Java

From code in Sublime Text to make it run:

1. Create a class
    a. Public (Modifier) class (*Name of the class*){}
2. Create a Method
    a. Static
        public static void main(){}
        main() = method
    b. Dynamic
        Code missing

Basics

Methods:

(block of code which only runs when called. Can pass data, parameters, into it. Used to perform certain actions, aka **functions**.)

- Must be declared within a class, defined by ***methodName*** followed by **()**
    - public class ***className***
      static void ***methodName***() {
      code
      }
        - static = method belong to ***className*** class and not object of it
            - Can access without creating an object of the class
        - void = method does not have a return value

Parameters

- Acts as variables inside a method
- Are specified after the method name inside parenthesis
- Can add as many as possible, just separate with comma (“,”)

Return

- Use a ***primitiveDataType*** instead of ***void***
- Use ***return*** inside Method
    - static ***variableType methodName*** (***variableType variableName***) {
      return number + ***variableName***;
      }

Comments:

- Short comment (only 1 line): //
- Long comment (more than 1 line): /* … */

Variables (must all be identified with unique names → Identifier):

- string: stores texts. Texts must be in double quotation marks (“ ”)
- int: stores integers. From 2^{-31}bits to $2^{31} - 1$ bit.
    - It is possible to create more than 1 variable: use “,” to separate them
- float: stores floating-point numbers, with decimals (1.99 or -1.99)
- char: stores single characters. In single quotation marks (‘ ’)
- boolean: stores values with 2 states (True or False)

To combine text and variable or variable and variable use “+”
Eg:
String name = “John”
System.out.println(“Hi” + name)

Data Types (variables must be a specified data type):

- Primitive data types: (1 byte = 8 bits)
    - Byte (1 byte: whole numbers from $-\ 2^7\ to\ 2^7\ -\ 1$)
    - Short (2 bytes: whole numbers from $-\ 2^{15}\ to\ 2^{15}\ -\ 1\ )$
    - Int (4 bytes: whole numbers from $-\ 2^{31}\ to\ 2^{31}\ -\ 1\ )$
    - Long (8 bytes: whole numbers from $-\ 2^{63}\ to\ 2^{63}\ -\ 1$)
    - Float (4 bytes: fractional numbers, sufficient for 6-7 decimal digits)
    - Double (8 bytes: fractional numbers, sufficient for 15 decimal digits)
    - Boolean (1 bit: store True & False → 0 and 1)
    - Char (2 bytes: single character/letter or ASCII values)
- Non-primitive data types
    - String
    - Array
    - Classes

Numbers:

- Integer types
    - byte
    - short
    - int (most used)
    - long (Should end with “L”)
- Floating point types (can use E, 10^):
    - float (should end with “f”)
    - double (most used) (should end with “d”)

Java Type Casting

- Widening Casting (automatically)
    - Converting smaller type to larger type size
        - byte -> short -> char -> int -> long -> float -> double
- Narrow Casting (manually)

- Converting larger type to smaller type size
  - double -> float -> long -> int -> char -> short -> byte
    - (smaller variableName) variableName

Operators

- Value = Operand (50, 100 etc.)
  -
- Operation = Operator (+)
  - 5 Groups (Operators):
    - Arithmetic operators
      - + Addition
        - Adds 2 values
      - - Subtraction
        - Subtracts 1 value from other
      - * Multiplication
        - Multiplies 2 values
      - / Division
        - Divide 1 value from another
      - % Modulus
        - Returns the division remainder
      - ++ Increment (++x = x+1)
        - Increases the value of a variable by 1
          - x++ vs ++x
            - x++ = number increase **after** another operation
            - ++x = number increase **before** another operation
      - -- Decrement (--x = x-1)
        - Decreases the value of a variable by 1
          - x-- = number decrease **after** another operation
          - --x = number decrease **before** another operation
    - Assignment operators
      - =
      - +=
      - -=
      - *=
      - /=
      - %=
      - &=
      - |=
      - ^=
      - >>=
        - Far greater than
      - <<=
        - Far less than
    - Comparison operators
      - == Equal to

        - != Not equal
        - > Greater than
        - < Less than
        - >= Greater than or equal to
        - <= Less than or equal to
    - Logical operators
        - && Logical and
            - True if both statement are true
        - || Logical or
            - True if one of statements is true
        - ! Logical not
            - Reverse the result, false if result is true
    - Bitwise operators

Operator Precedence

1. ! ++ --
2. * / %
3. + -
4. < > <= >=
5. == !=
6. &&
7. ||
8. = += -= *= /= %=

Booleans

- Values (declared with *boolean*)
    - True
    - False
- Expression
    - Returns a boolean value (True/False)
    - Use Comparison Operators

Escape Sequence

- \n = new line
- \" = double quote
- \\ = backslash

Control Structure

- switch Mechanism by which you make statement of a program run in a nonsequential order.

