CS620
Structured Programming
Introduction to Java

Day 7 - Lecture 1

General Recap

R. Voigt

Computer Science Department - NUI Maynooth

General Recap

- Refresh Topics that we've covered before
- Examine some things in greater detail
  - In light of other things we've learned meanwhile
- Opportunity for **Questions!**

Bits & Pieces - Comments

- Comments
  - Ignored by the compiler!
  - Added for the benefit of people reading the code (including the author!)
  - For documentation of the code
  - For keeping notes of what different parts of the code do

- Every bit as important as functional code
  - One of the major differences between ‘good’ programming and ‘bad’ programming!
  - Marks in this and other programming modules will depend on good use of comments

Bits & Pieces - Comments

- Different kinds of comments:
  - Single-line: *// Comment goes here*
  - Multi-line: */* Comment goes here *\/*
  - Documentation: */** Document info *\/*
    - Used by the javadoc tool to automatically generate written documentation for Java software from comments in the source code

```
1  /**
2   * The HelloWorldApp class implements an application that
3   * simply prints "Hello World!" to standard output.
4   */
5  class HelloWorldApp
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("Hello World!"); // Display the string.
10     }
11 }
```

Comments

- Use comments on code that you're reworking or heavily-altering.
- Comment out the old code, then copy & paste it for modification.
  - That way, if the new code doesn't work, you can still refer back to your previous code
- Use the //*/ … //*/ comment-style trick to make life easier!
  - Just one change to enable/disable a comment

Bits & Pieces – Blocks

- Blocks
  - The term ‘block’ refers to a section of code grouped together.
  - Classes, Methods, Conditionals and Loops are typically written as blocks
  - Blocks are usually enclosed by braces

```
1  /**
2   * The HelloWorldApp class implements an application that
3   * simply prints "Hello World!" to standard output.
4   */
5  class HelloWorldApp
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("Hello World!"); // Display the string.
10     }
11 }
```

Bits & Pieces – Braces

- Braces / Curly Brackets
  - These brackets denote the starts and end of a block
  - Everything between the curly brackets of the **HelloWorldApp** block is part of the **HelloWorldApp** class

```
1  /**
2   * The HelloWorldApp class implements an application that
3   * simply prints "Hello World!" to standard output.
4   */
5  class HelloWorldApp
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("Hello World!"); // Display the string.
10     }
11 }
```

Bits & Pieces – Braces

- Indentation
  - Like comments, indentation also makes the difference between '*good*' and '*bad*' code
  - **Readability**
    - A program will still work regardless of good or bad indentation, but don't expect anyone to enjoy reading *(or marking!)* it.

```
1  /**
2   * The HelloWorldApp class implements an application that
3   * simply prints "Hello World!" to standard output.
4   */
5  class HelloWorldApp
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("Hello World!"); // Display the string.
10     }
11 }
```

Braces

- Don’t add unnecessary braces
  - Only add braces to indicate the start and end of a block of code such as the body of a class, method, condition or loop.
  - Variables and Statements don’t need braces
- Line up your braces
- Create new braces in line with the beginning of the block of code you’ve just opened.
  - class, method(), if, for, while, do
  - Brace on the next line, aligned with the block

Braces

```
5     public someMethod()
6     {
7         // Method block - Code goes in here
8
9         if (someCondition)
10        {
11            // Code
12        }
13        else
14        {
15            // Code
16        }
17
18        while(someOtherCondition)
19        {
20            // Code
21
22            while(anotherCondition(
23            {
24                // Code
25            }
26
27        }
28    }
```

Bits & Pieces – Statements

- Statement
  - Think of a statement as being like a command
  - Telling the computer something you want to be taken as fact; or giving it a command
  - **Always** end with a semicolon ;
    - This tells the compiler where the end of the statement is

```
5   class HelloWorldAppString
6   {
7       public static void main(String[] args)
8       {
9           //System.out.println("Hello World!"); // Display the string. Old code commented out!
10          String helloStr = "Hello World! This is a string variable!";
11          System.out.println(helloStr); // Display the string VARIABLE.
12      }
13  }
```

Semicolons

- Don’t put semicolons after if/else/for/while
  - (Unless you have a very good reason to)
- Putting a semicolon directly after a condition or loop will make the compiler ignore the body.
- Yes, semicolons are a pain!

