        static void Main()
        {
            Console.WriteLine("This is "in
     ↪  quotes"");
            // This is parsed as if the string was
             ↪  "This is "
            // followed by in quotes, which is not
             ↪  valid C#,
            // followed by the empty string "".
        }
    }
    ```

  - This program will display the sentence including the quotation marks:

    ```
    using System;

    class Welcome
    {
      static void Main()
      {
        Console.WriteLine("This is \"in quotes\"");
      }
    }
    ```

- Note that all escape sequences begin with a backslash character (\), called the "escape character"
- General format is `\[key letter]` – the letter after the backslash is like a "keyword" indicating which special character to display. You can refer to the full list on microsoft documentation[322].
- If you want to put an actual backslash in your string, you need the escape sequence `\\`, which prints a single backslash
    - This will result in a compile error because `\U` is not a valid escape sequence:

      ```
      Console.WriteLine("Go to C:\Users\Edward");
      ```

    - This will display the path correctly:

      ```
      Console.WriteLine("Go to C:\\Users\\Edward");
      ```

Datatypes and Variables

Datatype Basics

- Recall the basic structure of a program
    - Program receives input from some source, uses input to make decisions, produces output for the outside world to see
    - In other words, the program reads some data, manipulates data, and writes out new data
    - In C#, data is stored in objects during the program's execution, and manipulated using the methods of those objects
- This data has **types**
    - Numbers (the number 2) are different from text (the word "two")
    - Text data is called "strings" because each letter is a **character** and a word is a *string of characters*
    - Within "numeric data," there are different types of numbers
        - Natural numbers ($\mathbb{N}$): 0, 1, 2, …
        - Integers ($\mathbb{Z}$): … -2, -1, 0, 1, 2, …
        - Real numbers ($\mathbb{R}$): 0.5, 1.333333…, -1.4, etc.
- Basic Datatypes in C#
    - C# uses keywords to name the types of data
    - Text data:
        - `string`: a string of characters, like `"Hello world!"`
        - `char`: a single character, like `'e'` or `'t'`

[322]https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/#string-escape-sequences

- Numeric data:
    * `int`: An integer, as defined previously
    * `uint`: An *unsigned* integer, in other words, a natural number (positive integers only)
    * `float`: A "floating-point" number, which is a real number with a fractional part, such as 3.85
    * `double`: A floating-point number with "double precision" – also a real number, but capable of storing more significant figures
    * `decimal`: An "exact decimal" number – also a real number, but has fewer rounding errors than `float` and `double` (we will explore the difference later) [323]

Literals and Variables

Literals and their types

- A **literal** is a data value written in the code
- A form of "input" provided by the programmer rather than the user; its value is fixed throughout the program's execution
- Literal data must have a type, indicated by syntax:
    - `string` literal: text in double quotes, like `"hello"`
    - `char` literal: a character in single quotes, like `'a'`
    - `int` literal: a number without a decimal point, with or without a minus sign (e.g. `52`)
    - `long` literal: just like an `int` literal but with the suffix `l` or `L`, e.g. `4L`
    - `double` literal: a number with a decimal point, with or without a minus sign (e.g. `-4.5`)
    - `float` literal: just like a `double` literal but with the suffix `f` or `F` (for "float"), e.g. `4.5f`
    - `decimal` literal: just like a `double` literal but with the suffix `m` or `M`(for "deciMal"), e.g. `6.01m`

Variables overview

- Variables store data that can *vary* (change) during the program's execution

[323]At this point, you may wonder "why don't we always use the most precise datatype instead of using imprecise ones?". There are three dimensions to consider to answer this question: first, using `decimal` takes more memory, hence more time, than the other numerical datatypes. Second, they are a bit more cumbersome to manipulate, as we will see later on. Last, you generally don't need to be *that* precise: for example, it would not make sense to use a floating-point number to account for human beings or other indivisible units. Even decimal may be an overkill for floating-point values sometimes: for instance, the NASA uses `3.141592653589793` as an approximation of pi[324] for their calculations. A `double` can hold such a value[325], so there is no need to be more precise.

- They have a type, just like literals, and also a name
- You can use literals to write data that gets stored in variables
- Sample program with variables:

  ```
  using System;

  class MyFirstVariables
  {
    static void Main()
    {
      // Declaration
      int myAge;
      string myName;
      // Assignment
      myAge = 29;
      myName = "Edward";
      // Displaying
      Console.WriteLine(
        $"My name is {myName} and I am {myAge} years
   ↪  old."
      );
    }
  }
  ```

  This program shows three major operations you can do with variables.

  - First it **declares** two variables, an `int`-type variable named "myAge" and a `string`-type variable named "myName"
  - Then, it **assigns** values to each of those variables, using literals of the same type. `myAge` is assigned the value 29, using the `int` literal `29`, and `myName` is assigned the value "Edward", using the `string` literal `"Edward"`
  - Finally, it **displays** the current value of each variable by using the `Console.WriteLine` method and **string interpolation**, in which the values of variables are inserted into a string by writing their names with some special syntax (a $ character at the beginning of the string, and braces around the variable names)

Variable Operations

Declaration

- This is when you specify the *name* of a variable and its *type*
- The syntax is the type keyword, a space, the name of the variable, then a semi-colon.

- Examples: `int myAge;`, `string myName;`, `double winChance;`.
- A variable name is an identifier, so it should follow the rules and conventions
  - Can only contain letters and numbers
  - Must be unique among all variable, method, and class names
  - Should use CamelCase if it contains multiple words
- Note that the variable's type is not part of its name: two variables cannot have the same name *even if* they are different types
- Multiple variables can be declared in the same statement: `string myFirstName, myLastName;` would declare *two* strings called respectively `myFirstName` and `myLastName`

Assignment

- The act of changing the value of a variable
- Uses the symbol =, which is the *assignment operator*, not a statement of equality – it does not mean "equals"
- Direction of assignment is **right to left**: the variable goes on the left side of the = symbol, and its new value goes on the right
- Syntax: `variable_name = value;`
- Example: `myAge = 29;`
- Value *must* match the type of the variable. If `myAge` was declared as an `int`-type variable, you cannot write `myAge = "29";` because `"29"` is a `string`

Initialization (Declaration + Assignment)

- Initialization statement combines declaration and assignment in one single statement (it is just a shortcut, a.k.a. some "syntactical sugar"[326], and not something new)
- Creates a new variable and also gives it an initial value
- The syntax is the datatype of the variable, the name of the variable, the = sign, the value we want to store, and a semi-colon
- Example: `string myName = "Edward";`
- Can only be used once per variable, since you can only declare a variable once

Assignment Details

- Assignment replaces the "old" value of the variable with a "new" one; it is how variables *vary*
  - If you initialize a variable with `int myAge = 29;` and then write `myAge = 30;`, the variable `myAge` now stores the value

[326]https://www.wikiwand.com/en/Syntactic_sugar

30

- You can assign a variable to another variable: just write a variable name on both sides of the = operator
  - This will take a "snapshot" of the current value of the variable on the right side, and store it into the variable on the left side
  - For example, in this code:

    ```
    int a = 12;
    int b = a;
    a = -5;
    ```

    the variable b gets the value 12, because that's the value that a had when the statement `int b = a` was executed. Even though a was then changed to -5 afterward, b is still `12`.

Displaying

- Only text (strings) can be displayed in the console
- When we want to print a mixture of text and variables with `Console.WriteLine`, we need to convert all of them to a string
- **String interpolation**: a mechanism for converting a variable's value to a `string` and inserting it into the main string
  - Syntax: `$"text {variable} text"` – begin with a $ symbol, then put variable's name inside brackets within the string
  - Example: `$"I am {myAge} years old"`
  - When this line of code is executed, it reads the variable's current value, converts it to a string (`29` becomes `"29"`), and inserts it into the surrounding string
  - Displayed: `I am 29 years old`
- If the argument to `Console.WriteLine` is the name of a variable, it will automatically convert that variable to a `string` before displaying it
- For example, `Console.WriteLine(myAge);` will display "29" in the console, as if we had written `Console.WriteLine($"{myAge}");`
- When string interpolation converts a variable to a string, it must call a "string conversion" method supplied with the data type (`int`, `double`, etc.). All built-in C# datatypes come with string conversion methods, but when you write your own data types (classes), you'll need to write your own string conversions – string interpolation will not magically "know" how to convert `MyClass` variables to `string`s

On a final note, observe that you can write statements mixing multiple declarations and assignments, as in `int myAge = 10, yourAge, ageDifference;` that declares three variables of type `int` and set the value of the first one. It is generally recommended to separate those instructions in different statements as you begin, to ease debugging and have a better understanding of the "atomic steps" your program should perform.

Format Specifiers

- Formats for displaying numbers
  - There are lots of possible ways to display a number, especially a fraction (how many decimal places to use?)
  - String interpolation has a default way to format numbers, but it might not always be the best
  - For example, consider this program:

    ```
    decimal price = 19.99m;
    decimal discount = 0.25m;
    decimal salePrice = price - discount * price;
    Console.WriteLine($"{price} with a discount of " +
        $"{discount} is {salePrice}");
    ```

    It will display this output:

    ```
    19.99 with a discount of 0.25 is 14.9925
    ```

    But this isn't the best way to display prices and discounts. Obviously, the prices should have dollar signs, but also, it does not make sense to show a price with fractional cents (14.9925) – it should be rounded to two decimal places. You might also prefer to display the discount as 25% instead of 0.25, since people usually think of discounts as percentages.
- Improving interpolation with format specifiers
  - You can change how numbers are displayed by adding a format specifier to a variable's name in string interpolation
  - **Format specifier**: A special letter indicating how a numeric value should be converted to a string
  - General format is `{[variable]:[format specifier]}`, e.g. `{numVar:N}`
  - Common format specifiers:

Format specifier	Description
N or n	Adds a thousands separator, displays 2 decimal places (by default)
E or e	Uses scientific notation, displays 6 decimal places (by default)
C or c	Formats as currency: Adds a currency symbol, adds thousands separator, displays 2 decimal places (by default)
P or p	Formats as percentage with 2 decimal places (by default)

- Example usage with our "discount" program:
  ```
  decimal price = 19.99m;
  decimal discount = 0.25m;
  decimal salePrice = price - discount * price;
  Console.WriteLine($"{price:C} with a discount of
   ↪   " +
      $"{discount:P} is {salePrice:C}");
  ```
  will display
  ```
  $19.99 with a discount of 25.00% is $14.99
  ```
- Format specifiers with custom rounding
  - Each format specifier uses a default number of decimal places, but you can change this with a precision specifier
  - **Precision specifier**: A number added after a format specifier indicating how many digits past the decimal point to display
  - Format is `{[variable]:[format specifier][precision specifier]}`, e.g. `{numVar:N3}`. Note there is no space or other symbol between the format specifier and the precision specifier, and the number can be more than one digit (`{numVar:N12}` is valid)
  - Examples:
    * Given the declarations
      ```
      double bigNumber = 1537963.666;
      decimal discount = 0.1337m;
      ```

Statement	Display
`Console.WriteLine($"{bigNumber:N}");`	`1,537,963.67`
`Console.WriteLine($"{bigNumber:N3}");`	`1,537,963.666`
`Console.WriteLine($"{bigNumber:N1}");`	`1,537,963.7`
`Console.WriteLine($"{discount:P1}");`	`13.4%`
`Console.WriteLine($"{discount:P4}");`	`13.3700%`
`Console.WriteLine($"{bigNumber:E}");`	`1.537964E+006`
`Console.WriteLine($"{bigNumber:E2}");`	`1.54E+006`

Variables in Memory

- A variable names a memory location
- Data is stored in memory (RAM), so a variable "stores data" by storing it in memory
- Declaring a variable reserves a memory location (address) and gives it a name
- Assigning to a variable stores data to the memory location (address) named by that variable

Sizes of Numeric Datatypes

- Numeric datatypes have different sizes
- Amount of memory used/reserved by each variable depends on the variable's type
- Amount of memory needed for an integer data type depends on the size of the number
  - `int` uses 4 bytes of memory, can store numbers in the range $[-2^{31}, 2^{31} - 1]$
  - `long` uses 8 bytes of memory can store numbers in the range $[-2^{63}, 2^{63} - 1]$
  - `short` uses 2 bytes of memory, can store numbers in the range $[-2^{15}, 2^{15} - 1]$
  - `sbyte` uses only 1 bytes of memory, can store numbers in the range $[-128, 127]$
- Unsigned versions of the integer types use the same amount of memory, but can store larger positive numbers
  - `byte` uses 1 byte of memory, can store numbers in the range $[0, 255]$
  - `ushort` uses 2 bytes of memory, can store numbers in the range $[0, 2^{16} - 1]$
  - `uint` uses 4 bytes of memory, can store numbers in the range $[0, 2^{32} - 1]$
  - `ulong` uses 8 bytes of memory, can store numbers in the range $[0, 2^{64} - 1]$
  - This is because in a signed integer, one bit (digit) of the binary number is needed to represent the sign (+ or -). This means the actual number stored must be 1 bit smaller than the size of the memory (e.g. 31 bits out of the 32 bits in 4 bytes). In an unsigned integer, there is no "sign bit", so all the bits can be used for the number.
- Amount of memory needed for a floating-point data type depends on the precision (significant figures) of the number
  - `float` uses 4 bytes of memory, can store positive or negative numbers in a range of approximately $[10^{-45}, 10^{38}]$, with 7 significant figures of precision
  - `double` uses 8 bytes of memory, and has both a wider range (10^{-324} to 10^{308}) and more significant figures (15 or 16)
  - `decimal` uses 16 bytes of memory, and has 28 or 29 significant figures of precision, but it actually has the smallest range (10^{-28} to 10^{28}) because it stores decimal fractions exactly
- Difference between binary fractions and decimal fractions
  - `float` and `double` store their data as binary (base 2) fractions, where each digit represents a power of 2
    - The binary number 101.01 represents $4 + 1 + 1/4$, or 5.25

in base 10

- More specifically, they use binary scientific notation: A mantissa (a binary integer), followed by an exponent assumed to be a power of 2, which is applied to the mantissa
    * 10101e-10 means a mantissa of 10101 (i.e. 21 in base 10) with an exponent of -10 (i.e. 2^{-2} in base 10), which also produces the value 101.01 or 5.25 in base 10
- Binary fractions cannot represent all base-10 fractions, because they can only represent fractions that are negative powers of 2. $1/10$ is not a negative power of 2 and cannot be represented as a sum of $1/16$, $1/32$, $1/64$, etc.
- This means some base-10 fractions will get "rounded" to the nearest finite binary fraction, and this will cause errors when they are used in arithmetic
- On the other hand, `decimal` stores data as a base-10 fraction, using base-10 scientific notation
- This is slower for the computer to calculate with (since computers work only in binary) but has no "rounding errors" with fractions that include 0.1
- Use `decimal` when working with money (since money uses a lot of 0.1 and 0.01 fractions), `double` when working with non-money fractions

Summary of numeric data types and sizes:

Type	Size	Range of Values	Precision
`sbyte`	1 bytes	$-128...127$	N/A
`byte`	1 bytes	$0...255$	N/A
`short`	2 bytes	$-2^{15}...2^{15}-1$	N/A
`ushort`	2 bytes	$0...2^{16}-1$	N/A
`int`	4 bytes	$-2^{31}...2^{31}-1$	N/A
`uint`	4 bytes	$0...2^{32}-1$	N/A
`long`	8 bytes	$-2^{63}...2^{63}-1$	N/A
`ulong`	8 bytes	$0...2^{64}-1$	N/A
`float`	4 bytes	$\pm1.5 \cdot 10^{-45}... \pm 3.4 \cdot 10^{38}$	7 digits
`double`	8 bytes	$\pm5.0 \cdot 10^{-324}... \pm 1.7 \cdot 10^{308}$	15-16 digits
`decimal`	16 bytes	$\pm1.0 \cdot 10^{-28}... \pm 7.9 \cdot 10^{28}$	28-29 digits

Value and Reference types

- Value and reference types are different ways of storing data in memory
- Variables name memory locations, but the data that gets stored at the named location is different for each type
- For a **value type** variable, the named memory location stores the exact data value held by the variable (just what you'd expect)
- Value types: all the numeric types (`int`, `float`, `double`, `decimal`, etc.), `char`, and `bool`
- For a **reference type** variable, the named memory location stores a *reference* to the data, not the data itself
  - The contents of the memory location named by the variable are the address of another memory location
  - The *other* memory location is where the variable's data is stored
  - To get to the data, the computer first reads the location named by the variable, then uses that information (the memory address) to find and read the other memory location where the data is stored
- Reference types: `string`, `object`, and all objects you create from your own classes
- Assignment works differently for reference types
  - Assignment always copies the value in the variable's named memory location - but in the case of a reference type that's just a memory address, not the data
  - Assigning one reference-type variable to another copies the memory address, so now both variables "refer to" the same data
  - Example:

    ```
    string word = "Hello";
    string word2 = word;
    ```

    Both `word` and `word2` contain the same memory address, pointing to the same memory location, which contains the string "Hello". There is only one copy of the string "Hello"; `word2` does not get its own copy.

Operators

Arithmetic Operators

Variables can be used to do math. All the usual arithmetic operations are available in C#:

Operation	C# Operator	C# Expression
Addition	`+`	`myVar + 7`
Subtraction	`-`	`myVar - 7`
Multiplication	`*`	`myVar * 7`
Division	`/`	`myVar / 7`
Remainder (a.k.a. modulo)	`%`	`myVar % 7`

Note: the "remainder" or "modulo" operator represents the remainder after doing integer division between its two operands.
For example, 44 % 7 = 2 because 44/7 = 6 when rounded down, then do 7*6 to get 42 and 44 - 42 = 2.

Arithmetic and variables

- The result of an arithmetic expression (like those shown in the table) is a numeric value
  - For example, the C# expression `3 * 4` has the value `12`, which is `int` data
- A numeric value can be assigned to a variable of the same type, just like a literal: `int myVar = 3 * 4;` initializes the variable `myVar` to contain the value `12`
- A numeric-type variable can be used in an arithmetic expression
- When a variable is used in an arithmetic expression, its current value is read, and the math is done on that value
- Example:

  ```
  int a = 4;
  int b = a + 5;
  a = b * 2;
  ```

  - To execute the second line of the code, the computer will first evaluate the expression on the right side of the = sign. It reads the value of the variable `a`, which is 4, and then computes the result of `4 + 5`, which is 9. Then, it assigns this value to the variable `b` (remember assignment goes right to left).

  - To execute the third line of code, the computer first evaluates the expression on the right side of the = sign, which means reading the value of `b` to use in the arithmetic operation. `b` contains 9, so the expression is `9 * 2`, which evaluates to 18. Then it assigns the value 18 to the variable `a`, which now contains 18 instead of 4.

- A variable can appear on both sides of the = sign, like this:

  ```
  int myVar = 4;
  myVar = myVar * 2;
  ```

  This looks like a paradox because `myVar` is assigned to itself, but it has a clear meaning because assignment is evaluated right to left. When executing the second line of code, the computer evaluates the right side of the = before doing the assignment. So it first reads the current ("old") value of `myVar`, which is 4, and computes `4 * 2` to get the value 8. Then, it assigns the new value to `myVar`, overwriting its old value.

Compound assignment operators

- The pattern of "compute an expression with a variable, then assign the result to that variable" is common, so there are shortcuts for doing it
- The **compound assignment operators** change the value of a variable by adding, subtracting, etc. from its current value, equivalent to an assignment statement that has the value on both sides:

Statement	Equivalent
`x += 2;`	`x = x + 2;`
`x -= 2;`	`x = x - 2;`
`x *= 2;`	`x = x * 2;`
`x /= 2;`	`x = x / 2;`
`x %= 2;`	`x = x % 2;`

Increment and Decrement Operators

Increment and decrement basics

- In C#, we have already seen multiple ways to add 1 to a numeric variable:

```
int myVar = 1;
myVar = myVar + 1;
myVar += 1
```

These two lines of code have the same effect; the += operator is "shorthand" for "add and assign"

- The **increment operator**, ++, is an even shorter way to add 1 to a variable. It can be used in two ways:

```
myVar++;
++myVar;
```

- Writing ++ after the name of the variable is called a **postfix increment**, while writing ++ before the name of the variable is called a **prefix increment**. They both have the same effect on the variable: its value increases by 1.

- Similarly, there are multiple ways to subtract 1 from a numeric variable:

```
int myVar = 10;
myVar = myVar - 1;
myVar -= 1;
```

- The **decrement operator**, --, is a shortcut for subtracting 1 from a variable, and is used just like the increment operator:

```
myVar--;
--myVar;
```

- To summarize, the increment and decrement operators both have a prefix and postfix version:

	Increment	Decrement
Postfix	`myVar++`	`myVar--`
Prefix	`++myVar`	`--myVar`

Difference between prefix and postfix

- The prefix and postfix versions of the increment and decrement operators both have the same effect on the variable: Its value increases or decreases by 1
- The difference between prefix and postfix is whether the "old" or "new" value of the variable is *returned* by the expression
- With postfix increment/decrement, the operator returns the value of the variable, *then* increases/decreases it by 1
- This means the value of the increment/decrement expression is the *old* value of the variable, before it was incremented/decremented
- Consider this example:

```
int a = 1;
Console.WriteLine(a++);
Console.WriteLine(a--);
```

- The expression a++ returns the current value of a, which is 1, to be used in Console.WriteLine. *Then* it increments a by 1, giving it a new value of 2. Thus, the first Console.WriteLine displays "1" on the screen.
- The expression a-- returns the current value of a, which is 2, to be used in Console.WriteLine, and *then* decrements a by 1. Thus, the second Console.WriteLine displays "2" on the screen.
- With prefix increment/decrement, the operator increases/decreases the value of the variable by 1, *then* returns its value
- This means the value of the increment/decrement expression is the *new* value of the variable, after the increment/decrement
- Consider the same code, but with prefix instead of postfix operators:

```
int a = 1;
Console.WriteLine(++a);
Console.WriteLine(--a);
```

- The expression ++a increments a by 1, then returns the value of a for use in Console.WriteLine. Thus, the first Console.WriteLine displays "2" on the screen.
- The expression --a decrements a by 1, then returns the value of a for use in Console.WriteLine. Thus, the second Console.WriteLine displays "1" on the screen.

Using increment/decrement in expressions

- The ++ and -- operators have higher precedence than the other math operators, so if you use them in an expression they will get executed first
- The "result" of the operator, i.e. the value that will be used in the rest of the math expression, depends on whether it is the prefix or postfix increment/decrement operator: The prefix operator returns the variable's new value, while the postfix operator returns the variable's old value
- Consider these examples:

```
int a = 1;
int b = a++;
int c = ++a * 2 + 4;
int d = a-- + 1;
```

- The variable b gets the value 1, because a++ returns the "old" value of a (1) and then increments a to 2
- In the expression ++a `* 2 + 4`, the operator ++a executes first, and it returns the new value of a, which is 3. Then the multiplication executes (`3 * 2`, which is 6), then the addition (`6 + 4`, which is 10). Thus c gets the value 10.
- In thee expression a`-- + 1`, the operator a`--` executes first, and it returns the *old* value of a, which is 3 (even though a is now 2). Then the addition executes, so d gets the value 4.

Arithmetic on Mixed Data Types

- The math operators (+, -, *, /) are defined separately for each data type: There is an `int` version of + that adds `int`s, a `float` version of + that adds `float`s, etc.
- Each operator expects to get two values of the same type on each side, and produces a result of that same type. For example, `2.25 + 3.25` uses the `double` version of +, which adds the two `double` values to produce a `double`-type result, 5.5.
- Most operators have the same effect regardless of their type, except for /
- The `int`/`short`/`long` version of / does **integer division**, which returns only the quotient and drops the remainder: In the statement `int result = 21 / 5;`, the variable `result` gets the value 4, because $21 \div 5$ is 4 with a remainder of 1. If you want the fractional part, you need to use the floating-point version (for `float`, `double`, and `decimal`): `double fracDiv = 21.0 / 5.0;` will initialize `fracDiv` to 4.2.

Implicit conversions in math

- If the two operands/arguments to a math operator are not the same type, they must become the same type – one must be converted
- C# will first try implicit conversion to "promote" a less-precise or smaller value to a more precise, larger type
- Example: with the expression `double fracDiv = 21 / 2.4;`
  - Operand types are `int` and `double`
  - `int` is smaller/less-precise than `double`
  - 21 gets implicitly converted to 21.0, a `double` value
  - Now the operands are both `double` type, so the `double` version of the / operator gets executed
  - The result is 8.75, a `double` value, which gets assigned to the variable `fracDiv`

- Implicit conversion also happens in assignment statements, which happen *after* the math expression is computed
- Example: with the expression `double fraction = 21 / 5;`
  - Operand types are `int` and `int`
  - Since they match, the `int` version of / gets executed
  - The result is 4, an `int` value
  - Now this value is assigned to the variable `fraction`, which is `double` type
  - The `int` value is implicitly converted to the `double` value 4.0, and `fraction` is assigned the value 4.0

Explicit conversions in math

- If the operands are `int` type, the `int` version of / will get called, even if you assign the result to a `double`

- You can "force" floating-point division by explicitly converting one operand to `double` or `float`

- Example:

  ```
  int numCookies = 21;
  int numPeople = 6;
  double share = (double) numCookies / numPeople;
  ```

  Without the cast, `share` would get the value 3.0 because `numCookies` and `numPeople` are both `int` type (just like the `fraction` example above). With the cast, `numCookies` is converted to the value 21.0 (a `double`), which means the operands are no longer the same type. This will cause `numPeople` to be implicitly converted to `double` in order to make them match, and the `double` version of / will get called to evaluate `21.0 / 6.0`. The result is 3.5, so `share` gets assigned 3.5.

- You might also *need* a cast to ensure the operands are the same type, if implicit conversion does not work

- Example:

  ```
  decimal price = 3.89;
  double shares = 47.75;
  decimal total = price * (decimal) shares;
  ```

  In this code, `double` cannot be implicitly converted to `decimal`, and `decimal` cannot be implicitly converted to `double`, so the multiplication `price * shares` would produce a compile error. We need an explicit cast to `decimal` to make both operands the same type (`decimal`).

Order of Operations

- Math operations in C# follow PEMDAS from math class: Parentheses, Exponents, Multiplication, Division, Addition, Subtraction
  - Multiplication/division are evaluated together, as are addition/subtraction
  - Expressions are evaluated left-to-right
  - Example: `int x = 4 = 10 * 3 - 21 / 2 - (3 + 3);`
    - Parentheses: `(3 + 3)` is evaluated, returns 6
    - Multiplication/Division: `10 * 3` is evaluated to produce 30, then `21 / 2` is evaluated to produce 10 (left-to-right)
    - Addition/Subtraction: 4 + 30 - 10 - 6 is evaluated, result is 18
- Cast operator is higher priority than all binary operators
  - Example: `double share = (double) numCookies / numPeople;`
    - Cast operator is evaluated first, converts `numCookies` to a `double`
    - Division is evaluated next, but operand types do not match
    - `numPeople` is implicitly converted to `double` to make operand types match
    - Then division is evaluated, result is 21.0 / 6.0 = 3.5
- Parentheses always increase priority, even with casts
  - An expression in parentheses gets evaluated before the cast "next to" it
  - Example:
    ```
    int a = 5, b = 4;
    double result = (double) (a / b);
    ```
    The expression in parentheses gets evaluated first, then the result has the `(double)` cast applied to it. That means `a / b` is evaluated to produce 1, since `a` and `b` are both `int` type, and then that result is cast to a `double`, producing 1.0.

Conversions

We now discuss implicit and explicit conversions between datatypes: how C# can (or not!) convert a value from one datatype to another, and how we can "force" this conversion if C# does not do it automatically.

Assignments from different types

- The "proper" way to initialize a variable is to assign it a literal of the same type:

```
int myAge = 29;
double myHeight = 1.77;
```

```
float radius = 2.3f;
```

Note that `1.77` is a `double` literal, while `2.3f` is a `float` literal

- If the literal is not the same type as the variable, you will sometimes get an error – for example, `float radius = 2.3` will result in a compile error – but sometimes, it appears to work fine: for example `float radius = 2;` compiles and executes without error even though 2 is an `int` value.
- In fact, the value being assigned to the variable **must** be the same type as the variable, but some types can be **implicitly converted** to others

Implicit conversions

- Implicit conversion allows variables to be assigned from literals of the "wrong" type: the literal value is first implicitly converted to the right type
- In the statement `float radius = 2;`, the `int` value 2 is implicitly converted to an equivalent `float` value, `2.0f`. Then the computer assigns `2.0f` to the `radius` variable.
- Implicit conversion also allows variables to be assigned from other variables that have a different type:

```
int length = 2;
float radius = length;
```

When the computer executes the second line of this code, it reads the variable `length` to get an `int` value 2. It then implicitly converts that value to `2.0f`, and then assigns `2.0f` to the `float`-type variable `radius`.

- Implicit conversion only works between *some* data types: a value will only be implicitly converted if it is "safe" to do so without losing data
- Summary of possible implicit conversions:

Type	Possible Implicit Conversions
`short`	`int`, `long`, `float`, `double`, `decimal`
`int`	`long`, `float`, `double`, `decimal`
`long`	`float`, `double`, `decimal`
`ushort`	`uint`, `int`, `ulong`, `long`, `decimal`, `float`, `double`
`uint`	`ulong`, `long`, `decimal`, `float`, `double`
`ulong`	`decimal`, `float`, `double`
`float`	`double`

- In general, a data type can only be implicitly converted to one with a *larger range* of possible values
- Since an `int` can store any integer between -2^{31} and $2^{31}-1$, but a `float` can store any integer between -3.4×10^{38} and 3.4×10^{38} (as well as fractional values), it is always safe to store an `int` value in a `float`
- You *cannot* implicitly convert a `float` to an `int` because an `int` stores fewer values than a `float` – it cannot store fractions – so converting a `float` to an `int` will **lose data**
- Note that all integer data types can be implicitly converted to `float` or `double`
- Each integer data type can be implicitly converted to a larger integer type: `short` $\rightarrow$ `int` $\rightarrow$ `long`
- Unsigned integer data types can be implicitly converted to a *larger* signed integer type, but not the *same* signed integer type: `uint` $\rightarrow$ `long`, but **not** `uint` $\rightarrow$ `int`
- This is because of the "sign bit": a `uint` can store larger values than an `int` because it does not use a sign bit, so converting a large `uint` to an `int` might lose data

Explicit conversions

- Any conversion that is "unsafe" because it might lose data will not happen automatically: you get a compile error if you assign a `double` variable to a `float` variable
- If you want to do an unsafe conversion anyway, you must perform an **explicit conversion** with the **cast operator**
- Cast operator syntax: `([type name]) [variable or value]` – the cast is "right-associative", so it applies to the variable to the right of the type name
- Example: `(float) 2.8` or `(int) radius`
- Explicit conversions are often used when you (the programmer) know the conversion is actually "safe" – data will not actually be lost
- For example, in this code, we know that 2.886 is within the range of a `float`, so it is safe to convert it to a `float`:

```
float radius = (float) 2.886;
```

The variable `radius` will be assigned the value `2.886f`.

- For example, in this code, we know that 2.0 is safe to convert to an `int` because it has no fractional part:

```
double length = 2.0;
int height = (int) length;
```

The variable `height` will be assigned the value `2`.

- Explicit conversions only work if there exists code to perform the conversion, usually in the standard library. The cast operator isn't "magic" – it just calls a method that is defined to convert one type of data (e.g. `double`) to another (e.g. `int`).

- All the C# numeric types have explicit conversions to each other defined

- `string` does not have explicit conversions defined, so you cannot write `int myAge = (int) "29";`

- If the explicit conversion is truly unsafe (will lose data), data is lost in a specific way

- Casting from floating-point (e.g. `double`) types to integer types: fractional part of number is *truncated* (ignored/dropped)

- In `int length = (int) 2.886;`, the value 2.886 is truncated to 2 by the cast to `int`, so the variable `length` gets the value 2.

- Casting from more-precise to less-precise floating point type: number is *rounded* to nearest value that fits in less-precise type:

```
decimal myDecimal = 123456789.999999918m;
double myDouble = (double) myDecimal;
float myFloat = (float) myDouble;
```

In this code, `myDouble` gets the value 123456789.99999993, while `myFloat` gets the value `123456790.0f`, as the original `decimal` value is rounded to fit types with fewer significant figures of precision.

- Casting from a larger integer to a smaller integer: the most significant *bits* are truncated – remember that numbers are stored in binary format

- This can cause weird results, since the least-significant *bits* of a number in binary do not correspond to the least significant *digits* of the equivalent base-10 number

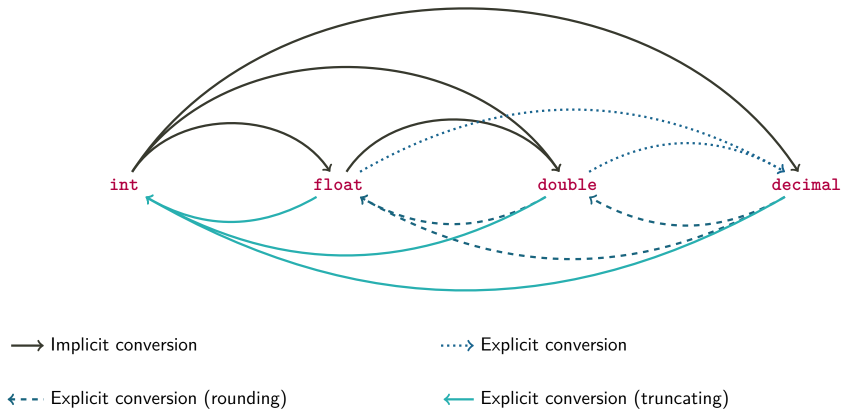
- Casting from another floating point type to `decimal`: Either value is stored precisely (no rounding), or *program crashes* with `System.OverflowException` if value is larger than `decimal`'s maximum value:

```
decimal fromSmall = (decimal) 42.76875;
double bigDouble = 2.65e35;
decimal fromBig = (decimal) bigDouble;
```

In this code, `fromSmall` will get the value `42.76875m`, but the program will crash when attempting to cast `bigDouble` to a `decimal` because 2.65×10^{35} is larger than `decimal`'s maximum value of 7.9×10^{28}

- `decimal` is more precise than the other two floating-point types (thus does not need to round), but has a smaller range (only 10^{28}, vs. 10^{308})

Summary of implicit and explicit conversions for the numeric datatypes:



int
float
double
decimal
Implicit conversion
Explicit conversion
Explicit conversion (rounding)
Explicit conversion (truncating)

Figure 13: "Implicit and Explicit Conversion Between Datatypes"

Refer to the "Result Type of Operations" chart from the cheatsheet[327] for more detail.

