Classes and Objects (Part 2)

Java Constructors

- A constructor in Java is a special method that is used to initialize objects.
- The constructor is called when an object of a class is created.
- It can be used to set initial values for object attributes:
- Note that the constructor name must match the class name, and it cannot have a return type (like void).
- Also note that the constructor is called when the object is created.
- All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

```
1  // Create a Main class
2  public class Main {
3    int x;
4    // Create a class constructor for the Main class
5    public Main() {
6      x = 5;
7    }
8    public static void main(String[] args) {
9      Main myObj = new Main();
10     System.out.println(myObj.x);
11   }
12 }
```

Constructor Parameters

- Constructors can also take parameters, which is used to initialize attributes.
- The following example adds an parameter to the constructor.

```
1  //filename: Main.java
2  public class Main {
3    int modelYear;
4    String modelName;
5    public Main(int year, String name) {
6      modelYear = year;
7      modelName = name;
8    }
9    public static void main(String[] args) {
10     Main myCar = new Main(1969, "Mustang");
11     System.out.println(myCar.modelYear + " " +
   myCar.modelName);
12   }
13 }
```

Access Modifiers/ Access Control

- The public keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.
- We divide modifiers into two groups:
  - Access Modifiers - controls the access level
  - Non-Access Modifiers - do not control access level, but provides other functionality
  - **Example of Access Modifiers**
- For Class: public
- For attributes, methods and constructors:
  - public
  - protected
  - private

Access Modifiers/ Access Control

- The public keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.
- We divide modifiers into two groups:
  - Access Modifiers - controls the access level
  - Non-Access Modifiers - do not control access level, but provides other functionality

Example of Non-Access Modifiers

- final : class cannot be inherited
- abstract: class cannot be used to create object

Setters and Getters

- Encapsulation
  - The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:
    - declare class variables/attributes as private
    - provide public get and set methods to access and update the value of a private variable

Setters and Getters

- **Getter and Setter**
- W know that private variables can only be accessed within the same class
- However, it is possible to access them if we provide public get and set methods.
- The get method returns the variable value, and the set method sets the value.
- **Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:**

```
1  public class Person {
2    private String name; // private = restricted access
3    // Getter
4    public String getName() {
5      return name;
6    }
7    // Setter
8    public void setName(String newName) {
9      this.name = newName;
10   }
11   public static void main(String[] args) {
12     Person myObj = new Person();
13     myObj.name = "John";
14     System.out.println(myObj.name);
15 }
16 }
```

Method Overloading

- With method overloading, multiple methods can have the same name with different parameters:

Method Overloading

```
public class Main {
  static int plusMethodInt(int x, int y) {
    return x + y;
  }
  static double plusMethodDouble(double x, double y) {
    return x + y;
  }
  public static void main(String[] args) {
    int myNum1 = plusMethodInt(8, 5);
    double myNum2 = plusMethodDouble(4.3, 6.26);
    System.out.println("int: " + myNum1);
    System.out.println("double: " + myNum2);
  }
}
```

```
public class Main {
  static int plusMethod(int x, int y) {
    return x + y;
  }
  static double plusMethod(double x, double y) {
    return x + y;
  }
  public static void main(String[] args) {
    int myNum1 = plusMethod(8, 5);
    double myNum2 = plusMethod(4.3, 6.26);
    System.out.println("int: " + myNum1);
    System.out.println("double: " + myNum2);
  }
}
```

Call by value, Call by reference

- *Definition of Call by Reference:*
  - *In call by value, a copy of the actual value of the argument is passed to the method.*

```
1  public class CallByValueExample {
2      public static void main(String[] args) {
3          int x = 10; // Primitive data type
4          System.out.println("Before calling method: " + x);
5          modifyValue(x);
6          System.out.println("After calling method: " + x);
7      }
8      static void modifyValue(int a) {
9          System.out.println("Inside method (before modification): " + a);
10         a = 20;
11         System.out.println("Inside method (after modification): " + a);
12     }
13 }
14
```