Semicolons

```
3     if(someCondition);
4     {
5         /*
6         * Some code.
7         * Nothing in here will work or run at all.
8         * The semicolon after the parentheses makes
9         * the compiler ignore everything here.
10        */
11    }
12
13    while(someCondition);
14    {
15        /*
16        * Some code.
17        * Nothing in here will work or run at all.
18        * The semicolon after the parentheses makes
19        * the compiler ignore everything here.
20        */
21    }
22
23    for(int i = 0; i < 10; i++);
24    {
25        /*
26        * Some code.
27        * Nothing in here will work or run at all.
28        * The semicolon after the parentheses makes
29        * the compiler ignore everything here.
30        */
31    }
```

Bits & Pieces – Variables

- Variables
  - Places in memory to hold information in a program
  - *helloStr* is a 'String' variable, meaning it holds a string of characters (i.e.; text)
  - Variables are *declared & defined*
  - Values are *assigned* to them
  - '*Passed*' to functions as *arguments*

```
1  class HelloWorldAppStringDecl
2  {
3      public static void main(String[] args)
4      {
5          String helloStr;                                    // Declaration
6          helloStr = "Hello World! This is a string variable!";   // Assignment
7          System.out.println(helloStr);                       // Passing
8      }
9  }
```

Bits & Pieces – Variables

- Passing variables to functions
  - In **HelloWorldApp** we're using the *System.out.println()* function to print our message to the screen.
  - The first argument to that function is a string we want to print.
  - We can concatenate multiple strings to be sent as one to the function:

```
1  class HelloWorldAppStringDeclConcat
2  {
3      public static void main(String[] args)
4      {
5          String helloStr = "Hello World! This is a string variable!";
6          String helloStr2 = "This is another string variable!";
7          System.out.println(helloStr + " " + helloStr2);
8      }
9  }
```

Keywords

- There are 50 keywords in Java that identify core parts of the language
- You can’t name your variables, classes or methods the same as any of these keywords.
- You don’t need to learn them off by heart, but you’ll become familiar with most of them over time just by working with Java.

Keywords

- Some commonly-encountered keywords:
  - break
  - case
  - switch
  - char
  - class
  - char / int / float / double
  - if / else
  - for / while
  - import
  - public / private / protected
  - this
  - void
  - new

Types

- Java is a ‘strongly typed’ language
- This means that it’s strict about what kind of data can be stored in different kinds of variables
- In practice, this means we need to declare special variables for different kinds of information

Types

- When we declare a variable, we tell the compiler what the variable's *type* is.

- The compiler converts that into an instruction for the computer to set aside a certain amount of space in memory for that variable.

Types

- Different *types* have different space requirements
  - We'll see this when we look at the different fundamental types more closely
- It's good practice for computational efficiency to choose the smallest *type* that will hold the information you want to store
  - For a simple program on a fast computer the difference will be entirely negligible
  - If your program is very complex and performance-intensive, or you're using a computer with limited performance, this matters much more!

Basic Types

- Integer
  - Holds **whole** numbers (only)
  - Keyword: ‘int’
  - Range: *-2,147,483,648* to *2,147,483,647*
  - If you try to store a decimal value in an integer it will be rounded to the nearest whole number
    - The rounding discards the decimal point and everything after it; it won’t round upwards!
    - Example:
      - `int x = 5 / 2`
      - The value of x will be 2, not 2.5!
  - Integers are suitable for basic counting and storing simple numbers

Basic Types

- Double
  - Holds **Decimal** numbers
  - Keyword: ‘double’
  - Range: **Huge**
    - The maximum range of a double-precision floating-point variable depends on the JVM and the architecture of the computer!
    - Suffice to say, a double is typically big enough to hold most decimal numbers that you’ll have to work with.
  - Doubles take up more space in the computer’s memory than ints
  - Generally speaking, use doubles any time you need to work with decimal numbers

