```
}
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 doesn't matter what values `myStringVar` or `myBoolVar` have, because the first if block (setting `x` to 10) will get executed.
- Example outcomes of running 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

11.4.0.4 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");
        }
    }
}
```

12 Switch Statements and the Conditional Operator

12.1 Switch Statements

12.1.0.1 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.");
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)
```

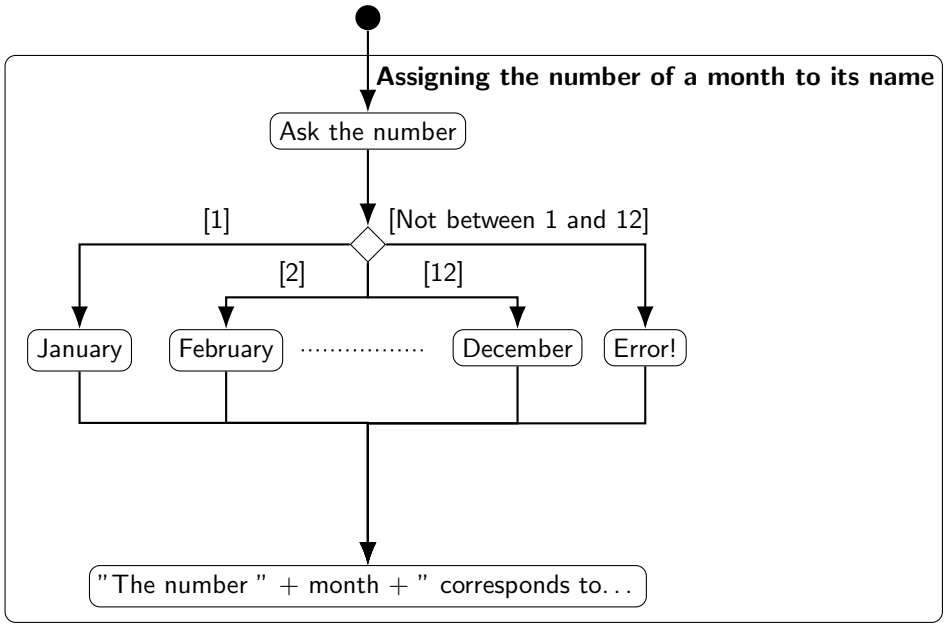


Figure 9: "A flowchart representation of the mapping between month number and name"

```
{
    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.

12.1.0.2 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 can't write two `case` statements with the same value)

12.1.0.3 Example `switch` statement

- This program has the same behavior as our previous example, but uses a **`switch`** statement instaed 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

12.1.0.4 `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";`

12.1.0.5 Intentionally omitting `break`

- Each block of code that starts with a `case` statement must end with a `break` statement; it won't 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!"
}
```

12.1.0.6 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`.

12.1.0.7 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.

12.2 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";
}
```

12.2.0.1 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

12.2.0.2 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";
  ```

13 Loops, Increment Operators, and Input Validation

13.1 The -- and ++ Operators

13.1.0.1 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`

13.1.0.2 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.

13.1.0.3 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.

13.2 While Loops

13.2.0.1 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*

13.2.0.2 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
```

13.2.0.3 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 runs, 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")

13.2.0.4 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.

13.2.0.5 Ensuring the loop ends

- If the loop condition is always true, the loop will never end, and your program will run "forever" (until you forcibly stop it, or the computer shuts down)
- Obviously, if you use the value `true` for the condition, the loop will run forever, like in this example:

```
int number = 1;
while (true)
    Console.WriteLine(number++);
```

- If you don't intend your loop to run 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 run 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 run 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 run 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 run 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` doesn't actually change the value of `number`: $2 \times 0 = 0$. So the loop body will never make the condition `number <= 64` false.

13.2.0.6 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.

13.3 Loops and Input Validation

13.3.0.1 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 can't 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 doesn't 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.

13.3.0.2 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 run 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.

13.3.0.3 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`

14 Do-While Loops and Loop Vocabulary

14.1 The `do-while` Statement

14.1.0.1 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"

14.1.0.2 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

14.1.0.3 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 doesn't 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.

14.1.0.4 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 can't 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>);
```

14.1.0.5 `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

14.2 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").

14.3 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);
```

15 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.

15.1 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

15.1.0.1 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 doesn't 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 can't 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.

15.1.0.2 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 can't 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`.

15.1.0.3 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 can't 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").

15.1.0.4 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`

15.1.0.5 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`

  - 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()`.

15.2 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

15.2.0.1 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'."

15.2.0.2 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()`.