Call by value, Call by reference

- *Definition of Call by Reference:*
  - *In Java, it's important to note that, there is no direct support for "call by reference" as it is defined in some other programming languages like C++.*
  - *However, when dealing with objects, you effectively pass the reference to the object by value.*

```
class MyObject {
    int value;
    MyObject(int value) {
        this.value = value;
    }
}
public class CallByReferenceExample {
    public static void main(String[] args) {
        MyObject obj = new MyObject(10);
        System.out.println("Before calling method: " + obj.value);
        modifyObject(obj);
        System.out.println("After calling method: " + obj.value);
    }
    static void modifyObject(MyObject myObject) {
        System.out.println("Inside method (before modification): " + myObject.value);
        myObject.value = 20;
        System.out.println("Inside method (after modification): " + myObject.value);
    }
}
```

this keyword

- Definition:
  - The this keyword in Java is a reference variable that refers to the current object.
- Usage:
  - It is often used to differentiate instance variables from local variables when they have the same name.
  - It is used to invoke the current object's method or constructor.

```
1  public class Person {
2      String name;
3      public Person(String name) {
4          this.name = name;
5      }
6      public void printDetails() {
7          System.out.println("Name: " + this.name);
8      }
9      public static void main(String[] args) {
10         Person person1 = new Person("Siri");
11         Person person2 = new Person("Google");
12         person1.printDetails();
13         person2.printDetails();
14     }
15 }
```

final Modifiers

- Definition:
  - The final modifier in Java is used to restrict the user.
  - It can be applied to variables, methods, and classes.
  - When applied to a variable, the final keyword indicates that the variable cannot be changed after initialization.
- Usage:
  - For variables, it makes them constants.
  - For methods, it prevents overriding in the subclasses.
  - For classes, it prevents inheritance.

```
1  public class Circle {
2      final double PI = 3.14;
3      final double radius;
4      public Circle(double radius) {
5          this.radius = radius;
6      }
7      public double calculateArea() {
8          return PI * radius * radius;
9      }
10     public static void main(String[] args) {
11         Circle circle = new Circle(5.0);
12         System.out.println("Radius: " + circle.radius);
13         System.out.println("Area: " + circle.calculateArea());
14     }
15 }
```

Nested class / inner classes

- In Java, it is also possible to nest classes (a class within a class).
- The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- To access the inner class, create an object of the outer class, and then create an object of the inner class:

```
class OuterClass {
  int x = 5;
  class InnerClass {
    int y = 10;
  }
}
public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Wrapper Classes in Java

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- The table aside shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Wrapper Classes in Java

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- The table aside shows the primitive type and the equivalent wrapper class:

```
public class WrapperClassExample {
    public static void main(String[] args) {
        // Using wrapper classes to convert primitive data types to objects
        Integer intObject = Integer.valueOf(42);
        Double doubleObject = Double.valueOf(3.14);
        Character charObject = Character.valueOf('A');
        Boolean booleanObject = Boolean.valueOf(true);
        // Using wrapper classes to convert objects to primitive data types
        int intValue = intObject.intValue();
        double doubleValue = doubleObject.doubleValue();
        char charValue = charObject.charValue();
        boolean booleanValue = booleanObject.booleanValue();
        // Printing values
        System.out.println("Integer Value: " + intValue);
        System.out.println("Double Value: " + doubleValue);
        System.out.println("Character Value: " + charValue);
        System.out.println("Boolean Value: " + booleanValue);
    }
}
```

Garbage Collection

- Garbage collection in Java is the process of automatically reclaiming memory occupied by objects that are no longer reachable or referenced by the program.
- Automatic Memory Management:
  - Java has an automatic garbage collector that runs in the background, identifying and reclaiming memory occupied by objects that are no longer needed.
  - The garbage collector helps simplify memory management for developers by handling memory deallocation automatically.

Garbage Collection

- Mark-and-Sweep Algorithm:
  - The garbage collector marks and sweeps through objects, identifying and removing those no longer reachable.
- Generational Approach:
  - Memory is divided into generations, and objects are promoted based on age, allowing for more efficient garbage collection.
- System.gc() and finalize():
  - System.gc() suggests garbage collection, but immediate execution is not guaranteed.
  - finalize() allows cleanup before object destruction, but relying on it is discouraged.
- Tuning and Monitoring:
  - Developers can tune garbage collection using JVM options.
  - Monitoring and analyzing garbage collection logs help optimize memory management for performance.

Garbage Collection: Example

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
 // Creating an object
  MyClass obj = new MyClass("My Object");
// Making the object eligible for garbage collection
        obj = null;
// Suggesting garbage collection (not always necessary)
        System.gc();
    }
}
```

```
class MyClass {
    String name;
    MyClass(String name) {
        this.name = name;
        System.out.println(name + " created.");
    }
//The finalize() method is called before an object is garbage collected
    @Override
    protected void finalize() throws Throwable {
    System.out.println(name + " is being garbage collected.");
    }
}
```