Inputs and Outputs

Reading Input from the User

- Input and output in CLI
  - Our programs use a command-line interface, where input and output come from text printed in a "terminal" or "console"

[327] https:/princomp.github.io/docs/programming_and_computer_usage/datatypes_in_csharp

    - We've already seen that `Console.WriteLine` prints text from your program on the screen to provide output to the user
    - The equivalent method for reading input is `Console.ReadLine()`, which waits for the user to type some text in the console and then returns it to your program
    - In general, the `Console` class represents the command-line interface
- Using `Console.ReadLine()`
    - Example usage:

      ```
      using System;

      class PersonalizedWelcomeMessage
      {
        static void Main()
        {
          string firstName;
          Console.WriteLine("Enter your first name:");
          firstName = Console.ReadLine();
          Console.WriteLine($"Welcome, {firstName}!");
        }
      }
      ```

      This program first declares a `string` variable named `firstName`. On the second line, it uses `Console.WriteLine` to display a message (instructions for the user). On the third line, it calls the `Console.ReadLine()` method, and assigns its return value (result) to the `firstName` variable. This means the program waits for the user to type some text and press "Enter", and then stores that text in `firstName`. Finally, the program uses string interpolation in `Console.WriteLine` to display a message including the contents of the `firstName` variable.
    - `Console.ReadLine` is the "inverse" of `Console.WriteLine`, and the way you use it is also the "inverse"
    - While `Console.WriteLine` takes an argument, which is the text you want to display on the screen, `Console.ReadLine()` takes no arguments because it does not need any input from your program – it will always do the same thing
    - `Console.WriteLine` has no "return value" - it does not give any output back to your program, and the only effect of calling it is that text is displayed on the screen
    - `Console.ReadLine()` does have a return value, specifically a `string`. This means you can use the result of this method to assign a `string` variable, just like you can use the result of an arithmetic expression to assign a numeric variable.
    - The `string` that `Console.ReadLine()` returns is **one line of text** typed in the console. When you call it, the computer will wait for the user to type some text and then press "Enter", and

everything the user typed before pressing "Enter" gets returned from `Console.ReadLine()`

Parsing user input

- `Console.ReadLine()` always returns the same type of data, a `string`, regardless of what the user enters
  - If you ask the user to enter a number, `ReadLine` will output that number as a `string`
  - For example, if you ask the user to enter his/her age, and the user enters 21, `Console.ReadLine()` will return the string `"21"`
- If we want to do any kind of arithmetic with a number provided by the user, we will need to convert that `string` to a numeric type like `int` or `double`. Remember that casting cannot be used to convert numeric data *to or from* `string` data.
- When converting numeric data to `string` data, we use string interpolation:

```
int myAge = 29;
//This does not work:
//string strAge = (string) myAge;
string strAge = $"{myAge}";
```

- In the other direction, we use a method called `Parse` to convert `string`s to numbers:

```
string strAge = "29";
//This does not work:
//int myAge = (int) strAge;
int myAge = int.Parse(strAge);
```

- The `int.Parse` method takes a `string` as an argument, and returns an `int` containing the numeric value written in that `string`
- Each built-in numeric type has its own `Parse` method
  - `int.Parse("42")` returns the value 42
  - `long.Parse("42")` returns the value `42L`
  - `double.Parse("3.65")` returns the value 3.65
  - `float.Parse("3.65")` returns the value `3.65f`
  - `decimal.Parse("3.65")` returns the value `3.65m`
- The Parse methods are useful for converting user input to useable data. For example, this is how to get the user's age as an `int`:

```
Console.WriteLine("Enter your age:");
string ageString = Console.ReadLine();
int age = int.Parse(ageString);
```

More detail on the `Parse` methods

- `Console.WriteLine` is a method that takes input from your program, in the form of an argument, but does not return any output. Meanwhile, `Console.ReadLine` is a method that does not have any arguments, but it returns output to your program (the user's string).
- `int.Parse` is a method that both takes input (the `string` argument) and returns output (the converted `int` value)
- When executing a statement such as

  ```
  int answer = int.Parse("42");
  ```

  the computer must evaluate the expression on the right side of the = operator before it can do the assignment. This means it calls the `int.Parse` method with the string `"42"` as input. The method's code then executes, converting `"42"` to an integer, and it returns a result, the `int` value `42`. This value can now be assigned to the variable `answer`.
- Since the return value of a `Parse` method is a numeric type, it can be used in arithmetic expressions just like a numeric-type variable or literal. For example, in this statement:

  ```
  double result = double.Parse("3.65") * 4;
  ```

  To evaluate the expression on the right side of the = operator, the computer must first call the method `double.Parse` with the input `"3.65"`. Then the method's return value, `3.65`, is used the math operation as if it was written `3.65 * 4`. So the computer implicitly converts `4` to a `double` value, performs the multiplication on `double`s, and gets the resulting value `14.6`, which it assigns to the variable `result`.
- Another example of using the result of `Parse` to do math:

  ```
  Console.WriteLine("Please enter the year.");
  string userInput = Console.ReadLine();
  int curYear = int.Parse(userInput);
  Console.WriteLine($"Next year it will be {curYear +
   ↪  1}");
  ```

  Note that in order to do arithmetic with the user's input (i.e. add 1), it must be a numeric type (i.e. `int`), not a `string`. This is why we often call a `Parse` method on the data returned by `Console.ReadLine()`.
- The previous example can be made shorter and simpler by combining the `Parse` and `ReadLine` methods in one statement. Specif-

ically, you can write:

```
int curYear = int.Parse(Console.ReadLine());
```

In this statement, the return value (output) of one method is used as the argument (input) to another method. When the computer executes the statement, it starts by evaluating the `int.Parse(...)` method call, but it cannot actually execute the `Parse` method yet because its argument is an expression, not a variable or value. In order to determine what value to send to the `Parse` method as input, it must first evaluate the `Console.ReadLine()` method call. Since this method has no arguments, the computer can immediately start executing it; the `ReadLine` method waits for the user to type a line of text, then returns that text as a `string` value. This return value can now be used as the argument to `int.Parse`, and the computer starts executing `int.Parse` with the user-provided string as input. When the `Parse` method returns an `int` value, this value becomes the value of the entire expression `int.Parse(Console.ReadLine())`, and the computer assigns it to the variable `curYear`.

- Notice that by placing the call to `ReadLine` inside the argument to `Parse`, we have eliminated the variable `userInput` entirely. The `string` returned by `ReadLine` does not need to be stored anywhere (i.e. in a variable); it only needs to exist long enough to be sent to the `Parse` method as input.

Correct input formatting

- The Parse methods *assume* that the string they are given as an argument (i.e. the user input) actually contains a valid number. But the user may not follow directions, and invalid input can cause a variety of errors.
- If the string does not contain a number at all – e.g. `int badIdea = int.Parse("Hello");` – the program will fail with the error `System.FormatException`
- If the string contains a number with a decimal point, but the `Parse` method is for an integer datatype, the program will also fail with `System.FormatException`. For example, `int fromFraction = int.Parse("52.5");` will cause this error. This will happen even if the number in the string ends in ".0" (meaning it has no fractional part), such as `int wholeNumber = int.Parse("45.0");`.
- If the string has extraneous text before or after the number, such as `"$18.95"` or `1999!`, the program will fail with the error `System.FormatException`
- If the string contains a number that cannot fit in the desired datatype (due to overflow or underflow), the behavior depends on the datatype:

  - For the integer types (`int` and `long`), the program will fail with the error `System.OverflowException`. For example, `int.Parse("3000000000")` will cause this error because 3000000000 is larger than $2^{31} - 1$ (the maximum value an `int` can store).
  - For the floating-point types (`float` and `double`), no error will be produced. Instead, the result will be the same as if an overflow or underflow had occurred during normal program execution: an overflow will produce the value `Infinity`, and an underflow will produce the value `0`. For example, `float tooSmall = float.Parse("1.5e-55");` will assign `tooSmall` the value 0, while `double tooBig = double.Parse("1.8e310");` will assign `tooBig` the value `double.Infinity`.
- Acceptable string formats vary slightly between the numeric types, due to the different ranges of values they can contain
  - `int.Parse` and `long.Parse` will accept strings in the format `([ws])([sign])[digits]([ws])`, where `[ws]` represents empty spaces and groups in parentheses are **optional**. This means that a string with leading or trailing spaces will not cause an error, unlike a string with other extraneous text around the number.
  - `decimal.Parse` will accept strings in the format `([ws])([sign])([digits],)[digits](` Note that you can optionally include commas between groups of digits, and the decimal point is also optional. This means a string like `"18,999"` is valid for `decimal.Parse` but not for `int.Parse`.
  - `float.Parse` and `double.Parse` will accept strings in the format `([ws])([sign])([digits],)[digits](.[digits])(e[sign][digits])([ws])`. As with `decimal`, you can include commas between groups of digits. In addition, you can write the string in scientific notation with the letter "e" or "E" followed by an exponent, such as `"-9.44e15"`.

Output with Variables

Converting from numbers to strings

- As we saw in a previous lecture (Datatypes and Variables), the `Console.WriteLine` method needs a `string` as its argument
- If the variable you want to display is not a `string`, you might think you could cast it to a `string`, but that will not work – there is no explicit conversion from `string` to numeric types
  - This code:

    ```
    double fraction = (double) 47 / 6;
    ```

```
string text = (string) fraction;
```

will produce a compile error

- You *can* convert numeric data to a `string` using string interpolation, which we've used before in `Console.WriteLine` statements:

  ```
  int x = 47, y = 6;
  double fraction = (double) x / y;
  string text = $"{x} divided by {y} is {fraction}";
  ```

  After executing this code, `text` will contain "47 divided by 6 is 7.8333333"

- String interpolation can convert any expression to a `string`, not just a single variable. For example, you can write:

  ```
  Console.WriteLine($"{x} divided by {y} is {(double) x
   ↪  / y}");
  Console.WriteLine($"{x} plus 7 is {x + 7}");
  ```

  This will display the following output:

  ```
  47 divided by 6 is 7.8333333
  47 plus 7 is 54
  ```

  Note that writing a math expression inside a string interpolation statement does not change the values of any variables. After executing this code, x is still 47, and y is still 6.

The `ToString()` method

- String interpolation does not "magically know" how to convert numbers to strings – it delegates the task to the numbers themselves
- This works because all data types in C# are objects, even the built-in ones like `int` and `double`
  - Since they are objects, they can have methods
- **All** objects in C# are guaranteed to have a method named `ToString()`, whose return value (result) is a `string`
- Meaning of `ToString()` method: "Convert this object to a `string`, and return that `string`"
- This means you can call the `ToString()` method on any variable to convert it to a `string`, like this:

  ```
  int num = 42;
  double fraction = 33.5;
  string intText = num.ToString();
  string fracText = fraction.ToString();
  ```

After executing this code, `intText` will contain the string "42", and `fracText` will contain the string "33.5"

- String interpolation calls `ToString()` on each variable or expression within braces, asking it to convert itself to a string
  - In other words, these three statements are all the same:

    ```
    Console.WriteLine($"num is {num}");
    Console.WriteLine($"num is {intText}");
    Console.WriteLine($"num is {num.ToString()}");
    ```

    Putting `num` within the braces is the same as calling `ToString()` on it.

String Concatenation

- Now that we've seen `ToString()`, we can introduce another operator: the concatenation operator
- Concatenation basics
  - Remember, the + operator is defined separately for each data type. The "`double + double`" operator is different from the "`int + int`" operator.
  - If the operand types are `string` (i.e. `string + string`), the + operator performs concatenation, not addition
  - You can concatenate `string` literals or `string` variables:

    ```
    string greeting = "Hi there, " + "John";
    string name = "Paul";
    string greeting2 = "Hi there, " + name;
    ```

    After executing this code, `greeting` will contain "Hi there, John" and `greeting2` will contain "Hi there, Paul"
- Concatenation with mixed types
  - Just like with the other operators, both operands (both sides of the +) must be the same type
  - If one operand is a `string` and the other is not a `string`, the `ToString()` method will automatically be called to convert it to a `string`
  - Example: In this code:

    ```
    int bananas = 42;
    string text = "Bananas: " + bananas;
    ```

    The + operator has a `string` operand and an `int` operand, so the `int` will be converted to a `string`. This means the computer will call `bananas.ToString()`, which returns the string "42". Then the `string + string` operator is called with the operands "Bananas:" and "42", which concatenates them into "Bananas: 42".

Output with concatenation

- We now have two different ways to construct a string for `Console.WriteLine`: Interpolation and concatenation
- Concatenating a string with a variable will automatically call its `ToString()` method, just like interpolation will. These two `WriteLine` calls are equivalent:

  ```
  int num = 42;
  Console.WriteLine($"num is {num}");
  Console.WriteLine("num is " + num);
  ```

- It's usually easier to use interpolation, since when you have many variables the + signs start to add up. Compare the length of these two equivalent lines of code:

  ```
  Console.WriteLine($"The variables are {a}, {b}, {c},
   ↪  {d}, and {e}");
  Console.WriteLine("The variables are " + a + ", " + b
   ↪  + ", " + c + ", " + d + ", and " + e);
  ```

- Be careful when using concatenation with numeric variables: the meaning of + depends on the types of its two operands
  - If both operands are numbers, the + operator does addition
  - If both operands are strings, the + operator does concatenation
  - If *one* argument is a string, the other argument will be converted to a string using `ToString()`
  - Expressions in C# are always evaluated **left-to-right**, just like arithmetic
  - Therefore, in this code:

    ```
    int var1 = 6, var2 = 7;
    Console.WriteLine(var1 + var2 + " is the result");
    Console.WriteLine("The result is " + var1 + var2);
    ```

    The first `WriteLine` will display "13 is the result", because `var1` and `var2` are both `int`s, so the first + operator performs addition on two `int`s (the resulting number,13, is then converted to a `string` for the second + operator). However, the second `WriteLine` will display "The result is 67", because both + operators perform concatenation: The first one concatenates a string with `var1` to produce a string, and then the second one concatenates this string with `var2`

- If you want to combine addition and concatenation in the same line of code, use parentheses to make the order and grouping of operations explicit. For example:

  ```
  int var1 = 6, var2 = 7;
  Console.WriteLine((var1 + var2) + " is the
   ↪  result");
  Console.WriteLine("The result is " + (var1 +
   ↪  var2));
  ```

  In this code, the parentheses ensure that `var1 + var2` is always interpreted as addition.

Introduction

Class and Object Basics

- Classes vs. Objects
  - A **class** is a specification, blueprint, or template for an object; it is the code that describes what data the object stores and what it can do
  - An **object** is a single instance of a class, created using its "template." It is executing code, with specific values stored in each variable
  - To **instantiate** an object is to create a new object from a class
- Object design basics
  - Objects have **attributes**: data stored in the object. This data is different in each instance, although the type of data is defined in the class.
  - Objects have **methods**: functions that use or modify the object's data. The code for these functions is defined in the class, but it is executed on (and modifies) a specific object
- Encapsulation: An important principle in class/object design
  - Attribute data is stored in **instance variables**, a special kind of variable
  - Called "instance" because each instance, i.e. object, has its own copy of them
  - **Encapsulation** means instance variables (attributes) are "hidden" inside an object: other code cannot access them directly
    * Only the object's own methods can access the instance variables
    * Other code must "ask permission" from the object in order to read or write the variables

Writing Our First Class

- Designing the class
  - Our first class will be used to represent rectangles; each instance (object) will represent one rectangle
  - Attributes of a rectangle:
    - Length
    - Width
  - Methods that will use the rectangle's attributes
    - Get length
    - Get width
    - Set length
    - Set width
    - Compute the rectangle's area
  - Note that the first four are a specific type of method called "getters" and "setters" because they allow other code to read (get) or write (set) the rectangle's instance variables while respecting encapsulation

The Rectangle class:

```
class Rectangle
{
  private int length;
  private int width;

  public void SetLength(int lengthParameter)
  {
    length = lengthParameter;
  }

  public int GetLength()
  {
    return length;
  }

  public void SetWidth(int widthParameter)
  {
    width = widthParameter;
  }

  public int GetWidth()
  {
    return width;
  }

  public int ComputeArea()
```

```
    {
      return length * width;
    }
}
```

Let's look at each part of this code in order.

- Attributes
    - Each attribute (length and width) is stored in an instance variable
    - Instance variables are declared similarly to "regular" variables, but with one additional feature: the **access modifier**
    - Syntax: `[access modifier] [type] [variable name]`
    - The access modifier can have several values, the most common of which are **`public`** and **`private`**. (There are other access modifiers, such as **`protected`** and **`internal`**, but in this class we will only be using **`public`** and **`private`**).
    - An access modifier of **`private`** is what enforces encapsulation: when you use this access modifier, it means the instance variable cannot be accessed by any code outside the `Rectangle` class
    - The C# compiler will give you an error if you write code that attempts to use a **`private`** instance variable anywhere other than a method of that variable's class
- SetLength method, an example of a "setter" method
    - This method will allow code outside the `Rectangle` class to modify a `Rectangle` object's "length" attribute
    - Note that the header of this method has an access modifier, just like the instance variable
    - In this case the access modifier is **`public`** because we *want* to allow other code to call the `SetLength` method
    - Syntax of a method declaration: `[access modifier] [return type] [method name](`
    - This method has one **parameter**, named `lengthParameter`, whose type is `int`. This means the method must be called with one **argument** that is `int` type.
        - Similar to how `Console.WriteLine` must be called with one argument that is `string` type – the `Console.WriteLine` declaration has one parameter that is `string` type.
        - Note that it is declared just like a variable, with a type and a name
    - A parameter works like a variable: it has a type and a value, and you can use it in expressions and assignment
    - When you call a method with a particular argument, like 15, the parameter is assigned this value, so within the method's code you can assume the parameter value is "the argument to this method"
    - The body of the `SetLength` method has one statement, which

    assigns the instance variable `length` to the value contained in the parameter `lengthParameter`. In other words, whatever argument `SetLength` is called with will get assigned to `length`
  - This is why it is called a "setter": `SetLength(15)` will set `length` to 15.
- GetLength method, an example of a "getter" method
  - This method will allow code outside the `Rectangle` class to read the current value of a `Rectangle` object's "length" attribute
  - The **return type** of this method is `int`, which means that the value it returns to the calling code is an `int` value
  - Recall that `Console.ReadLine()` returns a `string` value to the caller, which is why you can write `string userInput = Console.ReadLine()`. The `GetLength` method will do the same thing, only with an `int` instead of a `string`
  - This method has no parameters, so you do not provide any arguments when calling it. "Getter" methods never have parameters, since their purpose is to "get" (read) a value, not change anything
  - The body of `GetLength` has one statement, which uses a new keyword: `return`. This keyword declares what will be returned by the method, i.e. what particular value will be given to the caller to use in an expression.
  - In a "getter" method, the value we return is the instance variable that corresponds to the attribute named in the method. `GetLength` returns the `length` instance variable.
- SetWidth method
  - This is another "setter" method, so it looks very similar to `SetLength`
  - It takes one parameter (`widthParameter`) and assigns it to the `width` instance variable
  - Note that the return type of both setters is `void`. The return type `void` means "this method does not return a value." `Console.WriteLine` is an example of a `void` method we've used already.
  - Since the return type is `void`, there is no `return` statement in this method
- GetWidth method
  - This is the "getter" method for the width attribute
  - It looks very similar to `GetLength`, except the instance variable in the `return` statement is `width` rather than `length`
- The ComputeArea method
  - This is *not* a getter or setter: its goal is not to read or write a single instance variable
  - The goal of this method is to compute and return the rectangle's area

    - Since the area of the rectangle will be an int (it is the product of two ints), we declare the return type of the method to be int
    - This method has no parameters, because it does not need any arguments. Its only "input" is the instance variables, and it will always do the same thing every time you call it.
    - The body of the method has a **return** statement with an expression, rather than a single variable
    - When you write **return** [expression], the expression will be evaluated first, then the resulting value will be used by the **return** command
    - In this case, the expression length * width will be evaluated, which computes the area of the rectangle. Since both length and width are ints, the int version of the * operator executes, and it produces an int result. This resulting int is what the method returns.

Using Our Class

- We've written a class, but it does not do anything yet
    - The class is a blueprint for an object, not an object
    - To make it "do something" (i.e. execute some methods), we need to instantiate an object using this class
    - The code that does this should be in a separate file (e.g. Program.cs), not in Rectangle.cs
- Here is a program that uses our Rectangle class:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    Rectangle myRectangle = new Rectangle();
    myRectangle.SetLength(12);
    myRectangle.SetWidth(3);
    int area = myRectangle.ComputeArea();
    Console.WriteLine(
      "Your rectangle's length is "
        + $"{myRectangle.GetLength()}, and its width is "
        + $"{myRectangle.GetWidth()}, so its area is
         ↪  {area}."
    );
  }
}
```

- Instantiating an object
  - The first line of code creates a `Rectangle` object
  - The left side of the = sign is a variable declaration – it declares a variable of type `Rectangle`
    - Classes we write become new data types in C#
  - The right side of the = sign assigns this variable a value: a `Rectangle` object
  - We **instantiate** an object by writing the keyword `new` followed by the name of the class (syntax: `new [class name]()`). The empty parentheses are required, but we will explain why later.
  - This statement is really an initialization statement: It declares and assigns a variable in one line
  - The value of the `myRectangle` variable is the `Rectangle` object that was created by `new Rectangle()`
- Calling setters on the object
  - The next two lines of code call the `SetLength` and `SetWidth` methods on the object
  - Syntax: `[object name].[method name]([argument])`. Note the "dot operator" between the variable name and the method name.
  - `SetLength` is called with an argument of 12, so `lengthParameter` gets the value 12, and the rectangle's `length` instance variable is then assigned this value
  - Similarly, `SetWidth` is called with an argument of 3, so the rectangle's `width` instance variable is assigned the value 3
- Calling ComputeArea
  - The next line calls the `ComputeArea` method and assigns its result to a new variable named `area`
  - The syntax is the same as the other method calls
  - Since this method has a return value, we need to do something with the return value – we assign it to a variable
  - Similar to how you must do something with the result (return value) of `Console.ReadLine()`, i.e. `string userInput = Console.ReadLine()`
- Calling getters on the object
  - The last line of code displays some information about the rectangle object using string interpolation
  - One part of the string interpolation is the `area` variable, which we've seen before
  - The other interpolated values are `myRectangle.GetLength()` and `myRectangle.GetWidth()`
  - Looking at the first one: this will call the `GetLength` method, which has a return value that is an `int`. Instead of storing the return value in an `int` variable, we put it in the string interpolation brackets, which means it will be converted to a string using `ToString`. This means the rectangle's length will be inserted into the string and displayed on the screen

Flow of Control with Objects

- Consider what happens when you have multiple objects in the same program, like this:

```
class Program
{
  static void Main(string[] args)
  {
    Rectangle rect1;
    rect1 = new Rectangle();
    rect1.SetLength(12);
    rect1.SetWidth(3);
    Rectangle rect2 = new Rectangle();
    rect2.SetLength(7);
    rect2.SetWidth(15);
  }
}
```

  - First, we declare a variable of type `Rectangle`
  - Then we assign `rect1` a value, a new `Rectangle` object that we instantiate
  - We call the `SetLength` and `SetWidth` methods using `rect1`, and the `Rectangle` object that `rect1` refers to gets its `length` and `width` instance variables set to 12 and 3
  - Then we create another `Rectangle` object and assign it to the variable `rect2` . This object has its own copy of the `length` and `width` instance variables, not 12 and 3
  - We call the `SetLength` and `SetWidth` methods again, using `rect2` on the left side of the dot instead of `rect1`. This means the `Rectangle` object that `rect2` refers to gets its instance variables set to 7 and 15, while the other `Rectangle` remains unmodified

- The same method code can modify different objects at different times
  - Calling a method transfers control from the current line of code (i.e. in Program.cs) to the method code within the class (Rectangle.cs)
  - The method code is always the same, but the specific object that gets modified can be different each time
  - The variable on the left side of the dot operator determines which object gets modified
  - In `rect1.SetLength(12)`, `rect1` is the **calling object**, so `SetLength` will modify `rect1`
    - `SetLength` begins executing with `lengthParameter` equal to 12

    * The instance variable `length` in `length = lengthParameter` refers to `rect1`'s length
  - In `rect2.SetLength(7)`, `rect2` is the calling object, so `SetLength` will modify `rect2`
    * `SetLength` begins executing with `lengthParameter` equal to 7
    * The instance variable `length` in `length = lengthParameter` refers to `rect2`'s length

Accessing object members

- The "dot operator" that we use to call methods is technically the **member access operator**
- A **member** of an object is either a method or an instance variable
- When we write `objectName.methodName()`, e.g. `rect1.SetLength(12)`, we are using the dot operator to access the "SetLength" member of `rect1`, which is a method; this means we want to call (execute) the `SetLength` method of `rect1`
- We can also use the dot operator to access instance variables, although we usually do not do that because of encapsulation
- If we wrote the `Rectangle` class like this:

```
class Rectangle
{
    public int length;
    public int width;
}
```

  Then we could write a `Main` method that uses the dot operator to access the `length` and `width` instance variables, like this:

```
static void Main(string[] args)
{
    Rectangle rect1 = new Rectangle();
    rect1.length = 12;
    rect1.width = 3;
}
```

  But this code violates encapsulation, so we will not do this.

Method calls in more detail

- Now that we know about the member access operator, we can explain how method calls work a little better

- When we write `rect1.SetLength(12)`, the `SetLength` method is executed with `rect1` as the calling object – we are accessing the `SetLength` member of `rect1` in particular (even though every Rectangle has the same `SetLength` method)
- This means that when the code in `SetLength` uses an instance variable, i.e. `length`, it will automatically access `rect1`'s copy of the instance variable
- You can imagine that the `SetLength` method "changes" to this when you call `rect1.SetLength()`:

  ```
  public void SetLength(int lengthParameter)
  {
      rect1.length = lengthParameter;
  }
  ```

  Note that we use the "dot" (member access) operator on `rect1` to access its `length` instance variable.
- Similarly, you can imagine that the `SetLength` method "changes" to this when you call `rect2.SetLength()`:

  ```
  public void SetLength(int lengthParameter)
  {
      rect2.length = lengthParameter;
  }
  ```
- The calling object is automatically "inserted" before any instance variables in a method
- The keyword **`this`** is an explicit reference to "the calling object"
  - Instead of imagining that the calling object's name is inserted before each instance variable, you could write the `SetLength` method like this:

    ```
    public void SetLength(int lengthParameter)
    {
        this.length = lengthParameter;
    }
    ```
  - This is valid code (unlike our imaginary examples) and will work exactly the same as our previous way of writing `SetLength`
  - When `SetLength` is called with `rect1.SetLength(12)`, **`this`** becomes equal to `rect1`, just like `lengthParameter` becomes equal to 12
  - When `SetLength` is called with `rect2.SetLength(7)`, **`this`** becomes equal to `rect2` and `lengthParameter` becomes equal to 7

Methods and instance variables

- Using a variable in an expression means *reading* its value
- A variable only changes when it is on the left side of an assignment statement; this is *writing* to the variable
- A method that uses instance variables in an expression, but does not assign to them, will not modify the object
- For example, consider the `ComputeArea` method:

  ```
  public int ComputeArea()
  {
      return length * width;
  }
  ```

  It reads the current values of `length` and `width` to compute their product, but the product is returned to the method's caller. The instance variables are not changed.

- After executing the following code:

  ```
  Rectangle rect1 = new Rectangle();
  rect1.SetLength(12);
  rect1.SetWidth(3);
  int area = rect1.ComputeArea();
  ```

  `rect1` has a `length` of 12 and a `width` of 3. The call to `rect1.ComputeArea()` computes $12 \cdot 3 = 36$, and the `area` variable is assigned this return value, but it does not change `rect1`.

Methods and return values

- Recall the basic structure of a program: receive input, compute something, produce output
- A method has the same structure: it *receives input* from its parameters, *computes* by executing the statements in its body, then *produces output* by returning a value
  - For example, consider this method defined in the Rectangle class:

    ```
    public int LengthProduct(int factor)
    {
        return length * factor;
    }
    ```

    Its input is the parameter `factor`, which is an `int`. In the method body, it computes the product of the rectangle's length and `factor`. The method's output is the resulting product.

- The **`return`** statement specifies the output of the method: a variable, expression, etc. that produces some value
- A method call can be used in other code as if it were a value. The "value" of a method call is the method's return value.
  - In previous examples, we wrote `int area = rect1.ComputeArea();`, which assigns a variable (`area`) a value (the return value of `ComputeArea()`)
  - The `LengthProduct` method can be used like this:

    ```
    Rectangle rect1 = new Rectangle();
    rect1.SetLength(12);
    int result = rect1.LengthProduct(2) + 1;
    ```

    When executing the third line of code, the computer first executes the `LengthProduct` method with argument (input) 2, which computes the product $12 \cdot 2 = 24$. Then it uses the return value of `LengthProduct`, which is 24, to evaluate the expression `rect1.LengthProduct(2) + 1`, producing a result of 25. Finally, it assigns the value 25 to the variable `result`.
- When writing a method that returns a value, the value in the **`return`** statement **must** be the same type as the method's return type
  - If the value returned by `LengthProduct` is not an `int`, we will get a compile error
  - This will not work:

    ```
    public int LengthProduct(double factor)
    {
        return length * factor;
    }
    ```

    Now that `factor` has type `double`, the expression `length * factor` will need to implicitly convert `length` from `int` to `double` in order to make the types match. Then the product will also be a `double`, so the return value does not match the return type (`int`).
  - We could fix it by either changing the return type of the method to `double`, or adding a cast to `int` to the product so that the return value is still an `int`
- Not all methods return a value, but all methods must have a return type
  - The return type `void` means "nothing is returned"

- If your method does not return a value, its return type *must* be `void`. If the return type is not `void`, the method *must* return a value.
- This will cause a compile error because the method has a return type of `int` but no return statement:

  ```
  public int SetLength(int lengthP)
  {
      length = lengthP;
  }
  ```

- This will cause a compile error because the method has a return type of `void`, but it attempts to return something anyway:

  ```
  public void GetLength()
  {
      return length;
  }
  ```

Introduction to UML

- UML is a specification language for software
  - UML: Unified Modeling Language
  - Describes design and structure of a program with graphics
  - Does not include "implementation details," such as code statements
  - Can be used for any programming language, not just C#
  - Used in planning/design phase of software creation, before you start writing code
  - Process: Determine program requirements $\rightarrow$ Make UML diagrams $\rightarrow$ Write code based on UML $\rightarrow$ Test and debug program
- UML Class Diagram elements
  - Top box: Class name, centered
  - Middle box: Attributes (i.e. instance variables)
    - On each line, one attribute, with its name and type
    - Syntax: `[+/-] [name]: [type]`
    - Note this is the opposite order from C# variable declaration: type comes after name
    - Minus sign at beginning of line indicates "private member"
  - Bottom box: Operations (i.e. methods)
    - On each line, one method header, including name, parameters, and return type
    - Syntax: `[+/-] [name]([parameter name]: [parameter type]): [return type`

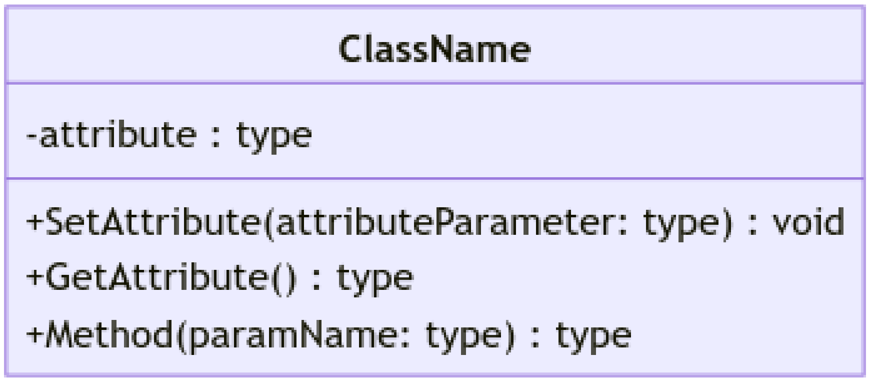

ClassName
-attribute : type
+SetAttribute(attributeParameter: type) : void
+GetAttribute() : type
+Method(paramName: type) : type

Figure 14: A UML diagram for the ClassName class (text version[328])

    * Also backwards compared to C# order: parameter types come after parameter names, and return type comes after method name instead of before it
    * Plus sign at beginning of line indicates "public", which is what we want for methods

- UML Diagram for the Rectangle class
  - Note that when the return type of a method is `void`, we can omit it in UML
  - In general, attributes will be private (- sign) and methods will be public (+ sign), so you can expect most of your classes to follow this pattern (-s in the upper box, +s in the lower box)
  - Note that there is no code or "implementation" described here: it does not say that `ComputeArea` will multiply `length` by `width`
- Writing code based on a UML diagram
  - Each diagram is one class, everything within the box is between the class's header and its closing brace
  - For each attribute in the attributes section, write an instance variable of the right name and type
    * See "- width: int", write `private int width;`
    * Remember to reverse the order of name and type
  - For each method in the methods section, write a method header with the matching return type, name, and parameters
    * Parameter declarations are like the instance variables: in UML they have a name followed by a type, in C# you write the type name first
  - Now the method bodies need to be filled in - UML just defined

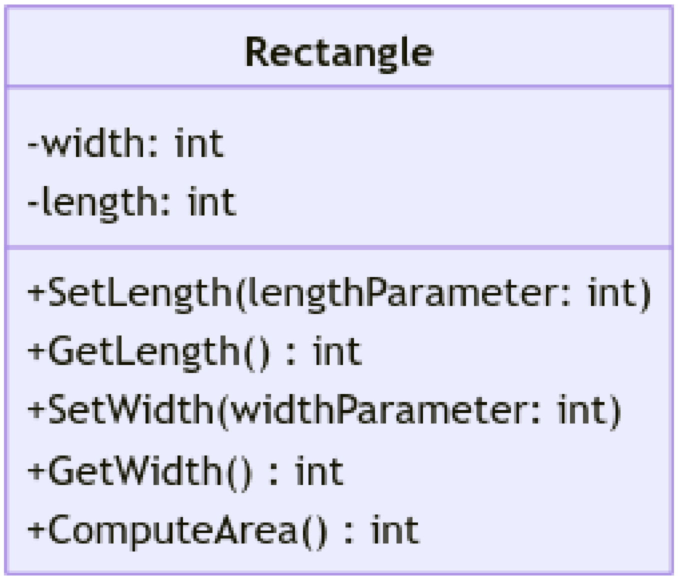


Figure 15: A UML diagram for the Rectangle class (text version[329])

the interface, now you need to write the implementation

Variable Scope

Instance variables vs. local variables

- Instance variables: Stored (in memory) with the object, shared by all methods of the object. Changes made within a method persist after method finishes executing.
- Local variables: Visible to only one method, not shared. Disappear after method finishes executing. Variables we've created before in the `Main` method (they are local to the `Main` method!).
- Example: In class Rectangle, we have these two methods:

```
public void SwapDimensions()
{
    int temp = length;
    length = width;
    width = temp;
}
public int GetLength()
{
    return length;
}
```

- temp is a local variable within SwapDimensions, while length and width are instance variables
- The GetLength method cannot use temp; it is visible only to SwapDimensions
- When SwapDimensions changes length, that change is persistent – it will still be different when GetLength executes, and the next call to GetLength after SwapDimensions will return the new length
- When SwapDimensions assigns a value to temp, it only has that value within the current call to SwapDimensions – after SwapDimensions finishes, temp disappears, and the next call to SwapDimensions creates a new temp

Definition of scope

- Variables exist only in limited **time** and **space** within the program
- Outside those limits, the variable cannot be accessed – e.g. local variables cannot be accessed outside their method
- Scope of a variable: The region of the program where it is accessible/visible
  - A variable is "in scope" when it is accessible
  - A variable is "out of scope" when it does not exist or cannot be accessed
- Time limits to scope: Scope begins *after* the variable has been declared
  - This is why you cannot use a variable before declaring it
- Space limits to scope: Scope is within the same *code block* where the variable is declared
  - Code blocks are defined by curly braces: everything between matching { and } is in the same code block
  - Instance variables are declared in the class's code block (they are inside `class Rectangle`'s body, but not inside anything else), so their scope extends to the entire class
  - Code blocks nest: A method's code block is inside the class's code block, so instance variables are also in scope within each method's code block
  - Local variables are declared inside a method's code block, so their scope is limited to that single method

- The scope of a parameter (which is a variable) is the method's code block - it is the same as a local variable for that method
- Scope example:

```
public void SwapDimensions()
{
    int temp = length;
    length = width;
    width = temp;
}
public void SetWidth(int widthParam)
{
    int temp = width;
    width = widthParam;
}
```

  - The two variables named `temp` have different scopes: One has a scope limited to the `SwapDimensions` method's body, while the other has a scope limited to the `SetWidth` method's body
  - This is why they can have the same name: variable names must be unique *within the variable's scope*. You can have two variables with the same name if they are in different scopes.
  - The scope of instance variables `length` and `width` is the body of class `Rectangle`, so they are in scope for both of these methods

Variables with overlapping scopes

- This code is legal (compiles) but does not do what you want:

```
class Rectangle
{
    private int length;
    private int width;
    public void UpdateWidth(int newWidth)
    {
        int width = 5;
        width = newWidth;
    }
}
```

- The instance variable `width` and the local variable `width` have different scopes, so they can have the same name
- But the instance variable's scope (the class `Rectangle`) *overlaps* with the local variable's scope (the method `UpdateWidth`)

- If two variables have the same name and overlapping scopes, the variable with the *closer* or *smaller* scope **shadows** the variable with the *farther* or *wider* scope: the name will refer *only* to the variable with the smaller scope
- In this case, that means `width` inside `UpdateWidth` refers only to the local variable named `width`, whose scope is smaller because it is limited to the `UpdateWidth` method. The line `width = newWidth` actually changes the local variable, not the instance variable named `width`.
- Since instance variables have a large scope (the whole class), they will always get shadowed by variables declared within methods
- You can prevent shadowing by using the keyword **`this`**, like this:

  ```
  class Rectangle
  {
      private int length;
      private int width;
      public void UpdateWidth(int newWidth)
      {
          int width = 5;
          this.width = newWidth;
      }
  }
  ```

  Since **`this`** means "the calling object", **`this`**`.width` means "access the `width` member of the calling object." This can only mean the instance variable `width`, not the local variable with the same name
- Incidentally, you can also use **`this`** to give your parameters the same name as the instance variables they are modifying:

  ```
  class Rectangle
  {
      private int length;
      private int width;
      public void SetWidth(int width)
      {
          this.width = width;
      }
  }
  ```

  Without **`this`**, the body of the `SetWidth` method would be `width = width;`, which does not do anything (it would assign the parameter `width` to itself).

Constants

- Classes can also contain constants
- Syntax: `[public/private] const [type] [name] = [value];`
- This is a named value that never changes during program execution
- Safe to make it `public` because it cannot change – no risk of violating encapsulation
- Can only be built-in types (`int`, `double`, etc.), not objects
- Can make your program more readable by giving names to "magic numbers" that have some significance
- Convention: constants have names in ALL CAPS
- Example:

  ```
  class Calendar
  {
      public const int MONTHS = 12;
      private int currentMonth;
      //...
  }
  ```

  The value "12" has a special meaning here, i.e. the number of months in a year, so we use a constant to name it.

- Constants are accessed using the name of the class, not the name of an object – they are the same for every object of that class. For example:

  ```
  Calendar myCal = new Calendar();
  decimal yearlyPrice = 2000.0m;
  decimal monthlyPrice = yearlyPrice / Calendar.MONTHS;
  ```

Reference Types: More Details

- Data types in C# are either value types or reference types
  - This difference was introduced in an earlier lecture (Datatypes and Variables)
  - For a **value type** variable (`int`, `long`, `float`, `double`, `decimal`, `char`, `bool`) the named memory location stores the exact data value held by the variable
  - For a **reference type** variable, such as `string`, the named memory location stores a *reference to the value*, not the value itself

  - All objects you create from your own classes, like `Rectangle`, are reference types
- Object variables are references
  - When you have a variable for a reference type, or "reference variable," you need to be careful with the assignment operation
  - Consider this code:
    ```
    using System;

    class Program
    {
      static void Main(string[] args)
      {
        Rectangle rect1 = new Rectangle();
        rect1.SetLength(8);
        rect1.SetWidth(10);
        Rectangle rect2 = rect1;
        rect2.SetLength(4);
        Console.WriteLine(
          $"Rectangle 1: {rect1.GetLength()} "
            + $"by {rect1.GetWidth()}"
        );
        Console.WriteLine(
          $"Rectangle 2: {rect2.GetLength()} "
            + $"by {rect2.GetWidth()}"
        );
      }
    }
    ```
  - The output is:
    ```
    Rectangle 1: 4 by 10
    Rectangle 2: 4 by 10
    ```
  - The variables `rect1` and `rect2` actually refer to the same `Rectangle` object, so `rect2.SetLength(4)` seems to change the length of "both" rectangles
  - The assignment operator copies the contents of the variable, but a reference variable contains a *reference* to an object – so that's what gets copied (in `Rectangle rect2 = rect1`), not the object itself
  - In more detail:
    * `Rectangle rect1 = new Rectangle()` creates a new Rectangle object somewhere in memory, then creates a reference variable named `rect1` somewhere else in memory. The variable named `rect1` is initialized with the memory address of the Rectangle object, i.e. a reference to the object
    * `rect1.SetLength(8)` reads the address of the Rectangle

object from the `rect1` variable, finds the object in memory, and executes the `SetLength` method on that object (changing its length to 8)
    * `rect1.SetWidth(10)` does the same thing, finds the same object, and sets its width to 10
    * `Rectangle rect2 = rect1` creates a reference variable named `rect2` in memory, but does not create a new Rectangle object. Instead, it initializes `rect2` with the same memory address that is stored in `rect1`, referring to the same Rectangle object
    * `rect2.SetLength(4)` reads the address of a Rectangle object from the `rect2` variable, finds that object in memory, and sets its length to 4 – but this is the exact same Rectangle object that `rect1` refers to
- Reference types can also appear in method parameters
  - When you call a method, you provide an argument (a value) for each parameter in the method's declaration
  - Since the parameter is really a variable, the computer will then assign the argument to the parameter, just like variable assignment
    * For example, when you write `rect1.SetLength(8)`, there is an implicit assignment `lengthParameter = 8` that gets executed before executing the body of the `SetLength` method
  - This means if the parameter is a reference type (like an object), the parameter will get a copy of the reference, not a copy of the object
  - When you use the parameter to modify the object, you will modify the same object that the caller provided as an argument
  - This means objects can change other objects!
  - For example, imagine we added this method to the Rectangle class:
    ```
    public void CopyToOther(Rectangle otherRect)
    {
        otherRect.SetLength(length);
        otherRect.SetWidth(width);
    }
    ```
    It uses the `SetLength` and `SetWidth` methods to modify its parameter, `otherRect`. Specifically, it sets the parameter's length and width to its own length and width.
  - The `Main` method of a program could do something like this:
    ```
    Rectangle rect1 = new Rectangle();
    Rectangle rect2 = new Rectangle();
    rect1.SetLength(8);
    rect1.SetWidth(10);
    ```

```
rect1.CopyToOther(rect2);
Console.WriteLine($"Rectangle 2:
↪  {rect2.GetLength()} "
   + $"by {rect2.GetWidth()}");
```

  * First it creates two different `Rectangle` objects (note the two calls to `new`), then it sets the length and width of one object, using `rect1.SetLength` and `rect1.SetWidth`
  * Then it calls the `CopyToOther` method with an argument of `rect2`. This transfers control to the method and (implicitly) makes the assignment `otherRect = rect2`
  * Since `otherRect` and `rect2` are now reference variables referring to the same object, the calls to `otherRect.SetLength` and `otherRect.SetWidth` within the method will modify that object
  * After the call to `CopyToOther`, the object referred to by `rect2` has a length of 8 and a width of 10, even though we never called `rect2.SetLength` or `rect2.SetWidth`

More Advanced Object Concepts

Default Values and the ClassRoom Class

- In lab, you were asked to execute a program like this:

  ```
  using System;

  class Program
  {
    static void Main(string[] args)
    {
      Rectangle myRect = new Rectangle();
      Console.WriteLine($"Length is
  ↪   {myRect.GetLength()}");
      Console.WriteLine($"Width is
  ↪   {myRect.GetWidth()}");
    }
  }
  ```

  Note that we create a Rectangle object, but do not use the `SetLength` or `SetWidth` methods to assign values to its instance variables. It displays the following output:

  ```
  Length is 0
  Width is 0
  ```

- This works because the instance variables `length` and `width` have a default value of 0, even if you never assign them a value

- Local variables, like the ones we write in the `Main` method, do *not* have default values. You must assign them a value before using them in an expression.
  - For example, this code will produce a compile error:

    ```
    int myVar1;
    int myVar2 = myVar1 + 5;
    ```

    You cannot assume `myVar1` will be 0; it has no value at all until you use an assignment statement.
- When you create (instantiate) a new object, its instance variables will be assigned specific default values based on their type:

Type	**Default Value**
Numeric types	0
`string`	`null`
objects	`null`
`bool`	`false`
`char`	`'\0'`

- Remember, `null` is the value of a reference-type variable that refers to "nothing" - it does not contain the location of any object at all. You cannot do anything with a reference variable containing `null`.

A class we will use for subsequent examples

- ClassRoom: Represents a room in a building on campus
- UML Diagram:
  - There are two attributes: the name of the building (a string) and the room number (an `int`)
  - Each attribute will have a "getter" and "setter" method
- Implementation:

  ```
  class ClassRoom
  {
    private string building;
    private int number;

    public void SetBuilding(string buildingParam)
    {
      building = buildingParam;
    }
  ```

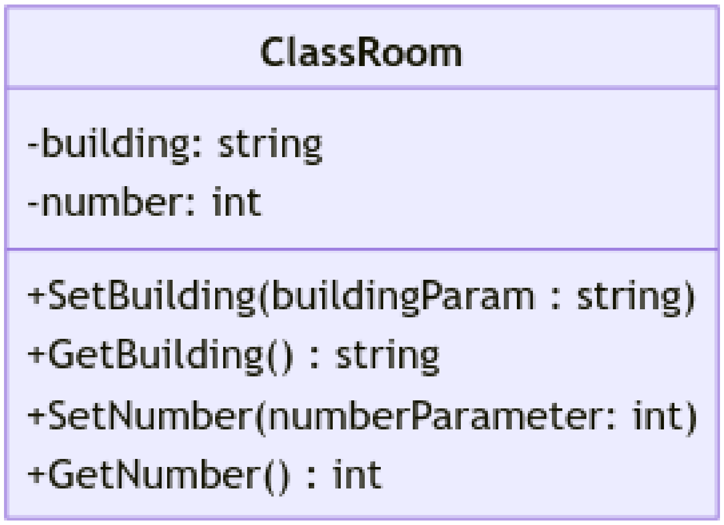


Figure 16: A UML diagram for the ClassRoom class (text version[330])

```
    public string GetBuilding()
    {
      return building;
    }

    public void SetNumber(int numberParam)
    {
      number = numberParam;
    }

    public int GetNumber()
    {
      return number;
    }
  }
```

- Each attribute is implemented by an instance variable with the same name
- To write the "setter" for the building attribute, we write a method whose return type is `void`, with a single `string`-type parameter. Its body assigns the `building` instance variable to the value in the parameter `buildingParam`
- To write the "getter" for the building attribute, we write a method whose return type is `string`, and whose body returns

the instance variable `building`

- Creating an object and using its default values:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    ClassRoom english = new ClassRoom();
    Console.WriteLine(
      $"Building is {english.GetBuilding()}"
    );
    Console.WriteLine(
      $"Room number is {english.GetNumber()}"
    );
  }
}
```

  This will print the following output:

```
Building is
Room number is 0
```

  Remember that the default value of a `string` variable is `null`. When you use string interpolation on `null`, you get an empty string.

Constructors

- Instantiation syntax requires you to write parentheses after the name of the class, like this:

```
ClassRoom english = new ClassRoom();
```

- Parentheses indicate a method call, like in `Console.ReadLine()` or `english.GetBuilding()`
- In fact, the instantiation statement `new ClassRoom()` does call a method: the **constructor**
- Constructor: A special method used to create an object. It "sets up" a new instance by **initializing its instance variables**.
- If you do not write a constructor in your class, C# will generate a "default" constructor for you – this is what's getting called when we write `new ClassRoom()` here
- The default constructor initializes each instance variable to its default value – that's where default values come from

Writing a constructor

- Example for ClassRoom:

```
public ClassRoom(string buildingParam, int
 ↪  numberParam)
{
    building = buildingParam;
    number = numberParam;
}
```

- To write a constructor, write a method whose name is *exactly the same* as the class name
- This method has *no return type*, not even `void`. It does not have a `return` statement either
- For `ClassRoom`, this means the constructor's header starts with `public ClassRoom`
  - You can think of this method as "combining" the return type and name. The name of the method is `ClassRoom`, and its output is of type `ClassRoom`, since the return value of `new ClassRoom()` is always a `ClassRoom` object
  - You do not actually write a `return` statement, though, because `new` will always return the new object after calling the constructor
- A custom constructor usually has parameters that correspond to the instance variables: for `ClassRoom`, it has a `string` parameter named `buildingParam`, and an `int` parameter named `numberParam`
  - Note that when we write a method with two parameters, we separate the parameters with a comma
- The body of a constructor must assign values to **all** instance variables in the object
- Usually this means assigning each parameter to its corresponding instance variable: initialize the instance variable to equal the parameter
  - Very similar to calling both "setters" at once
- Using a constructor
- An instantiation statement will call a constructor for the class being instantiated
- Arguments in parentheses must match the parameters of the constructor

- Example with the `ClassRoom` constructor:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    ClassRoom csci = new ClassRoom("Allgood East",
 ↪  356);
    Console.WriteLine($"Building is
 ↪  {csci.GetBuilding()}");
    Console.WriteLine($"Room number is
 ↪  {csci.GetNumber()}");
  }
}
```

  This program will produce this output:

```
Building is Allgood East
Room number is 356
```

- The instantiation statement `new ClassRoom("Allgood East", 356)` first creates a new "empty" object of type `ClassRoom`, then calls the constructor to initialize it. The first argument, "Allgood East", becomes the constructor's first parameter (`buildingParam`), and the second argument, 356, becomes the constructor's second parameter (`numberParam`).
- After executing the instantiation statement, the object referred to by `csci` has its instance variables set to these values, even though we never called `SetBuilding` or `SetNumber`

Methods with multiple parameters

- The constructor we wrote is an example of a method with two parameters
- The same syntax can be used for ordinary, non-constructor methods, if we need more than one input value
- For example, we could write this method in the `Rectangle` class:

```
public void MultiplyBoth(int lengthFactor, int
 ↪  widthFactor)
{
    length *= lengthFactor;
    width *= widthFactor;
}
```

- The first parameter has type `int` and is named lengthFactor. The second parameter has type `int` and is named `widthFactor`
- You can call this method by providing two arguments, separated by a comma:

  ```
  Rectangle myRect = new Rectangle();
  myRect.SetLength(5);
  myRect.SetWidth(10);
  myRect.MultiplyBoth(3, 5);
  ```

  The first argument, 3, will be assigned to the first parameter, `lengthFactor`. The second argument, 5, will be assigned to the second parameter, `widthFactor`
- The order of the arguments matters when calling a multi-parameter method. If you write `myRect.MultiplyBoth(5, 3)`, then `lengthFactor` will be 5 and `widthFactor` will be 3.
- The type of each argument must match the type of the corresponding parameter. For example, when you call the `ClassRoom` constructor we just wrote, the first argument must be a `string` and the second argument must be an `int`

Writing multiple constructors

- Remember that if you do not write a constructor, C# generates a "default" one with no parameters, so you can write `new ClassRoom()`
- Once you add a constructor to your class, C# will **not** generate a default constructor
  - This means once we write the `ClassRoom` constructor (as shown earlier), this statement will produce a compile error: `ClassRoom english = new ClassRoom();`
  - The constructor we wrote has 2 parameters, so now you always need 2 arguments to instantiate a `ClassRoom`
- If you still want the option to create an object with no arguments (i.e. `new ClassRoom()`), you must write a constructor with no parameters
- A class can have more than one constructor, so it would look like this:

  ```
  class ClassRoom
  {
      //...
      public ClassRoom(string buildingParam, int
       ↪  numberParam)
  ```

```
    {
        building = buildingParam;
        number = numberParam;
    }
    public ClassRoom()
    {
        building = null;
        number = 0;
    }
    //...
}
```

- The "no-argument" constructor must still initialize all the instance variables, even though it has no parameters
  - You can pick any "default value" you want, or use the same ones that C# would use (0 for numeric variables, **`null`** for object variables, etc.)
- When a class has multiple constructors, the instantiation statement must decide which constructor to call
- The instantiation statement will call the constructor whose parameters match the arguments you provide
  - For example, each of these statements will call a different constructor:

    ```
    ClassRoom csci = new ClassRoom("Allgood East",
    ↪   356);
    ClassRoom english = new ClassRoom();
    ```

    The first statement calls the two-parameter constructor we wrote, since it has a `string` argument and an `int` argument (in that order), and those match the parameters `(string buildingParam, int numberParam)`. The second statement calls the zero-parameter constructor since it has no arguments.
  - If the arguments do not match any constructor, it is still an error:

    ```
    ClassRoom csci = new ClassRoom(356, "Allgood
    ↪   East");
    ```

    This will produce a compile error, because the instantiation statement has two arguments in the order `int`, `string`, but the only constructor with two parameters needs the first parameter to be a `string`.

Writing ToString Methods

- ToString recap
  - String interpolation automatically calls the ToString method on each variable or value
  - ToString returns a string "equivalent" to the object; for example, if num is an int variable containing 42, num.ToString() returns "42".
  - C# datatypes already have a ToString method, but you need to write a ToString method for your own classes to use them in string interpolation
- Writing a ToString method
  - To add a ToString method to your class, you must write this header: **public override** string ToString()
  - The access modifier must be **public** (so other code, like string interpolation, can call it)
  - The return type must be string (ToString must output a string)
  - It must have no parameters (the string interpolation code will not know what arguments to supply)
  - The keyword **override** means your class is "overriding," or providing its own version of, a method that is already defined elsewhere – ToString is defined by the base object type, which is why string interpolation "knows" it can call ToString on any object
    * If you do not use the keyword **override**, then the pre-existing ToString method (defined by the base object type) will be used instead, which only returns the name of the class
  - The goal of ToString is to return a "string representation" of the object, so the body of the method should use all of the object's attributes and combine them into a string somehow
  - Example ToString method for ClassRoom:

```
public override string ToString()
{
    return building + " " + number;
}
```

    * There are two instance variables, building and number, and we use both of them
    * A natural way to write the name of a classroom is the building name followed by the room number, like "University Hall 124", so we concatenate the variables in that order
    * Note that we add a space between the variables
    * Note that building is already a string, but number is an int, so string concatenation will implicitly call number.ToString() – ToString methods can call other ToString methods

        * Another way to write the body would be `return $"{building} {number}";`
- Using a `ToString` method
    - Any time an object is used in string interpolation or concatenation, its `ToString` method will be called
    - You can also call `ToString` by name using the "dot operator," like any other method
    - This code will call the `ToString` method we just wrote for `ClassRoom`:

```
ClassRoom csci = new ClassRoom("Allgood East",
 ↪  356);
Console.WriteLine(csci);
Console.WriteLine($"The classroom is {csci}");
Console.WriteLine("The classroom is " +
 ↪  csci.ToString());
```

Method Signatures and Overloading

Name uniqueness in C#

- In general, variables, methods, and classes must have unique names, but there are several exceptions
- **Variables** can have the same name if they are in *different scopes*
    - Two methods can each have a local variable with the same name
    - A local variable (scope limited to the method) can have the same name as an instance variable (scope includes the whole class), but this will result in **shadowing**
- **Classes** can have the same name if they are in *different namespaces*
    - This is one reason C# has namespaces: you can name your classes anything you want. Otherwise, if a library (someone else's code) used a class name, you would be prevented from using that name
    - For example, imagine you were using a "shapes library" that provided a class named `Rectangle`, but you also wanted to write your own class named `Rectangle`
    - The library's code would use its own namespace, like this:

```
namespace ShapesLibrary
{
    class Rectangle
    {
        //instance variables, methods, etc.
    }
}
```

Then your own code could have a `Rectangle` class in your own namespace:

```
namespace MyProject
{
    class Rectangle
    {
        //instance variables, methods, etc.
    }
}
```

  - You can use both `Rectangle` classes in the same code, as long as you specify the namespace, like this:

    ```
    MyProject.Rectangle rect1 = new
     ↪  MyProject.Rectangle();
    ShapesLibrary.Rectangle rect2 = new
     ↪  ShapesLibrary.Rectangle();
    ```

- **Methods** can have the same name if they have *different signatures*; this is called **overloading**
  - We'll explain signatures in more detail in a minute
  - Briefly, methods can have the same name if they have different parameters
  - For example, you can have two methods named Multiply in the Rectangle class, as long as one has one parameter and the other has two parameters:

    ```
    public void Multiply(int factor)
    {
        length *= factor;
        width *= factor;
    }
    public void Multiply(int lengthFactor, int
     ↪  widthFactor)
    {
        length *= lengthFactor;
        width *= widthFactor;
    }
    ```

    C# understands that these are different methods, even though they have the same name, because their parameters are different. If you write `myRect.Multiply(2)` it can only mean the first "Multiply" method, not the second one, because there is only one argument.
  - We have used overloading already when we wrote multiple constructors – constructors are methods too. For example, these two constructors have the same name, but different parameters:

    ```
    public ClassRoom(string buildingParam, int
     ↪  numberParam)
    {
        building = buildingParam;
        number = numberParam;
    ```

```
}
public ClassRoom()
{
    building = null;
    number = 0;
}
```

Method signatures

- A method's **signature** has 3 components: its **name**, the **type** of each parameter, and the **order** the parameters appear in
- Methods are unique if their *signatures* are unique, which is why they can have the same name
- Signature examples:
  - `public void Multiply(int lengthFactor, int widthFactor)` – the signature is `Multiply(int, int)` (name is `Multiply`, parameters are `int` and `int` type)
  - `public void Multiply(int factor)` – signature is `Multiply(int)`
  - `public void Multiply(double factor)` – signature is `Multiply(double)`
  - These could all be in the same class since they all have different signatures
- Parameter *names* are not part of the signature, just their types
  - Note that the parameter names are omitted when I write down the signature
  - That means these two methods are not unique and could not be in the same class:
    ```
    public void SetWidth(int widthInMeters)
    {
        //...
    }
    public void SetWidth(int widthInFeet)
    {
        //...
    }
    ```
    Both have the same signature, `SetWidth(int)`, even though the parameters have different names. You might intend the parameters to be different (i.e. represent feet vs. meters), but any `int`-type parameter is the same to C#
- The method's return type is not part of the signature
  - So far all the examples have the same return type (`void`), but changing it would not change the signature
  - The signature of `public int Multiply(int factor)` is `Multiply(int)`, which is the same as `public void Multiply(int factor)`

- The signature "begins" with the name of the method; everything "before" that does not count (i.e. `public`, `int`)

- The order of parameters is part of the signature, as long as the types are different
  - Since parameter name is not part of the signature, only the type can determine the order
  - These two methods have different signatures:
    ```
    public int Update(int number, string name)
    {
        //...
    }
    public int Update(string name, int number)
    {
        //..
    }
    ```
    The signature of the first method is `Update(int, string)`. The signature of the second method is `Update(string, int)`.
  - These two methods have the same signature, and could not be in the same class:
    ```
    public void Multiply(int lengthFactor, int
     ↪   widthFactor)
    {
        //...
    }
    public void Multiply(int widthFactor, int
     ↪   lengthFactor)
    {
        //...
    }
    ```
    The signature for both methods is `Multiply(int, int)`, even though we switched the order of the parameters – the name does not count, and they are both `int` type
- Constructors have signatures too
  - The constructor `ClassRoom(string buildingParam, int numberParam)` has the signature `ClassRoom(string, int)`
  - The constructor `ClassRoom()` has the signature `ClassRoom()`
  - Constructors all have the same name, but they are unique if their signatures (parameters) are different

Calling overloaded methods

- Previously, when you used the dot operator and wrote the name of a method, the name was enough to determine which method to execute – `myRect.GetLength()` would call the `GetLength` method
- When a method is overloaded, you must use the entire signature to

determine which method gets executed
- A method call has a "signature" too: the name of the method, and the type and order of the arguments
- C# will execute the method whose signature matches the signature of the method call
- Example: `myRect.Multiply(4);` has the signature `Multiply(int)`, so C# will look for a method in the Rectangle class that has the signature `Multiply(int)`. This matches the method `public void Multiply(int factor)`
- Example: `myRect.Multiply(3, 5);` has the signature `Multiply(int, int)`, so C# will look for a method with that signature in the Rectangle class. This matches the method `public void Multiply(int lengthFactor, int widthFac`
- The same process happens when you instantiate a class with multiple constructors: C# calls the constructor whose signature matches the signature of the instantiation
- If no method or constructor matches the signature of the method call, you get a compile error. You still cannot write `myRect.Multiply(1.5)` if there is no method whose signature is `Multiply(double)`.

Constructors in UML

- Now that we can write constructors, they should be part of the UML diagram of a class
  - No need to include the default constructor, or one you write yourself that takes no arguments
  - Non-default constructors go in the operations section (box 3) of the UML diagram
  - Similar syntax to a method: `[+/-] <<constructor>> [name]([parameter name]: [par`
  - Note that the name will always match the class name
  - No return type, ever
  - Annotation "«constructor»" is nice, but not necessary: if the method name matches the class name, it is a constructor
- Example for ClassRoom:

Properties

Introduction

- Attributes are implemented with a standard "template" of code
- Remember, "attribute" is the abstract concept of some data stored in an object; "instance variable" is the way that data is actually stored

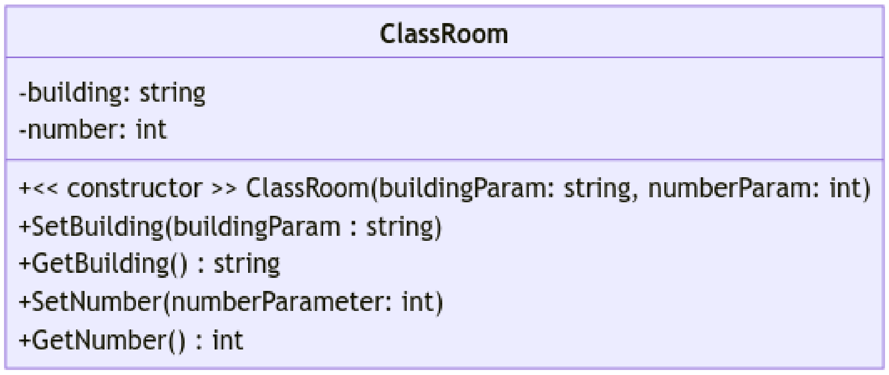

ClassRoom
-building: string
-number: int
+<< constructor >> ClassRoom(buildingParam: string, numberParam: int)
+SetBuilding(buildingParam : string)
+GetBuilding() : string
+SetNumber(numberParameter: int)
+GetNumber() : int

Figure 17: A UML diagram for the ClassRoom class (text version[331])

- First, declare an instance variable for the attribute
- Then write a "getter" method for the instance variable
- Then write a "setter" method for the instance variable
- With this combination of instance variable and methods, the object has an attribute that can be read (with the getter) and written (with the setter)
- For example, this code implements a "width" attribute for the class Rectangle:

```
class Rectangle
{
    private int width;
    public void SetWidth(int value)
    {
        width = value;
    }
    public int GetWidth()
    {
        return width;
    }
}
```

- Note that there is a lot of repetitive or "obvious" code here:
  - The name of the attribute is intended to be "width," so you must name the instance variable `width`, and the methods `GetWidth` and `SetWidth`, repeating the name three times.
  - The attribute is intended to be type `int`, so you must ensure that the instance variable is type `int`, the getter has a return

    type of `int`, and the setter has a parameter type of `int`. Similarly, this repeats the data type three times.
  - You need to come up with a name for the setter's parameter, even though it also represents the width (i.e. the new value you want to assign to the width attribute). We usually end up naming it "widthParameter" or "widthParam" or "newWidth" or "newValue."

- Properties are a "shorthand" way of writing this code: They implement an attribute with less repetition.

- Note that properties are not present in every object-oriented programming language: for example, Java does not have properties[332].

Writing properties

- Declare an instance variable for the attribute, like before

- A **property declaration** has 3 parts:

  - Header, which gives the property a name and type (very similar to variable declaration)
  - `get` accessor, which declares the "getter" method for the property
  - `set` accessor, which declares the "setter" method for the property

- Example code, implementing the "width" attribute for Rectangle (this replaces the code in the previous example):

```
class Rectangle
{
    private int width;
    public int Width
    {
        get
        {
            return width;
        }
        set
        {
            width = value;
        }
    }
}
```

[332]https://stackoverflow.com/questions/2701077/does-java-have-properties-that-work-the-same-way-properties-work-in-c

- Header syntax: `[public/private] [type] [name]`
- *Convention* (not rule) is to give the property the same name as the instance variable, but capitalized – C# is case sensitive
- `get` accessor: Starts with the keyword `get`, then a method body inside a code block (between braces)
  - `get` is like a method header that always has the same name, and its other features are implied by the property's header
  - Access modifier: Same as the property header's, i.e. `public` in this example
  - Return type: Same as the property header's type, i.e. `int` in this example (so imagine it says `public int get()`)
  - Body of `get` section is exactly the same as body of a "getter": return the instance variable
- `set` accessor: Starts with the keyword `set`, then a method body inside a code block
  - Also a method header with a fixed name, access modifier, return type, and parameter
  - Access modifier: Same as the property header's, i.e. `public` in this example
  - Return type: Always `void` (like a setter)
  - Parameter: Same type as the property header's type, name is always "value". In this case that means the parameter is `int value`; imagine the method header says `public void set(int value)`
  - Body of `set` section looks just like the body of a setter: Assign the parameter to the instance variable (and the parameter is always named "value"). In this case, that means `width = value`

Using properties

- Properties are members of an object, just like instance variables and methods
- Access them with the "member access" operator, aka the dot operator
  - For example, `myRect.Width` will access the property we wrote, assuming `myRect` is a Rectangle
- A complete example (available as a complete solution[333]), where the "length" and "width" attributes are implemented with properties:

[333]https:/princomp.github.io/code/projects/Properties_Example.zip

```
class Rectangle
{
  private int width;
  public int Width
  {
    get { return width; }
    set { width = value; }
  }
  private int length;
  public int Length
  {
    get { return length; }
    set { length = value; }
  }
}
```

- Properties "act like" variables: you can assign to them and read from them
- Reading from a property will *automatically* call the `get` accessor for that property
  - For example,

  ```
  Console.WriteLine(
      $"The width is {myRectangle.Width}");
  ```

  will call the `get` accessor inside the `Width` property, which in turn executes `return width` and returns the current value of the instance variable
  - This is equivalent to

  ```
  Console.WriteLine(
      $"The width is {myRectangle.GetWidth()}");
  ```

  using the "old" Rectangle code
- Assigning to (writing) a property will *automatically* call the `set` accessor for that property, with an argument equal to the right side of the = operator
  - For example, `myRectangle.Width = 15;` will call the `set` accessor inside the `Width` property, with `value` equal to 15
  - This is equivalent to `myRectangle.SetWidth(15);` using the "old" Rectangle code

In More Details

- Note that in a property, `value` is what is called a *contextual keyword*[334]: it is not a reserved word in C# (it could be used as an identifier), but *inside a property* it refers to something special, the name of the set method parameter.
- In the following code:

  ```
  private int width;
  public int Width
  {
      get
      {
          return width;
      }
      set
      {
          width = value;
      }
  }
  ```

  The attribute `width` is called *the Width's property backing field*: it holds the data assigned to the property.
- When the property's get *and* set accessors are trivial (like the ones above), we can simply omit them their body completely. That is, the previous `Width` property could be implemented using

  ```
  public int Width { get; set;  }
  ```

  This is called *auto-properties*. Note that in this case, we do not need to declare the property's backing field (that is, no need to have `private int width;`), but cannot refer to it!
- Conversely, get and set accessor can contains arbitrarily convoluted code:

  ```
  public int Length
  {
      get { return length; }
      set {
          if (value < 0) {
              length = -value;
          }
          else if (value == 0) {
              length = 1;
  ```

[334] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/lexical-structure#644-keywords

```
            }
            else {
                length = value;
            }
        }
    }
```

- Note however that if the set or get accessor is not the "trivial" one, then auto-properties cannot be used and the other accessor must be specified.
  - For example, in the above code, simply writing `get;` instead of `get { return length; }` would give a compilation error.
- Note that properties can exist without backing field, and they can be *read-only* (that is, without a set accessor) or *write-only* (that is, without a get accessor, but this is rarer).
  - An example of read-only property is as follows:

    ```
    class Circle
    {
        public decimal Diameter { get; set; }
        // The constructor below  sets the value
        // of the property's backing field through
        // the property's set accessor.
        public Circle(decimal dP)
        {
            Diameter = dP;
        }
        // The Radius property below is
        // 1. read-only (no set accessor),
        // 2. without a backing field.
        public decimal Radius {
            get { return Diameter / 2; }
        }
    }
    ```

- It is possible to set a "custom default value" for properties using a *property initializer*, as follows:

  ```
  public double Width { get; set; } = -1;
  ```

  In this case, the property's backing field value will be -1 by default. Properties with initializer can be read-only:

  ```
  public int MaximumValue { get; } = 999;
  ```

- Finally, properties can be **`static`** as well:

```
public static string Explanation { get; set; } = "A
↪  Circle has for radius its diameter divided by 2.";
```

Such a property can be accessed using for example

```
Console.WriteLine(Circle.Explanation);
```

and its value can be changed, for instance by appending a `string` to it:

```
Circle.Explanation += "\nIts circumference is π
↪  multiplied by its diameter.";
```

Properties in UML Class Diagrams

Simple Notation

- Since properties represent (or, rather, allow to access) attributes, they go in the "attributes" box (the second box)
- If a property will simply "get" and "set" an instance variable of the same name, you do *not* need to write the instance variable in the box
  - No need to write both the property `Width` and the instance variable `width`
- Syntax: `[+/-] <<property>> [name]: [type]`
- Note that the access modifier (+ or -) is for the property, not the instance variable, so it is + if the property is **`public`** (which it usually is)
- Example for `Rectangle`, assuming we converted both attributes to use properties instead of getters and setters:

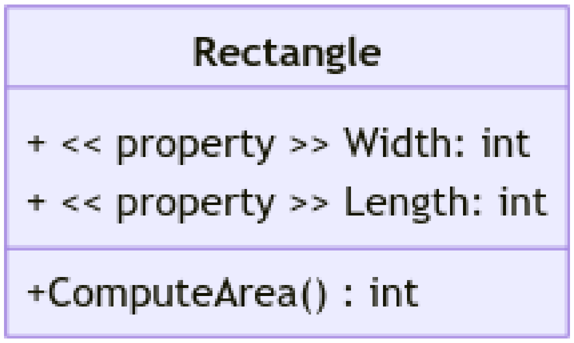


Figure 18: A UML diagram for the Rectangle class (text version[335])

- We no longer need to write all those setter and getter methods, since they are "built in" to the properties

More Accurate Notation In general, instead of writing for example

```
+ <<properties>> Explanation: string
```

one can write

```
+ <<get, set>> Explanation: string
```

or even

```
+ <<set>> Explanation: string
+ <<get>> Explanation: string
```

The benefit of this notation is that read-only properties can easily be integrated in the UML class diagram, by simply omitting the `<<set>>` line:

```
+ <<get>> Radius : decimal
```

The `static` Keyword

Static Methods

Different ways of calling methods

- Usually you call a method by using the "dot operator" (member access operator) on an object, like this:

  ```
  Rectangle rect = new Rectangle();
  rect.SetLength(12);
  ```

  The `SetLength` method is defined in the `Rectangle` class. In order to call it, we need an *instance* of that class, which in this case is the object `rect`.

- However, sometimes we have written code where we call a method using the dot operator on the name of a class, not an object. For example, the familiar `WriteLine` method:

  ```
  Console.WriteLine("Hello!");
  ```

  Notice that we have never needed to write `new Console()` to instantiate a `Console` object before calling this method.

- More recently, we learned about the `Array.Resize` method, which can be used to resize an array. Even though arrays are objects, we call the `Resize` method on the `Array` class, not the particular array object we want to resize:

```
int[] myArray = {10, 20, 30};
Array.Resize(ref myArray, 6);
```

- Methods that are called using the name of the class rather than an instance of that class are **static methods**

Declaring `static` methods

- Static methods are declared by adding the **`static`** keyword to the header, like this:

  ```
  class Console
  {
      public static void WriteLine(string value)
      {
          ...
      }
  }
  ```

- The **`static`** keyword means that this method belongs to the class “in general,” rather than an instance of the class
- Thus, you do not need an object (instance of the class) to call a static method; you only need the name of the class

`static` methods and instances

- Normal, non-static methods are always associated with a particular instance (object)
- When a normal method modifies an instance variable, it always “knows” which object to modify, because you need to specify the object when calling it
  - For example, the `SetLength` method is defined like this:

    ```
    class Rectangle
    {
        private int length;
        private int width;
        public void SetLength(int lengthParameter)
        {
            length = lengthParameter;
        }
    }
    ```

    When you call the method with `rect.SetLength(12)`, the `length` variable automatically refers to the `length` instance variable stored in `rect`.

- Static methods are not associated with any instance, and thus **cannot use instance variables**
- For example, we could attempt to declare the `ComputeArea` method of `Rectangle` as a static method, but this would not compile:

  ```
  class Rectangle
  {
      private int length;
      private int width;
      public void SetLength(int lengthParameter)
      {
          length = lengthParameter;
      }
      public static int ComputeArea()
      {
          return length * width;
      }
  }
  ```

  - To call this static method, you would write `Rectangle.ComputeArea();`
  - Since no `Rectangle` object is specified, which object's length and width should be used in the computation?

Uses for `static` methods

- Since static methods cannot access instance variables, they do not seem very useful
- One reason to use them: when writing a function that does not need to "save" any state, and just computes an output (its return value) based on some input (its parameters)
- Math-related functions are usually written as static methods. The .NET library comes with a class named `Math` that defines several static methods, like these:

  ```
  public static double Pow(double x, double y)
  {
      //Computes and returns x^y
  }
  public static double Sqrt(double x)
  {
      //Computes and returns the square root of x
  }
  public static int Max(int x, int y)
  {
  ```

```
    //Returns the larger of the two numbers x and y
}
public static int Min(int x, int y)
{
    //Returns the smaller of the two numbers x and y
}
```

Note that none of them need to use any instance variables.

- Defining several static methods in the same class (like in class `Math`) helps to group together similar or related functions, even if you never create an object of that class
- Static methods are also useful for providing the program's "entry point." Remember that your program must always have a `Main` method declared like this:

  ```
  class Program
  {
      static void Main(string[] args)
      {
          ...
      }
  }
  ```

  - When your program first starts, no objects exist yet, which means no "normal" methods can be called
  - The .NET run-time (the interpreter that runs a C# program) must decide what code to execute to make your program start running
  - It can call `Program.Main()` without creating an object, or knowing anything else about your program, because `Main` is a static method
- Static methods can be used to "help" other methods, both static and non-static
  - It's easy to call a static method from within the same class: You can just write the name of the method, without the class name, i.e. `MethodName(args)` instead of `ClassName.MethodName(args)`
  - For example, the `Array` class has a static method named `Copy` that copies the contents of one array into another array. This makes it very easy to write the `Resize` method:

    ```
    class Array
    {
        public static void Copy(Array source, Array
         ↪  dest, int length)
    ```

```
    {
        //Copy [length] elements from source to
         ↪  dest, in the same order
    }
    public static void Resize<T>(ref T[] array,
     ↪  int newSize)
    {
        T[] newArray = new T[newSize]
        Copy(array, newArray,
         ↪  Math.Min(array.Length, newSize));
        array = newArray;
    }
}
```

Since arrays are fixed-size, the only way to resize an array is to create a new array of the new size and copy the data from the old array into the new array. This Resize method is easy to read because the act of copying the data (which would involve a **`for`** loop) is written separately, in the `Copy` method, and Resize just needs to call `Copy`.

– Similarly, you can add additional static methods to the class that contains `Main`, and call them from within `Main`. This can help you separate a long program into smaller, easier-to-read chunks. It also allows you to re-use the same code multiple times without copying and pasting it.

```
class Program
{
    static void Main(string[] args)
    {
        int userNum1 = InputPositiveNumber();
        int userNum2 = InputPositiveNumber();
        int part1Result = DoPart1(userNum1,
         ↪  userNum2);
        DoPart2("Bananas", part1Result);
    }
    static int InputPositiveNumber()
    {
        int number;
        bool success;
        do
        {
            Console.WriteLine("Please enter a
 ↪  positive number");
            success =
 ↪  int.TryParse(Console.ReadLine(), out number);
```

```
        } while(!success || number < 0);
        return number;
    }
    static int DoPart1(int a, int b)
    {
        ...
    }
    static void DoPart2(string x, int y)
    {
        ...
    }
}
```

In this example, our program needs to read two different numbers from the user, so we put the input-validation loop into the `InputPositiveNumber` method instead of writing it twice in the `Main` method. It then has two separate "parts" (computing some result with the two user-input numbers, and combining that computed number with a string to display some output), which we write in the two methods `DoPart1` and `DoPart2`. This makes our actual `Main` method only 4 lines long.

Static Variables

Defining `static` variables

- The **`static`** keyword can be used in something that looks like an instance variable declaration:

  ```
  class Rectangle
  {
      private static int NumRectangles = 0;
      ...
  }
  ```

- This declares a variable that is stored with the class definition, not inside an object (it is *not* an instance variable)
- Unlike an instance variable, there is only one copy in the entire program, and any method that refers to `NumRectangles` will access the *same* variable, no matter which object the method is called on
- Since it is not an instance variable, it does not get initialized in the constructor. Instead, you must initialize it with a value when you declare it, more like a local variable (in this case, `NumRectangles` is initialized to 0).
- It's OK to declare a **`static`** variable with the **`public`** access modifier, because it is not part of any object's state. Thus, accessing

the variable from outside the class will not violate encapsulation, the principle that an object's state should only be modified by that object.

- For example, we could use the `NumRectangles` variable to count the number of rectangles in a program by making it **`public`**. We could define it like this:

  ```
  class Rectangle
  {
      public static int NumRectangles = 0;
      ...
  }
  ```

  and use it like this, in a `Main` method:

  ```
  Rectangle myRect = new Rectangle();
  Rectangle.NumRectangles++;
  Rectangle myOtherRect = new Rectangle();
  Rectangle.NumRectangles++;
  ```

Using `static` variables

- Since all instances of a class share the same static variables, you can use them to keep track of information about "the class as a whole" or "all the objects of this type"
- A common use for static variables is to count the number of instances of an object that have been created so far in the program
  - Instead of "manually" incrementing this counter, like in our previous example, we can increment it inside the constructor:

    ```
    class Rectangle
    {
        public static int NumRectangles = 0;
        private int length;
        private int width;
        public Rectangle(int lengthP, int widthP)
        {
            length = lengthP;
            width = widthP;
            NumRectangles++;
        }
    }
    ```

  - Each time this constructor is called, it initializes a new `Rectangle` object with its own copy of the `length` and `width` variables. It also increments the single copy of the

NumRectangles variable that is shared by all Rectangle objects.

  - The variable can still be accessed from the Main method (because it is public), where it could be used like this:

    ```
    Rectangle rect1 = new Rectangle(2, 4);
    Rectangle rect2 = new Rectangle(7, 5);
    Console.WriteLine(Rectangle.NumRectangles
        + " rectangle objects have been created");
    ```

    When rect1 is instantiated, its copy of length is set to 2 and its copy of width is set to 4, then the single NumRectangles variable is incremented to 1. Then, when rect2 is instantiated, its copy of length is set to 7 and its copy of width is set to 5, and the NumRectangles variable is incremented to 2.

- Static variables are also useful for **constants**
  - The const keyword, which we learned about earlier, is actually very similar to **static**
  - A const variable is just a **static** variable that cannot be modified
  - Like a **static** variable, it can be accessed using the name of the class where it is defined (e.g. Math.PI), and there is only one copy for the entire program

Static methods and variables

- Static methods cannot access instance variables, but they *can* access static variables
- There is no ambiguity when accessing a static variable: you do not need to know which object's variable to access, because there is only one copy of the static variable shared by all objects
- This means you can write a "getter" or "setter" for a static variable, as long as it is a static method. For example, we could improve our NumRectangles counter by ensuring that the Main method can only read it through a getter method, like this:

  ```
  class Rectangle
  {
      private static int NumRectangles = 0;
      private int length;
      private int width;
      public Rectangle(int lengthP, int widthP)
      {
          length = lengthP;
  ```

```
        width = widthP;
        NumRectangles++;
    }
    public static int GetNumRectangles()
    {
        return NumRectangles;
    }
}
```

- The `NumRectangles` variable is now declared **`private`**, which means only the Rectangle constructor will be able to increment it. Before, it would have been possible for the `Main` method to execute something liek `Rectangle.NumRectangles = 1;` and throw off the count.
- The `GetNumRectangles` method cannot access `length` or `width` because they are instance variables, but it can access `NumRectangles`
- The static method would be called from the `Main` method like this:

  ```
  Rectangle rect1 = new Rectangle(2, 4);
  Rectangle rect2 = new Rectangle(7, 5);
  Console.WriteLine(Rectangle.GetNumRectangles()
      + " rectangle objects have been created");
  ```

Summary of `static` access rules

- Static variables and instance variables are both **fields** of a class; they can also be called "static fields" and "non-static fields"
- This table summarizes how methods are allowed to access them:

	Static Field	Non-static Field
Static method	Yes	No
Non-static method	Yes	Yes

Static Classes

- The **`static`** keyword can also be used in a class declaration
- If a class is declared **`static`**, all of its members (fields and methods) must be static
- This is useful for classes that serve as "utility libraries" containing a collection of functions, and are not supposed to be instantiated and used as objects

- For example, the `Math` class is declared like this:

  ```
  static class Math
  {
      public static double Sqrt(double x)
      {
          ...
      }
      public static double Pow(double x, double y)
      {
          ...
      }
  }
  ```

  There is no need to ever create a `Math` object, but all of these methods belong together (within the same class) because they all implement standard mathematical functions.

Generic Type Parameter

Introduction

Imagine that you want to write a method that takes as an argument an array and returns an array of the same type, but with the values reversed. You may write the following code:

```
public class Helper{
    public static int[] Reverse(int[] arrayP)
    {
        int[] result = new int[arrayP.Length];
        int j = 0;
        for (int i = arrayP.Length - 1; i >= 0; i--)
        {
            result[j] = arrayP[i];
            j++;
        }
        return result;
    }
}
```

Then, this method could be used as follows:

```
int[] array1 = {0, 2, 3, 6};
int[] array1reversed = Helper.Reverse(array1);
```

And then `array1reversed` would contain 6, 3, 2, 0.

This method works as intended, but you can use it only with arrays of *integers*. If you want to use a similar method with arrays of, say, `char`, then you need to copy-and-paste the code above and to replace every occurrence of `int` by `char`. This is not very efficient, and it is error-prone.

Generic Types

There is a tool in C# to avoid having to be *too* specific, and to be able to tell the compiler that the method will work "with some type", called generic type parameter[336], using the keyword T. In essence, <T> is affixed after the name of the method to signal that the method will additionally require to instantiate T with a particular type.

The previous method would become:

```
public class Helper{
    public static T[] Reverse<T>(T[] arrayP)
    {
        T[] result = new T[arrayP.Length];
        int j = 0;
        for (int i = arrayP.Length - 1; i >= 0; i--)
        {
            result[j] = arrayP[i];
            j++;
        }
        return result;
    }
}
```

where three occurrences of `int[]` were replaced by `T[]`, and <T> was additionally added between the name of the method and its parameters. This method is used as follows:

```
int[] array1 = {0, 2, 3, 6};
int[] array1reversed = Helper.Reverse<int>(array1);

char[] array2 = {'a', 'b', 'c'};
char[] array2reversed = Helper.Reverse<char>(array2);
```

In essence, `Reverse<int>` tells C# that `Reverse` will be used with `T` being `int` (not `int[]`, as the method uses `T[]` for its argument and return type). Note that to use *the same method* with `char`, we simply use `Reverse<char>`, and then we provide an array of `char` as parameters, and obtain an array of `char` in return.

[336]https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters

Implicitly Typed Local Variables

Sometimes, the body of the method needs to declare variable with the same type as T. Indeed, imagine, for example, that we want to add to our Helper class a method that returns a string description of an array. We can write the following:

```
public static string Description(int[] arrayP)
{
    string returned = "";
    foreach (int element in arrayP)
    {
        returned += element + " ";
    }
    return returned;
}
```

but this method is specific to arrays of int, and we would have to write another one for char, for example. Making the header generic is "easy", as we can use, as before:

```
public static string Description<T>(T[] arrayP)
```

but the body is problematic: what should be the type of the element variable in the header of the **foreach**? We cannot simply use T, but we can use *implicitly typed variable*. This technique, that uses the keyword var essentially tells C# to ... figure out the type of the variable. In that case, since C# knows the type of the array you are passing, it can easily infer the type of its elements.

We can then rewrite the previous method as follows:

```
public static string Description<T>(T[] arrayP)
{
    string returned = "";
    foreach (var element in arrayP)
    {
        returned += element + " ";
    }
    return returned;
}
```

and use it with

```
Console.WriteLine(Helper.Display<char>(array2);
```

for example.

Introduction

Decisions are a constant occurrence in daily life. For instance consider an instructor teaching CSCI 1301. At the beginning of class the instructor may

- Ask if there are questions. If a student has a question, then the instructor will answer it, and ask again ("Anything else?").
- When there are no more questions, the instructor will move on to the next step.
- If there is a quiz scheduled, the next step will be distributing the quiz.
- If there is no quiz scheduled or the quiz is complete (and collected), the instructor may introduce the lecture topic ("Today, we will be discussing Decisions and Decision Structures") and start the class.
- etc.

This type of "branching" between multiple choices can be represented with an activity diagram[337]:

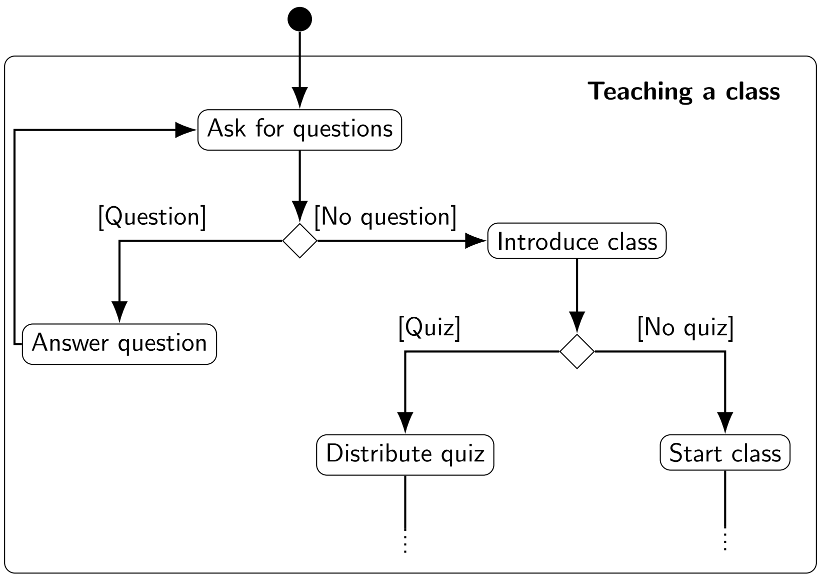


Figure 19: "An Activity Diagram on Teaching a Class"

In C#, we will express

337 https://www.wikiwand.com/en/Activity_diagram

- repetition (or "loops") ("As long as there are questions…") with the **`while`**, `do…while` and **`for`** keywords,
- branching ("If there is a quiz…") with the **`if`**, `if…else` and **`switch`** keywords.

Both structures need a datatype to express the result of a decision ("Is it *true* that there are questions.", or "Is it *false* that there is a quiz.") called Booleans. Boolean values can be set with conditions, that can be composed in different ways using three operators ("and", "or" and "not"). For example, "If today is a Monday or Wednesday, and it is not past 10:10 am, the class will also include a brief reminder about the upcoming exam."

Booleans

Variables

We can store if something is true or false ("The user has reached the age of majority", "The switch is on", "The user is using Windows", "This computer's clock indicates that we are in the afternoon", …) in a variable of type boolean, which is also known as a boolean *flag*. Note that **`true`** and **`false`** are the only possible two values for boolean variables: there is no third option!

We can declare, assign, initialize and display a boolean variable (flag) as with any other variable:

```
bool learning_how_to_program = true;
Console.WriteLine(learning_how_to_program);
```

Operations on Boolean Values

Boolean variables have only two possible values (**`true`** and **`false`**), but we can use three operations to construct more complex booleans:

1. "and" (`&&`, conjunction),
2. "or" (`||`, disjunction),
3. "not" (`!`, negation).

Each has the precise meaning described here:

1. the condition "A and B" is true if and only if A is true, and B is true,
2. "A or B" is false if and only if A is false, and B is false (that is, it takes only one to make their disjunction true),
3. "not A" is true if and only if A is false (that is, "not" "flips" the value it is applied to).

The expected results of these operations can be displayed in *truth tables*, as follows:

Operation	Value
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Operation	Value
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Operation	Value
`!true`	`false`
`!false`	`true`

These tables can also be written in 2-dimensions, as can be seen for conjunction on wikipedia[338].

Equality and Relational Operators

Boolean values can also be set through expressions, or tests, that "evaluate" a condition or series of conditions as **`true`** or **`false`**. For instance, you can write an expression meaning "variable `myAge` has the value 12" which will evaluate to **`true`** if the value of `myAge` is indeed 12, and to **`false`** otherwise. *To ease your understanding*, we will write "expression $\rightarrow$ **`true`**" to indicate that "expression" evaluates to **`true`** below, but this is *not* part of C#'s syntax.

Here we use two kinds of operators:

- Equality operators test if two values (literal or variable) are the same. This works on all datatypes.
- Relational operators test if a value (literal or variable) is greater or smaller (strictly or largely) than an other value or variable.

Relational operators will be primarily used for numerical values.

[338] https://www.wikiwand.com/en/Truth_table#Logical_conjunction_(AND)

Equality Operators

In C#, we can test for equality and inequality using two operators, `==` and `!=`.

Mathematical Notation	C# Notation	Example
$=$	`==`	`3 == 4` $\to$ `false`
$\neq$	`!=`	`3!=4` $\to$ `true`

Note that testing for equality uses *two equal signs*: C# already uses a single equal sign for assignments (e.g. `myAge = 12;`), so it had to pick another notation! It is fairly common across programing languages to use a single equal sign for assignments and double equal for comparisons.

Writing `a != b` ("a is not the same as b") is actually logically equivalent to writing `!(a == b)` ("it is not true that a is the same as b"), and both expressions are acceptable in C#.

We can test numerical values for equality, but actually any datatype can use those operators. Here are some examples for `int`, `string`, `char` and `bool`:

```
int myAge = 12;
string myName = "Thomas";
char myInitial = 'T';
bool cs_major = true;
Console.WriteLine("My age is 12: " + (myAge == 12));
Console.WriteLine("My name is Bob: " + (myName == "Bob"));
Console.WriteLine("My initial is Q: " + (myInitial ==
 ↪  'Q'));
Console.WriteLine("My major is Computer Science: " +
 ↪  cs_major);
```

This program will display

```
My age is 12: True
My name is Bob: False
My initial is Q: False
My major is Computer Science: True
```

Remember that C# is case-sensitive, and that applies to the equality operators as well: for C#, the string `Thomas` is not the same as the string `thomas`. This also holds for characters like `a` versus `A`.

```
Console.WriteLine("C# is case-sensitive for string
 ↪  comparison: " + ("thomas" != "Thomas"));
```

```
Console.WriteLine("C# is case-sensitive for character
 ↪  comparison: " + ('C' != 'c'));
Console.WriteLine("But C# does not care about 0 decimal
 ↪  values: " + (12.00 == 12));
```

This program will display:

```
C# is case-sensitive for string comparison: True
C# is case-sensitive for character comparison: True
But C# does not care about 0 decimal values: True
```

Relational Operators

We can test if a value or a variable is greater than another, using the following *relational* operators.

Mathematical Notation	C# Notation	Example
$>$	`>`	`3 > 4` $\longrightarrow$ **`false`**
$<$	`<`	`3 < 4` $\longrightarrow$ **`true`**
$\geq$ or $\geqslant$	`>=`	`3 >= 4` $\longrightarrow$ **`false`**
$\leq$ or $\leqslant$	`<=`	`3 <= 4` $\longrightarrow$ **`true`**

Relational operators can also compare `char`, but the order is a bit complex (you can find it explained, for instance, in this stack overflow answer[339]).

Precedence of Operators

All of the operators have a "precedence", which is the order in which they are evaluated. The precedence is as follows:

Operator	
!	is evaluated before
*, /, and %	which are evaluated before
+ and -	which are evaluated before
<, >, <=, and >=	which are evaluated before
== and !=	which are evaluated before
&&	which is evaluated before
||	which comes last.

[339]https://stackoverflow.com/a/14967721/

- Operators with higher precedence (higher in the table) are evaluated before operators with lower precedence (lower in the table). For instance, in an expression like `2*3+4`, `2*3` will have higher precedence than `3+4`, and thus be evaluated first: `2*3+4` is to be read as `(2*3)+4 = 6 + 4 = 10` and *not* as `2*(3+4) = 2*7 = 14`.
- Operators on the same row have equal precedence and are evaluated in the order they appear, from left to right: in `1-2+3`, `1-2` will be evaluated before `2+3`: `1-2+3` is to be read as `(1-2)+3 = -1 + 3 = 2` and *not* as `1-(2+3) = 1-5 = 4`.
- Forgetting about precedence can lead to errors that can be hard to debug: for instance, an expression such as `! 4 == 2` will give the error

```
The `!' operator cannot be applied to operand of type
 ↪  `int'
```

Since `!` has a higher precedence than `==`, C# first attempts to compute the result of `!4`, which corresponds to "not 4". As negation (`!`) is an operation that can be applied only to booleans, this expression does not make sense and C# reports an error. The expression can be rewritten to change the order of evaluation by using parentheses, e.g. you can write `!(4 == 2)`, which will correctly be evaluated to **`true`**.

if

if Statements

Introduction

- Recall from a previous lecture (Booleans and Comparisons) that decision structures change the flow of code execution based on conditions
- Now that we know how to write conditions in C#, we can write decision structures
- Our first decision structure is the **if statement**, which executes a block of code *only if a condition is true*

Example code with an `if` statement

```
Console.WriteLine("Enter your age");
int age = int.Parse(Console.ReadLine());
if (age >= 18)
{
    Console.WriteLine("You can vote!");
}
Console.WriteLine("Goodbye");
```

- After the keyword **`if`** is a condition, in parentheses: `age >= 18`
- On the next line after the **`if`** statement, the curly brace begins a code block. The code in this block is "controlled" by the **`if`** statement.
- If the condition `age >= 18` is true, the code in the block (the WriteLine statement with the text "You can vote!") gets executed, then execution proceeds to the next line (the WriteLine statement that prints "Goodbye")
- If the condition `age >= 18` is false, the code in the block gets *skipped*, and execution proceeds directly to the line that prints "Goodbye"
- The behavior of this program can be represented by this flowchart:

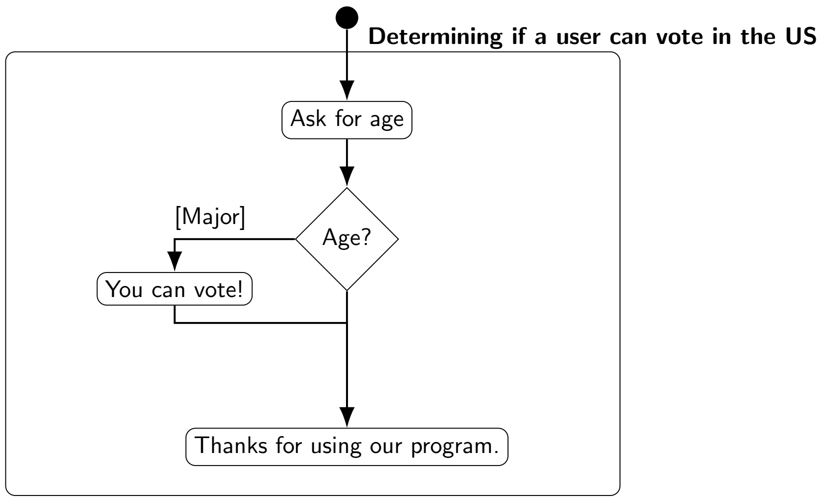

Determining if a user can vote in the US
Ask for age
Age?
[Major]
You can vote!
Thanks for using our program.

Figure 20: "A flowchart representation of an if statement"

- Example interaction 1:

```
Enter your age
20
You can vote!
Goodbye
```

When the user enters "20", the value 20 is assigned to the `age` variable, so the condition `age >= 18` is true. This means the code inside the **`if`** statement's block gets executed.

- Example interaction 2:

  ```
  Enter your age
  17
  Goodbye
  ```

  When the user enters "17", the value 17 is assigned to the `age` variable, so the condition `age >= 18` is false, and the **`if`** statement's code block gets skipped.

Syntax and rules for if statements

- Formally, the syntax for an **`if`** statement is this:

  ```
  if (<condition>)
  {
      <statements>
  }
  ```

- The "condition" in parentheses can be any expression that produces a `bool` value, including all of the combinations of conditions we saw in the previous lecture (Booleans and Comparisons). It can even be a `bool` variable, since a `bool` variable "contains" a `bool` value.

- Note that there is no semicolon after the `if (<condition>)`. It's a kind of "header" for the following block of code, like a method header.

- The statements in the code block will be executed if the condition evaluates to **`true`**, or skipped if it evaluates to **`false`**

- If the code block contains only *one* statement, the curly braces can be omitted, producing the following syntax:

  ```
  if(<condition>)
      <statement>
  ```

  For example, the **`if`** statement in our previous example could be written like this, since there was only one statement in the code block:

  ```
  if(age >= 18)
      Console.WriteLine("You can vote!");
  Console.WriteLine("Goodbye");
  ```

- Omitting the curly braces is slightly dangerous, though, because it makes it less obvious which line of code is controlled by the **`if`** statement. It is up to you, the programmer, to remember to indent the line after the **`if`** statement, and then de-indent the line after that; indentation is just a convention. Curly braces make it easier to see where the **`if`** statement starts and ends.

if-else Statements

Example:

```
if(age >= 18)
{
    Console.WriteLine("You can vote!");
}
else
{
    Console.WriteLine("You are too young to vote");
}
Console.WriteLine("Goodbye");
```

- The **if-else statement** is a decision structure that chooses *which* block of code to execute, based on whether a condition is true or false
- In this example, the condition is `age >= 18` again
- The first block of code (underneath the **`if`**) will be executed if the statement is true – the console will display "You can vote!"
- The *second* block of code, which comes after the keyword **`else`**, will be executed if the statement is *false* – so if the user's age is less than 18, the console will display "You are too young to vote"
- Only one of these blocks of code will be executed; the other will be skipped
- After executing one of the two code blocks, execution continues at the next line after the **`else`** block, so in either case the console will next display "Goodbye"
- The behavior of this program can be represented by this flowchart:

Syntax and comparison

- Formally, the syntax for an **`if-else`** statement is this:

  ```
  if (<condition>)
  {
      <statement block 1>
  }
  else
  {
  ```

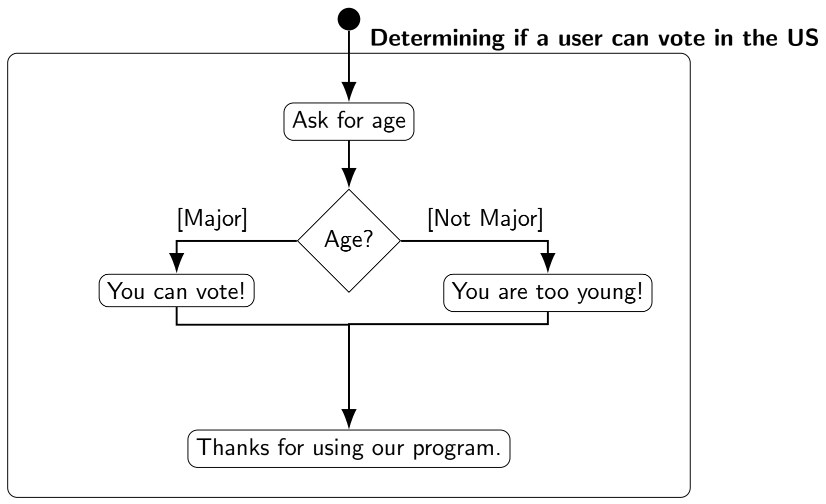

Determining if a user can vote in the US
Ask for age
Age?
[Major]
[Not Major]
You can vote!
You are too young!
Thanks for using our program.

Figure 21: "A flowchart representation of an if-else statement"

```
    <statement block 2>
}
```

- As with the **if** statement, the condition can be anything that produces a `bool` value
- Note that there is no semicolon after the **else** keyword
- If the condition is true, the code in statement block 1 is executed (this is sometimes called the "if block"), and statement block 2 is skipped
- If the condition is false, the code in statement block 2 is executed (this is sometimes called the "else block"), and statement block 1 is skipped
- This is very similar to an if statement; the difference is what happens if the condition is false
  - With an **if** statement, the "if block" is executed if the condition is true, but *nothing happens* if the condition is false.
  - With an **if-else** statement, the code in the "else block" is executed if the condition is false, so something always happens - one of the two code blocks will get executed

Nested if-else Statements

- If-else statements are used to change program flow based on a condition; they represent making a decision
- Sometimes decisions are more complex than a single yes/no question: once you know whether a certain condition is true or false, you then need to ask another question (check another condition) based on the outcome
- For example, we could improve our voting program to ask the user whether he/she is a US citizen, as well as his/her age. This means there are two conditions to evaluate, as shown in this flowchart:

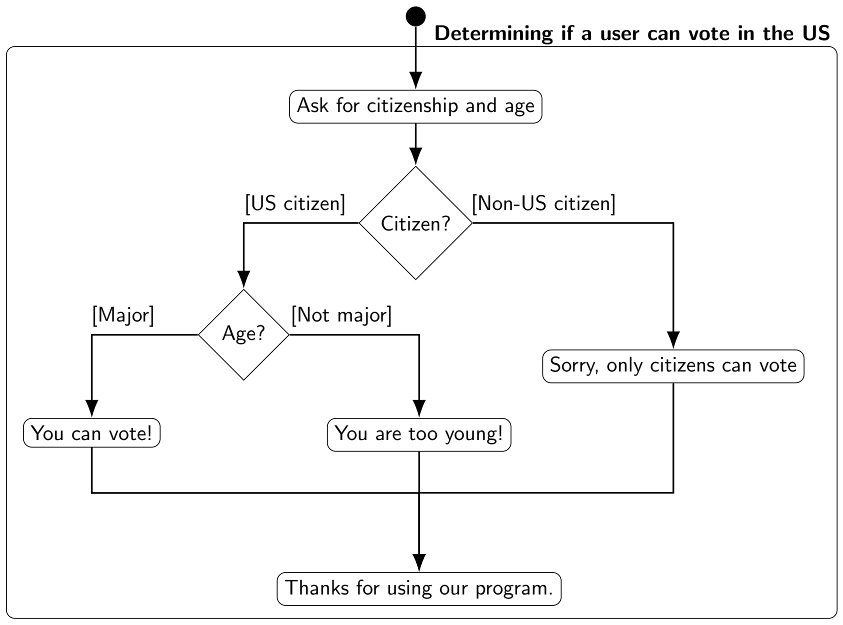

Determining if a user can vote in the US
Ask for citizenship and age
Citizen?
[US citizen]
[Non-US citizen]
Age?
[Major]
[Not major]
Sorry, only citizens can vote
You can vote!
You are too young!
Thanks for using our program.

Figure 22: "A flowchart representation of the nested if-else statement"

  - First, the program should test whether the user is a citizen. If not, there is no need to check the user's age, since he/she cannot vote anyway
  - If the user is a citizen, the program should then test whether the user is over 18 to determine if he/she is old enough to vote.

Using nested if statements

- An if statement's code block can contain any kind of statements, including another if statement
- Putting an if statement inside an if block represents making a sequence of decisions - once execution has reached the inside of an if block, your program "knows" that the if condition is true, so it can proceed to make the next decision
- For the voting example, we can implement the decision structure from the flowchart above with this code, assuming age is an int and usCitizen is a bool:

```
if(usCitizen == true)
{
    if(age >= 18)
    {
        Console.WriteLine("You can vote!");
    }
    else
    {
        Console.WriteLine("You are too young to
 ↪  vote");
    }
}
else
{
    Console.WriteLine("Sorry, only citizens can
 ↪  vote");
}
Console.WriteLine("Goodbye");
```

  - First, the program tests the condition `usCitizen == true`, and if it is true, the code in the first "if block" is executed
  - Within this if block is another if statement that tests the condition `age >= 18`. This represents checking the user's age after determining that he/she is a US citizen - execution only reaches this second if statement if the first one evaluated to true. So "You can vote" is printed if both `usCitizen == true` and `age >= 18`
  - If the condition `usCitizen == true` is false, the if block is skipped and the else block is executed instead, so the entire inner if statement is never executed – the user's age does not matter if he/she isn't a citizen
  - Note that the condition `usCitizen == true` could also be expressed by just writing the name of the variable usCitizen (i.e., the if statement would be `if(usCitizen)`), because

usCitizen is a bool variable. We do not need the equality comparison operator to test if it is **true**, because an **if** statement already tests whether its condition is **true** (and a bool variable by itself is a valid condition)

  - Note that indentation helps you match up an **else** block to its corresponding **if** block. The meaning of **else** depends on which **if** statement it goes with: the "outer" **else** will be executed if the condition `usCitizen == true` is false, while the "inner" **else** will be executed if the condition `age >= 18` is false.

- Nested **if** statements do not need to be the *only* code in the if block; you can still write other statements before or after the nested **if**
- For example, we could change our voting program so that it only asks for the user's age if he/she is a citizen:

```
if(usCitizen == true)
{
    Console.WriteLine("Enter your age");
    int age = int.Parse(Console.ReadLine());
    if(age >= 18)
    {
        Console.WriteLine("You can vote!");
    }
    else
    {
        Console.WriteLine("You are too young to
 ↪  vote");
    }
}
else
{
    Console.WriteLine("Sorry, only citizens can
 ↪  vote");
}
Console.WriteLine("Goodbye");
```

if-else-if Statements

- Sometimes your program needs to test multiple conditions at once, and take different actions depending on which one is true
- Example: We want to write a program that tells the user which floor a `ClassRoom` object is on, based on its room number
  - If the room number is between 100 and 200 it is on the first floor; if it is between 200 and 300 it is on the second floor; if it is greater

  than 300 it is on the third floor
- There are 3 ranges of numbers to test, and 3 possible results, so we cannot do it with a single if-else statement

If-else-if syntax

- An if-else-if statement looks like this:

```
if(<condition 1>)
{
    <statement block 1>
}
else if(<condition 2>)
{
    <statement block 2>
}
else if(<condition 3>)
{
    <statement block 3>
}
else
{
    <statement block 4>
}
```

- Unlike an **`if`** statement, there are multiple conditions
- They are evaluated *in order*, top to bottom
- Just like with **`if-else`**, exactly one block of code will get executed
- If condition 1 is true, statement block 1 is executed, and everything else is skipped
- If condition 1 is false, statement block 1 is skipped, and execution proceeds to the first **`else if`** line; condition 2 is then evaluated
- If condition 2 is true, statement block 2 is executed, and everything else is skipped
  - Thus, statement block 2 is only executed if condition 1 is false *and* condition 2 is true
- Same process repeats for condition 3: If condition 2 is false, condition 3 is evaluated, and statement block 3 is either executed or skipped
- If *all* the conditions are false, the final else block (statement block 4) is executed

Using if-else-if to solve the "floors problem"

- Assuming `myRoom` is a `ClassRoom` object, this code will display which floor it is on:

  ```
  if(myRoom.GetNumber() >= 300)
  {
      Console.WriteLine("Third floor");
  }
  else if(myRoom.GetNumber() >= 200)
  {
      Console.WriteLine("Second floor");
  }
  else if(myRoom.GetNumber() >= 100)
  {
      Console.WriteLine("First floor");
  }
  else
  {
      Console.WriteLine("Invalid room number");
  }
  ```

- If the room number 300 or greater (e.g. 365), the first "if" block is executed, and the rest are skipped. The program prints "Third floor"
- If the room number is less than 300, the program continues to the line `else if(myRoom.GetNumber() >= 200)` and evaluates the condition
- If `myRoom.GetNumber() >= 200` is true, it means the room number is between 200 and 299, and the program will print "Second floor." Even though the condition only tests whether the room number is >= 200, this condition is only evaluated if the first one was false, so we know the room number must be < 300.
- If the second condition is false, the program continues to the line `else if(myRoom.GetNumber() >= 100)`, evaluates the condition, and prints "First floor" if it is true.
- Again, the condition `myRoom.GetNumber() >= 100` is only evaluated if the first two conditions have already been tested and turned out false, so we know the room number is less than 300 and less than 200.
- In the final `else` block, the program prints "Invalid room number" because this block is only executed if the room number is less than 100 (all three conditions were false).

if-else-if with different conditions

- We often use if-else-if statements to test the same variable multiple times, but there is no requirement for the conditions to use the same variable
- An if-else-if statement can use several different variables, and its conditions can be completely unrelated, like this:

```
int x;
if(myIntVar > 12)
{
    x = 10;
}
else if(myStringVar == "Yes")
{
    x = 20;
}
else if(myBoolVar)
{
    x = 30;
}
else
{
    x = 40;
}
```

- Note that the order of the else-if statements still matters, because they are evaluated top-to-bottom. If `myIntVar` is 15, it does not matter what values `myStringVar` or `myBoolVar` have, because the first if block (setting x to 10) will get executed.
- Example outcomes of executing this code (which value x is assigned) based on the values of `myIntVar`, `myStringVar`, and `myBoolVar`:

`myIntVar`	`myStringVar`	`myBoolVar`	x
12	"Yes"	**true**	20
15	"Yes"	**false**	10
-15	"yes"	**true**	30
10	"yes"	**false**	40

if-else-if vs. nested if

- Sometimes a nested **if** statement can be rewritten as an **if-else-if** statement

- This reduces the amount of indentation in your code, which makes it easier to read
- To convert a nested **if** statement to **if-else-if**, you'll need to combine the conditions of the "outer" and "inner" **if** statements, using the logical operators
- A nested **if** statement inside an **if** block is testing whether the outer **if**'s condition is true *and* its own condition is true, so combine them with the && operator
- The **else** block of the inner **if** statement can be rewritten as an **else if** by combining the outer **if**'s condition with the *opposite* of the inner **if**'s condition, since "else" means "the condition is false." We need to explicitly write down the "false condition" that is normally implied by **else**.
- For example, we can rewrite this nested **if** statement:

```
if(usCitizen == true)
{
    if(age >= 18)
    {
        Console.WriteLine("You can vote!");
    }
    else
    {
        Console.WriteLine("You are too young to
 ↪  vote");
    }
}
else
{
    Console.WriteLine("Sorry, only citizens can
 ↪  vote");
}
```

as this **if-else-if** statement:

```
if(usCitizen == true && age >= 18)
{
    Console.WriteLine("You can vote!");
}
else if(usCitizen == true && age < 18)
{
    Console.WriteLine("You are too young to vote");
}
else
{
    Console.WriteLine("Sorry, only citizens can
↪   vote");
}
```

- Note that the **else** from the inner if statement becomes `else if(usCitizen == true && age < 18)` because we combined the outer if condition (`usCitizen == true`) with the opposite of the inner if condition (`age >= 18`).
- Not all nested **if** statements can be rewritten this way. If there is additional code in a block, other than the nested **if** statement, it is harder to convert it to an if-else-if
- For example, in this nested **if** statement:

  ```
  if(usCitizen == true)
  {
      Console.WriteLine("Enter your age");
      int age = int.Parse(Console.ReadLine());
      if(age >= 18)
      {
          Console.WriteLine("You can vote!");
      }
      else
      {
          Console.WriteLine("You are too young to
  ↪   vote");
      }
  }
  else
  {
      Console.WriteLine("Sorry, only citizens can
  ↪   vote");
  }
  Console.WriteLine("Goodbye");
  ```

  the code that asks for the user's age executes after the outer **if** condition is determined to be true, but before the inner **if** condition is tested. There would be nowhere to put this code if we tried to convert it to an if-else-if statement, since both conditions must be tested at the same time (in `if(usCitizen == true && age >= 18)`).
- On the other hand, any if-else-if statement can be rewritten as a nested **if** statement
- To convert an if-else-if statement to a nested **if** statement, rewrite each **else if** as an **else** block with a nested **if** statement inside it – like you're splitting the "if" from the "else"

- This results in a lot of indenting if there are many `else if` lines, since each one becomes another nested `if` inside an `else` block
- For example, the "floors problem" could be rewritten like this:

```
if(myRoom.GetNumber() >= 300)
{
    Console.WriteLine("Third floor");
}
else
{
    if(myRoom.GetNumber() >= 200)
    {
        Console.WriteLine("Second floor");
    }
    else
    {
        if(myRoom.GetNumber() >= 100)
        {
            Console.WriteLine("First floor");
        }
        else
        {
            Console.WriteLine("Invalid room number");
        }
    }
}
```

Switch

Switch Statements

Multiple equality comparisons

- In some situations, your program will need to test if a variable is equal to one of several values, and perform a different action based on which value the variable matches
- For example, you have an `int` variable named `month` containing a month number, and want to convert it to a `string` with the name of the month. This means your program needs to take a different action depending on whether `month` is equal to 1, 2, 3, … or 12:
- One way to do this is with a series of `if-else-if` statements, one for each possible value, like this:

```
Console.WriteLine("Enter the month as a number
 ↪  between 1 and 12.");
```

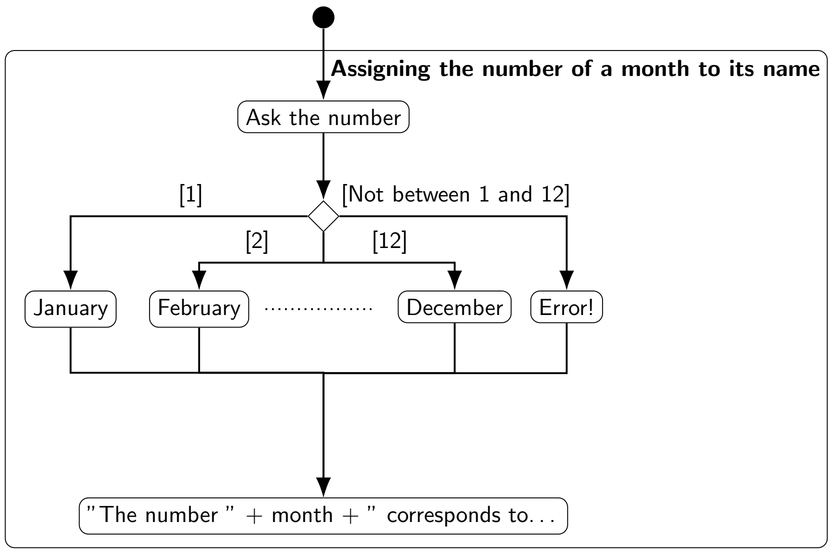

Assigning the number of a month to its name
Ask the number
[1]
[Not between 1 and 12]
[2]
[12]
January
February
December
Error!
"The number " + month + " corresponds to...

Figure 23: "A flowchart representation of the mapping between month number and name"

```
int month = int.Parse(Console.ReadLine());
string monthName;
if(month == 1)
{
    monthName = "January";
}
else if(month == 2)
{
    monthName = "February";
}
else if(month == 3)
{
    monthName = "March";
}
else if(month == 4)
{
    monthName = "April";
}
else if(month == 5)
{
    monthName = "May";
}
else if(month == 6)
{
    monthName = "June";
}
else if(month == 7)
{
    monthName = "July";
}
else if(month == 8)
{
    monthName = "August";
}
else if(month == 9)
{
    monthName = "September";
}
else if(month == 10)
{
    monthName = "October";
}
else if(month == 11)
{
    monthName = "November";
}
else if(month == 12)
{
    monthName = "December";
}
else
{
    monthName = "Error!"; // Invalid month
}
Console.WriteLine("The number " + month + "
 ↪  corresponds to the month " + monthName + ".")
```

- This code is very repetitive, though: every **`else if`** statement is almost the same, with only the number changing. The text "**`if`**`(month ==`" is copied over and over again.

Syntax for `switch` statements

- A **`switch`** statement is a simpler, easier way to compare a single variable against multiple possible values
- It is written like this:

```
switch (<variable name>)
{
    case <value 1>:
        <statement block 1>
        break;
    case <value 2>:
        <statement block 2>
        break;
    …
    default:
        <statement block n>
        break;
}
```

- First, the "header" of the **switch** statement names the variable that will be compared
- The "body" of the switch statement is enclosed in curly braces, and contains multiple **case** statements
- Each **case** statement gives a possible value the variable could have, and a block of statements to execute if the variable equals that value. Statement block 1 is executed if the variable is equal to value 1, statement block 2 is executed if the variable is equal to value 2, etc.
- The statement "block" within each **case** is **not** enclosed in curly braces, unlike **if** and **else if** blocks. Instead, it begins on the line after the **case** statement, and ends with the keyword **break**.
- The **default** statement is like the **else** statement: It defines code that gets executed if the variable does not match any of the values in the **case** statements.
- The values in each **case** statement must be **literals**, not variables, and they must be **unique** (you cannot write two **case** statements with the same value)

Example `switch` statement

- This program has the same behavior as our previous example, but uses a **switch** statement instead of an **if-else-if** statement:

```
Console.WriteLine("Enter the month as a number
 ↪  between 1 and 12.");
int month = int.Parse(Console.ReadLine());
string monthName;
switch(month)
{
    case 1:
        monthName = "January";
        break;
    case 2:
        monthName = "February";
        break;
    case 3:
        monthName = "March";
        break;
    case 4:
        monthName = "April";
        break;
    case 5:
        monthName = "May";
        break;
    case 6:
        monthName = "June";
        break;
    case 7:
        monthName = "July";
        break;
    case 8:
        monthName = "August";
        break;
    case 9:
        monthName = "September";
        break;
    case 10:
        monthName = "October";
        break;
    case 11:
        monthName = "November";
        break;
    case 12:
        monthName = "December";
        break;
    default:
        monthName = "Error!"; // Invalid month
        break;
}
Console.WriteLine("The number " + month + "
 ↪  corresponds to the month " + monthName + ".")
```

- Since the variable in the **switch** statement is `month`, each **case** statement means, effectively, **if** `(month == <value>)`. For example, **case** `1:` has the same effect as **if** `(month == 1)`

- The values in each `case` statement must be `int` literals, since `month` is an `int`
- The `default` statement has the same effect as the final `else` in the `if-else-if` statement: it contains code that will be executed if `month` did not match any of the values

`switch` with multiple statements

- So far, our examples have used only one line of code in each `case`
- Unlike `if-else`, you do not need curly braces to put multiple lines of code in a `case`
- For example, imagine our "months" program needed to convert a month number to both a month name and a three-letter abbreviation. The `switch` would look like this:

```
string monthName;
string monthAbbrev;
switch(month)
{
    case 1:
        monthName = "January";
        monthAbbrev = "Jan";
        break;
    case 2:
        monthName = "February";
        monthAbbrev = "Feb";
        break;
    // and so on, with all the other months...
}
```

- The computer knows which statements are included in each case because of the `break` keyword. For the "1" case, the block of statements starts after `case 1:` and ends with the `break;` after `monthAbbrev = "Jan";`

Intentionally omitting break

- Each block of code that starts with a `case` statement must end with a `break` statement; it will not automatically end at the next `case` statement
  - The `case` statement only defines where code execution *starts* when the variable matches a value (like an open `{`). The `break` statement defines where it *ends* (like a close `}`).

- However, there is one exception: A **case** statement with *no body* (code block) after it does not need a matching **break**
- If there is more than one value that should have the same behavior, you can write **case** statements for both values above a single block of code, with no **break** between them. If *either one* matches, the computer will execute that block of code, and then stop at the **break** statement.
- In a switch statement with this structure:

  ```
  switch(<variable>)
  {
      case <value 1>:
      case <value 2>:
          <statement block 1>
          break;
      case <value 3>:
      case <value 4>:
          <statement block 2>
          break;
      default:
          <statement block 3>
          break;
  }
  ```

  The statements in block 1 will execute if the variable matches value 1 *or* value 2, and the statements in block 2 will execute if the variable matches value 3 *or* value 4.
- For example, imagine our program needs to tell the user which season the month is in. If the month number is 1, 2, or 3, the season is the same (winter), so we can combine these 3 cases. This code will correctly initialize the string `season`:

  ```
  switch(month)
  {
      case 1:
      case 2:
      case 3:
          season = "Winter";
          break;
      case 4:
      case 5:
      case 6:
          season = "Spring";
          break;
      case 7:
      case 8:
  ```

```
    case 9:
        season = "Summer";
        break;
    case 10:
    case 11:
    case 12:
        season = "Fall";
        break;
    default:
        season = "Error!"
        break;
}
```

If month is equal to 1, execution will start at **case** 1:, but the computer will continue past **case** 2 and **case** 3 and execute season = "Winter". It will then stop when it reaches the **break**, so season gets the value "Winter". Similarly, if month is equal to 2, execution will start at **case** 2:, and continue until the **break** statement, so season will also get the value "Winter".

- This syntax allows **switch** statements to have conditions with a logical OR, equivalent to an **if** condition with an ||, like **if**(x == 1 || x == 2)
- For example, the "seasons" statement could also be written as an **if**-**else**-**if** with || operators, like this:

```
if(month == 1 || month == 2 || month == 3)
{
    season = "Winter";
}
else if(month == 4 || month == 5 || month == 6)
{
    season = "Spring";
}
else if(month == 7 || month == 8 || month == 9)
{
    season = "Summer";
}
else if(month == 10 || month == 11 || month == 12)
{
    season = "Fall"
}
else
{
    season = "Error!"
}
```

Scope and `switch`

- In C#, the scope of a variable is defined by curly braces (recall that local variables defined in a method have a scope that ends with the } at the end of the method)
- Since the **`case`** statements in a **`switch`** do not have curly braces, they are all in the same scope: the one defined by the **`switch`** statement's curly braces
- This means you cannot declare a "local" variable within a **`case`** statement – it will be in scope (visible) to all the other **`case`** statements
- For example, imagine you wanted to use a local variable named `nextMonth` to do some local computation within each case in the "months" program. This code will not work:

  ```
  switch(month)
  {
      case 1:
          int nextMonth = 2;
          monthName = "January";
          // do something with nextMonth...
          break;
      case 2:
          int nextMonth = 3;
          monthName = "February";
          // do something with nextMonth...
          break;
      //...
  }
  ```

  The line `int nextMonth = 3` would cause a compile error because a variable named `nextMonth` already exists – the one declared within `case 1`.

Limitations of `switch`

- Not all **`if-else-if`** statements can be rewritten as **`switch`** statements
- **`switch`** can only test equality, so in general, only **`if`** statements whose condition uses == can be converted to **`switch`**
- For example, imagine we have a program that determines how much of a fee to charge a rental car customer based on the number of miles the car was driven. A variable named `mileage` contains the number of miles driven, and it is used in this **`if-else-if`**

statement:

```
decimal fee = 0;
if(mileage > 1000)
{
    fee = 50.0M;
}
else if(mileage > 500)
{
    fee = 25.0M;
}
```

- This `if-else-if` statement could not be converted to `switch(mileage)` because of the condition `mileage > 1000`. The `switch` statement would need to have a `case` for each number greater than 1000, which is infinitely many.

While Loops

Introduction to `while` loops

- There are two basic types of decision structures in all programming languages. We've just learned about the first, which is the "selection structure," or `if` statement. This allows the program to choose whether or not to execute a block of code, based on a condition.
- The second basic decision structure is the loop, which allows the program to execute the same block of code repeatedly, and choose when to stop based on a condition.
- The **while statement** executes a block of code repeatedly, *as long as a condition is true*. You can also think of it as executing the code repeatedly *until a condition is false*

Example code with a `while` loop

```
int counter = 0;
while(counter <= 3)
{
    Console.WriteLine("Hello again!");
    Console.WriteLine(counter);
    counter++;
}
Console.WriteLine("Done");
```

- After the keyword `while` is a condition, in parentheses: `counter <= 3`
- On the next line after the `while` statement, the curly brace begins

a code block. The code in this block is "controlled" by the **while** statement.

- The computer will repeatedly execute that block of code as long as the condition `counter <= 3` is true
- Note that inside this block of code is the statement `counter++`, which increments `counter` by 1. So eventually, `counter` will be greater than 3, and the loop will stop because the condition is false.
- This program produces the following output:

```
Hello again!
0
Hello again!
1
Hello again!
2
Hello again!
3
Done
```

Syntax and rules for `while` loops

- Formally, the syntax for a **while** loop is this:

```
while(<condition>)
{
    <statements>
}
```

- Just like with an **if** statement, the condition is any expression that produces a `bool` value (including a `bool` variable by itself)
- When the computer encounters a **while** loop, it first evaluates the condition
- If the condition is false, the loop body (code block) is skipped, just like with an **if** statement
- If the condition is true, the loop body is executed
- After executing the loop body, the computer goes back to the **while** statement and evaluates the condition again to decide whether to execute the loop again
- Just like with an **if** statement, the curly braces can be omitted if the loop body is just one statement:

```
while(<condition>)
    <statement>
```

- Examining the example in detail
- When our example program executes, it initializes `counter` to 0, then it encounters the loop
- It evaluates the condition `counter <= 0`, which is true, so it executes the loop's body. The program displays "Hello again!" and "0" on the screen.
- At the end of the code block (after `counter++`) the program returns to the **`while`** statement and evaluates the condition again. 1 is less than 3, so it executes the loop's body again.
- This process repeats two more times, and the program displays "Hello again!" with "2" and "3"
- After displaying "3", `counter++` increments `counter` to 4. Then the program returns to the **`while`** statement and evaluates the condition, but `counter <= 3` is false, so it skips the loop body and executes the last line of code (displaying "Done")

While loops may execute zero times

- You might think that a "loop" always repeats code, but nothing requires a while loop to execute at least once
- If the condition is false when the computer first encounters the loop, the loop body is skipped
- For example, if we initialize `counter` to 5 with our previous loop:

  ```
  int counter = 5;
  while(counter <= 3)
  {
      Console.WriteLine("Hello again!");
      Console.WriteLine(counter);
      counter++;
  }
  Console.WriteLine("Done");
  ```

  The program will only display "Done," because the body of the loop never executes. `counter <= 3` is false the first time it is evaluated, so the program skips the code block and continues on the next line.

Ensuring the loop ends

- If the loop condition is always true, the loop will never end, and your program will execute "forever" (until you forcibly stop it, or the computer shuts down)

- Obviously, if you use the value **`true`** for the condition, the loop will execute forever, like in this example:

  ```
  int number = 1;
  while (true)
      Console.WriteLine(number++);
  ```

- If you do not intend your loop to execute forever, you must ensure the statements in the loop's body do something to *change a variable* in the loop condition, otherwise the condition will stay true

- For example, this loop will execute forever because the loop condition uses the variable `counter`, but the loop body does not change the value of `counter`:

  ```
  int counter = 0;
  while(counter <= 3)
  {
      Console.WriteLine("Hello again!");
      Console.WriteLine(counter);
  }
  Console.WriteLine("Done");
  ```

- This loop will also execute forever because the loop condition uses the variable `num1`, but the loop body changes the variable `num2`:

  ```
  int num1 = 0, num2 = 0;
  while(num1 <= 5)
  {
      Console.WriteLine("Hello again!");
      Console.WriteLine(num1);
      num2++;
  }
  Console.WriteLine("Done");
  ```

- It's not enough for the loop body to simply change the variable; it must change the variable in a way that will eventually *make the condition false*

  - For example, if the loop condition is `counter <= 5`, then the loop body must increase the value of `counter` so that it is eventually greater than 5

  - This loop will execute forever, even though it changes the right variable, because it changes the value in the wrong "direction":

    ```
    int number = 10;
    while(number >= 0)
    {
        Console.WriteLine("Hello again!");
    ```

```
    Console.WriteLine(number);
    number++;
}
```

The loop condition checks to see whether `number` is ≥ 0, and `number` starts out at the value 10. But the loop body increments `number`, which only moves it further away from 0 in the positive direction. In order for this loop to work correctly, we need to *decrement* `number` in the loop body, so that eventually it will be less than 0.

– This loop will execute forever, even though it uses the right variable in the loop body, because it multiplies the variable by 0:

```
int number = 0;
while (number <= 64)
{
    Console.WriteLine(number);
    number *= 2;
}
```

Since `number` was initialized to 0, `number *= 2` does not actually change the value of `number`: $2 \times 0 = 0$. So the loop body will never make the condition `number <= 64` false.

Principles of writing a while loop

- When writing a **while** loop, ask yourself these questions about your program:
  1. When (under what conditions) do I want the loop to continue?
  2. When (under what conditions) do I want the loop to stop?
  3. How will the body of the loop bring it closer to its ending condition?
- This will help you think clearly about how to write your loop condition. You should write a condition (Boolean expression) that will be **true** in the circumstances described by (1), and **false** in the circumstances described by (2)
- Keep your answer to (3) in mind as you write the body of the loop, and make sure the actions in your loop's body match the condition you wrote.

While Loop With Complex Conditions

In the following example, a complex boolean expression is used in the *while* statement. The program gets a value and tries to parse it as an integer. If the value can not be converted to an integer, the program tries again, but not more than three times.

```
int c;
string message;
int count;
bool res;

Console.WriteLine("Please enter an integer.");
message = Console.ReadLine();
res = int.TryParse(message, out c);
count = 0; // The user has 3 tries: count will be 0, 1, 2,
 ↪  and then we default.
while (!res && count < 3)
{
    count++;
    if (count == 3)
    {
        c = 1;
        Console.WriteLine("I'm using the default value
 ↪  1.");
    }
    else
    {
        Console.WriteLine("The value entered was not an
 ↪  integer.");
        Console.WriteLine("Please enter an integer.");
        message = Console.ReadLine();
        res = int.TryParse(message, out c);
    }
}
Console.WriteLine("The value is: " + c);
```

do while

Comparing `while` and `if` statements

- **`while`** and **`if`** are very similar: Both test a condition, execute a block of code if the condition is true, and skip the block of code if the condition is false
- There is only a difference if the condition is true: **`if`** statements only execute the block of code once if the condition is true, but **`while`** statements may execute the block of code multiple times if the condition is true

- Compare these snippets of code:

```
if(number < 3)
{
    Console.WriteLine("Hello!");
    Console.WriteLine(number);
    number++;
}
Console.WriteLine("Done");
```

and

```
while(number < 3)
{
    Console.WriteLine("Hello!");
    Console.WriteLine(number);
    number++;
}
Console.WriteLine("Done");
```

- If `number` is 4, then both will do the same thing: skip the block of code and display "Done".
- If `number` is 2, both will also do the same thing: Display "Hello!" and "2", then increment `number` to 3 and print "Done".
- If `number` is 1, there is a difference: The **`if`** statement will only display "Hello!" once, but the **`while`** statement will display "Hello! 2" and "Hello! 3" before displaying "Done"

Code duplication in `while` loops

- Since the **`while`** loop evaluates the condition before executing the code in the body (like an **`if`** statement), you sometimes end up duplicating code

- For example, consider an input-validation loop like the one we wrote for Item prices:

```
Console.WriteLine("Enter the item's price.");
decimal price = decimal.Parse(Console.ReadLine());
while(price < 0)
{
    Console.WriteLine("Invalid price. Please enter a
 ↪  non-negative price.");
    price = decimal.Parse(Console.ReadLine());
}
Item myItem = new Item(desc, price);
```

- Before the **`while`** loop, we wrote two lines of code to prompt the user for input, read the user's input, convert it to `decimal`, and store it in `price`

- In the body of the **`while`** loop, we also wrote two lines of code to prompt the user for input, read the user's input, convert it to `decimal`, and store it in `price`
- The code before the **`while`** loop is necessary to give `price` an initial value, so that we can check it for validity in the **`while`** statement
- It would be nice if we could tell the **`while`** loop to execute the body first, and then check the condition

Introduction to do-while

- The **`do-while`** loop executes the loop body **before** evaluating the condition
- Otherwise works the same as a **`while`** loop: If the condition is true, execute the loop body again; if the condition is false, stop the loop
- This can reduce repeated code, since the loop body is executed *at least once*
- Example:

```
decimal price;
do
{
    Console.WriteLine("Please enter a non-negative
 ↪  price.");
    price = decimal.Parse(Console.ReadLine());
} while(price < 0);
Item myItem = new Item(desc, price);
```

- The keyword **`do`** starts the code block for the loop body, but it does not have a condition, so the computer simply starts executing the body
- In the loop body, we prompt the user for input, read and parse the input, and store it in `price`
- The condition `price < 0` is evaluated at the end of the loop body, so `price` has its initial value by the time the condition is evaluated
- If the user entered a valid price, and the condition is false, execution simply proceeds to the next line
- If the user entered a negative price (the condition is true), the computer returns to the beginning of the code block and executes the loop body again

- This has the same effect as the **while** loop: the user is prompted repeatedly until he/she enters a valid price, and the program can only reach the line `Item myItem = new Item(desc, price)` when `price < 0` is false

- Note that the variable `price` must be declared before the **do-while** loop so that it is in scope after the loop. It would not be valid to declare `price` inside the body of the loop (e.g. on the line with `decimal.Parse`) because then its scope would be limited to inside that code block.

Formal syntax and details of do-while

- A **do-while** loop is written like this:

```
do
{
    <statements>
} while(<condition>);
```

- The **do** keyword does nothing, but it is required to indicate the start of the loop. You cannot just write a `{` by itself.
  - Unlike a **while** loop, a semicolon is required after `while(<condition>)`
  - It's a convention to write the **while** keyword on the same line as the closing `}`, rather than on its own line as in a **while** loop
  - When the computer encounters a **do-while** loop, it first executes the body (code block), then evaluates the condition
  - If the condition is true, the computer jumps back to the **do** keyword and executes the loop body again
  - If the condition is false, execution continues to the next line after teh **while** keyword
  - If the loop body is only a single statement, you can omit the curly braces, but not the semicolon:

```
do
<statement>
while(<condition>);
```

do-while loops with multiple conditions

- We can combine both types of user-input validation in one loop: Ensuring the user entered a number (not some other string), and ensuring the number is valid. This is easier to do with a **do-while** loop:

```
decimal price;
bool parseSuccess;
do
{
    Console.WriteLine("Please enter a price (must be
 ↪  non-negative).");
    parseSuccess = decimal.TryParse(Console.ReadLine(),
 ↪  out price);
} while(!parseSuccess || price < 0);
Item myItem = new Item(desc, price);
```

- There are two parts to the loop condition: (1) it should be true if the user did not enter a number, and (2) it should be true if the user entered a negative number.
- We combine these two conditions with `||` because either one, by itself, represents invalid input. Even if the user entered a valid number (which means `!parseSuccess` is false), the loop should not stop unless `price < 0` is also false.
- Note that both variables must be declared before the loop begins, so that they are in scope both inside and outside the loop body

Input Validation

Valid and invalid data

- Depending on the purpose of your program, each variable might have a limited range of values that are "valid" or "good," even if the data type can hold more
- For example, a `decimal` variable that holds a price (in dollars) should have a positive value, even though it is legal to store negative numbers in a `decimal`
- Consider the `Item` class, which represents an item sold in a store. It has a `price` attribute that should only store positive values:

```
class Item
{
  private string description;
  private decimal price;

  public Item(string initDesc, decimal initPrice)
  {
    description = initDesc;
    price = initPrice;
  }
  public decimal GetPrice()
  {
    return price;
  }

  public void SetPrice(decimal p)
  {
    price = p;
  }

  public string GetDescription()
  {
    return description;
  }

  public void SetDescription(string desc)
  {
    description = desc;
  }
}
```

- When you write a program that constructs an `Item` from literal values, you (the programmer) can make sure you only use positive prices. However, if you construct an `Item` based on input provided by the user, you cannot be certain that the user will follow directions and enter a valid price:

```
Console.WriteLine("Enter the item's description");
string desc = Console.ReadLine();
Console.WriteLine("Enter the item's price (must be
 ↪  positive)");
decimal price = decimal.Parse(Console.ReadLine());
Item myItem = new Item(desc, price);
```

In this code, if the user enters a negative number, the `myItem` object will have a negative price, even though that does not make sense.

- One way to guard against "bad" user input values is to use an **`if`** statement or a conditional operator, as we saw in the previous lecture (Switch and Conditional), to provide a default value if the user's input is invalid. In our example with Item, we could add a conditional operator to check whether `price` is negative before providing it to the `Item` constructor:

```
decimal price = decimal.Parse(Console.ReadLine());
Item myItem = new Item(desc, (price >= 0) ? price : 0);
```

In this code, the second argument to the `Item` constructor is the result of the conditional operator, which will be 0 if `price` is negative.

- You can also put the conditional operator inside the constructor, to ensure that an `Item` with an invalid price can never be created. If we wrote this constructor inside the `Item` class:

```
public Item(string initDesc, decimal initPrice)
{
    description = initDesc;
    price = (initPrice >= 0) ? initPrice : 0;
}
```

then the instantiation `new Item(desc, price)` would never be able to create an object with a negative price. If the user provides an invalid price, the constructor will ignore their value and initialize the `price` instance variable to 0 instead.

Ensuring data is valid with a loop

- Another way to protect your program from "bad" user input is to check whether the data is valid as soon as the user enters it, and prompt him/her to re-enter the data if it is not valid
- A `while` loop is the perfect fit for this approach: you can write a loop condition that is true when the user's input is *invalid*, and ask the user for input in the body of the loop. This means your program will repeatedly ask the user for input until he/she enters valid data.
- This code uses a `while` loop to ensure the user enters a non-negative price:

```
Console.WriteLine("Enter the item's price.");
decimal price = decimal.Parse(Console.ReadLine());
while(price < 0)
{
    Console.WriteLine("Invalid price. Please enter a
 ↪  non-negative price.");
    price = decimal.Parse(Console.ReadLine());
}
Item myItem = new Item(desc, price);
```

- The condition for the `while` loop is `price < 0`, which is true when the user's input is invalid
- If the user enters a valid price the first time, the loop will not execute at all – remember that a `while` loop will skip the code block if the condition is false
- Inside the loop's body, we ask the user for input again, and assign the result of `decimal.Parse` to the same `price` variable we use in

the loop condition. This is what ensures that the loop will end: the variable in the condition gets changed in the body.

- If the user still enters a negative price, the loop condition will be true, and the body will execute again (prompting them to try again)
- If the user enters a valid price, the loop condition will be false, so the program will proceed to the next line and instantiate the Item
- Note that the *only* way for the program to "escape" from the `while` loop is for the user to enter a valid price. This means that `new Item(desc, price)` is guaranteed to create an Item with a non-negative price, even if we did not write the constructor that checks whether `initPrice >= 0`. On the next line of code after the end of a `while` loop, you can be certain that the loop's condition is false, otherwise execution would not have reached that point.

Ensuring the user enters a number with `TryParse`

- Another way that user input might be invalid: When asked for a number, the user could enter something that is not a number
- The `Parse` methods we have been using assume that the `string` they are given (in the argument) is a valid number, and produce a run-time error if it is not
- For example, this program will crash if the user enters "hello" instead of a number:

```
Console.WriteLine("Guess a number"):
int guess = int.Parse(Console.ReadLine());
if(guess == favoriteNumber)
{
    Console.WriteLine("That's my favorite number!");
}
```

- Each built-in data type has a **TryParse method** that will *attempt* to convert a `string` to a number, but will not crash (produce a run-time error) if the conversion fails. Instead, TryParse indicates failure by returning the Boolean value `false`
- The `TryParse` method is used like this:

```
string userInput = Console.ReadLine();
int intVar;
bool success = int.TryParse(userInput, out intVar);
```

- The first parameter is a `string` to be parsed (`userInput`)
- The second parameter is an **out parameter**, and it is the name of a variable that will be assigned the result of the conversion. The

keyword **out** indicates that a method parameter is used for *output* rather than *input*, and so the variable you use for that argument will be changed by the method.

- The return type of TryParse is bool, not int, and the value returned indicates whether the input string was successfully parsed
- If the string userInput contains an integer, TryParse will assign that integer value to intVar and return **true** (which gets assigned to success)
- If the string userInput does not contain an integer, TryParse will assign 0 to intVar and return **false** (which gets assigned to success)
- Either way, the program will not crash, and intVar will be assigned a new value
- The other data types have TryParse methods that are used the same way. The code will follow this general format:

```
bool success = <numeric datatype>.TryParse(<string to
 ↪  convert>, out <numeric variable to store result>)
```

Note that the variable you use in the out parameter must be the same type as the one whose TryParse method is being called. If you write decimal.TryParse, the out parameter must be a decimal variable.

- A more complete example of using TryParse:

```
Console.WriteLine("Please enter an integer");
string userInput = Console.ReadLine();
int intVar;
bool success = int.TryParse(userInput, out intVar);
if(success)
{
    Console.WriteLine($"The value entered was an integer:
 ↪  {intVar}");
}
else
{
    Console.WriteLine($"\"{userInput}\" was not an
 ↪  integer");
}
Console.WriteLine(intVar);
```

- The TryParse method will attempt to convert the user's input to an int and store the result in intVar
- If the user entered an integer, success will be **true**, and the program will display "The value entered was an integer:" followed by

the user's value

- If the user entered some other string, `success` will be **`false`**, and the program will display a message indicating that it was not an integer
- Either way, `intVar` will be assigned a value, so it is safe to write `Console.WriteLine(intVar)`. This will display the user's input if the user entered an integer, or "0" if the user did not enter an integer.
- Just like with `Parse`, you can use `Console.ReadLine()` itself as the first argument rather than a `string` variable. Also, you can declare the output variable within the out parameter, instead of on a previous line. So we can read user input, declare an `int` variable, and attempt to parse the user's input all on one line:

```
bool success = int.TryParse(Console.ReadLine(), out int
 ↪  intVar);
```

- You can use the return value of `TryParse` in a **`while`** loop to keep prompting the user until they enter valid input:

```
Console.WriteLine("Please enter an integer");
bool success = int.TryParse(Console.ReadLine(), out int
 ↪  number);
while(!success)
{
    Console.WriteLine("That was not an integer, please
 ↪  try again.");
    success = int.TryParse(Console.ReadLine(), out
 ↪  number);
}
```

- The loop condition should be true when the user's input is *invalid*, so we use the negation operator `!` to write a condition that is true when `success` is **`false`**
- Each time the loop body executes, both `success` and `number` are assigned new values by `TryParse`

The foreach Loop

- When writing a **`for`** loop that accesses each element of an array once, you will end up writing code like this:

```
for(int i = 0; i < myArray.Length; i++)
{
    <do something with myArray[i]>;
}
```

- In some cases, this code has unnecessary repetition: If you are not using the counter `i` for anything other than an array index, you still need to declare it, increment it, and write the condition with `myArray.Length`
- The **foreach loop** is a shortcut that allows you to get rid of the counter variable and the loop condition. It has this syntax:

  ```
  foreach(<type> <variableName> in <arrayName>)
  {
      <do something with variable>
  }
  ```

  - The loop will repeat exactly as many times as there are elements in the array
  - On each iteration of the loop, the variable will be assigned the next value from the array, in order
  - The variable must be the same type as the array
- For example, this loop accesses each element of `homeworkGrades` and computes their sum:

  ```
  int sum = 0;
  foreach(int grade in homeworkGrades)
  {
      sum += grade;
  }
  ```

  - The variable `grade` is declared with type `int` since `homeworkGrades` is an array of `int`
  - `grade` has a scope limited to the body of the loop, just like the counter variable `i`
  - In successive iterations of the loop `grade` will have the value `homeworkGrades[0]`, then `homeworkGrades[1]`, and so on, through `homeworkGrades[homeworkGrades.Length - 1]`
- A `foreach` loop is **read-only** with respect to the array: The loop's variable cannot be used to *change* any elements of the array. This code will result in an error:

  ```
  foreach(int grade in homeworkGrades)
  {
      grade = int.Parse(Console.ReadLine());
  }
  ```

For Loops

Counter-controlled loops

- Previously, when we learned about loop vocabulary, we looked at counter-controlled **`while`** loops
- Although counter-controlled loops can perform many different kinds of actions in the body of the loop, they all use very similar code for managing the counter variable
- Two examples of counter-controlled **`while`** loops:

  ```
  int i = 0;
  while(i < 10)
  {
      Console.WriteLine($"{i}");
      i++;
  }
  Console.WriteLine("Done");
  ```

  ```
  int num = 1, total = 0;
  while(num <= 25)
  {
      total += num;
      num++;
  }
  Console.WriteLine($"The sum is {total}");
  ```

  Notice that in both cases, we've written the same three pieces of code:

  - Initialize a counter variable (`i` or `num`) before the loop starts
  - Write a loop condition that will become false when the counter reaches a certain value (`i < 10` or `num <= 25`)
  - Increment the counter variable at the end of each loop iteration, as the last line of the body

`for` loop example and syntax

- This **`for`** loop does the same thing as the first of the two **`while`** loops above:

  ```
  for(int i = 0; i < 10; i++)
  {
      Console.WriteLine($"{i}");
  }
  Console.WriteLine("Done");
  ```

    - The `for` statement actually contains 3 statements in 1 line; note that they are separated by semicolons
    - The code to initialize the counter variable has moved inside the `for` statement, and appears first
    - Next is the loop condition, `i < 10`
    - The third statement is the increment operation, `i++`, which no longer needs to be written at the end of the loop body

- In general, `for` loops have this syntax:

```
for(<initialization>; <condition>; <update>)
{
    <statements>
}
```

    - The initialization statement is executed once, when the program first reaches the loop. This is where you declare and initialize the counter variable.
    - The condition statement works exactly the same as a `while` loop's condition statement: Before executing the loop's body, the computer checks the condition, and skips the body (ending the loop) if it is false.
    - The update statement is code that will be executed each time the loop's body *ends*, before checking the condition again. You can imagine that it gets inserted right before the closing `}` of the loop body. This is where you increment the counter variable.

- Examining the example in detail

    - When the computer executes our example `for` loop, it first creates the variable `i` and initializes it to 0
    - Then it evaluates the condition `i < 10`, which is true, so it executes the loop's body. The computer displays "0" in the console.
    - At the end of the code block for the loop's body, the computer executes the update code, `i++`, and changes the value of `i` to 1.
    - Then it returns to the beginning of the loop and evaluates the condition again. Since it is still true, it executes the loop body again.
    - This process repeats several more times. On the last iteration, `i` is equal to 9. The computer displays "9" on the screen, then increments `i` to 10 at the end of the loop body.
    - The computer returns to the `for` statement and evaluates the condition, but `i < 10` is false, so it skips the loop body and proceeds to the next line of code. It displays "Done" in the console.

Limitations and Pitfalls of Using `for` Loops

Scope of the `for` loop's variable

- When you declare a counter variable in the **`for`** statement, its scope is limited to *inside* the loop
- Just like method parameters, it is as if the variable declaration happened just inside the opening `{`, so it can only be accessed inside that code block
- This means you cannot use a counter variable after the end of the loop. This code will produce a compile error:

```
int total = 0;
for(int count = 0; count < 10; count++)
{
    total += count;
}
Console.WriteLine($"The average is {(double) total /
 ↪  count}");
```

- If you want to use the counter variable after the end of the loop, you must declare it *before* the loop
- This means your loop's initialization statement will need to assign the variable its starting value, but not declare it
- This code works correctly, since `count` is still in scope after the end of the loop:

```
int total = 0;
int count;
for(count = 0; count < 10; count++)
{
    total += count;
}
Console.WriteLine($"The average is {(double) total /
 ↪  count}");
```

Accidentally re-declaring a variable

- If your **`for`** loop declares a new variable in its initialization statement, it cannot have the same name as a variable already in scope
- If you want your counter variable to still be in scope after the end of the loop, you cannot also declare it in the **`for`** loop. This is why we had to write `for(count = 0...` instead of `for(int count = 0...` in the previous example: the name `count` was already being used.

- Since counter variables often use short, common names (like `i` or `count`), it is more likely that you'll accidentally re-use one that's already in scope
- For example, you might have a program with many **`for`** loops, and in one of them you decide to declare the counter variable outside the loop because you need to use it after the end of the loop. This can cause an error in a different **`for`** loop much later in the program:

  ```
  int total = 0;
  int i;
  for(i = 0; i < 10; i++)
  {
      total += i;
  }
  Console.WriteLine($"The average is {(double) total /
   ↪  i}");
  // Many more lines of code
  // ...
  // Some time later:
  for(int i = 0; i < 10; i++)
  {
      Console.WriteLine($"{i}");
  }
  ```

  The compiler will produce an error on the second **`for`** loop, because the name "i" is already being used.
- On the other hand, if all of your **`for`** loops declare their variables inside the **`for`** statement, it is perfectly fine to reuse the same variable name. This code does not produce any errors:

  ```
  int total = 0;
  for(int i = 0; i < 10; i++)
  {
      total += i;
  }
  Console.WriteLine($"The total is {total}");
  // Some time later:
  for(int i = 0; i < 10; i++)
  {
      Console.WriteLine($"{i}");
  }
  ```

Accidentally double-incrementing the counter

- Now that you know about **for** loops, you may want to convert some of your counter-controlled **while** loops to **for** loops
- Remember that in a **while** loop the counter must be incremented in the loop body, but in a **for** loop the increment is part of the loop's header
- If you just convert the header of the loop and leave the body the same, you will end up incrementing the counter *twice* per iteration. For example, if you convert this **while** loop:

  ```
  int i = 0;
  while(i < 10)
  {
      Console.WriteLine($"{i}");
      i++;
  }
  Console.WriteLine("Done");
  ```

  to this **for** loop:

  ```
  for(int i = 0; i < 10; i++)
  {
      Console.WriteLine($"{i}");
      i++;
  }
  Console.WriteLine("Done");
  ```

  it will not work correctly, because `i` will be incremented by both the loop body and the loop's update statement. The loop will seem to "skip" every other value of `i`.

More Ways to use `for` Loops

Complex condition statements

- The condition in a **for** loop can be any expression that results in a `bool` value
- If the condition compares the counter to a variable, the number of iterations depends on the variable. If the variable comes from user input, the loop is also user-controlled, like in this example:

  ```
  Console.WriteLine("Enter a positive number.");
  int numTimes = int.Parse(Console.ReadLine());
  for(int c = 0; c < numTimes; c++)
  {
  ```

```
    Console.WriteLine("**********");
}
```

- The condition can compare the counter to the result of a method call. In this case, the method will get called on every iteration of the loop, since the condition is re-evaluated every time the loop returns to the beginning. For example, in this loop:

  ```
  for(int i = 1; i <= (int) myItem.GetPrice(); i++)
  {
      Console.WriteLine($"${i}");
  }
  ```

  the `GetPrice()` method of `myItem` will be called every time the condition is evaluated.

Complex update statements

- The update statement can be anything, not just an increment operation
- For example, you can write a loop that only processes the even numbers like this:

  ```
  for(int i = 0; i < 19; i += 2)
  {
      Console.WriteLine($"{i}");
  }
  ```

- You can write a loop that decreases the counter variable on every iteration, like this:

  ```
  for(int t = 10; t > 0; t--)
  {
      Console.Write($"{t}...");
  }
  Console.WriteLine("Liftoff!");
  ```

Complex loop bodies

- The loop body can contain more complex statements, including other decision structures
- **`if`** statements can be nested inside **`for`** loops, and they will be evaluated again on every iteration
- For example, in this program:

  ```
  for(int i = 0; i < 8; i++)
  {
  ```

```
    if(i % 2 == 0)
    {
        Console.WriteLine("It's my turn");
    }
    else
    {
        Console.WriteLine("It's your turn");
    }
    Console.WriteLine("Switching players...");
}
```

On even-numbered iterations, the computer will display "It's my turn" followed by "Switching players...", and on odd-numbered iterations the computer will display "It's your turn" followed by "Switching players..."

- **for** loops can contain other **for** loops. This means the "inner" loop will execute all of its iterations each time the "outer" loop executes one iteration.
- For example, this program prints a multiplication table:

```
for(int r = 0; r < 11; r++)
{
    for(int c = 0; c < 11; c++)
    {
        Console.Write($"{r} x {c} = {r * c} \t");
    }
    Console.Write("\n");
}
```

The outer loop prints the rows of the table, while the inner loop prints the columns. On a single iteration of the outer **for** loop (i.e. when `r = 2`), the inner **for** loop executes its body 11 times, using values of `c` from 0 to 10. Then the computer executes the `Console.Write("\n")` to print a newline before the next "row" iteration.

Combining `for` and `while` loops

- **while** loops are good for sentinel-controlled loops or user-input validation, and **for** loops are good for counter-controlled loops
- This program asks the user to enter a number, then uses a **for** loop to print that number of asterisks on a single line:

```
string userInput;
do
{
        Console.WriteLine("Enter a positive number, or
    ↪   \"Q\" to stop");
        userInput = Console.ReadLine();
        int inputNum;
        int.TryParse(userInput, out inputNum);
        if(inputNum > 0)
        {
            for(int c = 0; c < inputNum; c++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    } while(userInput != "Q");
```

- The sentinel value "Q" is used to end the program, so the outer **while** loop repeats until the user enters this value
- Once the user enters a number, that number is used in the condition for a **for** loop that prints asterisks using `Console.Write()`. After the **for** loop ends, we use `Console.WriteLine()` with no argument to end the line (print a newline).
- Since the user could enter either a letter or a number, we need to use `TryParse` to convert the user's input to a number
- If `TryParse` fails (because the user entered a non-number), `inputNum` will be assigned the value 0. This is also an invalid value for the loop counter, so we do not need to check whether `TryParse` returned **true** or **false**. Instead, we simply check whether `inputNum` is valid (greater than 0) before executing the **for** loop, and skip the **for** loop entirely if `inputNum` is negative or 0.

Loop Vocabulary

Variables and values can have multiple roles, but it is useful to mention three different roles in the context of loops:

Counter Variable that is incremented every time a given event occurs.

```
int i = 0; // i is a counter
while (i < 10){
    Console.WriteLine($"{i}");
    i++;
}
```

Sentinel Value A special value that signals that the loop needs to end.

```
Console.WriteLine("Give me a string.");
string ans = Console.ReadLine();
while (ans != "Quit") // The sentinel value is "Quit".
{
    Console.WriteLine("Hi!");
    Console.WriteLine("Enter \"Quit\" to quit, or
 ↪  anything else to continue.");
    ans = Console.ReadLine();
}
```

Accumulator Variable used to keep the total of several values.

```
int i = 0, total = 0;
while (i < 10){
    total += i; // total is the accumulator.
    i++;
}

Console.WriteLine($"The sum from 0 to {i} is {total}.");
```

We can have an accumulator and a sentinel value at the same time:

```
Console.WriteLine("Enter a number to sum, or \"Done\" to
 ↪  stop and print the total.");
string enter = Console.ReadLine();
int sum = 0;
while (enter != "Done")
{
    sum += int.Parse(enter);
    Console.WriteLine("Enter a number to sum, or \"Done\"
 ↪  to stop and print the total.");
    enter = Console.ReadLine();
}
Console.WriteLine($"Your total is {sum}.");
```

You can have counter, accumulator and sentinel values at the same time:

```
int a = 0;
int sum = 0;
int counter = 0;
Console.WriteLine("Enter an integer, or N to quit.");
string entered = Console.ReadLine();
while (entered != "N") // Sentinel value
{
    a = int.Parse(entered);
    sum += a; // Accumulator
    Console.WriteLine("Enter an integer, or N to quit.");
    entered = Console.ReadLine();
    counter++; // counter
}
Console.WriteLine($"The average is {sum /
 ↪  (double)counter}");
```

We can distinguish between three "flavors" of loops (that are not mutually exclusive):

Sentinel controlled loop The exit condition tests if a variable has (or is different from) a *specific value*.

User controlled loop The number of iterations depends on the *user*.

Count controlled loop The number of iterations depends on a *counter*.

Note that a user-controlled loop can be sentinel-controlled (that is the example we just saw), but also count-controlled ("Give me a value, and I will iterate a task that many times").

Combining Classes and Decision Structures

Now that we have learned about decision structures, we can revisit classes and methods. Decision structures can make our methods more flexible, useful, and functional.

Using `if` Statements with Methods

There are several ways we can use `if-else` and `if-else-if` statements with methods:

- For input validation in setters and properties
- For input validation in constructors
- With Boolean parameters to change a method's behavior
- Inside a method to evaluate instance variables

Setters with Input Validation

- Recall that getters and setters are used to implement **encapsulation**: an object's attributes (instance variables) can only be changed by code in that object's class
- For example, this Item class (which represents an item for sale in a store) has two attributes, a price and a description. Code outside the Item class (e.g. in the `Main` method) can only change these attributes by calling `SetPrice` and `SetDescription`

```
class Item
{
  private string description;
  private decimal price;

  public Item(string initDesc, decimal initPrice)
  {
    description = initDesc;
    price = initPrice;
  }

  public decimal GetPrice()
  {
    return price;
  }

  public void SetPrice(decimal p)
  {
    price = p;
  }

  public string GetDescription()
  {
    return description;
  }

  public void SetDescription(string desc)
  {
    description = desc;
  }
}
```

- Right now, it is possible to set the price to any value, including a negative number, but a negative price does not make sense. If we add an **if** statement to SetPrice, we can check that the new value is a valid price before changing the instance variable:

```
public void SetPrice(decimal p)
{
    if(p >= 0)
    {
        price = p;
    }
    else
    {
        price = 0;
    }
}
```

  - If the parameter `p` is less than 0, we do not assign it to `price`; instead we set `price` to the nearest valid value, which is 0.
  - Since code outside the Item class cannot access `price` directly, this means it is now impossible to give an item a negative price: If your code calls `myItem.SetPrice(-90m)`, `myItem`'s price will be 0, not -90.

- Alternatively, we could write a setter that simply ignores invalid values, instead of changing the instance variable to the "nearest valid" value
- For example, in the `Rectangle` class, the length and width attributes must also be non-negative. We could write a setter for width like this:

  ```
  public void SetWidth(int newWidth)
  {
      if(newWidth >= 0)
      {
          width = newWidth
      }
  }
  ```

  - This means if `myRectangle` has a width of 6, and your code calls `myRectangle.SetWidth(-18)`, then `myRectangle` will still have a width of 6.

- A setter with input validation is a good example of where a conditional operator can be useful. We can write the `SetPrice` method with one line of code using a conditional operator:

  ```
  public void SetPrice(decimal p)
  {
      price = (p >= 0) ? p : 0;
  }
  ```

  The instance variable `price` is assigned to the result of the conditional operator, which is either `p`, if `p` is a valid price, or 0, if `p` is not a valid price.

- If you have a class that uses properties instead of getters and setters, the same kind of validation can be added to the `set` component of a property
  - For example, the "price" attribute could be implemented with a property like this:

    ```
    public decimal Price
    {
    ```

```
        get
        {
            return price;
        }
        set
        {
            price = value;
        }
    }
```

  - We can add an **if** statement or a conditional operator to the set accessor to ensure the price is not set to a negative number:

    ```
    public decimal Price
    {
        get
        {
            return price;
        }
        set
        {
            price = (value >= 0) ? value : 0;
        }
    }
    ```

- If a class's attributes have a more limited range of valid values, we might need to write a more complex condition in the setter. For example, consider the Time class:

  ```
  class Time
  {
      private int hours;
      private int minutes;
      private int seconds;
  }
  ```

- In a Time object, hours can be any non-negative number, but minutes and seconds must be between 0 and 59 for it to represent a valid time interval

- The SetMinutes method can be written as follows:

  ```
  public void SetMinutes(int newMinutes)
  {
      if(newMinutes >= 0 && newMinutes < 60)
      {
          minutes = newMinutes;
      }
  ```

```
        else if(newMinutes >= 60)
        {
            minutes = 59;
        }
        else
        {
            minutes = 0;
        }
    }
```

  - If the parameter `newMinutes` is between 0 and 59 (both greater than or equal to 0 and less than 60), it is valid and can be assigned to `minutes`
  - If `newMinutes` is 60 or greater, we set `minutes` to the largest possible value, which is 59
  - If `newMinutes` is less than 0, we set `minutes` to the smallest possible value, which is 0
  - Note that we need an if-else-if statement because there are two different ways that `newMinutes` can be invalid (too large or too small) and we need to distinguish between them. When the condition `newMinutes >= 0 && newMinutes < 60` is false, it could either be because `newMinutes` is less than 0 or because `newMinutes` is greater than 59. The **`else if`** clause tests which of these possibilities is true.

Constructors with Input Validation

- A constructor's job is to initialize the object's instance variables, so it is very similar to a "setter" for all the instance variables at once

- If the constructor uses parameters to initialize the instance variables, it can use **`if`** statements to ensure the instance variables are not initialized to "bad" values

- Returning to the `Item` class, this is how we could write a 2-argument constructor that initializes the price to 0 if the parameter `initPrice` is not a valid price:

  ```
  public Item(string initDesc, decimal initPrice)
  {
      description = initDesc;
      price = (initPrice >= 0) ? initPrice : 0;
  }
  ```

  With both this constructor and the `SetPrice` method we wrote earlier, we can now guarantee that it is impossible for an Item object to have a negative price. This will make it easier to write a large program that uses many Item objects without introducing bugs: the program cannot accidentally reduce an item's price below 0, and it can add up the prices of all the items and be sure to get the correct answer.

- Recall the `ClassRoom` class from an earlier lecture, which has a room number as one of its attributes. If we know that no classroom building has more than 3 floors, then the room number must be between 100 and 399. The constructor for `ClassRoom` could check that the room number is valid using an if-else-if statement, as follows:

```
public ClassRoom(string buildingParam, int
 ↪  numberParam)
{
    building = buildingParam;
    if(numberParam >= 400)
    {
        number = 399;
    }
    else if(numberParam < 100)
    {
        number = 100;
    }
    else
    {
        number = numberParam;
    }
}
```

  - Here, we have used similar logic to the `SetMinutes` method of the Time class, but with the conditions tested in the opposite order
  - First, we check if `numberParam` is too large (greater than 399), and if so, initialize `number` to 399
  - Then we check if `numberParam` is too small (less than 100), and if so, initialize `number` to 100
  - If both of these conditions are false, it means `numberParam` is a valid room number, so we can initialize `number` to `numberParam`

- The `Time` class also needs a constructor that checks if its parameters are within a valid range, since both minutes and seconds must be between 0 and 59

- However, with this class we can be "smarter" about the way we handle values that are too large. If a user attempts to construct a Time object with a value of 0 hours and 75 minutes, the constructor could "correct" this to 1 hour and 15 minutes and initialize the Time object with these equivalent values. In other words, this code:

```
Time classTime = new Time(0, 75, 0);
Console.WriteLine($"{classTime.GetHours()} hours,
 ↪  {classTime.GetMinutes()} minutes");
```

should produce the output "1 hours, 15 minutes", not "0 hours, 59 minutes"

- Here's a first attempt at writing the Time constructor:

```
public Time(int hourParam, int minuteParam, int
 ↪  secondParam)
{
    hours = (hourParam >= 0) ? hourParam : 0;
    if(minuteParam >= 60)
    {
        minutes = minuteParam % 60;
        hours += minuteParam / 60;
    }
    else if(minuteParam < 0)
    {
        minutes = 0;
    }
    else
    {
        minutes = minuteParam;
    }
    if(secondParam >= 60)
    {
        seconds = secondParam % 60;
        minutes += secondParam / 60;
    }
    else if(secondParam < 0)
    {
        seconds = 0;
    }
    else
    {
        seconds = secondParam;
    }
}
```

  - First, we initialize `hours` using `hourParam`, unless `hourParam` is negative. There is no upper limit on the value of `hours`
  - If `minuteParam` is 60 or greater, we perform an integer division by 60 and add the result to `hours`, while using the remainder after dividing by 60 to initialize `minutes`. This separates the value

  into a whole number of hours and a remaining, valid, number of minutes. Since `hours` has already been initialized, it is important to use += (to add to the existing value).
- Similarly, if `secondParam` is 60 or greater, we divide it into a whole number of minutes and a remaining number of seconds, and add the number of minutes to `minutes`
- With all three parameters, any negative value is replaced with 0

- This constructor has an error, however: If `minuteParam` is 59 and `secondParam` is 60 or greater, `minutes` will be initialized to 59, but then the second if-else-if statement will increase `minutes` to 60. There are two ways we can fix this problem.

  - One is to add a nested **`if`** statement that checks if `minutes` has been increased past 59 by `secondParam`:

```
public Time(int hourParam, int minuteParam, int
 ↪   secondParam)
{
    hours = (hourParam >= 0) ? hourParam : 0;
    if(minuteParam >= 60)
    {
        minutes = minuteParam % 60;
        hours += minuteParam / 60;
    }
    else if(minuteParam < 0)
    {
        minutes = 0;
    }
    else
    {
        minutes = minuteParam;
    }
    if(secondParam >= 60)
    {
        seconds = secondParam % 60;
        minutes += secondParam / 60;
        if(minutes >= 60)
        {
            hours += minutes / 60;
            minutes = minutes % 60;
        }
    }
    else if(secondParam < 0)
    {
        seconds = 0;
    }
    else
    {
        seconds = secondParam;
    }
}
```

- Another is to use the AddMinutes method we have already written to increase minutes, rather than the += operator, because this method ensures that minutes stays between 0 and 59 and increments hours if necessary:

  ```
  public Time(int hourParam, int minuteParam, int
   ↪  secondParam)
  {
      hours = (hourParam >= 0) ? hourParam : 0;
      if(minuteParam >= 60)
      {
          AddMinutes(minuteParam);
      }
      else if(minuteParam < 0)
      {
          minutes = 0;
      }
      else
      {
          minutes = minuteParam;
      }
      if(secondParam >= 60)
      {
          seconds = secondParam % 60;
          AddMinutes(secondParam / 60);
      }
      else if(secondParam < 0)
      {
          seconds = 0;
      }
      else
      {
          seconds = secondParam;
      }
  }
  ```

  Note that we can also use AddMinutes in the first **if** statement, since it will perform the same integer division and remainder operations that we originally wrote for minuteParam.

Boolean Parameters

- When writing a method, we might want a single method to take one of two different actions depending on some condition, instead of doing the same thing every time. In this case we can declare the method with a `bool` parameter, whose value represents whether the method should (true) or should not (false) have a certain behavior.
- For example, in the `Room` class we wrote in lab, we wrote two separate methods to compute the area of the room: `ComputeArea()` would compute and return the area in meters, while `ComputeAreaFeet()` would compute and return the area in feet. Instead, we could write a single method that computes the area in either feet or meters depending on a parameter:

```
public double ComputeArea(bool useMeters)
{
    if(useMeters)
        return length * width;
    else
        return GetLengthFeet() * GetWidthFeet();
}
```

  - If the `useMeters` parameter is **`true`**, this method acts like the original ComputeArea method and returns the area in meters
  - If the `useMeters` parameter is **`false`**, this method acts like ComputeAreaFeet and returns the area in feet
  - We can use the method like this:

```
Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
if(userChoice == 'f')
{
    Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(false)}");
}
else if(userChoice == 'm')
{
    Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(true)}");
}
else
{
    Console.WriteLine("Invalid choice");
}
```

Regardless of whether the user requests feet or meters, we can call the same method. Instead of calling `ComputeAreaFeet()` when the user requests the area in feet, we call `ComputeArea(false)`

- Note that the `bool` argument to `ComputeArea` can be any expression that results in a Boolean value, not just true or false. This means that we can actually eliminate the **`if`** statement from the previous example:

```
Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
bool wantsMeters = userChoice == 'm';
Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(wantsMeters)}");
```

  The expression `userChoice == 'm'` is true if the user has requested to see the area in meters. Instead of testing this expression in an **`if`** statement, we can simply use it as the argument to `ComputeArea` – if the expression is true, we should call `ComputeArea(true)` to get the area in meters.

- Constructors are also methods, and we can add Boolean parameters to constructors as well, if we want to change their behavior. Remember that the parameters of a constructor do not need to correspond directly to instance variables that the constructor will initialize.

- For example, in the lab we wrote two different constructors for the `Room` class: one that would interpret its parameters as meters, and one that would interpret its parameters as feet. Since parameter names ("meters" or "feet") are not part of a method's signature, we ensured the two constructors had different signatures by omitting the "name" parameter from the feet constructor.

  – Meters constructor:

```
public Room(double lengthMeters, double
 ↪  widthMeters, string initName)
```

  – Feet constructor:

```
public Room(double lengthFeet, double widthFeet)
```

  – The problem with this approach is that the feet constructor cannot initialize the name of the room; if we gave it a `string` parameter for the room name, it would have the same signature as the meters constructor.

  – Using a Boolean parameter, we can write a single constructor that accepts either meters or feet, and is equally capable of initializing the name attribute in both cases:

```
public Room(double lengthP, double widthP, string
 ↪   nameP, bool meters)
{
    if(meters)
    {
        length = lengthP;
        width = widthP;
    }
    else
    {
        length = lengthP * 0.3048;
        width = widthP * 0.3048;
    }
    name = nameP;
}
```

- If the parameter `meters` is true, this constructor interprets the length and width parameters as meters (acting like the previous "meters constructor"), but if `meters` is false, this constructor interprets the length and width parameters as feet (acting like the previous "feet constructor").

Ordinary Methods Using `if`

- Besides enhancing our "setter" methods, we can also use **`if`** statements to write other methods that change their behavior based on conditions
- For example, we could add a `GetFloor` method to `ClassRoom` that returns a string describing which floor the classroom is on. This looks very similar to the example **`if`**-**`else`**-**`if`** statement we wrote in a previous lecture, but inside the `ClassRoom` class rather than in a `Main` method:

```
public string GetFloor()
{
    if(number >= 300)
    {
        return "Third floor";
    }
    else if(number >= 200)
    {
        return "Second floor";
    }
    else if(number >= 100)
```

```
        {
            return "First floor";
        }
        else
        {
            return "Invalid room";
        }
    }
```

   - Now we can replace the **if-else-if** statement in the `Main` method with a single statement: `Console.WriteLine(myRoom.GetFloor());`

- We can add a `MakeCube` method to the `Prism` class that transforms the prism into a cube by "shrinking" two of its three dimensions, so that all three are equal to the smallest dimension. For example, if `myPrism` is a prism with length 4, width 3, and depth 6, `myPrism.MakeCube()` should change its length and depth to 3.

```
public void MakeCube()
{
    if(length <= width && length <= depth)
    {
        width = length;
        depth = length;
    }
    else if(width <= length && width <= depth)
    {
        length = width;
        depth = width;
    }
    else
    {
        length = depth;
        width = depth;
    }
}
```

   - This **if-else-if** statement first checks to see if `length` is the smallest dimension, and if so, sets the other two dimensions to be equal to `length`
   - Similarly, if `width` is the smallest dimension, it sets both other dimensions to `width`
   - No condition is necessary in the **else** clause, because one of the three dimensions must be the smallest. If the first two conditions are false, `depth` must be the smallest dimension.
   - Note that we need to use <= in both comparisons, not <: if `length` is equal to `width`, but smaller than `depth`, we should still set all dimensions to be equal to `length`

Boolean Instance Variables

- A class might need a `bool` instance variable if it has an attribute that can only be in one of two states, e.g. on/off, feet/meters, on sale/not on sale
- For example, we can add an instance variable called "taxable" to the Item class to indicate whether or not the item should have sales tax added to its price at checkout. The UML diagram for Item with this instance variable would look like this:

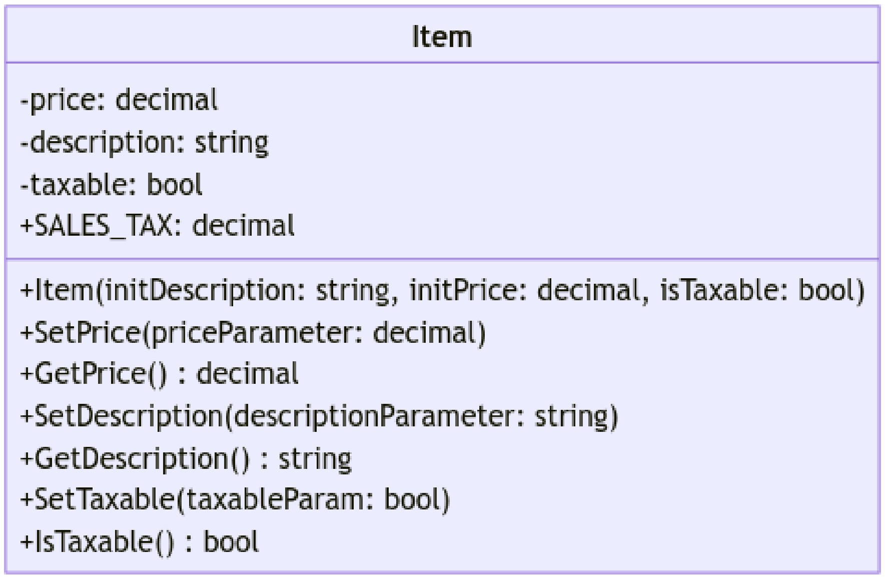

Item
-price: decimal
-description: string
-taxable: bool
+SALES_TAX: decimal
+Item(initDescription: string, initPrice: decimal, isTaxable: bool)
+SetPrice(priceParameter: decimal)
+GetPrice() : decimal
+SetDescription(descriptionParameter: string)
+GetDescription() : string
+SetTaxable(taxableParam: bool)
+IsTaxable() : bool

Figure 24: A UML diagram for the Item class (text version[340])

  - Note that the "getter" for a Boolean variable is conventionally named with a word like "Is" or "Has" rather than "Get"
  - We will add a constant named SALES_TAX to the Item class to store the sales tax rate that should be applied if the item is taxable. The sales tax rate is not likely to change during the program's execution, but it is better to store it in a named variable instead of writing the same literal value (e.g. `0.08m`) every time we want to compute a total price with tax.

- The instance variables and constructor for `Item` now look like this:

```
class Item
{
    private string description;
    private decimal price;
    private bool taxable
    public const decimal SALES_TAX = 0.08m;

    public Item(string initDesc, decimal initPrice,
     ↪  bool isTaxable)
    {
        description = initDesc;
        price = (initPrice >= 0) ? initPrice : 0;
        taxable = isTaxable;
    }
...
}
```

- We can use this instance variable in a `Main` method to compute the final price of an Item based on whether or not it is taxable:

  ```
  Item myItem = new Item("Blue Polo Shirt", 19.99m,
   ↪  true);
  decimal totalPrice = myItem.GetPrice();
  if(myItem.isTaxable())
  {
      totalPrice = totalPrice + (totalPrice *
   ↪  Item.SALES_TAX);
  }
  Console.WriteLine($"Final price: {totalPrice:C}");
  ```

- However, if we were writing a program that handled large numbers of items, we might find it tedious to write this **`if`** statement every time. To make it easier to compute the "real" (with tax) price of an item, we could instead modify the `GetPrice()` method to automatically include sales tax if applicable:

  ```
  public decimal GetPrice()
  {
      if(taxable)
          return price + (price * SALES_TAX);
      else
          return price;
  }
  ```

  Now, `myItem.GetPrice()` will return the price with tax if the item is taxable, so our `Main` method can simply use `myItem.GetPrice()` as the total price without needing to check `myItem.isTaxable()`.

Using `while` Loops with Classes

There are several ways that **`while`** loops are useful when working with classes and methods:

- To validate input before calling a method
- Inside a method, to interact with the user
- Inside a method, to take repeated action based on the object's attributes
- To control program behavior based on the return value of a method

Input Validation with Objects

- As we have seen in a previous section (Loops and Input Validation), **while** loops can be used with the `TryParse` method to repeatedly prompt the user for input until he/she enters a valid value

- This is a useful technique to use before initializing an object's attributes with user-provided data

- For example, the length and width of a `Rectangle` object should be non-negative integers. If we want to create a `Rectangle` with a length and width provided by the user, we can use a **while** loop for each attribute to ensure the user enters valid values before constructing the `Rectangle`.