15.2.0.3 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
```

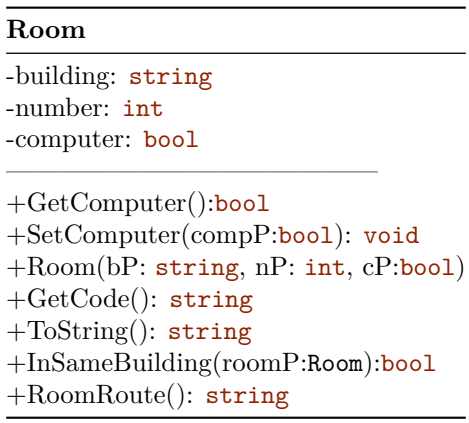
- Note that there are `while` loops to validate each number the user inputs for hours, minutes, and seconds, as well as an outer `while` loop that validates the Time object as a whole.
- The outer loop will continue until the user enters values that make `userTime.ComesBefore(maximumTime)` return `true`.

15.3 Examples

15.3.1 The Room Class

The class and its associated Main method presented in this archive[23] show how you can use classes, methods, constructors and decision structures 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:



15.3.2 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[24] asks you to add the user input validation code, and you can download the following code in this archive[25].

[23] /labs/ValidatingInput/Room.zip
[24] /labs/ValidatingInput
[25] /labs/ValidatingInput/LoanCalculator.zip

```
using System;

class Loan
{
    private string account;
    private char type;
    private int cscore;
    private decimal amount;
    private decimal rate;

    public Loan()
    {
        account = "Unknown";
        type = 'o';
        cscore = -1;
        amount = -1;
        rate = -1;
    }

    public Loan(string nameP, char typeP, int cscoreP, decimal needP, decimal downP)
    {
        account = nameP;
        type = typeP;
        cscore = cscoreP;
        if (cscore < 300)
        {
            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 " + account + $", you borrowed {amount:C} at {rate:P} for "
            + typeName + ".";
    }
}

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; ;
        Console.WriteLine();

        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";
        }

        Console.WriteLine($"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);
    }

}
```

16 Arrays

Arrays are structures that allow you to store multiple values in memory using a single name and indexes.

- Usually all the elements of an array have the same type.
- You limit the type of array elements when you declare the array.
- If you want the array to store elements of any type, you can specify object as its type.

An array can be:

- Single-Dimensional
- Multidimensional (not covered)
- Jagged (not covered)

Arrays are useful, for instance,

- When you want to store a collection of related values,
- When you don't know in advance how many variables we need.
- When you need too many variables of the same type.

16.1 Single-Dimensional Arrays

You can define a single-dimensional array as follow:

```
<type>[] arrayName;
```

where

- `<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:

```
arrayName[<index>] = <value>;  // Assigns <value> to the <index> element of the
 ↪  array arrayName.

Console.WriteLine(arrayName[<index>]); // Display the <index> element of the array
 ↪  arrayName.
```

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[26].

16.1.1 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
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()
```

16.1.2 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];
```

[26] https://cseducators.stackexchange.com/a/5026

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" };
Rectangle[] arrayOfRectangle = new Rectangle[5];  // Assume there is a class called
↪ Rectangle
```

16.1.3 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()};
```

16.2 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 runtime, 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 can't 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.

16.2.1 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.

16.2.2 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.

16.2.3 Example

```
int counter2 = 0;
while (counter2 < customArray.Length)
{
    Console.WriteLine($"{counter2}: {customArray[counter2]}.");
    counter2++;
}
```

Observe that this code doesn't 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.

16.2.4 Loops with Arrays of Objects

In the following example, we will ask the user how many `Item` objects 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;
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 can't 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.

16.3 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.

16.3.1 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.

17 For Loops

17.0.0.1 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

17.0.0.2 `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.

17.1 Limitations and Pitfalls of Using `for` Loops

17.1.0.1 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 can't 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}");
```

17.1.0.2 Accidentally re-declaring a variable

- If your `for` loop declares a new variable in its initialization statement, it can't 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 can't 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}");
}
```

17.1.0.3 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`.

17.2 More Ways to use `for` Loops

17.2.0.1 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.

17.2.0.2 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!");
```

17.2.0.3 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.

17.2.0.4 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 don't 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.

17.3 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 doesn't 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.

18 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());
}
```

18.1 `break` and `continue`

18.1.0.1 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.

18.1.0.2 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.

18.1.0.3 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 can't 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

18.1.0.4 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;
}
```

19 The `static` Keyword

19.1 Static Methods

19.1.0.1 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**

19.1.0.2 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

19.1.0.3 `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 **can't 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?

19.1.0.4 Uses for `static` methods

- Since static methods can't access instance variables, they don't seem very useful
- One reason to use them: when writing a function that doesn't 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 runtime (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.

19.2 Static Variables

19.2.0.1 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++;
```

19.2.0.2 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 can't 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

19.2.0.3 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 don't 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 can't 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");
```

19.2.0.4 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

19.3 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.

20 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[27] class or System.Security.Cryptography.Rando

- 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.

    ```
    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!

[27] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?view=net-5.0
[28] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.randomnumbergenerator?view=net-5.0

```
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;
    }
}
```