Basic Types

- Float
  - Holds **Decimal** numbers, but smaller than a double
  - Keyword: ‘float’
  - Range: **Still Huge**
    - The maximum range of a floating-point variable depends on the JVM and the architecture of the computer!
    - Suffice to say, a float is typically big enough to hold most decimal numbers that you’ll have to work with.
  - Floats take up more space in the computer’s memory than *int*s, but less than *double*s
  - You would use floats in a very performance-intensive program where speed matters and where you know your decimals will be small enough to fit!

Basic Types

- Byte / Short / Long
  - Hold whole numbers
  - Similar to '*int*' but different ranges
  - Keywords: *byte*, *short*, *long*
  - Ranges:
    - Byte: -128 to +127
    - Short: -32,768 to 32'767
    - Long: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
      - (You don't need to know these numbers off by heart..)

Basic Types

- Boolean
  - Holds boolean values: true / false **only**
  - Keyword: *boolean*
  - Range:
    - *true* or *false*
  - Booleans are used to store the results of comparisons
  - Example:
    - 4 == 5 returns the boolean value: *false*
    - 3 < 6 returns the boolean value: *true*

Basic Types

- Char
  - Holds single characters (text)
  - Keyword: *char*
  - Chars are treated as symbols
  - If you store a number as a char, such as:
    - char x = 5
    - The computer will see this as the **symbol** ‘5’ or the **letter** ’5’, even
    - You cannot perform maths on *chars*!
  - You will rarely work with individual chars

Not-So Basic Types - Strings

- String
  - Holds multiple *chars*
  - Keyword: *String*
    - Note the capital S
      - This indicates that the String is not a ‘fundamental’ type
      - A String is a constructed type, made up of multiple *chars* stored as a string of text
  - Any piece of text longer than one character should be stored as a string
  - Typically, you’ll store even single characters in strings for simplicity, even though it is not as efficient!

Variables

- Setting-up, managing and working with variables

Variable Declaration

- Declaring a variable:
  - Specify its type, specify its identifier, end with a semicolon;
  - Examples:
    - int x;
    - int someNumber
    - char y;
    - char aSingleCharacter;
    - double z;
    - double someBigDecimalNumber;
  - It’s good practice to give your variables descriptive names
    - “int aNumberUsedToCountSomething” is better than “int x”

Variable Assignment

- Assigning values to variables:
  - Use the assignment operator: ‘=‘
  - In maths, equals is how we ‘get’ our answer
    - 5 + 5 = ?
      - Answer: 10 (Just in case you were wondering!)
  - In programming, equals is how we assign a value
    - x = 5 + 5
    - The value of x is now 10
  - This might take some getting used to
    - A handy way to think of it is that we put our answer on the left in programming:
    - Answer = expression

Variable Assignment

- Assigning values to variables:
  - If the variable doesn't already exist:
    - Declare it and give it a value at the same time
    - `int someNumber = 10;`
  - If the variable already exists (it's been declared already):
    - Just give it a value, don't re-specify the type
    - `someNumber = 10;`
    - If you specify the type, you'll either get an error claiming that the variable already exists; or you'll create a new variable with the same name somewhere else in memory!
    - This depends on 'scope', which we'll talk about later.

Variable Initialization

- It's necessary to initialize your variables
- This means giving them a value to start out with, or making sure to assign them a value at least once before using them for anything else
- Uninitialized variables typically contain 'junk' data left over from other variables that used the same space in memory some point previously on the computer.

Variable Initialization

- We saw previously that we can declare a variable without a value such as:
  - `int someNumber;`
  - What happens when you try to do something with an uninitialized variable?
  - What's the result of "someNumber + 10;" ?

- Problems! Errors!
  - The Java Compiler won't let you compile a program that tries to use an uninitialized variable.
  - This prevents bugs creeping into code!

Operators

- Doing stuff with variables!

