**BCA-DS-212: PROGRAMMING IN JAVA *BCA IV Semester***

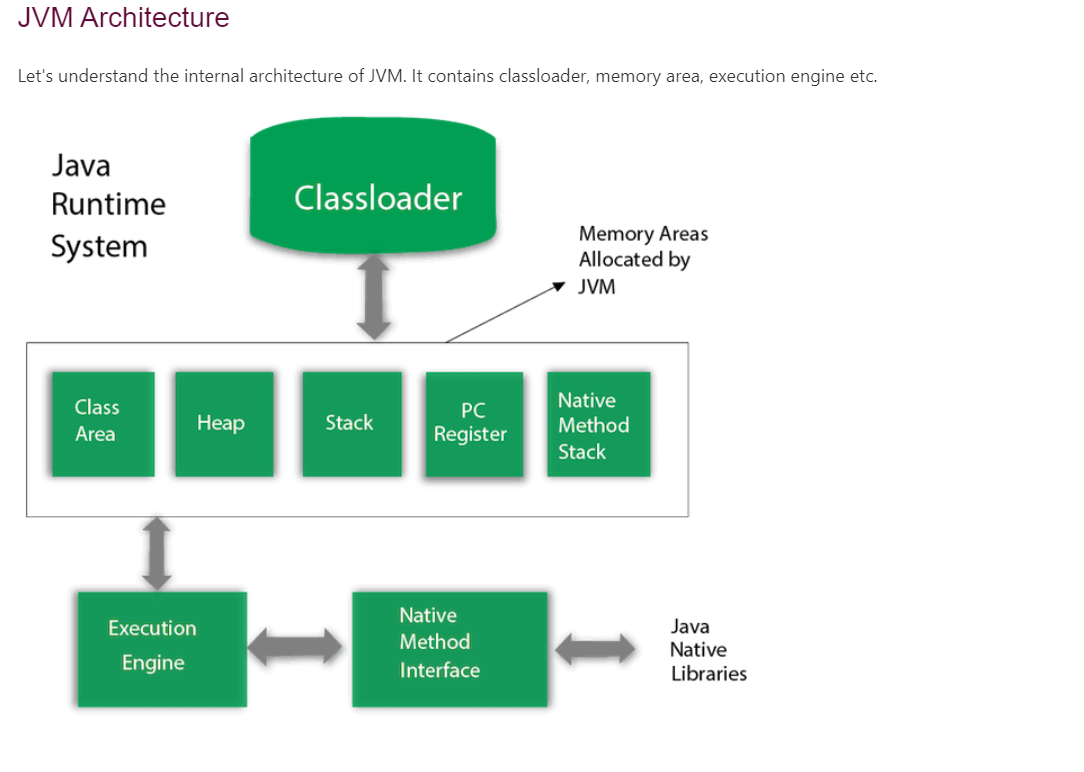
**UNIT 2**

**JAVA VIRTUAL MACHINE CONCEPTS**

**JVM**

**A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

1. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
2. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.



### **1) Classloader**

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside $JAVA\_HOME/jre/lib/ext directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

### **2) Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### **3) Heap**

It is the runtime data area in which objects are allocated.

### **4) Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

### **5) Program Counter Register**

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### **6) Native Method Stack**

It contains all the native methods used in the application.

### **7) Execution Engine**

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

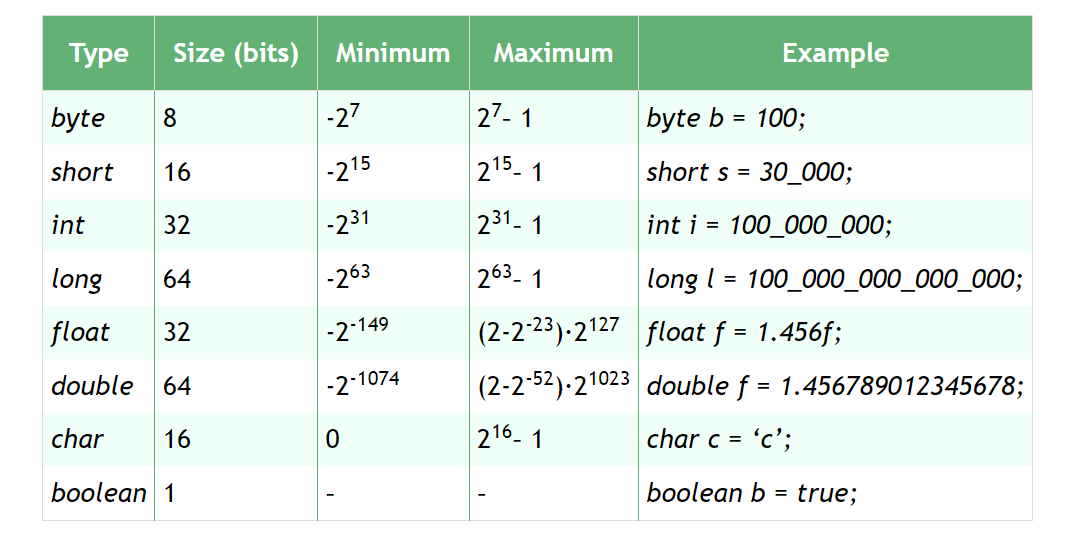
Primitive Type in Java

The Java Programming Language features eight primitive data types.

