**Lesson 1. Variables and Jave Data Types**

|  |  |
| --- | --- |
| **Variables** | containers for sorting data values |
| **String** – surrounded by double quote.  **int** – whole numbers w/o decimals.  **float** – numbers with decimals.  **char** – single character, single quotes  **boolean** – true or false |
| **Declaring Variables** | Syntax  **type variable = value** |
| **Naming Conventions** | make programs more understandable by making them easier to read. |
| **Hungarian Notation** | the name of a variable or function indicates its intention or kind. |
| specifies a variable with data type and its description. |
| **Data Types** | * **Primitive Data Types** * **Non-Primitive Data Types** |
| **Primitive Data Types** | byte, short, int, long, float, double, boolean, char |
| predefined (already defined) in Java. Specifies the size and type of variable values, and it has no additional methods. |
| **8 primitive data types in Java:** |
| **Non-Primitive Data Types** | String, Arrays, and Classes |
| created by the programmer and is not defined by Java (except for String). Can be used to call methods to perform certain operations. |
| **Java Type Casting** | Assign a value of one primitive data type to another type. |
|  | Two Types of Casting   * **Widening Casting (automatically)** * **Narrowing Casting (manually)** |
| **Widening Casting (automatically)** | converting a smaller type to a larger type size.  byte > short > char > int > long > float > double |
| **Narrowing Casting (manually)** | converting a larger type to smaller size type  double > float > long > int > char > short > byte |
| **Java Operators** | used to perform operations on variables and values. |
| * **Arithmetic Operators** * **Assignment Operators** * **Comparison Operators** * **Logical Operators** * **Bitwise Operators** |
| **Arithmetic Operations** | used to perform common mathematical operations. |
| **Assignment Operators** | used to assign values to variables. |
| **Comparison Operators** | used to compare two values. |
| **Logical Operators** | used to determine the logic between variables and values. |
| **Strings Length** | contain methods that can perform certain operations on strings.  the length of a string can be found with the **length()** method. |
| **Change Case** | * toUpperCase() * toLowerCase() |
| **indexOf()**  **method** | methods return the index (the position) of the first occurrence of a specified text in a string (including whitespace). |
| **String Concatenation** | the + operator can be used between strings to combine them. |
| **concat()**  **method** | method to concatenate two strings |
| **Backslash (\)** | turns special characters into string characters. |
| **contains() method** | checks whether a string contains a sequence of characters. |
| **substring()**  **method** | extract characters, between to indices (positions), from string, and returns the substring. |
| does not change the original string |
| **trim()**  **method** | Removes whitespace from both ends of a string. |

**Lesson 2. Control and Iteration**

|  |  |
| --- | --- |
| A program executes from top to bottom, except when we use control statements. Then, we can control the order of execution of the program, based on logic and the values. | |
| three **Control Statements** categories | * **Selection statement** * **Iteration statement** * **Jump statement** |
| **Selection Statements** | allow you to control the flow of program execution, on the basis of the outcome of an expression or state of a variable, known during runtime |
| **Categories of Selection Statements** | * **If statements** * **If-else statements** * **If-else-if statements** * **Switch statements** |
| **If statements** | The first contained statement (that can be a block) of an if statement, only executes when the specified condition is true. |
| If the condition is false and there is no keyword, then the first contained statement will be skipped and execution continues with the rest of the program. |
| **If-else statements** | if the specified condition in the if the statement is false, then the statement after the else keyword (that can be a block) will execute. |
| **If-else-if statements** | Whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true, then the else block will execute. |
| **Switch statements** | multi-way branch statement |
| The switch statement of Java is another selection statement that defines multiple paths of execution, of a program. It provides a better alternative than a large series of if-else-if statements. |
| **Iteration statements** | Repeating the same code fragment several times, until a specified condition is satisfied, is called iteration. |
| execute the same set of instructions, until a termination condition is met. |
| Java provides the following loop for iteration statements: | * **While loop** * **For loop** * **Do-while loop** * **For-each loop** |
| **While loop** | It continually executes a statement (that is usually a block) while a condition is true. The condition must return a boolean value. |
| **Do-while loop** | do-while evaluates its expression at the bottom of the loop, instead of the top. |
| The do-while loop executes at least one time, then it will check the expression prior to the next iteration. |
| **For loop** | executes a statement (that is usually a block) as long as the boolean condition evaluates to true. |
| combination of the three elements initialization statement, boolean expression and increment or decrement statement |
| **Syntax:**  **for** ( < initialization > ; < condition > ; < increment or decrement statement > ) {  < block of code >  } |
|  | The initialization block executes first before the loop starts. It is used to initialize the loop variable.  The condition statement evaluates every time prior to when the statement (that is usually a block) executes. If the condition is true then only the statement (that is usually a block) will execute.  The increment or decrement statement executes every time, after the statement (that is usually a block). |
| **For each loop** | This loop is basically used to traverse the array or collection elements. |
| **Jump Statements** | are used to unconditionally transfer the program control to another part of the program. |
| Java provides the following jump statements: | * **Break statements** * **Continue statements** * **Return statements** |
| **Break Statements** | immediately quits the current iteration and goes to the first statement, following the loop. Another form of break is used in the switch statement. |
| Unlabeled Break Statement | This is used to jump program control out of the **specific loop on the specific condition.** |
| Labelled Break Statement | This is used for when we want to jump the program control out of **nesting loops or multiple loops**. |
| **Continue Statements** | used when you want to **continue running the loop**, with the next iteration, and want to skip the rest of the statements of the body, for the current iteration. |
| Unlabeled Continue Statement | This statement **skips the current iteration of the innermost** for, while and do-while loop |
| Labeled Continue Statement | This statement **skips the current iteration of the loop with the specified label.** |
| **Return Statements** | used to immediately quit the current method and return to the calling method. |
| It is mandatory to use a return statement for non-void methods to return a value. |

