**Primitive Data Types:** Primitive Data Types. The eight primitives defined in Java are i**nt, byte, short, long, float, double, boolean and char**. These aren't considered objects and represent raw values. They're stored directly on the stack (check out this article for more information about memory management in Java). A primitive data type is pre-defined by the programming language. The size and type of variable values are specified, and it has no additional methods.

**Is array primitive data type?** => **No, arrays are not primitive datatypes in Java**. They are container objects which are created dynamically. All methods of class Object may be invoked on an array

**Is string primitive in Java?** => String is an object, in android or java **it isn't a primitive type at all**. you can use strings to store in SharedPreferences.

**Non-Primitive Data Types:** Non-Primitive data types refer to objects and hence they are called reference types. Examples of non-primitive types include Strings, Arrays, Classes, Interface, etc. These data types are not actually defined by the programming language but are created by the programmer. They are also called “reference variables” or “object references” since they reference a memory location which stores the data.

**Immutable:** An [immutable class](https://howtodoinjava.com/java/basics/how-to-make-a-java-class-immutable/) is one whose state can not be changed once created. Here, the state of the object essentially means the values stored in the instance variable in class whether they are primitive types or reference types.

**Pass by Value/Reference**

* **Pass by Value:**It is a process in which the function parameter values are copied to another variable and instead this object copied is passed. This is known as call by Value.
* **Pass by Reference:** It is a process in which the actual copy of reference is passed to the function. This is called by Reference.

**Java support only Pass by Value and not pass by reference**

public class Tester {

   public static void main(String[] args) {

      Point point = new Point();

      System.out.println("X: " +point.x + ", Y: " + point.y); //x = 0, Y=0

      updatePoint(point);

      System.out.println("X: " +point.x + ", Y: " + point.y); //x = 100, Y=100

    }

    public static void updatePoint(Point point) {

      point.x = 100;

      point.y = 100;

    }

}

class Point {

   public int x, y;

}

## Difference between String Objects and Literals?

When we create String with new() it’s created in heap and also added into [string pool](https://howtodoinjava.com/java/string/string-constant-pool/), while String created using literal are created in String pool area only.

String str1 = **new** String("test"); *//String Object*

String str2 = "test"; *//String Literal*

## When do we Override *hashCode()* and *equals()* Methods?

**equals()** : This method checks if some other object passed to it as an argument is equal the object in which this method is invoked.

**hashCode()**: This method returns a hashCode() value as an Integer and is supported for the benefit of hashing based java.util.Collection classes

Note:

* If two objects are equal as per the equals() method, then calling the hashCode() method in each of the two objects must return the same integer result. So, If a field is not used in equals(), then it must not be used in hashCode() method.
* If two objects are unequal as per the equals() method, each of the two objects can return either two different integer results or same integer results (i.e. if 2 objects have the same hashCode() result does not mean that they are equal, but if two objects are equal then they must return the same hashCode() result).
* equals() and hashCode() are different methods and hashCode method should not be used to check if two object references are same. **Reason:** hashCode just returns int value for an Object, even two different objects can have same hashCode integer. The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal. **equals()** checks if the two object references are same. If two objects are equal then their hashCode must be the same, but the reverse is not true.

## Refrance equality operator ('==') or structural equality operator ('===') Equals Operator ( == ): The comparison x == y with equals operator, where x and y are operands, can produce boolean result which is either true or false.

Note: For example, if we try to compare a string value to a number value, then string value will be converted to first into the number type, and then the comparison will happen.

**Strict Equals Operator ( === ):** The strict comparison x === y with equals operator, where x and y are values, produces true or false only when –

1. x and y are of the same type
2. x and y are have the same value

**0**==**false** // true, because false is equivalent of 0

**0**===**false** // false, because both operands are of different type

**2**=="2" // true, auto type coercion, string converted into number

**2**==="2" // false, since both operands are not of same type

**Synchronization:**

1. When a method is declared as synchronized; the thread holds the monitor for that method’s object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.
2. Synchronization in Java is achieved using synchronized keyword. we can use *synchronized* keyword in our class or methods or blocks. Keywords can not be used with variables or attributes in the class definition.