Operators

- Operators in java are generally the same as symbols/signs in mathematics

- ‘+’, ‘-’, ‘*’, ‘/’
  - Addition, Subtraction, Multiplication, Division

  - The asterisk and slash symbols are used in most programming languages instead of the traditional ‘×’ for multiply and ‘÷’ (*obelus*) for divide, respectively

Operators

- Not all of the operators have the same meanings as you might be used to, however.
  - ‘+’ Means addition
    - *‘+’ Also means ‘positive’*
    - ‘+’ Also means ‘string concatenation’
      - (i.e.; glue two strings together as one)
      - *“Some Words”* + *“Some other words*” in java terms gives: *“Some words Some other words”*
  - *‘-’ Means subtraction*
    - *‘-’ Also means ‘negative’*
  - *‘*’ Means multiplication*
  - *‘/’ Means division*

Operators

- *Some operators you might not be familiar with:*
  - *'%' Means modulo* – To calculate the remainder
    - *Example:*
      - *10 % 5 gives remainder 0*
      - *11 % 5 gives remainder 1*
    - *Modulo is useful for checking if a number is odd or even:*
      - *If the remainder of a number divided by 2 is not 0, then the number must be odd!*

Operators

- *Some operators you might not be familiar with:*
  - '++' means 'increment'
    - *int x = 5;*
    - *x++;*
    - Result: x is now 6
    - This is the same as the code:
      - int x = 5;
      - x = x + 1;
      - Result: x is now 6
  - *'--' means 'decrement'*
    - *int x = 5;*
    - *x --;*
    - *Result: x is now 4*
    - This is the same as the code:
      - int x = 5;
      - x = x - 1;
      - Result: x is now 4

Logic Operators

- More operators:
  - '==' *(is Equal to)*
    - Used with conditionals
    - **NOT** the same as '='
    - More on this the next day!
  - '!=' *(is not Equal to)*
    - Used with conditionals
  - *> (Greater than)*
    - *>= (Greater than or equal to)*
  - *< (Less than)*
    - *<= (Less than or equal to)*
- We'll look at these operators in more detail when we deal with conditionals and boolean algebra later

Conditional Control Structures

- If
- Then
- Else
- Switch

Conditional Control Structures

- Also known as ‘program flow-control’
  - Think of a pipe with flowing water and valves to direct the flow in one direction or another
- Provides a huge increase in the potential for complex programs
- Crucial for **decision-making** in software
- Alter the **sequence of execution**

Sequence of Execution

- When a program typically runs, it executes instructions (lines of code) in sequence
- Control structures can change the order of execution
- Control structures can enable/disable sections of code

If-Then

- The most basic control structure is **if-then**:
  - `if (someBoolean)`
  - `{`
    - `// Do stuff`
  - `}`
  - Tells your program to execute a certain section of code *only if* a particular test evaluates to true.
  - The opening brace is the equivalent of ‘then’ when saying ***if x, then do y***

If-Then

- If *isMoving* is false, the execution jumps to the end of the ‘if’ block.

```
4   // A method somewhere in a 'Bike' class
5   void applyBrakes()
6   {
7       // the "if" clause: bicycle must be moving
8       if (isMoving)
9       {
10          // the "then" clause: decrease current speed
11          currentSpeed--;
12      }
13  }
```

- Braces can be left out **if** there’s only one statement inside the block:

```
3   void applyBrakes()
4   {
5       // same as above, but without braces
6       if (isMoving)
7           currentSpeed--;
8   }
```

- It’s good practice to use braces anyway

If-Then-Else

- The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

- You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is **not** in motion.