**Lesson 3. Methods**

|  |  |
| --- | --- |
| **Method** | block of code which only runs when it is called. You can pass data, known as parameters, into a method. |
| Methods are used to perform certain actions, and they are also known as **functions**. |
| A method must be declared within a class |
| It is defined with the name of the method, followed by parentheses (). |
| Example  Create a method inside Main:  public class Main {  static void myMethod() {  // code to be executed  }  } |
| **static** | means that the method belongs to the Main class and not an object of the Main class. |
| **void** | means that this method does not have a return value. |
| **Call a Method** | To call a method in Java, write the method's name followed by two parentheses () and a semicolon; |
|  | A method can also be called multiple times: |
| **parameter** | Information can be passed to methods as \_\_\_\_\_ |
| **Parameter** | act as variables inside the method. |
| are specified after the method name, inside the parentheses |
| You can add as many parameters as you want, just separate them with a comma. |
| **Argument** | When a parameter is passed to the method, it is called an argument |
| public class Main {  static void myMethod(String fname) {  System.out.println(fname + " Refsnes");  }  public static void main(String[] args) {  myMethod("Liam");  myMethod("Jenny");  myMethod("Anja");  }  }  // Liam Refsnes  // Jenny Refsnes  // Anja Refsnes | |
| **Multiple Parameters** | You can have as many parameters as you like |
| Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order. |
| **Return Values** | The *void* keyword indicates that the method should not return a value. |
| If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method |
|  | public class Main {  static int myMethod(int x) {  return 5 + x;  }  public static void main(String[] args) {  System.out.println(myMethod(3));  }  }  // Outputs 8 |
| **Java Scope** | In Java, variables are only accessible inside the region they are created. This is called scope. |
| **Method Scope** | Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared |
| **Block Scope** | refers to all of the code between curly braces *{}*. Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared |
| **Java Recursion** | Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve |
| **Infinite Recursion** | Never stops calling itself |
| **Halting Condition** | Every recursive function should have a halting condition, which is the condition where the function stops calling itself |

**Lesson 4. Class and Object**

|  |  |
| --- | --- |
| **OOP** | stands for Object-Oriented Programming |
| **Procedural Programming** | is about writing procedures or methods that perform operations on the data |
| **Object-Oriented Programming** | is about creating objects that contain both data and methods. |
| Object-oriented programming has several **advantages** over procedural programming: | • OOP is faster and easier to execute  • OOP provides a clear structure for the programs  • OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug  • OOP makes it possible to create full reusable applications with less code and shorter development time |
| **"Don't Repeat Yourself" (DRY) principle** | is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it. |
| **Classes and Objects** | are the two main aspects of object-oriented programming |
| **Class** | Is a template for objects |
| **Object** | Is an instance of a class |
| When the individual objects are created, they inherit all the variables and methods from the class. | |
|  | **Java** is an **object-oriented programming language**. |
|  | Everything in Java is associated with classes and objects, along with its attributes and methods. |
|  | A **Class** is like an object constructor, or a "blueprint" for creating objects. |
| **Create a Class** | To create a class, use the keyword class: Main.java |
| A **class** should always start with an **uppercase first letter**, and that the name of the java file should match the class name. | |
| **Create an Object** | In Java, an object is created from a class. |
|  | To create an object of Main, specify the class name, followed by the object name, and use the keyword new: |
|  | **Example**  Create an object called "myObj" and print the value of x:  public class Main {  int x = 5;  public static void main(String[] args) {  Main myObj = new Main();  System.out.println(myObj.x);  }  } |
| **Accessing Object of Other Classes**  You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)). | |
| **Java Class Attributes** | In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an attribute of the class. Or you could say that **class attributes** are variables within a class:  Example  Create a class called "Main" with two attributes: x and y:  public class Main {  int x = 5;  int y = 3;  }  Another term for class attributes is **fields**. |
| **Accessing Attributes** | You can access attributes by creating an object of the class, and by using the dot syntax (.): |
| **Modify Attributes** | You can also modify attribute values:  **Example**  Set the value of x to 40:  public class Main {  int x;  public static void main(String[] args) {  Main myObj = new Main();  myObj.x = 40;  System.out.println(myObj.x);  }  }  Or override existing values:  **Example**  Change the value of x to 25:  public class Main {  int x = 10;  public static void main(String[] args) {  Main myObj = new Main();  myObj.x = 25; // x is now 25  System.out.println(myObj.x);  }  }  If you don't want the ability to override existing values, declare the attribute as final:  **Example**  public class Main {  final int x = 10;  public static void main(String[] args) {  Main myObj = new Main();  myObj.x = 25; // will generate an error: cannot assign a value to a final variable  System.out.println(myObj.x);  }  }  The **final** keyword is useful when you want a variable to always store the same value.  The **final** keyword is called a "**modifier**". |
| **Java Class Methods** | You learned from the Java Methods chapter that methods are *declared within a class*, and that they are *used to perform certain actions*: |
|  | **myMethod**() prints a text (the action), when it is called. To call a method, write the method's name followed by two parentheses () and a semicolon; |
| **static method** | which means that it can be accessed without creating an object of the class |
| **public** | which can only be accessed by objects  **Public** methods must be called by creating objects |
| **dot (.)** | is used to access the object's attributes and methods. |
| To call a method in Java, write the method name followed by a set of parentheses (), followed by a semicolon (;). | |
| A class must have a matching filename (Main and Main.java). | |