2 types:

- Decision making (based on truth value decide which path to follow)
    - if (execute a block of code, if true)
        - if (***boolean expression***)

{
statement
}

- **Statement** will be executed only if the **boolean expression** is true

- else (execute a block of code, if condition false)
  - if (***boolean expression***)
    {
    statement
    }
    else
    {
    statement
    }
    - If **boolean expression** is true, the **statement** following it is executed
    - If **boolean expression** is false, **statement** after else is executed
- else if (specify a new condition to test, if 1st condition false)
  - if (***boolean expression***)
    {
    statement
    } else if (***boolean expression***)
    {
    statement
    }
    - Short Hand
      - variable = (condition) ? expressionTrue : expressionFalse;
- Switch (specify many alternative blocks of code to be executed)
  - switch(***boolean expression***) {
    case x:
    statement
    break;
    case y:
    statement
    break;
    default:
    statement
  - Is evaluated once
  - Value of Boolean expression is compared with the value of each case
  - If there is a match, associated block of code is executed
  - Break and default are optional

Break

- Stop execution of more code in case testing inside block
- → no more testing

Default

- Run a code if there is no case match

Loop (Block of code executed as a specified condition is reached)

- While (loops as long as condition is true)
    - while (***condition***) {
      statement
      }
    - **Do/While (Variant of while, execute code once, before checking if condition is true)**
        - do {
          statement
          }
          while (***condition)***;
- For (know the quantity of loop)
    - for (***statement 1***; ***statement 2***; ***statement 3***) {
      statement
      }
    - For-Each (exclusively to loop through elements in an array)
        - for (***variableType*** : ***arrayName***) {
          statement
          }
            - ***variableType***[] cars = {***variable***};
              for (String i : cars) {
                System.out.println(i);
              }
                - Output = all elements in Array car
- Iteration

Continue (breaks one iteration, if a specified condition occurs and continues in the next iteration)

- for (int i = 0; i < 10; i++) {
  If (i == 4) {
  Continue;
  }
  System.out.println(i);
  }

Break/Continue in While loop

- int = 0;
  while (i < 10) {
  System.out.println(i);
  i++;
  if (i == 4) {
  break/continue;

```
    }
}
```

Arrays (used to store multiple values in a single variable to declare use "[]")

- ***variableType***[] ***variableName*** = {***String***}
- ***variableType***[] ***variableName*** = {***Integers***}

Access Elements of Array (access by referring to the index number)

Note: Index starts with 0 (1st elements)

- ***variableType variableName*** = {***String***}
  System.out.println{***variableName***[***indexNumber***]};

Change Array Element (refer to index number)

- ***variableName***[***indexNumber***] = ***newString/Integer***

Array Length (how many elements an array has by using *length* property)

- ***variableType***[] ***variableName*** = {***variable***};
  System.out.println(***variable.length***);

Loop through Array

- **For**
  - ***variableType***[] ***variableName*** = {***variable***};
    for (int i = 0; i < ***variable.length***; i++) {
    System.out.println(***variable***[i]);
    }

Loop through Array with For-Each

- Already mentioned
- Comparison between for and for-each
  - For each is **easier to write**, not require counter and more readable

Multidimensional Arrays (Array containing one or more Arrays, to create: add array in its own [])

- ***variableType***[][] ***variableName*** = { {***variable***}, {***variable***} };

Exceptions

- **try** (block of code to be tested for errors while executed)
- **catch** (block of code to be executed, if error occurs in the try block)
  - try {
    statement
    }
    catch (***Exception e***) {
    statement
    }

- **finally** (execute code after try... catch, regardless of result)
    - try {
      statement
      }
      catch (***Exception e***) {
      statement
      }
      finally {
      statement
      }
- **throw** (allow to create custom error)
    - Used together with an **exception type:**
        - ArithmeticException
        - FileNotFoundException
        - ArrayIndexOutOfBoundsException
        - SecurityException
        - …
    - throw **new** ExceptionType("...")

Java Object-Oriented Programming

- OOP = faster and easier to execute
- OOP → clear structure for programs
- OOP → keep Java code DRY (Don't Repeat Yourself) and easier to maintain, modify and debug
- OOP → full reusable application with less code and less development time

Class vs Objects

- Class = template for object
- Object = instance of a class

To create an object:

- ***className objectName*** **=** **new** ***className*****();**
    - The created object belongs to that following ***className***()
- Can create multiple objects in class

Multiple classes

- Used for better organization (but in the same directory → relate it)

    - 1 class hold all the **attributes** and **methods**
    - Other class holds **main**() method

Multiple Objects

- With multiple objects, you can change value of 1 object without affecting the other one

Class Attributes (or fields)

- Can specify as many as possible
- Attributes are variables inside a Class (eg. x = 5)
    - Access
        - ***objectName.variable***
    - Modify (also override)
        - ***objectName.variable*** **=** new assigned value
        - If don't want to override the existing value
            - **final** ***variableName***
                - final is useful when you want to always store the same value
                - Final is also called **modifier**

Class Method

Static vs Non-Static

- Static = can be accessed without creating an object of the class
    - Can use ***methodName*****();**
- Public = can be accessed only by the objects
    - Can only use ***objName.methodName*****();**

Access Methods with an Object

- Create object that addresses a class
- Can access other class if in a directory

Constructors (or Methods)

- Special method used to initialize objects
- Called when an object of a class is created

Parameters

- Constructors can also take parameters → initialize attributes
- Can add as many parameters as you want

Modifiers

- public = **access modifier** → used to set access level for classes, attributes, methods and constructors
- 2 groups
    - **Access Modifier** - Control access level
    - **Non-Access Modifiers** - do not control level, but provide other functionality

Access Modifiers

For **classes** can use either **public** or **default**.

