Contents

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 `int`s), 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 `int`s, 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]`

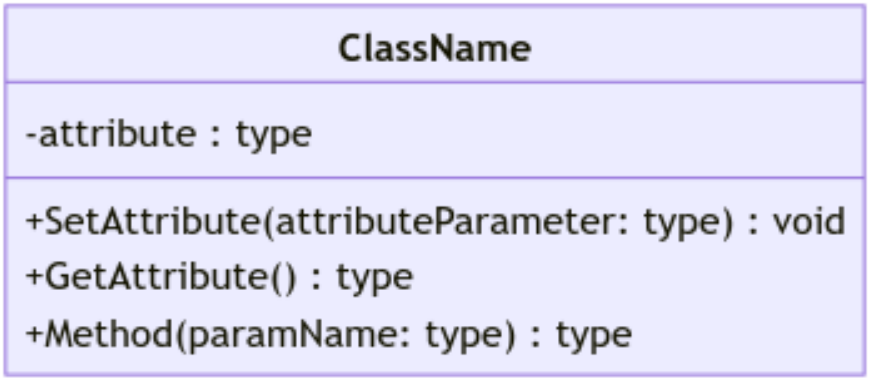

Figure 1: A UML diagram for the ClassName class (text version[1])

    - 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

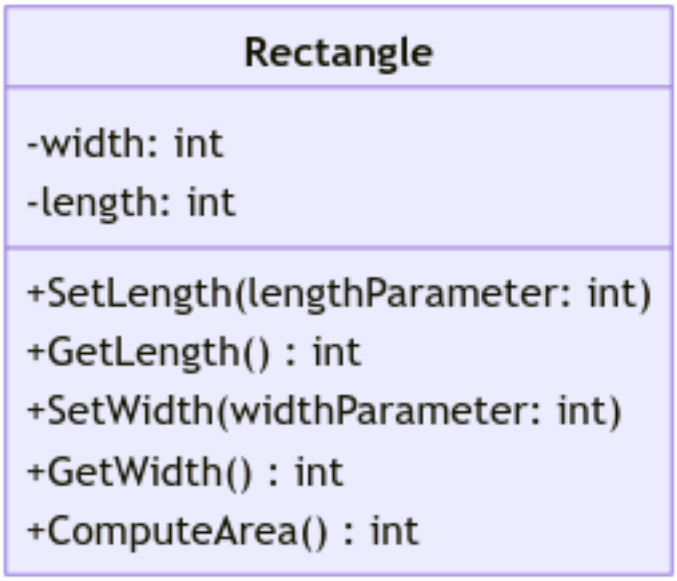

Figure 2: A UML diagram for the Rectangle class (text version[2])

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`