Contents

do while

Comparing `while` and `if` statements

- **`while`** and **`if`** are very similar: Both test a condition, execute a block of code if the condition is true, and skip the block of code if the condition is false
- There is only a difference if the condition is true: **`if`** statements only execute the block of code once if the condition is true, but **`while`** statements may execute the block of code multiple times if the condition is true
- Compare these snippets of code:

```
if(number < 3)
{
    Console.WriteLine("Hello!");
    Console.WriteLine(number);
    number++;
}
Console.WriteLine("Done");
```

and

```
while(number < 3)
{
    Console.WriteLine("Hello!");
    Console.WriteLine(number);
    number++;
}
Console.WriteLine("Done");
```

- If `number` is 4, then both will do the same thing: skip the block of code and display "Done".
- If `number` is 2, both will also do the same thing: Display "Hello!" and "2", then increment `number` to 3 and print "Done".

- If `number` is 1, there is a difference: The **`if`** statement will only display "Hello!" once, but the **`while`** statement will display "Hello! 2" and "Hello! 3" before displaying "Done"

Code duplication in `while` loops

- Since the **`while`** loop evaluates the condition before executing the code in the body (like an **`if`** statement), you sometimes end up duplicating code
- For example, consider an input-validation loop like the one we wrote for Item prices:

```
Console.WriteLine("Enter the item's price.");
decimal price = decimal.Parse(Console.ReadLine());
while(price < 0)
{
    Console.WriteLine("Invalid price. Please enter a
↪   non-negative price.");
    price = decimal.Parse(Console.ReadLine());
}
Item myItem = new Item(desc, price);
```

- Before the **`while`** loop, we wrote two lines of code to prompt the user for input, read the user's input, convert it to `decimal`, and store it in `price`
- In the body of the **`while`** loop, we also wrote two lines of code to prompt the user for input, read the user's input, convert it to `decimal`, and store it in `price`
- The code before the **`while`** loop is necessary to give `price` an initial value, so that we can check it for validity in the **`while`** statement
- It would be nice if we could tell the **`while`** loop to execute the body first, and then check the condition

Introduction to `do-while`

- The **`do-while`** loop executes the loop body **before** evaluating the condition
- Otherwise works the same as a **`while`** loop: If the condition is true, execute the loop body again; if the condition is false, stop the loop
- This can reduce repeated code, since the loop body is executed *at least once*
- Example:

```
decimal price;
do
{
    Console.WriteLine("Please enter a non-negative
 ↪  price.");
    price = decimal.Parse(Console.ReadLine());
} while(price < 0);
Item myItem = new Item(desc, price);
```

- The keyword **do** starts the code block for the loop body, but it does not have a condition, so the computer simply starts executing the body
- In the loop body, we prompt the user for input, read and parse the input, and store it in `price`
- The condition `price < 0` is evaluated at the end of the loop body, so `price` has its initial value by the time the condition is evaluated
- If the user entered a valid price, and the condition is false, execution simply proceeds to the next line
- If the user entered a negative price (the condition is true), the computer returns to the beginning of the code block and executes the loop body again
- This has the same effect as the **while** loop: the user is prompted repeatedly until he/she enters a valid price, and the program can only reach the line `Item myItem = new Item(desc, price)` when `price < 0` is false
- Note that the variable `price` must be declared before the **do-while** loop so that it is in scope after the loop. It would not be valid to declare `price` inside the body of the loop (e.g. on the line with `decimal.Parse`) because then its scope would be limited to inside that code block.

Formal syntax and details of `do-while`

- A **do-while** loop is written like this:

```
do
{
    <statements>
} while(<condition>);
```

- The **do** keyword does nothing, but it is required to indicate the start of the loop. You cannot just write a `{` by itself.
  - Unlike a **while** loop, a semicolon is required after **while**`(<condition>)`

- It's a convention to write the **while** keyword on the same line as the closing }, rather than on its own line as in a **while** loop
- When the computer encounters a **do-while** loop, it first executes the body (code block), then evaluates the condition
- If the condition is true, the computer jumps back to the **do** keyword and executes the loop body again
- If the condition is false, execution continues to the next line after teh **while** keyword
- If the loop body is only a single statement, you can omit the curly braces, but not the semicolon:

```
do
<statement>
while(<condition>);
```

do-while loops with multiple conditions

- We can combine both types of user-input validation in one loop: Ensuring the user entered a number (not some other string), and ensuring the number is valid. This is easier to do with a **do-while** loop:

```
decimal price;
bool parseSuccess;
do
{
    Console.WriteLine("Please enter a price (must be
 ↪  non-negative).");
    parseSuccess = decimal.TryParse(Console.ReadLine(),
 ↪  out price);
} while(!parseSuccess || price < 0);
Item myItem = new Item(desc, price);
```

- There are two parts to the loop condition: (1) it should be true if the user did not enter a number, and (2) it should be true if the user entered a negative number.
- We combine these two conditions with || because either one, by itself, represents invalid input. Even if the user entered a valid number (which means `!parseSuccess` is false), the loop should not stop unless `price < 0` is also false.
- Note that both variables must be declared before the loop begins, so that they are in scope both inside and outside the loop body