- **public**: class accessible by any other class
- **default**: class only accessible by classes in the same package. Used when you don't specify a modifier.

For **attributes**, **methods**, and **constructors** can use:

- **public**: code accessible for all classes
- **private**: code only accessible within the declared class
- **default**: code only accessible in the same package. Used when you don't specify a modifier.
- **protected**: code accessible in the same package and **subclasses**.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Non-Access Modifiers

For **classes**, can use either **final** or **abstract**:

- **final**: class cannot be inherited by other classes
- **abstract**: cannot be used to create objects. To access, it must be inherited from another class.

For **attributes** and **methods** can use:

- **final**: attributes and methods cannot be modified
- **static**: attributes and methods belongs to the class (not an object)
- **abstract**: can be used only in an abstract class, and can be used only on methods
    - Method doesn't have a body (body provided by subclass)
        - **extends** is used to create a Subclass (inherit abstract class)
            -
- **transient**: attributes and methods are skipped when serializing the object containing them
- **synchronized**: methods can be accessed only one thread at a time
- **volatile**: value of an attribute is not cached thread-locally, always read from "main memory"

Encapsulation

- Make sure that "sensitive data is hidden from the user.
    - Declare class **private**
    - Provide public **get** and **set** methods
        - **get** returns the variable value
            - myObj.**get*className*( )**;
        - **set** sets the value
            - myObj.**set*className*(" ")**;
        - **this** = used to refer to current object

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made **read-only (**if only use **get** method), or **write-only** (if only use **set** method)
- Flexible: can change part of program without affecting other parts
- Increase security of data.

Packages

Group of related classes, **folder in a file directory**.

- 2 categories
    - Built-in Packages (from Java API)
        - Java API is a library of prewritten classes (Free to use, including JDE).
            - Divided into (to use a class, use **import**):
                - **Packages**
                - import ***package.name.****;
                - **Classes**
                - import ***package.name.*****Class**;
    - User-defined Packages (own packages)
        - To create, use **package**
            - package ***packageName***; (***packageName*** in lower in order to avoid conflict with other classes)
        - Compile package
            - javac **-d .** ***className***.java;
                - -d = specify destination
                - Use any directory name
                - . = keep package within same directory
        - Run package
            - java ***packageName.className***;

Inheritance (inherit attributes and methods from another class)

Use **extends** to inherit from a class

Subclass

- Class that inherits from another class

Superclass

- Class being inherited from

Use **final** to avoid other classes to inherit from a class

Polymorphism

Many classes that are related to each other by inheritance

- Example:

```
class Animal {
```

```
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal
object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }

}
```

Why and When use Inheritance and Polymorphism:

- For reusability: reuse attributes and methods of an existing class when creating new class.

Inner Classes

- It is possible to nest classes → group classes that belong together
    - Code more readable & maintainable

Access Inner class from Outer Class

- To access inner class, create an object of the outer class + create an object of inner class.

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}
```

```
public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Private Inner Class

- Inner class can be **private** or **protected** (unlike regular class). If don't want outside objects to access the inner class, declare class as **private**.
    - When the object of outside class accesses the inner class, error will occur.

Static Inner Class

- Inner class can be **static**, access it without creating an object of the outer class.
    - OuterClass.InnerClass myInner = new OuterClass.InnerClass();

Access Outer Class From Inner Class

- Inner classes can access attributes and methods of outer classes.

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class MyMainClass {
  public static void main(String args[]) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}
```

Abstraction

Hide certain details and show only essential information to user

Abstract Class

- Restricted class that cannot be used to create objects (must be inherited from another class)

Interface

- A completely "abstract class" used to relate methods with empty bodies

    - Do not have a body
- Can be accessed by using ***implements*** (instead of ***extends***)
- Methods are by default **abstract** and **public**
- Attributes are by default **public**, **static** and **final**
- To implement multiple interfaces, separate them with ","

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a
body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Why Interface?

- achieve security - hide certain details & show only the important details of an object
- Cannot have "multiple inheritance"
    - But can have multiple **Interfaces**

Abstract Method

- Can only be used in an abstract class, and it doesn't have a body. Body is provided by the subclass (inherited from).

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
```

```
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Enums

User Input

ArrayList

LinkedList

HashMap

HashSet

Iterator

Wrapper

RegEx

Threads

Java File Handling

Files

Create/Write

Read

Delete