Date:26/11/24

Name:Venkatesh Kumar B

Primitive Types:

Primitive types are the most basic data types that hold simple values. They are not objects and are stored directly in memory. Java has 8 primitive data types

byte: 8-bit signed integer.

short: 16-bit signed integer.

int: 32-bit signed integer.

long: 64-bit signed integer.

float: 32-bit floating point

double: 64-bit floating point.

char: 16-bit Unicode character.

boolean: Represents **true** or **false.**

Code:

```
public class PrimitiveTypesExample {
                        public static void main(String[] args) {
                          byte b = 100;  // 8-bit integer
                          System.out.println("byte value: " + b);
                           short s = 15000;  // 16-bit integer
                          System.out.println("short value: " + s);
```

```
        int i = 100000;  // 32-bit integer
        System.out.println("int value: " + i);

        long l = 10000000000L;
        System.out.println("long value: " + l);


        float f = 5.75f;
        System.out.println("float value: " + f);


        double d = 19.99;  // 64-bit floating point
        System.out.println("double value: " + d);

        char c = 'A';  // 16-bit character
        System.out.println("char value: " + c);

        boolean isJavaFun = true;
        System.out.println("boolean value: " + isJavaFun);
    }
}
```

Wrapper Class:

wrapper classes are used to convert the **primitive data types** (such as int, char, double, etc.)

Into **objects**. Each primitive type has a corresponding wrapper class in the java.lang package.

These wrapper classes are used when you need to work with objects instead of primitive types, such as when using collections (like ArrayList), or for methods that require objects rather than primitives.

The 8 Wrapper Classes in Java:

byte –Byte

short – Short

int –Integer

long – Long

float – Float

double -Double

char –Character

boolean –Boolean

Code:

```
public class WrapperClassExample {
public static void main(String[] args) {
  int primitiveInt = 10;
  Integer wrapperInt = Integer.valueOf(primitiveInt);
  System.out.println("Wrapper Integer: " + wrapperInt);
  int unboxedInt = wrapperInt.intValue();
  System.out.println("Unboxed int: " + unboxedInt);
  double primitiveDouble = 20.5;
  Double wrapperDouble = Double.valueOf(primitiveDouble);
  System.out.println("Wrapper Double: " + wrapperDouble);
```

```
        double unboxedDouble = wrapperDouble.doubleValue();
    System.out.println("Unboxed double: " + unboxedDouble);
         Integer autoBoxedInteger = primitiveInt;
    System.out.println("Auto-boxed Integer: " + autoBoxedInteger);



    int autoUnboxedInt = autoBoxedInteger;
    System.out.println("Auto-unboxed int: " + autoUnboxedInt);
  }
}
```

Autoboxing & Unboxing :

Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class object.

Unboxing is the automatic conversion of a wrapper class object back to its corresponding primitive type.

Code:

```
  public class AutoBoxingUnboxingExample {
  public static void main(String[] args) {
        int primitiveInt = 42;
    Integer wrapperInt = primitiveInt;  // Autoboxing occurs here
    System.out.println("Autoboxed Integer: " + wrapperInt);
    Integer anotherWrapperInt = new Integer(100);
    int unboxedInt = anotherWrapperInt;
```

```
        System.out.println("Unboxed int: " + unboxedInt);
        Integer[] integerArray = new Integer[3];
        integerArray[0] = primitiveInt;
        System.out.println("Value in array (autoboxed): " + integerArray[0]);
        int result = wrapperInt + 10;  // Unboxing wrapperInt and adding to 10
        System.out.println("Unboxed and added: " + result);
    }
}
```

Autoboxing in Collections:

When you assign a primitive type to a collection that holds objects (like an Integer[] array), autoboxing automatically converts the primitive to its corresponding wrapper class.

Example: integerArray[0] = primitiveInt; — primitiveInt is automatically boxed into an Integer object.

Unboxing in Expressions:

When you perform arithmetic or other operations with wrapper class objects, Java automatically unboxes the object to use the primitive value in the operation.

Example: int result = wrapperInt + 10; — wrapperInt is automatically unboxed to an int, and the result is calculated

Aggregation:

Aggregation is a concept in object-oriented programming (OOP) that represents a "has-a" relationship between two objects

It is a type of association where one object (the "whole") contains references to other objects (the "parts").

unlike composition, in aggregation, the contained objects can exist independently of the containing object.

Aggregation is a loose relationship where the "whole" object can exist without the "part" objects.

It is often described as a "has-a" relationship.

The "part" objects can exist independently of the "whole" object.

Code:

```
class Teacher {
   private String name;

   public Teacher(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}

class Department {
   private String name;
   private Teacher[] teachers;
   public Department(String name, Teacher[] teachers) {
      this.name = name;
      this.teachers = teachers;
```

```
    }

    public void printDepartmentDetails() {
        System.out.println("Department: " + name);
        System.out.println("Teachers in this department:");
        for (Teacher teacher : teachers) {
            System.out.println(teacher.getName());
        }
    }
}

public class AggregationExample {
    public static void main(String[] args) {
        Teacher teacher1 = new Teacher("John");
        Teacher teacher2 = new Teacher("Alice");
        Teacher teacher3 = new Teacher("Bob");

        Teacher[] teachers = {teacher1, teacher2, teacher3
}
        Department department = new Department("Computer Science", teachers);
        department.printDepartmentDetails();
    }
}
```

Composition:

Composition is a strong relationship where the contained objects cannot exist without the parent object.