If-Then-Else

- In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
3   void applyBrakes()
4   {
5       if (isMoving)
6       {
7           currentSpeed--;
8       }
9       else
10      {
11          System.err.println("The bicycle has already stopped!");
12      }
13  }
```

A more complex example:

```
1  class IfElseDemo
2  {
3      public static void main(String[] args)
4      {
5          int testscore = 76;
6          char grade;
7
8          if (testscore >= 90)
9          {
10             grade = 'A';
11         }
12         else if (testscore >= 80)
13         {
14             grade = 'B';
15         }
16         else if (testscore >= 70)
17         {
18             grade = 'C';
19         }
20         else if (testscore >= 60)
21         {
22             grade = 'D';
23         }
24         else
25         {
26             grade = 'F';
27         }
28         System.out.println("Grade : " + grade);
29     }
30 }
```

A more complex example:

- The output from the program is:
  - `Grade = C`
- You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60.
- However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

Another Example

```
2   void someMethod()
3   {
4
5       int someNumber = 3;
6
7       if (someNumber < 2)
8       {
9           // Do something
10      }
11      else if (someNumber > 5)
12      {
13          // Do something else
14      }
15      else if (someNumber <= 4)
16      {
17          // Do something else
18      }
19      else
20      {
21          // Do some default action, such as giving an error
22      }
23  }
```

Another Example

- What happens if we take out the ‘else’ keywords?

```
2   void someMethod()
3   {
4
5       int someNumber = 3;
6
7       if (someNumber < 2)
8       {
9           // Do something
10      }
11      if (someNumber > 5)
12      {
13          // Do something else
14      }
15      if (someNumber <= 4)
16      {
17          // Do something else
18      }
19  }
```

Switch

- Switch allows multiple execution paths depending on a single variable.

- With switch, you set up a variable to determine which path to take; then define a ‘case’ for each path you want.

Switch - Example

```
4       int month = 8;
5       String monthString;
6       switch (month)
7       {
8           case 1:  monthString = "January";
9                    break;
10          case 2:  monthString = "February";
11                   break;
12          case 3:  monthString = "March";
13                   break;
14          case 4:  monthString = "April";
15                   break;
16          case 5:  monthString = "May";
17                   break;
18          case 6:  monthString = "June";
19                   break;
20          case 7:  monthString = "July";
21                   break;
22          case 8:  monthString = "August";
23                   break;
24          case 9:  monthString = "September";
25                   break;
26          case 10: monthString = "October";
27                   break;
28          case 11: monthString = "November";
29                   break;
30          case 12: monthString = "December";
31                   break;
32          default: monthString = "Invalid month";
33                   break;
34      }
35      System.out.println(monthString);
```

Switch

- In this case, August is printed to standard output.
- The body of a switch statement is known as a *switch block*.
- A statement in the switch block can be labelled with one or more case or *default* labels.
- The switch statement evaluates its expression, then executes all statements that follow the matching case label.

Switch

- You could also display the name of the month with if-then-else statements:

```
int month = 8;
if (month == 1)
{
    System.out.println("January");
}
else if (month == 2)
{
    System.out.println("February");
} ... // and so on
```

- Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.

- An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Switch - Break Statement

- Another point of interest is the break statement.
- Each break statement terminates the enclosing switch statement.
- Control flow continues with the first statement following the switch block.
- The break statements are necessary because without them, statements in switch blocks *fall through*:
  - All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered.

Loops

- A very useful conditional control structure, Loops are used to do something over and over in a program.

- You will often find a use for a piece of code that runs repetitively until a certain condition is met, or to count through a list, etc.

Loops

- Computers in general run in a constant loop, cycling over and over.

- When your computer is idle on the desktop it appears to be doing nothing, but is in fact running many continuous loops and constantly updating millions of variables and running functions accordingly.

While

- The while statement continually executes a block of statements while a particular condition is true.
- Its syntax can be expressed as:
  - `while (expression) { statement(s) }`
- The **while** statement evaluates *expression*, which must return a boolean value.
- If the expression evaluates to true, the **while** statement executes the *statement*(s) in the while block.
- The **while** statement continues testing the expression and executing its block until the expression evaluates to false.

While - Infinite Loop

- The following code will cause an *infinite loop*; otherwise known as a **crash.**
  - `while(true) { doSomething(); }`
- The expression ‘true’ will never evaluate as ‘false’ under any circumstances, so the loop never stops.
- Because a program executes its instructions in sequence and only ever does one thing at a time, it gets *‘stuck’* inside the infinite loop and appears to freeze up!

While - Example

- The following program will print 10 messages to the screen by executing the same statements 10 times over

- The repeated statements are the ones inside the **while** loop's block.