```
int length, width;
bool isInt;
do
{
    Console.WriteLine("Enter a positive length");
    isInt = int.TryParse(Console.ReadLine(), out
 ↪  length);
} while(!isInt || length < 0);
do
{
    Console.WriteLine("Enter a positive width");
    isInt = int.TryParse(Console.ReadLine(), out
 ↪  width);
} while(!isInt || width < 0);
Rectangle myRectangle = new Rectangle(length, width);
```

  - Each loop asks the user to enter a number, and repeats if the user enters a non-integer (`TryParse` returns **false**) or enters a negative number (`length` or `width` is less than 0).
  - Note that we can re-use the `bool` variable `isInt` to contain the return value of `TryParse` in the second loop, since it would otherwise have no purpose or meaning after the first loop ends.
  - After both loops have ended, we know that `length` and `width` are sensible values to use to construct a `Rectangle`

- Similarly, we can use **while** loops to validate user input before calling a non-constructor method that takes arguments, such as

Rectangle's Multiply method or Item's SetPrice method

- For example, if a program has an already-initialized Item object named myItem and wants to use SetPrice to change its price to a user-provided value, we can use a **while** loop to keep prompting the user for input until he/she enters a valid price.

```
bool isNumber;
decimal newPrice;
do
{
    Console.WriteLine($"Enter a new price for
 ↪  {myItem.GetDescription()}");
    isNumber = decimal.TryParse(Console.ReadLine(),
 ↪  out newPrice);
} while(!isNumber || newPrice < 0);
myItem.SetPrice(newPrice);
```

  - Like with our previous example, the **while** loop's condition will be **true** if the user enters a non-numeric string, or a negative value. Thus the loop will only stop when newPrice contains a valid price provided by the user.
  - Although it is "safe" to pass a negative value as the argument to SetPrice, now that we added an **if** statement to SetPrice, it can still be useful to write this **while** loop
  - The SetPrice method will use a default value of 0 if its argument is negative, but it will not alert the user that the price they provided is invalid or give them an opportunity to provide a new price

- The ComputeArea method that we wrote earlier for the Room class demonstrates another situation where it is useful to write a **while** loop before calling a method
  - Note that in the version of the code that passes the user's input directly to the ComputeArea method, instead of using an **if-else-if** statement, there is nothing to ensure the user enters one of the choices "f" or "m":

```
Console.WriteLine("Compute area in feet (f) or
 ↪  meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
Console.WriteLine($"Area:
 ↪  {myRoom.ComputeArea(userChoice == 'm')}");
```

  - This means that if the user enters a multiple-letter string the program will crash (char.Parse throws an exception if its input string is larger than one character), and if the user enters a letter other than "m" the program will act as if he/she entered "f"

- Instead, we can use `TryParse` and a **`while`** loop to ensure that `userChoice` is either "f" or "m" and nothing else

  ```
  bool validChar;
  char userChoice;
  do
  {
      Console.WriteLine("Compute area in feet (f) or
   ↪  meters (m)?");
      validChar = char.TryParse(Console.ReadLine(),
   ↪  out userChoice);
  } while(!validChar || !(userChoice == 'f' ||
   ↪  userChoice == 'm'));
  Console.WriteLine($"Area:
   ↪  {myRoom.ComputeArea(userChoice == 'm')}");
  ```

- This loop will prompt the user for input again if `TryParse` returns **`false`**, meaning he/she did not enter a single letter. It will also prompt again if the user's input was not equal to `'f'` or `'m'`.

- Note that we needed to use parentheses around the expression `!(userChoice == 'f' || userChoice == 'm')` in order to apply the `!` operator to the entire "OR" condition. This represents the statement "it is not true that userChoice is equal to 'f' or 'm'." We could also write this expression as `(userChoice != 'f' && userChoice != 'm')`, which represents the equivalent statement "userChoice is not equal to 'f' and also not equal to 'm'."

Using Loops Inside Methods

- A class's methods can contain **`while`** loops if they need to execute some code repeatedly. This means that when you call such a method, control will not return to the `Main` program until the loop has stopped.

- Reading input from the user, validating it, and using it to set the attributes of an object is a common task in the programs we have been writing. If we want to do this for several objects, we might end up writing many very similar **`while`** loops in the `Main` method. Instead, we could write a method that will read and validate user input for an object's attribute every time it is called.

  - For example, we could add a method `SetLengthFromUser` to the `Rectangle` class:

```
public void SetLengthFromUser()
{
    bool isInt;
    do
    {
        Console.WriteLine("Enter a positive
 ↪  length");
        isInt = int.TryParse(Console.ReadLine(),
 ↪  out length);
    } while(!isInt || length < 0);
}
```

  - This method is similar to a setter, but it has no parameters because its only input comes from the user
  - The **`while`** loop is just like the one we wrote before constructing a `Rectangle` in a previous example, except the **`out`** parameter of `TryParse` is the instance variable `length` instead of a local variable in the `Main` method
  - `TryParse` will assign the user's input to the `length` instance variable when it succeeds, so by the time the loop ends, the Rectangle's length has been set to the user-provided value
  - Assuming we wrote a similar method `SetWidthFromUser()` (substituting `width` for `length` in the code), we would use these methods in the `Main` method like this:

    ```
    Rectangle rect1 = new Rectangle();
    Rectangle rect2 = new Rectangle();
    rect1.SetLengthFromUser();
    rect1.SetWidthFromUser();
    rect2.SetLengthFromUser();
    rect2.SetWidthFromUser();
    ```

    After executing this code, both `rect1` and `rect2` have been initialized with length and width values the user entered.

- Methods can also contain **`while`** loops that are not related to validating input. A method might use a **`while`** loop to repeat an action several times based on the object's instance variables.
  - For example, we could add a method to the `Rectangle` class that will display the Rectangle object as a rectangle of asterisks on the screen:

    ```
    public void DrawInConsole()
    {
        int counter = 1;
        while(counter <= width * length)
    ```

```
        {
            Console.Write(" * ");
            if(counter % width == 0)
            {
                Console.WriteLine();
            }
            counter++;
        }
    }
```

  - This **while** loop prints a number of asterisks equal to the area of the rectangle. Each time it prints `width` of them on the same line, it adds a line break with `WriteLine()`.

Using Methods to Control Loops

- Methods can return Boolean values, as we showed previously in the section on Boolean instance variables
- Other code can use the return value of an object's method in the loop condition of a **while** loop, so the loop is controlled (in part) by the state of the object
- For example, recall the `Time` class, which stores hours, minutes, and seconds in instance variables.
  - In a previous example we wrote a `GetTotalSeconds()` method to convert these three instance variables into a single value:

    ```
    public int GetTotalSeconds()
    {
        return hours * 60 * 60 + minutes * 60 +
         ↪  seconds;
    }
    ```

  - We can now write a method `ComesBefore` that compares two Time objects:

    ```
    public bool ComesBefore(Time otherTime)
    {
        return GetTotalSeconds() <
         ↪  otherTime.GetTotalSeconds();
    }
    ```

    This method will return **true** if the calling object (i.e. **this** object) represents a smaller amount of time than the other Time object passed as an argument

- Since it returns a Boolean value, we can use the `ComesBefore` method to control a loop. Specifically, we can write a program that asks the user to enter a Time value that is smaller than a specified maximum, and use `ComesBefore` to validate the user's input.

  ```
  Time maximumTime = new Time(2, 45, 0);
  Time userTime;
  do
  {
      Console.WriteLine($"Enter a time less than
   ↪  {maximumTime}");
      int hours, minutes, seconds;
      do
      {
          Console.Write("Enter the hours: ");
      } while(!int.TryParse(Console.ReadLine(), out
       ↪  hours));
      do
      {
          Console.Write("Enter the minutes: ");
      } while(!int.TryParse(Console.ReadLine(), out
       ↪  minutes));
      do
      {
          Console.Write("Enter the seconds: ");
      } while(!int.TryParse(Console.ReadLine(), out
       ↪  seconds));
      userTime = new Time(hours, minutes, seconds);
  } while(!userTime.ComesBefore(maximumTime));
  //At this point, userTime is valid Time object
  ```

- Note that there are **`while`** loops to validate each number the user inputs for hours, minutes, and seconds, as well as an outer **`while`** loop that validates the Time object as a whole.

- The outer loop will continue until the user enters values that make `userTime.ComesBefore(maximumTime)` return **`true`**.

Examples

The Room Class

The class and its associated `Main` method presented in this archive[341] show how you can use classes, methods, constructors and decision struc-

[341] https:/princomp.github.io/code/projects/Room.zip

tures all in the same program. It also exemplifies how a method can take *an object* as a parameter with `InSameBuilding`.

The corresponding UML diagram is:

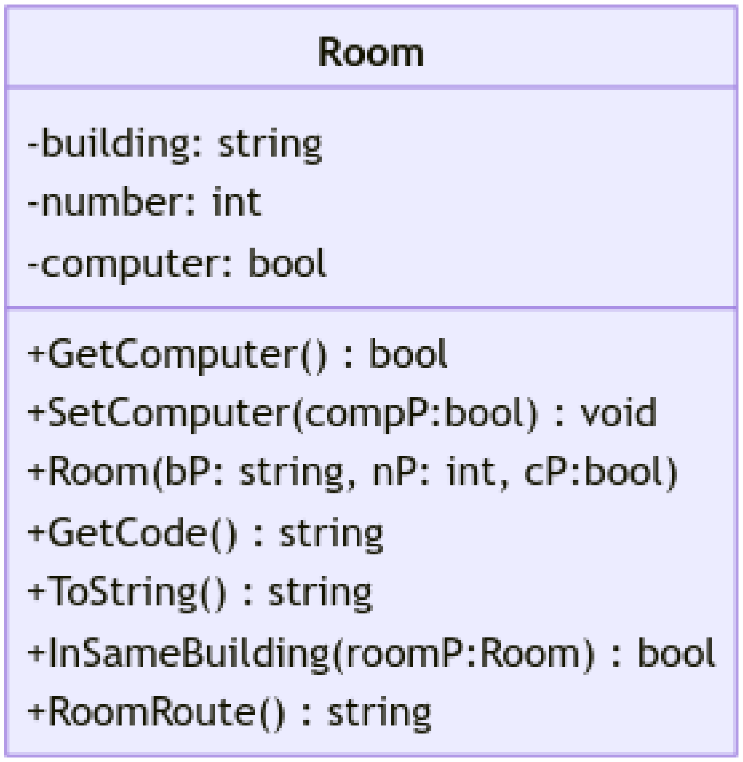


Figure 25: A UML diagram for the Room class (text version[342])

The Loan Class

Similarly, this class and its associated `Main` method show how you can use classes, methods, constructors, decision structures, and user input validation all in the same program. This lab[343] asks you to add the user input validation code, and you can download the following code in this archive[344].

```
/*
 * Application program for the "Loan" class.
```

[343]https:/princomp.github.io/labs/ValidatingInput
[344]https:/princomp.github.io/code/projects/LoanCalculator.zip

```
 * This program gathers from the user all the information
 ↪  needed
 * to create a "proper" Loan object.
*/

using System;

class Program
{
  static void Main()
  {
    Console.WriteLine("What is your name?");
    string name = Console.ReadLine();

    Console.WriteLine(
      "Do you want a loan for an Auto (A, a), a House (H,
       ↪  h), or for some Other (O, o) reason?"
    );
    char type = Console.ReadKey().KeyChar; // This part
     ↪  of the code reads *a char* from the user.
    // We haven't studied it, but it's pretty
     ↪  straightforward.
    Console.WriteLine();

    /*
     * The part of the code that follows
     * does the convertion from the character
     * to the corresponding string.
     * We could have a method in the Loan
     * class that does it for us, but
     * we'll just do it "by hand" here
     * for simplicity.
     */
    string typeOfLoan;

    if (type == 'A' || type == 'a')
    {
      type = 'a';
      typeOfLoan = "an auto";
    }
    else if (type == 'H' || type == 'h')
    {
      type = 'h';
      typeOfLoan = "a house";
    }
    else
    {
      type = 'o';
      typeOfLoan = "some other reason";
    }

    // We display the information back to the user, and
     ↪  ask the next question:
    Console.WriteLine(
      $"{name}, you need money for {typeOfLoan},
 ↪  great.\nWhat is your current credit score?"
    );
    int cscore = int.Parse(Console.ReadLine());

    Console.WriteLine("How much do you need, total?");
    decimal need = decimal.Parse(Console.ReadLine());

    Console.WriteLine("What is your down payment?");
    decimal down = decimal.Parse(Console.ReadLine());

    Loan myLoan = new Loan(name, type, cscore, need,
 ↪  down);
    Console.WriteLine(myLoan);
  }
}

/*
 * "Loan" class.
 * This class helps primarily in computing
 * an APR based on information provided from the user.
 * A ToString method is provided.
 */
using System;

class Loan
{
  private string name; // For the name of the loan holder.
  private char type; // For the type ('a'uto, 'h'ouse or
   ↪  'o'ther) of the loan
  private int cscore; // For the credit score.
  private decimal amount; // For the amount of money
   ↪  loaned.
  private decimal rate; // For the A.P.R., the interest
   ↪  rate.

  /*
   * Our constuctor will compute the amount and the rate
   * based on the information given as arguments.
   * The name, type and credit score will simply be given
↪   as arguments.
   */
  public Loan(
    string nameP,
    char typeP,
    int cscoreP,
    decimal needP,
    decimal downP
  )
  {
    name = nameP;
    type = typeP;
    cscore = cscoreP;
    if (cscore < 421)
    {
      Console.WriteLine(
        "Sorry, we can't accept your application."
      );
      amount = -1;
      rate = -1;
    }
    else
    {
      amount = needP - downP;

      switch (type)
      {
        case ('a'):
          rate = .05M;
          break;

        case ('h'):
          if (cscore > 600 && amount < 1000000M)
            rate = .03M;
          else
            rate = .04M;
          break;

        case ('o'):
          if (cscore > 650 || amount < 10000M)
            rate = .07M;
          else
            rate = .09M;
          break;
      }
    }
  }

  public override string ToString()
  {
    string typeName = "";
    switch (type)
    {
      case ('a'):
        typeName = "an auto";
        break;

      case ('h'):
        typeName = "a house";
        break;
      case ('o'):
        typeName = "another reason";
        break;
    }
    return "Dear "
      + name
      + $", you borrowed {amount:C} at {rate:P} for "
      + typeName
      + ".";
  }
}
```

Break and continue

Conditional iteration

- Sometimes, you want to write a loop that will skip some iterations if a certain condition is met
- For example, you may be writing a **for** loop that iterates through an array of numbers, but you only want to use *even* numbers from the array
- One way to accomplish this is to nest an **if** statement inside the **for** loop that checks for the desired condition. For example:

```
int sum = 0;
for(int i = 0; i < myArray.Length; i++)
{
    if(myArray[i] % 2 == 0)
    {
        Console.WriteLine(myArray[i]);
        sum += myArray[i];
    }
}
```

Since the entire body of the **for** loop is contained within an **if** statement, the iterations where `myArray[i]` is odd will skip the body and do nothing.

Skipping iterations with `continue`

- The **continue** keyword provides another way to conditionally skip an iteration of a loop
- When the computer encounters a **continue**; statement, it immediately returns to the beginning of the current loop, skipping the rest of the loop body
- Then it executes the update statement (if the loop is a **for** loop) and checks the loop condition again
- A **continue**; statement inside an **if** statement will end the current iteration only if that condition is true
- For example, this code will skip the odd numbers in `myArray` and use only the even numbers:

```
int sum = 0;
for(int i = 0; i < myArray.Length; i++)
{
    if(myArray[i] % 2 != 0)
        continue;
    Console.WriteLine(myArray[i]);
    sum += myArray[i];
}
```

If `myArray[i]` is odd, the computer will execute the **continue** statement and immediately start the next iteration of the loop. This means that the rest of the loop body (the other two statements) only gets executed if `myArray[i]` is even.

- Using a **continue** statement instead of putting the entire body within an **if** statement can reduce the amount of indentation in your code, and it can sometimes make your code's logic clearer.

Loops with multiple end conditions

- More advanced loops may have multiple conditions that affect whether the loop should continue

- Attempting to combine all of these conditions in the loop condition (i.e. the expression after **while**) can make the loop more complicated

- For example, consider a loop that processes user input, which should end either when a sentinel value is encountered or when the input is invalid. This loop ends if the user enters a negative number (the sentinel value) or a non-numeric string:

```
int sum = 0, userNum = 0;
bool success = true;
while(success && userNum >= 0)
{
    sum += userNum;
    Console.WriteLine("Enter a positive number to add it.
 ↪  "
    + "Enter anything else to stop.");
    success = int.TryParse(Console.ReadLine(), out
 ↪  userNum);
}
Console.WriteLine($"The sum of your numbers is {sum}");
```

- The condition `success && userNum >= 0` is true if the user entered a valid number that was not negative
- In order to write this condition, we needed to declare the extra variable `success` to keep track of the result of `int.TryParse`
- We cannot use the condition `userNum > 0`, hoping to take advantage of the fact that if `TryParse` fails it assigns its **out** parameter the value 0, because 0 is a valid input the user could give

Ending the loop with break

- The **break** keyword provides another way to write an additional end condition

- When the computer encounters a `break;` statement, it immediately ends the loop and proceeds to the next statement after the loop body

- This is the same **break** keyword we used in **switch** statements

- In both cases it has the same meaning: stop execution here and skip to the end of this code block (the ending } for the **switch** or the loop)

- Using a **break** statement inside an **if-else** statement, we can rewrite the previous **while** loop so that the variable `success` is not needed:

```
int sum = 0, userNum = 0;
while(userNum >= 0)
{
    sum += userNum;
    Console.WriteLine("Enter a positive number to add it.
↪   "
    + "Enter anything else to stop.");
    if(!int.TryParse(Console.ReadLine(), out userNum))
        break;
}
Console.WriteLine($"The sum of your numbers is {sum}");
```

- Inside the body of the loop, the return value of `TryParse` can be used directly in an **`if`** statement instead of assigning it to the `success` variable
- If `TryParse` fails, the **`break`** statement will end the loop, so there is no need to add `success` to the **`while`** condition
- We can also use the **`break`** statement with a **`for`** loop, if there are some cases where the loop should end before the counter reaches its last value
- For example, imagine that our program is given an `int` array that a user *partially* filled with numbers, and we need to find their product. The "unused" entries at the end of the array are all 0 (the default value of `int`), so the **`for`** loop needs to stop before the end of the array if it encounters a 0. A **`break`** statement can accomplish this:

```
int product = 1;
for(int i = 0; i < myArray.Length; i++)
{
    if(myArray[i] == 0)
        break;
    product *= myArray[i];
}
```

- If `myArray[i]` is 0, the loop stops before it can multiply the product by 0
- If all of the array entries are nonzero, though, the loop continues until `i` is equal to `myArray.Length`
- Note that in this example, we access each array element once and do not modify them, so we could also write it with a **`foreach`** loop:

```
int product = 1;
foreach(int number in myArray)
{
    if(number == 0)
```

```
        break;
    product *= number;
}
```

The Conditional Operator

- There are many situations where we need to assign a variable to a different value depending on the result of a condition
- For example, the `if-else-if` and `switch` statements in the previous section were used to decide which value to assign to the variable `monthName`
- A simpler example: Imagine your program needs to tell the user whether a number is even or odd. You need to initialize a `string` variable to either "Even" or "Odd" depending on whether `myInt % 2` is equal to 0. We could write an `if` statement to do this:

```
string output;
if(myInt % 2 == 0)
{
    output = "Even";
}
else
{
    output = "Odd";
}
```

Assignment with the conditional operator

- If the only thing an `if` statement does is assign a value to a variable, there is a much shorter way to write it
- The **conditional operator** `?:` tests a condition, and then outputs one of two values based on the result
- Continuing the "even or odd" example, the conditional operator is used like this:

```
string output = (myInt % 2 == 0) ? "Even" : "Odd";
```

When this line of code is executed:

- The condition `(myInt % 2 == 0)` is evaluated, and the result is either true or false
- If the condition is true, the conditional operator returns (outputs) the value `"Even"` (the left side of the `:`)

- If the condition is false, the operator returns the value `"Odd"` (the right side of the `:`)
- This value, either "Even" or "Odd", is used in the initialization statement for `string output`
- Thus, `output` gets assigned the value `"Even"` if `(myInt % 2 == 0)` is true, or `"Odd"` if `(myInt % 2 == 0)` is false
- In general, the syntax for the conditional operator is:

```
condition ? true_expression : false_expression;
```

- The "condition" can be any expression that produces a `bool` when evaluated, just like in an `if` statement
- `true_expression` and `false_expression` can be variables, values, or more complex expressions, but they must both produce the same *type* of data when evaluated
- For example, if `true_expression` is `myInt * 1.5`, then `false_expression` must also produce a `double`
- When the conditional operator is evaluated, it returns either the value of `true_expression` or the value of `false_expression` (depending on the condition) and this value can then be used in other operations such as assignment

Conditional operator examples

- The `true_expression` and `false_expression` can both be mathematical expressions, and only one of them will get computed. For example:

```
int answer = (myInt % 2 == 0) ? myInt / 2 : myInt + 1;
```

If `myInt` is even, the computer will evaluate `myInt / 2` and assign the result to `answer`. If it is odd, the computer will evaluate `myInt + 1` and assign the result to `answer`.

- Conditional operators can be used with user input to quickly provide a "default value" if the user's input is invalid. For example, we can write a program that asks the user their height, but uses a default value of 0 if the user enters a negative height:

```
Console.WriteLine("What is your height in meters?");
double userHeight = double.Parse(Console.ReadLine());
double height = (userHeight >= 0.0) ? userHeight : 0.0;
```

- The condition can be a Boolean variable by itself, just like in an if statement. This allows you to write code that looks kind of like English, due to the question mark in the conditional operator. For example,

```
bool isAdult = age >= 18;
decimal price = isAdult ? 5.0m : 2.5m;
string closingTime = isAdult ? "10:00 pm" : "8:00 pm";
```

Introduction

Arrays are structures that allow you to store multiple values in memory using a single name and indexes. Internally, an array contains a fixed number of variables (called *elements*) of a particular type[345]. The elements in an array are always stored in a contiguous block of memory, providing fast and efficient access.

An array can be:

- Single-Dimensional,
- Multidimensional.

Multidimensional arrays can be

- Jagged,
- Rectangular.

Arrays are useful, for instance,

- When you want to store a collection of related values,
- When you do not know in advance how many variables will be needed,
- When you need a large number of variables (say, 10) of the same type,
- When you want to represent matrices (as you can use an array of arrays to represent 2-dimensional objects).

Single-Dimensional Arrays

Introduction

You can define a single-dimensional array as follow:

```
<type>[] arrayName;
```

where

[345]Usually, all the elements of an array have the same type, but an array can store elements of different types if `object` is its type, since any element is actually of type `object`.

- <`type`> can be any data-type and specifies the data-type of the array elements.
- `arrayName` is an identifier that you will use to access and modify the array elements.

Before using an array, you must specify the number of elements in the array as follows:

```
arrayName = new <type>[<number of elements>];
```

where <`type`> is a type as before, and <`number of elements`>, called the *size declarator*, is a strictly positive integer which will correspond to the size of the array.

- An element of a single-dimensional array can be accessed and modified by using the name of the array and the index of the element as follows:

```
// Assigns <value> to the <index> element of the array
 ↪  arrayName.
arrayName[<index>] = <value>;

// Display the <index> element of the array arrayName.
Console.WriteLine(arrayName[<index>]);
```

The index of the first element in an array is always *zero*; the index of the second element is one, and the index of the last element is the size of the array minus one. As a consequence, if you specify an index greater or equal to the number of elements, a run-time error will happen.

Indexing starting from 0 may seem surprising and counter-intuitive, but this is a largely respected convention across programing languages and computer scientists. Some insights on the reasons behind this (collective) choice can be found in this answer on Computer Science Educators[346].

Example

In the following example, we define an array named `myArray` with three elements of type integer, and assign 10 to the first element, 20 to the second element, and 30 to the last element.

```
int[] myArray;
myArray = new int[3]; // 3 is the size declarator
// We can now store 3 ints in this array,
// at index 0, 1 and 2

myArray[0] = 10; // 0 is the subscript, or index
```

[346] https://cseducators.stackexchange.com/a/5026

```
myArray[1] = 20;
myArray[2] = 30;
```

If we were to try to store a fourth value in our array, at index 3, using e.g.

```
myArray[3] = 40;
```

our program would compile just fine, which may seems surprising. However, when executing this program, *array bounds checking* would be performed and detect that there is a mismatch between the size of the array and the index we are trying to use, resulting in a quite explicit error message:

```
Unhandled Exception: System.IndexOutOfRangeException:
 ↪  Index was outside the bounds of the array at
 ↪  Program.Main()
```

Abridged Syntaxes

If you know the number of elements when you are defining an array, you can combine declaration and assignment on one line as follows:

```
<type>[] arrayName = new <type>[<number of elements>];
```

So, we can combine the first two lines of the previous example and write:

```
int[] myArray = new int[3];
```

We can even initialize *and* give values on one line:

```
int[] myArray = new int[3] { 10, 20, 30 };
```

And that statement can be rewritten as any of the following:

```
int[] myArray = new int[] { 10, 20, 30 };
int[] myArray = new[] { 10, 20, 30 };
int[] myArray = { 10, 20, 30 };
```

But, we should be careful, the following would cause an error:

```
int[] myArray = new int[5];
myArray = { 1, 2 ,3, 4, 5}; // ERROR
```

If we use the shorter notation, we *have to* give the values at initialization, we cannot re-use this notation once the array has been created.

Other datatypes, and even objects, can be stored in arrays in a perfectly similar way:

```
string[] myArray = { "Bob", "Mom", "Train", "Console" };

// Assume there is a class called Rectangle.
Rectangle[] arrayOfRectangle = new Rectangle[5];
```

Simple Loops and Length

Custom Size and Loops

One of the benefits of arrays is that they allow you to specify the number of their elements at run-time: the size declarator can be a variable, not just an integer literal. Hence, depending on run-time conditions such as user input, we can have enough space to store and process any number of values.

In order to access the elements of whose size is not known until run-time, we will need to use a loop. If the size of `myArray` comes from user input, it wouldn't be safe to try to access a specific element like `myArray[5]`, because we cannot guarantee that the array will have at least 6 elements. Instead, we can write a loop that uses a counter variable to access the array, and use the loop condition to ensure that the variable does not exceed the size of the array.

Example

In the following example, we get the number of elements at run-time from the user, create an array with the appropriate size, and fill the array.

```
Console.WriteLine("What is the size of the array that you
 ↪  want?");
int size = int.Parse(Console.ReadLine());
int[] customArray = new int[size];

int counter = 0;
while (counter < size)
{
    Console.WriteLine($"Enter the {counter + 1}th value");
    customArray[counter] = int.Parse(Console.ReadLine());
    counter++;
}
```

Observe that:

- If the user enters a negative value or a string that does not correspond to an integer for the `size` value, our program will crash: we are not performing any user-input validation here, to keep our example compact.
- The loop condition is `counter < size` because we do *not* want the loop to execute when `counter` is equal to `size`. The last valid index in `customArray` is `size - 1`.
- We are asking for the `{counter +1}th` value because we prefer not to confuse the user by asking for the "0th" value. Note that a

more sophisticated program would replace "th" with "st", "nd" and "rd" for the first three values.

The Length Property

Every single-dimensional array has a property called `Length` that returns the number of the elements in the array (or size of the array).

To process an array whose size is not fixed at compile-time, we can use this property to find out the number of elements in the array.

Example

```
int counter2 = 0;
while (counter2 < customArray.Length)
{
    Console.WriteLine($"{counter2}:
 ↪  {customArray[counter2]}.");
    counter2++;
}
```

Observe that this code does not need the variable `size`.

Note: You *cannot* use the length property to change the size of the array, that is, entering

```
int[] test = new int[10];
test.Length = 9;
```

would return, at compile time,

```
Compilation error (line 8, col 3): Property or indexer
 ↪  'System.Array.Length' cannot be assigned to --it is
 ↪  read only.
```

When a field is marked as 'read only,' it means the attribute can only be initialized during the declaration or in the constructor of a class. We receive this error because the array attribute, 'Length,' can not be changed once the array is already declared. Resizing arrays will be discussed in the section: Changing the Size.

For Loops With Arrays

- Previously, we learned that you can iterate over the elements of an array using a **`while`** loop. We can also process arrays using **`for`** loops, and in many cases they are more concise than the equivalent **`while`** loop.

- For example, consider this code that finds the average of all the elements in an array:

```
int[] homeworkGrades = {89, 72, 88, 80, 91};
int counter = 0;
int sum = 0;
while(counter < 5)
{
    sum += homeworkGrades[counter];
    counter++
}
double average = sum / 5.0;
```

- This can also be written with a **for** loop:

```
int sum = 0;
for(int i = 0; i < 5; i++)
{
    sum += homeworkGrades[i];
}
double average = sum / 5.0;
```

- In a **for** loop that iterates over an array, the counter variable is also used as the array index
- Since we did not need to use the counter variable outside the body of the loop, we can declare it in the loop header and limit its scope to the loop's body
- Using a **for** loop to access array elements makes it easy to process "the whole array" when the size of the array is user-provided:

```
Console.WriteLine("How many grades are there?");
int numGrades = int.Parse(Console.ReadLine());
int[] homeworkGrades = new int[numGrades];
for(int i = 0; i < numGrades; i++)
{
    Console.WriteLine($"Enter grade for homework {i+1}");
    homeworkGrades[i] = int.Parse(Console.ReadLine());
}
```

- You can use the `Length` property of an array to write a loop condition, even if you did not store the size of the array in a variable. For example, this code does not need the variable `numGrades`:

```
int sum = 0;
for(int i = 0; i < homeworkGrades.Length; i++)
{
    sum += homeworkGrades[i];
}
double average = (double) sum / homeworkGrades.Length;
```

- In general, as long as the loop condition is in the format `i < <arrayName>.Length` (or, equivalently, `i <= <arrayName>.Length - 1`), the loop will access each element of the array.

Default Values and Resizing

When created, arrays have a fixed size and are populated with some *default values*. We discuss here what those default values are, how an array can be resized, and how we can avoid resizing an array.

Default Values

If we initialize an array but do not assign any values to its elements, each element will get the default value for that element's data type. (These are the same default values that are assigned to instance variables if we do not write a constructor, as we learned in "More Advanced Object Concepts"). In the following example, each element of `myArray` gets initialized to 0, the default value for `int`:

```
int[] myArray = new int[5];
Console.WriteLine(myArray[2]); // Displays "0"
myArray[1]++;
Console.WriteLine(myArray[1]); // Displays "1"
```

However, remember that the default value for any *object* data type is **`null`**, which is an object that does not exist. Attempting to call a method on a **`null`** object will cause a run-time error of the type `System.NullReferenceException`;

```
Rectangle[] shapes = new Rectangle[3];
shapes[0].SetLength(5);  // ERROR
```

Before we can use an array element that should contain an object, we must instantiate an object and assign it to the array element. For our array of `Rectangle` objects, we could either write code like this:

```
Rectangle[] shapes = new Rectangle[3];
shapes[0] = new Rectangle();
shapes[1] = new Rectangle();
shapes[2] = new Rectangle();
```

or use the abridged initialization syntax as follows:

```
Rectangle[] shapes = {new Rectangle(), new Rectangle(),
 ↪   new Rectangle()};
```

Changing the Size

There is a class named `Array` that can be used to resize an array. Upon expanding an array, the additional indices will be filled with the default value of the corresponding type. Shrinking an array will cause the data in the removed indices (those beyond the new length) to be lost.

Example

```
Array.Resize(ref myArray, 4); //myArray[3] now contains 0
myArray[3] = 40;
Array.Resize(ref myArray, 2);
```

In the above example, all data starting at index 2 is lost.

Partially Filled Arrays

To avoid resizing an array, it also possible to declare it *larger than it needs to be*, and then to manipulate an accompanying integer variable that holds the number of elements that are actually stored in the array. The solution to the todo list project[347] illustrates this behavior in detail, the general idea is that you want to let the user store some elements without having to say ahead of time how many, and without having to resize the array constantly. The drawback is that the `Length` property becomes less useful, and that you have to manipulate a custom "accounting" variable to keep track of the actual number of elements manipulated.

