Contents

Constructors and Methods

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:

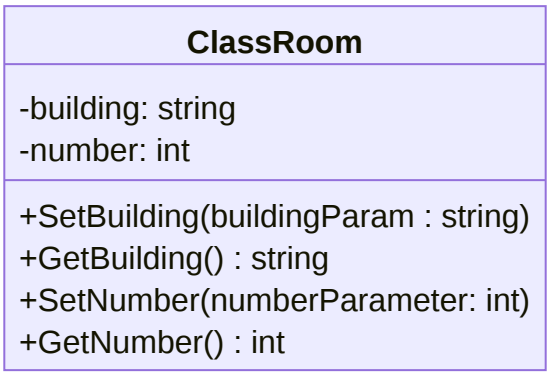

Figure 1: A UML diagram for the ClassRoom class (text version[1])

  - 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;
  }

  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`
    (where <> is sometimes replaced with «constructor»)
  - 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:

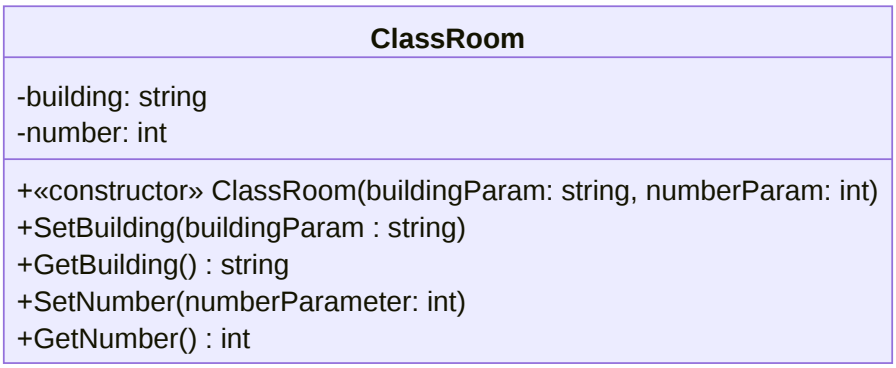

Figure 2: A UML diagram for the ClassRoom class (text version[2])