```
1  class WhileDemo
2  {
3      public static void main(String[] args)
4      {
5          int count = 1;
6          while (count <= 10)
7          {
8              System.out.println("Count is: " + count);
9              count++;
10         }
11     }
12 }
```

Do-While

- Java programming also provides a **do-while** statement, which can be expressed as follows:
  - `do { statement(s) } while (expression);`
- The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top.
- Therefore, the statements within the do block are always executed at least once.

Do-While

```
1   class DoWhileDemo
2   {
3       public static void main(String[] args)
4       {
5           int count = 1;
6           do
7           {
8               System.out.println("Count is: " + count);
9               count++;
10          } while (count < 11);
11      }
12  }
```

For

- **For** provides a compact way to iterate over a range of values.
- Programmers often refer to it as the *"for loop"* because of the way in which it repeatedly loops until a particular condition is satisfied.
- It can do the same things as a while loop, but in a more compact manner.

For

- A typical **for** loop:

```
1  class ForDemo
2  {
3      public static void main(String[] args)
4      {
5          for(int i=1; i<11; i++)
6          {
7              System.out.println("Count is: " + i);
8          }
9      }
10 }
```

- The above example does exactly the same as our first **while** loop:

```
1  class WhileDemo
2  {
3      public static void main(String[] args)
4      {
5          int count = 1;
6          while (count <= 10)
7          {
8              System.out.println("Count is: " + count);
9              count++;
10         }
11     }
12 }
```

Loops in Reverse

- Loops can work ‘backwards’ too
- In fact, you simply need to set them up so that they check a condition that will eventually reach ‘false’ for the loop to not be infinite.
- ‘Reverse’ **for** loop:

```
1  class ForDemo_Reverse
2  {
3      public static void main(String[] args)
4      {
5          for(int i=10; i>=0; i--)
6          {
7              System.out.println("Count is: " + i);
8          }
9      }
10 }
```

Loops in Reverse

- 'Reverse' **while** loop:

```
1   class WhileDemo_Reverse
2   {
3       public static void main(String[] args)
4       {
5           int count = 10;
6           while (count >= 1)
7           {
8               System.out.println("Count is: " + count);
9               count--;
10          }
11      }
12  }
```

Break

- We saw **break** used before with switch
- It can also be used in loops to exit the loop ‘early’

```
3   for (int i = 0; i <= 10; i++)
4   {
5       if (i == 5)
6       {
7           System.out.println("Reached i==5!");
8           break;
9       }
10  }
```

- This loop is set up to run 11 times, but will exit after 5 iterations because of **break;**

Return

- Functions similarly to **break**
- However, whereas **break** simply exits a loop or condition early, **return** serves the purpose of **returning** a variable or value from a function/method
- **Return** used inside a loop/conditional will break out of the whole **method**, not just that loop/conditional.

Return

- Method return Example

Scope

- Refers to the lifetime and accessibility of a variable
- A variable's scope depends on where it is declared
- A variable declared at the top of a class will be available throughout that class (in all its methods)
- A variable declared in a method will only be usable within that method
- A variable out of scope no longer 'exists'!

Scope Example

```
2   public class AllAboutHours
3   {
4       private final int NUMBER_OF_HOURS_IN_A_DAY = 24;
5
6       public int calculateHoursInDays(int days)
7       {
8           return days * NUMBER_OF_HOURS_IN_A_DAY;
9       }
10
11      public int calculateHoursInWeeks(int weeks)
12      {
13          final int NUMBER_OF_DAYS_IN_A_WEEK = 7;
14          return weeks * NUMBER_OF_DAYS_IN_A_WEEK * NUMBER_OF_HOURS_IN_A_DAY;
15      }
16
17      public void errorMethod()
18      {
19          //Error - NUMBER_OF_DAYS_IN_A_WEEK doesn't exist here
20          System.out.println(NUMBER_OF_DAYS_IN_A_WEEK);
21      }
22  }
```