```
using System;

public class Program
{
  public static void Main(string[] args)
  {
    // We decide that the maximum number of input is 10.
    const int MAXSIZE = 10;

    int[] inputs = new int[MAXSIZE];

    // The following variable will contain the number of
     ↪  input actually given.
    int numberOfInputs = 0;

    // The following variable will hold the user input.
    string uInput;
```

[347] https:/princomp.github.io/projects/todolist/solution

```
    do
    {
      Console.WriteLine(
        "What is your input #"
          + (numberOfInputs + 1)
          + "? Enter \"done\" when you are done."
      );
      uInput = Console.ReadLine();
      if (uInput != "done")
      {
        inputs[numberOfInputs] = int.Parse(uInput);
        numberOfInputs++; // We increment the number of
↪   items in the list.
      }
      if (numberOfInputs == MAXSIZE)
      {
        Console.WriteLine(
          "You have reached the maximum number of inputs."
        );
      }
    } while (uInput != "done" && numberOfInputs <
      ↪   MAXSIZE);
    /*
     * When the user enters "done", or if the user reached
↪   the maximum number of inputs, we exit this loop.
     */
  }
}
```

Searching in Arrays

We now discuss how we can search for values in an array.

Finding the Maximum Value

To find the greatest value in an array of integer, one needs a comparison point, a variable holding "the greatest value *so far*". Once this value is set, then one "just" have to inspect each value in the array, and to update "the greatest value *so far*" if the value currently inspected is greater, and then to move on to the next value. Once we reach the end of the array, we know that "the greatest value *so far*" is actually the greatest value (period) in the array.

The problem is to find the starting point: one cannot assume that "the greatest value *so far*" is 0 (what if the array contains only negative values?), so the best strategy is simply to assume that "the greatest value *so far*" is the first one in the array (after all, it *is* the greatest value we have seen *so far*).

Using **foreach**, we have for example the following:

```
int[] arrayExample = { 1, 8, -12, 9, 10, 1, 30, 1, 32, 3
 ↪  };

int maxSoFar = arrayExample[0];
foreach (int i in arrayExample)
    if (i > maxSoFar) maxSoFar = i;

Console.WriteLine("The greatest value is "
    + maxSoFar + ".");
```

Finding a Particular Value

Suppose we want to set a particular Boolean variable to **true** if a particular value `target` is present in an arrayy `arrayExample`. The simplest way to perform such a search is to

1. Set the Boolean variable to **false**,
2. Inspect the values in `arrayExample` one by one, comparing them to `target`, and setting the Boolean variable to **true** if they are identical.

```
int[] arrayExample = { 1, 8, -12, 9, 10, 1, 30, 1, 32, 3
 ↪  };

bool foundTarget = false;
int target = 8;

for (int i = 0; i < arrayExample.Length; i++)
{
    if (arrayExample[i] == target) foundTarget = true;
}
Console.WriteLine(target + " is in the array: " +
 ↪  foundTarget + ".");
```

Note that in the particular example above, we could have stopped exploring the array after the second index, since the target value was found. A slightly different logic would allow to exit prematurely the loop when the `target` value is found:

```
int[] arrayExample = { 1, 8, -12, 9, 10, 1, 30, 1, 32, 3
 ↪  };
```

```
bool foundYet = false;
int target = 30;
int index = 0;

do
{
    if (arrayExample[index] == target) foundYet = true;
    index++;
}
while (index < arrayExample.Length && !foundYet);
Console.WriteLine(target + " is in the array: " +
 ↪  foundYet +
"\nNumber of elements inspected: " + (index) +".");
```

This code would display:

```
30 is in the array: True
Number of elements inspected: 7.
```

Both codes are examples of *linear* (or *sequential*) *search*: the array is parsed one element after the other, and potentially all elements are inspected.

Finding a Particular Value in a Sorted Array

If the array is *sorted* (that is, the value at index i is less than the value at index $i + 1$), then the search for a particular value can be sped up by using *binary search*.

Sorted Arrays

A way of making sure that an array is sorted is given below. Note that, as above when trying to find the maximum value, we decide that the array is "sorted so far" unless proven otherwise, in which case we exit prematurely the loop. Note also that the condition contains `index + 1 < arrayExample.Length`: we need to make sure that "the next value" actually exists before comparing it with the current value.

```
int[] arrayExample = { 1, 10, 12, -1};
bool sortedSoFar = true;
int index = 0;

while (index + 1 < arrayExample.Length && sortedSoFar)
{
    if (arrayExample[index] > arrayExample[index+1])
     ↪  sortedSoFar = false;
    index++;
}
Console.WriteLine("The array is sorted: " + sortedSoFar
 ↪  +".");
```

Binary Search

Introduction *Binary* (*half-interval* or *logarithmic*) *search* leverages the fact that the array is sorted to speed up the search for a particular value. It goes as follows:

The algorithm compares the `target` value to the middle element of the array.

1. If they are equal, we are done.
2. If they are not equal, then there are two cases:
   (a) If the middle element is greater than the `target`, then the algorithm restarts, but looking for the value only in the left half of the array,
   (b) If the middle element is less than the `target`, then the algorithm restarts, but looking for the value only in the right half of the array.
3. If the search ends with the remaining half being empty, the `target` is not in the array.

First Example An example of implementation (and of execution) is as follows:

```
int[] arrayExample = { 1, 10, 12, 129, 190, 220, 230,
 ↪  310, 320, 340, 400, 460};
bool foundSoFar = false;

int target = 340;

int start = 0;
int end = arrayExample.Length - 1;
int mid;
while (start <= end && !foundSoFar)
{
    mid = (start + end) / 2;
    /*
     * This is integer division: if start + end is odd,
     * then it will be truncated. In our example,
     * (0 + 11) / 2 gives 5.
     */
    Console.WriteLine("The middle index is " + mid + ".");
    if (target == arrayExample[mid])
    {
        foundSoFar = true;
    }
    else if (target > arrayExample[mid])
    {
        start = mid + 1;
        Console.WriteLine("I keep looking right.");
    }
    else
    {
        end = mid - 1;
        Console.WriteLine("I keep looking left.");
    }
}
Console.WriteLine("Found the value: " + foundSoFar +".");
```

This code would display:

```
The middle index is 5.
I keep looking right.
The middle index is 8.
I keep looking right.
The middle index is 10.
I keep looking left.
The middle index is 9.
Found the value: True.
```

Second Example Remembering that characters are such that `'A'` is less than `'a'`, and `'a'` is less than `'b'`, we can run a binary search on a sorted array of characters. The code below is the same algorithm as above, only the information logged changes:

```
char[] arrayExample = { 'A', 'B', 'D', 'Z', 'a', 'b', 'd'
 ↪  };
char target = 'D';
bool foundSoFar = false;
int start = 0;
int end = arrayExample.Length - 1;
int mid;
while (start <= end && !foundSoFar)
{
    Console.WriteLine("Range: " + start + " -- " + end);
    mid = (start + end) / 2;
    Console.WriteLine("Mid: " + mid);
    if (target == arrayExample[mid])
    {
        foundSoFar = true;
    }
    else if (target > arrayExample[mid])
    {
        start = mid + 1;
    }
    else
    {
        end = mid - 1;
    }
}
Console.WriteLine("Found the value: " + foundSoFar + ".");
```

This will display:

```
Range: 0 -- 6
Mid: 3
Range: 0 -- 2
Mid: 1
Range: 2 -- 2
Mid: 2
Found the value: True
```

Observe that if we were to replace `start <= end` with `start < end` then the algorithm would not have correctly terminated in the example above.

Arrays of Objects

An array can contain more than simple datatypes: it can contains object. It can be objects from a custom class, or even … arrays, which are themselves objects!

Array of Objects From a Custom Class

In the following example, we will ask the user how many `Item` objects (the details of the implementation does not matter, but can be inspired by this example[348]) they want to create, then fill an array with `Item` objects initialized from user input:

```
Console.WriteLine("How many items would you like to
 ↪  stock?");
Item[] items = new Item[int.Parse(Console.ReadLine())];
int i = 0;
```

[348]https:/princomp.github.io/lectures/flow/control_flow_and_classes#setters-with-input-validation

```
while(i < items.Length)
{
    Console.WriteLine($"Enter description of item
 ↪  {i+1}:");
    string description = Console.ReadLine();
    Console.WriteLine($"Enter price of item {i+1}:");
    decimal price = decimal.Parse(Console.ReadLine());
    items[i] = new Item(description, price);
    i++;
}
```

Observe that, since we do not perform any user-input validation, we can simply use the result of `int.Parse()` as the size declarator for the `items` array - no `size` variable is needed at all.

We can also use **`while`** loops to search through arrays for a particular value. For example, this code will find and display the lowest-priced item in the array `items`, which was initialized by user input:

```
Item lowestItem = items[0];
int i = 1;
while(i < items.Length)
{
    if(items[i].GetPrice() < lowestItem.GetPrice())
    {
        lowestItem = items[i];
    }
    i++;
}
Console.WriteLine($"The lowest-priced item is
 ↪  {lowestItem}");
```

Note that the `lowestItem` variable needs to be initialized to refer to an `Item` object before we can call the `GetPrice()` method on it; we cannot call `GetPrice()` if `lowestItem` is **`null`**. We could try to create an `Item` object with the "highest possible" price, but a simpler approach is to initialize `lowestItem` with `items[0]`. As long as the array has at least one element, `0` is a valid index, and the first item in the array can be our first "guess" at the lowest-priced item.

Arrays of Arrays

An array of arrays is called a multi-dimensional array. A multi-dimensional array can be rectangular (it then represents an n-dimensional block of memory) or jagged (in that case, it is an array of arrays).

Rectangular Multi-Dimensional Array

Also called 2-dimensional arrays, their syntax is very close to 1-dimensional arrays:

```
int[,] matrix = new int[2, 3];
```

where 2 is the number of rows, and 3 is the number of columns. They can be accessed with `matrix.GetLength(0)` and `matrix.GetLength(1)` respectively.

Assignment is as for 1-dimensional arrays, starting at 0:

```
matrix[0, 0] = 1;
matrix[0, 1] = 2;
matrix[0, 2] = 3;
matrix[1, 0] = 4;
matrix[1, 1] = 5;
matrix[1, 2] = 6;
```

This will produce a matrix as follows:

	0th col.	1st col.	2nd col.
0th row	1	2	3
1st row	4	5	6

We could also have used a shortened notation to declare this 2-dimensional array, as follows:

```
int[,] matrix = new int[,]
{
    {1,2,3},
    {4,5,6}
};
```

or even simply

```
int[,] matrix = {{1,2,3},{4,5,6}};
```

To display such an array, nested loops are needed:

```
for (int row = 0; row < matrix.GetLength(0); row++)
{
    for (int col = 0; col < matrix.GetLength(1); col++)
        Console.Write(matrix[row, col] + " ");
    Console.WriteLine();
}
```

Jagged Array

A jagged array is an array of arrays. The difference with rectangular arrays is that the arrays stored can be of varying size.

The syntax is straightforward once understood that jagged arrays are *exactly* arrays of arrays:

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[3] { 1, 2, 3};
jaggedArray[1] = new int [2]{ 4, 5};
jaggedArray[2] = new int[5] { 6, 7, 8, 9, 10};

for (int row = 0; row < jaggedArray.Length; row++)
{
    Console.Write("The row #" + row + " contain: ");
    for (int arrayCell = 0; arrayCell <
     ↪  jaggedArray[row].Length; arrayCell++)
    {
        Console.Write(jaggedArray[row][arrayCell]+ " " );
    }
    Console.WriteLine("");
}
```

In this example, it should be clear that `jaggedArray[row]` is itself an array, and hence that we can use e.g., `jaggedArray[row].Length` or `jaggedArray[row][arrayCell]`.

Manipulating Rectangular Arrays

We present below some simple algorithms to manipulate 2-dimensional (rectangular) arrays.

Summing the values row per row

The following code sum the values contained in a 2-dimensional array row per row, and display the result each time before moving on to the next row:

```
int[,] numbers =
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8}
    };

int acc;
```

```
for (int row = 0; row < numbers.GetLength(0); row++)
    {
    acc = 0;
    for (int col = 0; col < numbers.GetLength(1); col++)
    {
        acc += numbers[row, col];
    }
    Console.WriteLine("Total for row #" + row +
        " is " + acc + ".");
}
```

This code can easily be adapted to compute the sums *column per column* if needed.

Computing Magic Square

A magic square[349] is a square matrix where the sums of the numbers in each row, each column, and both the diagonal and the anti-diagonal are the same.

The following is an example of a magic square:

```
int[,] magicSquare = {
    { 4, 9, 2 },
    { 3, 5, 7 },
    { 8, 1, 6 }
};
```

as we have, diagonally,

$$4 + 5 + 6 = 15$$

and anti-diagonally,

$$2 + 5 + 8 = 15$$

and on the rows,

$$4 + 9 + 2 = 15$$

$$3 + 5 + 7 = 15$$

$$8 + 1 + 6 = 15$$

and finally on the columns

$$4 + 3 + 8 = 15$$

$$9 + 5 + 1 = 15$$

[349]https://en.wikipedia.org/wiki/Magic_square

$$2 + 7 + 6 = 15$$

A method to return **true** if the 2d-matrix of int passed as an argument is a magic square is as follows:

```
bool isMagic(int[,] arrayP)
{
    bool magicSoFar = true;
    if (arrayP.GetLength(0) == arrayP.GetLength(1))
    { // The array is a square.
        int magicConstant = 0;
        for (int i = 0; i < arrayP.GetLength(1); i++)
        {
            magicConstant += arrayP[i, i];
        }
        int testedValue = 0;
        for (int i = 0; i < arrayP.GetLength(1); i++)
        {
           testedValue += arrayP[i, arrayP.GetLength(1) -
 ↪  i - 1];
        }
        if (testedValue == magicConstant)
        {// The diagonal and anti-diagonal have the same
         ↪  sums.
            // We test the rows.
            for (int row = 0; row < arrayP.GetLength(0);
             ↪  row++)
            {
                testedValue = 0;
                for (int col = 0; col <
                 ↪  arrayP.GetLength(1); col++)
                {
                    testedValue += arrayP[row, col];

                }

                if (testedValue != magicConstant)
                {
                    magicSoFar = false;
                }
            }
            // We test the columns.
            for (int col = 0; col < arrayP.GetLength(1);
             ↪  col++)
            {
                testedValue = 0;
                for (int row = 0; row <
                 ↪   arrayP.GetLength(0); row++)
                {
                    testedValue += arrayP[row, col];
                }

                if (testedValue != magicConstant)
                {
                    magicSoFar = false;
                }
            }
        }
        else
        {// The diagonal and anti-diagonal have different
         ↪   same sums.
            magicSoFar = false;
        }
    }
    else
    { // The array is not a square.
        magicSoFar = false;
    }

    return magicSoFar;
}
```

That code can be tested using for example this code[350].

Over and Underflow

Overflow

- Assume a car has a 4-digit odometer, and currently, it shows `9999`. What does the odometer show if you drive the car another mile? As you might guess, it shows `0000` while it should show `10000`. The reason is the odometer does not have a counter for the fifth digit. Similarly, in C#, when you do arithmetic operations on integral data, the result may not fit in the corresponding data type. This situation is called an **overflow** error.
- In an unsigned data type variable with N bits, we can store the numbers from 0 to $2^N - 1$. In signed data type variables, the high order bit represents the sign of the number as follows:

[350]https:/princomp.github.io/code/snippets/magicSquare_test.cs

- 0 means zero or a positive value
- 1 means a negative value
- With the remaining $N-1$ bits, we can represent $2^(N-1)$ values. Hence, considering the sign bit, we can store a number from $-2^(N-1)$ to $2^(N-1)-1$ in the variable.
- In some programming languages like C and C++, overflow errors cause undefined behavior, and can crash your program. In C#, however, the extra bits are just ignored, and the program will continue executing even though the value in the variable may not make sense. If the programmer is not careful to check for the possibility of overflow errors, they can lead to unwanted program behavior and even severe security problems.
- For example, assume a company gives loans to its employee. Couples working for the company can get loans separately, but the total amount cannot exceed $10,000. The following program looks like it checks loan requests to ensure they are below the limit, but it can be attacked using an overflow error. (This program uses notions you may have not studied yet, but that should not prevent you from reading the source code and executing it.)

```
using System;

class Program
{
  static void Main()
  {
    uint n1,
      n2;

    Console.WriteLine(
      "Enter the requested loan amount for the first
       ↪  person:"
    );
    n1 = uint.Parse(Console.ReadLine());

    Console.WriteLine(
      "Enter the requested loan amount for the second
       ↪  person:"
    );
    n2 = uint.Parse(Console.ReadLine());

    if (n1 + n2 < 10000)
    {
      Console.WriteLine($"Pay ${n1} to the first person");
      Console.WriteLine($"Pay ${n2} to the second
  ↪   person");
    }
    else
    {
      Console.WriteLine(
        "Error: the sum of the loans exceeds the maximum
          ↪  allowance."
      );
    }
  }
}
```

- If the user enters 2 and 4,294,967,295, we expect to see the error message ("Error: the sum of loans exceeds the maximum allowance."). However, this is not what will happen, and the request will be accepted even though it should not have. The reason can be explained as follows:
- `uint` is a 32-bit data type.
- The binary representation of 2 and 4,294,967,295 are `00000000000000000000000000000010` and `11111111111111111111111111111111`.
- Therefore, the sum of these numbers should be `100000000000000000000000000000001`, which needs 33 bits.
- Nevertheless, there are only 32 bits available for the result, and the extra bits will be dropped, so the result will be `00000000000000000000000000000001`. This is less than 10,000, so the program will conclude that the sum of the loan values is less than 10,000.

Underflow

- Sometimes, the result of arithmetic operations over floating-point numbers is smaller than the minimum value that can be stored in the corresponding data type. This problem is known as the **underflow** problem.
- In C#, in case of an underflow, the result will be zero.
- For example, the smallest value that can be stored in a `float` variable is $1.5 \cdot 10^{-45}$. If we attempt to divide this value by 10, the variable will get the value 0, not $1.5 \cdot 10^{-46}$:

```
using System;

class Program
{
  static void Main()
  {
    float myNumber;
```

```
        myNumber = 1E-45f;
        Console.WriteLine(myNumber); //outputs 1.401298E-45
        myNumber = myNumber / 10;
        Console.WriteLine(myNumber); //outputs 0
        myNumber = myNumber * 10;
        Console.WriteLine(myNumber); //outputs 0
        myNumber = (1E-45f / 10) * 10;
        Console.WriteLine(myNumber); //outputs 0
    }
}
```

- An underflow error can result in "losing" data in the middle of a series of operations: even if you immediately multiply by 10 again, the intermediate result was less than $1.5 \cdot 10^{-45}$, so the final result is still 0.

Random

- Random Number Generation
  - Produce a number within some bounds following some statistical rules.
  - A true random number is a number that is **nondeterministically** selected from a set of numbers wherein each possible selection has an equal probability of occurrence.
  - Usually in computer science we contend with **pseudo-random** numbers. These are not truly nondeterministic, but an approximation of random selection based on some algorithm.
  - Since pseudo-random selections are "determined" by an algorithm, or set of rules, they are technically **deterministic**.
- Random Class in C#
  - Instantiate a random number generator and use to select numbers:

    ```
    Random rand = new Random();
    Random randB = new Random(seed_int);
    ```

  - Notice that we can create a generator with or without an argument. The argument is called a **seed** for the generator.
  - A seed tells the generator where to start its sequence. Using the same seed will always reproduce the same sequence of numbers.
  - The default constructor still has a seed value, but it is a hidden value pulled from the clock time during instantiation.

  - Time-based seeds only reset approximately every 15 milliseconds.
  - The random class is not "random enough" for cryptography.
  - For cryptographic randomness, use the RNGCryptoServiceProvider[351] class or System.Security.Cryptography.RandomNumberGenerator[352].
- Using Random
  - Next() method returns a pseudo-random number between 0 and 2,147,483,647 (max signed `int`), inclusive.
  - By default, the number is always non-negative and within that range.

    ```
    int randomInt = rand.Next();
    ```

  - What if we wanted to create a random number between 0 and 100?
  - We could use rand.Next() and then use modulo to cut down the answer range!
  - Alternatively, we could give the Next() method an `int` argument to set a ceiling.

    ```
    int randomUpto100 = rand.Next(101);
    ```

  - The ceiling value is **exclusive**, so remember to use one number higher than what you want to be your max number.
  - We can also pass two arguments in order to set a range for the values.

    ```
    int random50to100 = rand.Next(50,101);
    ```

  - The ceiling value is still exclusive, but the floor is **inclusive**.
  - NextDouble() returns a **normalized** value (value between 0.0 and 1.0 inclusive).
  - What if we want a different range? Adjust with math!

    ```
    double randNeg2to3 - (rand.NextDouble()*5)-2;
    ```

  - NextBytes() method takes a `byte` array as an argument and generates a random `byte` value for each index.
  - Remember, a `byte` has an unsigned value between 0 and 255 inclusive.

[351] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?view=net-5.0

[352] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.randomnumbergenerator?view=net-5.0

```
byte[] byteArray = new byte[10];
rand.NextBytes(byteArray);
```

- Creating Random Strings
  - What if we want to construct random strings made of a, b, c, and d?
  - Other techniques are available, but we can use a loop and switch!

```
Random rand = new Random();
string answer = "";
int selection = 0;

for(int i = 0; i < 10; i++)
{
    selection = rand.Next(4);
    switch(selection){
    case(0):
        answer+="a";
        break;
    case(1):
        answer+="b";
        break;
    case(2):
        answer+="c";
        break;
    default:
        answer+="d";
        break;
    }
}
```

Exceptions

Introduction

- At *execution time* programs can run into unspecified behaviour, such as
  - having to divide by zero,
  - having to access an array at an index greater than its length.
- For example, the following instructions would compile just fine, but *at execution time* the program would "explode":

```
int zero = 0;
Console.WriteLine($"Let's divide by zero: {1 /
 ↪  zero}.");
```

```
int[] test = new int[2];
test[2] = 3;
```

  - In the first case, a "System.DivideByZeroException has been thrown" error message would be displayed.
  - In the second case, a "System.IndexOutOfRangeException has been thrown" error message would be displayed.
  - Those are examples of exceptions thrown by operations[353].

- Methods can also throw exceptions. For example, the following statement:

  ```
  int x = int.Parse("This is not a number.");
  ```

  will display a "System.FormatException has been thrown" error message. This is because the `Parse` method can *throw an exception*[354].

- Of course, a programmer would not *purposely* introduce such strange instructions in their code, but they may arise after interacting with the "outside world", that is, a user, file, or other external factor.

- C# allows *exception handling*, which are ways of recovering when such exceptions are thrown, so that the program can keep on executing. Stated differently, they instruct the program what to do, for example, if it is asked to perform a division by 0. This is handled by **`catch`** blocks.

- C# also allows **`finally`** block, which contain code executed unconditionally, that is, regardless of if the exception was thrown or not.

Syntax and Rules for `try`...`catch`...`finally` Statements

- In a first approximation, the syntax of a **`try`**...**`catch`**...**`finally`** statement is as follows:

  ```
  try{
      <statement block 1>
  }
  catch{
  ```

[353] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions#215-common-exception-classes

[354] https://learn.microsoft.com/en-us/dotnet/api/system.int32.parse?view=net-8.0#system-int32-parse(system-string)

```
    <statement block 2>
}
finally{
    <statement block 3>
}
```

- When executed, `<statement block 1>` will be executed statement by statement. If, at any point, one of the statement in `<statement block 1>` throws an exception, then the rest of the statements in `<statement block 1>` will be discarded and `<statement block 2>` will execute. After all of the statements in `<statement block 1>` have executed, or after all of the statements in `<statement block 2>` have executed, then `<statement block 3>` will execute.
- A simple example is

```
try
{
    Console.WriteLine("Enter a number.");
    Console.WriteLine($"Your number is
 ↪  {int.Parse(Console.ReadLine())}.");
}
catch
{
    Console.WriteLine("Something was off.");
}
finally
{
    Console.WriteLine("Did it worked?");
}
```

  - If the user enters a string that contains only numbers (say, "10"), then the program will display "Your number is 10." then "Did it worked?".
  - If the user enters a string that does *not* contain only numbers (say, "No"), then the program will display "Something was off." then "Did it worked?".
- Both the **`catch`** and the **`finally`** parts of the statement are optional: they can be both present, or only one can occur in the **`try`** block statement (however, you have to have one or the other).
- A **`try`** block can have multiple **`catch`**, as follows:

```
try
{
    Console.WriteLine("Enter a number");
    int uInput = int.Parse(Console.ReadLine());
```

```
        Console.WriteLine($"Your number is {uInput}.");
        Console.WriteLine($"Ten divided by your number is
    ↪   {10 / uInput}.");

    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("You tried to divide by zero.");
    }
    catch (FormatException)
    {
        Console.WriteLine("You have tried to convert a
    ↪   string "
            + " containing non-numerical characters to a
            ↪   number.");
    }
    finally
    {
        Console.WriteLine("Did it worked?");
    }
```

- This allows a more fine-grained handling of the exceptions that can be thrown.
- In the example, if a `DivideByZeroException` exception is thrown, it is because the user entered "0" and the operation `{10 / uInput}` failed. In this case, we can display an appropriate error message ("You tried to divide by zero").
- In the example, if a `FormatException` exception is thrown, it is because the user entered a string containing non-numerical characters, and we can similarly return an appropriate error message.
- Writing `catch{…}` is the same as `catch (Exception){…}`: by default, a `catch` block catches all the exceptions that can be thrown, not the exceptions of a particular class. Note that, if specifying multiple `catch` blocks, the order matter, as a `catch (Exception)`, if placed first, will always execute before the `catch` blocks put after.

Exception Class and Objects

- Technically speaking, an exception is an object in a particular class that inherits from the exception class[355].
- We can assign an identifier to it in the `catch` block, to be able to ac-

[355] https://learn.microsoft.com/en-us/dotnet/standard/exceptions/exception-class-and-properties

cess some of its properties such as the `Message` and a `StackTrace` properties.

- For example, the `IndexOutOfRangeException` object returned when trying to access an array outside of its bound can be named ex and used to display particular information:

```
try
{
    int[] test = new int[10];
    for (int i = 0; i <= test.Length; i++)
    {
        test[i] = i;
    }
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
}
```

  - When the statement `test[10] = 10;` gets executed, the exception is thrown, named ex, and we display its message ("Index was outside the bounds of the array.") and Stack-Trace ("at Program.Main (System.String() args) (0x0000f) in <`path`>/Program.cs:<`line`>", with <`path`> the path of the Program.cs file, and <`line`> the line where the error occurs).

Purpose of the `finally` Block

- The difference between

```
try{
    <statement block 1>
}
catch{
    <statement block 2>
}
finally{
    <statement block 3>
}
```

  and

```
try{
    <statement block 1>
}
catch{
    <statement block 2>
}
<statement block 3>
```

is that in the second case, `<statement block 3>` may be skipped if `<statement block 1>` or `<statement block 2>` return a value, throw an exception that is not caught, or break the flow of control (using for example **`break`**`;`). In the first case, `<statement block 3>` will *always*[356] get executed, no matter which block gets executed and even if it breaks the control flow or throws another exception.

- For example,

```
static bool GuessGame(string guessP)
{
    const int valueToGuess = 12;
    try
    {
        int guessV = int.Parse(guessP);
        if (guessV == valueToGuess)
        {
            Console.WriteLine("You guessed it!");
            return true;
        }
        else
        {
            Console.WriteLine("Try again!");
            return false;
        }
    }
    catch (FormatException)
    {
        Console.WriteLine("Please, provide a string
 ↪  containing only numbers");
        return false;
    }
    finally
    {
        Console.WriteLine("Thank you for playing!");
    }
}
```

will always display “Thank you for playing!”. If this last statement was *not* in the **`finally`** block, but was simply inserted after the **`try`** …

[356]That is, unless the program crashes or loops forever.

catch statement, then this message would actually never be displayed.

Scoping in **try** ... **catch**... **finally** Statements

- Understanding the scope of statements in **try** ... **catch**... **finally** statements can be tricky.
- The general rules are:
  - Variables declared in **try**, **catch** or **finally** blocks will not be accessible outside of them,
  - Variables whose value are set in the **try** block will keep the value they had when the **try** block threw an exception.
- For example, in the following code,

```
int zero = -1;
try
{
    zero = 0;
    int x = 1 / zero;
    zero = 2;
}
catch (DivideByZeroException)
{
    Console.WriteLine("You tried to divide by " +
 ↪  zero + ".");
    zero = 3;
}
finally
{
    Console.WriteLine("The variable holds " + zero +
 ↪  ".");
}
```

  - This program will display

    ```
    You tried to divide by 0.
    The variable holds 3.
    ```

  - The variable `x` would not be accessible to the **catch** or **finally** blocks.
  - If we were to remove the `zero = 0;` statement, then the program would display "The variable holds 2.".

When To Use `try` … `catch` and When To Use `TryParse`?

- If something goes wrong in a method, that method can either return some error code or throw an exception.
- Returning an error code means possibly cluttering the signature of the method with some extra parameters, as in the `TryParse` methods.
- `TryParse` is "baking in" a way of signaling that something went wrong because
  1. This type of error is simple, common and predictable,
  2. It decided not to care about *why* the parsing fails (it can be either because the input is **`null`**, because it is not in valid format, or because it produces an overflow).
- However, exceptions can handle those cases differently thanks to different **`catch`** blocks:

```
Console.WriteLine("Test with" +
    "\n\t- nothing (ctrl + d on linux, ctrl + z on
     ↪  windows), " +
    "\n\t- \"No\"," +
    "\n\t-  " + int.MaxValue + "+ 1 = 2147483648.");
try
{
    int.Parse(Console.ReadLine());
}
catch (ArgumentNullException)
{
    Console.WriteLine("No argument provided.");
}
catch (FormatException)
{
    Console.WriteLine("The string does not contain
 ↪  only number characters.");
}
catch (OverflowException)
{
    Console.WriteLine("The number is greater than
 ↪  what an integer can store.");
}
```

- So, in summary, `TryParse` is in general better if there is no need to handle the different exceptions differently.

Throwing an Exception

- You can explicitly throw an exception by
  - Creating an `Exception` object,
  - Throwing it, using the **`throw`** keyword.
- For example, the property setter in the following class can explicitly throw an `ArgumentOutOfRangeException` object if we try to create a `Circle` object with a negative diameter:

```
using System;

class Circle
{
  private decimal diameter;
  public decimal Diameter
  {
    get { return diameter; }
    set
    {
      if (value <= 0)
      {
        throw new ArgumentOutOfRangeException();
      }
      else
      {
        diameter = value;
      }
    }
  }

  public Circle(decimal dP)
  {
    Diameter = dP;
  }

  public override string ToString()
  {
    return "Diameter: " + diameter;
  }
}
```

- To use this class properly, every time the `Diameter` value is set (using the set accessor, possibly *via* the constructor), a **`try`** … **`catch`** statement should be used to handle a possible exception, with possibly a loop around it, as follows:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    Circle circle1 = new Circle(1);
    Console.WriteLine(circle1);

    try
    {
      circle1 = new Circle(-10);
      Console.WriteLine("circle1: " + circle1);
    }
    catch (ArgumentOutOfRangeException)
    {
      Console.WriteLine($"Error: value was out of
 ↪  range.");
    }
    Console.WriteLine(circle1);

    bool circle_modified = false;
    do
    {
      try
      {
        Console.WriteLine("Enter the circle new
 ↪  diameter.");
        int uInput = int.Parse(Console.ReadLine());
        circle1.Diameter = uInput;
        Console.WriteLine(circle1);
        circle_modified = true;
      }
      catch (ArgumentOutOfRangeException)
      {
        Console.WriteLine(
          $"Error: value was out of range."
        );
      }
      catch
      {
        Console.WriteLine("Something went wrong.");
        throw;
      }
    } while (!circle_modified);
  }
}
```

- In the last **`catch`** block above, the **`throw`**`;` (without argument) will pass the exception to the calling environment. It is indeed possible to catch the exception, do something with it (typically, log it or display an error message), and then "pass" that exception to the surrounding environment.

The List collections

Introduction

The `List` class serves a similar purpose than arrays, but with a few notable differences:

- Lists do not need to have a number of elements fixed ahead of time,
- Lists automatically expand when elements are added,
- Lists automatically shrink when elements are removed,
- Lists require to have the **`using`** `System.Collections.Generic;` statement at the beginning of the file,
- Lists have many built-in methods.

Syntax

Creation

The syntax to create an empty list of `string` named `nameList` and a list of `int` named `valueList` containing 1, 2 and 3 is:

```
List<string> nameList = new List<string>();
List<int> valueList = new List<int>() { 1, 2, 3 };
```

Adding Elements

Adding an element to the list is done using the `Add` method, and counting the number of elements is done using the `Count` property:

```
Console.WriteLine("nameList has " + nameList.Count + "
 ↪  element.");
nameList.Add("Bob");
Console.WriteLine("nameList has " + nameList.Count + "
 ↪  element.");
nameList.Add("Sandrine");
Console.WriteLine("nameList has " + nameList.Count + "
 ↪  elements.");
```

Note that we did not need to resize the `nameList` manually: its size went from 0 to 1 after we added "Bob", and from 1 to 2 after we added "Sandrine".

Accessing Elements

Using the `[]` operator Accessing an element can be done using the same operator as with arrays (the `[]` operator):

```
Console.Write(nameList[0]);
```

will display "Bob". Note that this syntax can be used to change the value of an element that already exist. For example,

```
nameList[0] = "Robert";
```

would replace the first value in the list ("Bob") with "Robert".

Note that while accessing or replacing an element using the `[]` operator inside a list is fine, *you cannot add new elements to the list using this syntax*. For example,

```
nameList[2] = "Sandrine";
```

would raise an exception since there is no third element to our list.

Using `foreach` Another way of accessing the elements in a list is to use **`foreach`** loops:

```
foreach (string name in nameList)
{
    Console.WriteLine(name);
}
```

Removing Elements

An element can be removed from the list using the `RemoveAt` method. If `nameList` contains "Robert, Sandrine", then after the following statement,

```
nameList.RemoveAt(0);
```

it would only contain "Sandrine" and its size would be 1. That is, the first element would be deleted and the list would shrink.

Another way of removing an element is to use the `Remove` method. Suppose we have the following list:

```
List<int> valueList = new List<int>() {-1, 0, 1, 2, 3, 2,
 ↪  5 };
```

then using

```
valueList.Remove(1);
```

would remove "1" from the list, and the list would become -1, 0, 2, 3, 2, 5.

Observe that `Remove` returns a `bool`, so that for instance the following

```
if(valueList.Remove(0)){
    Console.WriteLine("0 was removed.");
}
```

would not only remove 0 from the list, but also display "0 was removed".

Finally, if the value is present multiple times in the list, then only its first occurrence is removed. For example, if the list is -1, 2, 3, 2, 5, then after executing

```
valueList.Remove(2);
```

it would become -1, 3, 2, 5.

A Custom Implementation of Lists

A "custom" implementation of list can be found in this project[357].

[357] https:/princomp.github.io/code/projects/CList.zip