If the parent object is destroyed, all its contained objects are also destroyed.

Example: A Car has Engine and Wheels, but the engine and wheels cannot exist without the car.

Code:

```
class Engine {
    public Engine() {
        System.out.println("Engine created.");
    }

    public void start() {
        System.out.println("Engine started.");
    }
}

class Wheel {
    public Wheel() {
        System.out.println("Wheel created.");
    }

    public void rotate() {
        System.out.println("Wheel is rotating.");
    }
}
```

```
class Car {
    private Engine engine; // Engine is a part of Car
    private Wheel[] wheels; // Car has Wheels

    public Car() {
        engine = new Engine(); // Creating an engine object inside the Car class
        wheels = new Wheel[4]; // Creating an array to hold 4 wheels
        for (int i = 0; i < wheels.length; i++) {
            wheels[i] = new Wheel(); // Each wheel is created for the car
        }
    }
    public void startCar() {
        engine.start();
        for (Wheel wheel : wheels) {
            wheel.rotate();
        }
    }
}

public class CompositionExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();
    }
}
```

Multithreading:

Multithreading in Java is a technique where multiple threads execute independently but share resources such as memory.

Each thread represents a separate path of execution, allowing the program to perform multiple tasks concurrently, which can improve performance on multi-core systems.

By extending the Thread class.

By implementing the Runnable interface.

Code:

```
class Thread1 extends Thread {
   @Override
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Thread 1: " + i);
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            System.out.println(e);
         }
      }
   }
}

class Thread2 extends Thread {
```

```
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread 2: " + i);
            try {
                Thread.sleep(700);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
 public class MultithreadingExample {
    public static void main(String[] args) {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();
             thread1.start();
        thread2.start();
    }
}
```

LifeCycle in Threading

The life cycle of a thread refers to the various stages that a thread goes through during its existence, from its creation to its termination.

The thread life cycle is managed by the Java Virtual Machine (JVM) and the thread scheduler.

Thread Life Cycle Stages:

New (Born):

When a thread is created using the Thread class or by implementing Runnable, it is in the **New** state. At this point, the thread has not yet started execution.

Runnable:

After the thread's start() method is invoked, it enters the **Runnable** state. This does not mean the thread is currently running; it is now eligible to run, and the thread scheduler decides when to actually run it

Blocked:

A thread enters the **Blocked** state when it is waiting for a resource that is currently unavailable, like I/O operations or waiting for another thread to release a lock (synchronization).

Waiting:

A thread enters the **Waiting** state when it is waiting for another thread to perform a specific action, like calling Thread.join() or when Thread.sleep() is used.

Timed Waiting:

A thread enters the **Timed Waiting** state when it is waiting for a specific period of time. This can happen with methods like Thread.sleep(long millis) or Object.wait(long millis).

Terminated:

A thread enters the **Terminated** state when it has completed its execution, or if it has been terminated due to an exception or error.

Code:

```
class MyThread extends Thread {

  @Override
```

```
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is in state: " +
    Thread.currentThread().getState());
            Thread.sleep(1000); // Timed Waiting
            System.out.println(Thread.currentThread().getName() + " is in state: " +
    Thread.currentThread().getState());
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + " is printing: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(Thread.currentThread().getName() + " is in state: " +
    Thread.currentThread().getState());
    }
}

public class ThreadLifeCycleExample {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + " is in state: " +
    Thread.currentThread().getState());
MyThread thread = new MyThread();
        System.out.println(thread.getName() + " is in state: " + thread.getState()); //
    NEW state
```

```
        thread.start();


        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " (Main thread) is
    printing: " + i);
            try {
                Thread.sleep(700);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

File Handling Concept:

Create/Write:

Create a new file or overwrite an existing file and write data to it.

Read:

Read data from a file.

Delete/Manipulate:

Delete or manipulate files and directories (using the File class).

Code:

```
import java.io.File;
import java.io.FileWriter;
```

```
import java.io.BufferedWriter;
import java.io.IOException;

public class FileHandlingExample {
   public static void main(String[] args) {
      String filename = "example.txt";  // Specify the file name

      try (FileWriter fileWriter = new FileWriter(filename);
          BufferedWriter bufferedWriter = new BufferedWriter(fileWriter)) {
          bufferedWriter.write("Hello, this is a test.");
         bufferedWriter.newLine();  // Adds a new line
         bufferedWriter.write("This is the second line.");
           System.out.println("Data has been written to the file: " + filename);

      } catch (IOException e) {
         System.out.println("An error occurred while writing to the file.");
         e.printStackTrace();
      }
   }
}
```