2. Primitive Data Types

The eight primitives defined in Java are int, byte, short, long, float, double, boolean and char. These aren’t considered objects and represent raw values.

They’re stored directly on the stack



2.1. int

The first primitive data type we’re going to cover is int. Also known as an integer, int type holds a wide range of non-fractional number values.

Specifically, Java stores it using 32 bits of memory. In other words, it can represent values from -2,147,483,648 (-231) to 2,147,483,647 (231-1).

In Java 8, it’s possible to store an unsigned integer value up to 4,294,967,295 (232-1) by using new special helper functions.

We can simply declare an int:

int x = 424\_242;

int y;

The default value of an int declared without an assignment is 0.

If the variable is defined in a method, we must assign a value before we can use it.

We can perform all standard arithmetic operations on ints. Just be aware that decimal values will be chopped off when performing these on integers.

2.2. byte

byte is a primitive data type similar to int, except it only takes up 8 bits of memory. This is why we call it a byte. Because the memory size is so small, byte can only hold the values from -128 (-27) to 127 (27 – 1).

Here’s how we can create byte:

byte b = 100;

byte empty;

The default value of byte is also 0.

2.3. short

The next stop on our list of primitive data types in Java is short.

If we want to save memory and byte is too small, we can use the type halfway between byte and int: short.

At 16 bits of memory, it’s half the size of int and twice the size of byte. Its range of possible values is -32,768(-215) to 32,767(215 – 1).

short is declared like this:

short s = 20\_020;

short s;

Also similar to the other types, the default value is 0. We can use all standard arithmetic on it as well.

2.4. long

Our last primitive data type related to integers is long.

long is the big brother of int. It’s stored in 64 bits of memory, so it can hold a significantly larger set of possible values.

The possible values of a long are between -9,223,372,036,854,775,808 (-263) to 9,223,372,036,854,775,807 (263 – 1).

We can simply declare one:

long l = 1\_234\_567\_890;

long l;

As with other integer types, the default is also 0. We can use all arithmetic on long that works on int.

2.5. float

We represent basic fractional numbers in Java using the float type. This is a single-precision decimal number. This means that if we get past six decimal points, the number becomes less precise and more of an estimate.

In most cases, we don’t care about the precision loss. But if our calculation requires absolute precision (e.g., financial operations, landing on the moon, etc.), we need to use specific types designed for this work. For more information, check out the Java class Big Decimal.

This type is stored in 32 bits of memory just like int. However, because of the floating decimal point, its range is much different. It can represent both positive and negative numbers. The smallest decimal is 1.40239846 x 10-45, and the largest value is 3.40282347 x 1038.

We declare floats the same as any other type:

float f = 3.145f;

float f;

And the default value is 0.0 instead of 0. Also, notice we add the f designation to the end of the literal number to define a float. Otherwise, Java will throw an error because the default type of a decimal value is double.

We can also perform all standard arithmetic operations on floats. However, it’s important to note that we perform floating point arithmetic very differently than integer arithmetic.

2.6. double

Next, we look at double. Its name comes from the fact that it’s a double-precision decimal number.

It’s stored in 64 bits of memory. This means it represents a much larger range of possible numbers than float.

Although, it does suffer from the same precision limitation as float does. The range is 4.9406564584124654 x 10-324 to 1.7976931348623157 x 10308. That range can also be positive or negative.

Declaring double is the same as other numeric types:

double d = 3.13457599923384753929348D;

double d;

The default value is also 0.0 as it is with float. Similar to float, we attach the letter D to designate the literal as a double.

2.7. boolean

The simplest primitive data type is boolean. It can contain only two values: true or false. It stores its value in a single bit.

However, for convenience, Java pads the value and stores it in a single byte.

freestar

Here’s how we declare boolean:

boolean b = true;

boolean b;

Declaring it without a value defaults to false. boolean is the cornerstone of controlling our programs flow. We can use boolean operators on them (e.g., and, or, etc.).

2.8. char

The final primitive data type to look at is char.

Also called a character, char is a 16-bit integer representing a Unicode-encoded character. Its range is from 0 to 65,535. In Unicode, this represents ‘\u0000’ to ‘\uffff’.

For a list of all possible Unicode values, check out sites such as Unicode Table.

Let’s now declare a char:

char c = 'a';

char c = 65;

char c;

When defining our variables, we can use any character literal, and they will get automatically transformed into their Unicode encoding for us. A character’s default value is ‘/u0000’.

2.9. Overflow

The primitive data types have size limits. But what happens if we try to store a value that’s larger than the maximum value?

We run into a situation called overflow.

When an integer overflows, it rolls over to the minimum value and begins counting up from there.

Floating point numbers overflow by returning Infinity:

int i = Integer.MAX\_VALUE;

int j = i + 1;

// j will roll over to -2\_147\_483\_648

double d = Double.MAX\_VALUE;

double o = d + 1;

// o will be Infinity

Underflow is the same issue except it involves storing a value smaller than the minimum value. When the numbers underflow, they return 0.0.

2.10. Autoboxing

Each primitive data type also has a full Java class implementation that can wrap it. For instance, the Integer class can wrap an int. There is sometimes a need to convert from the primitive type to its object wrapper (e.g., using them with generics).

Luckily, Java can perform this conversion for us automatically, a process called Autoboxing:

Character c = 'c';

Integer i = 1;



A diagram of mathematical equations

Description automatically generated

A diagram of clouds with text

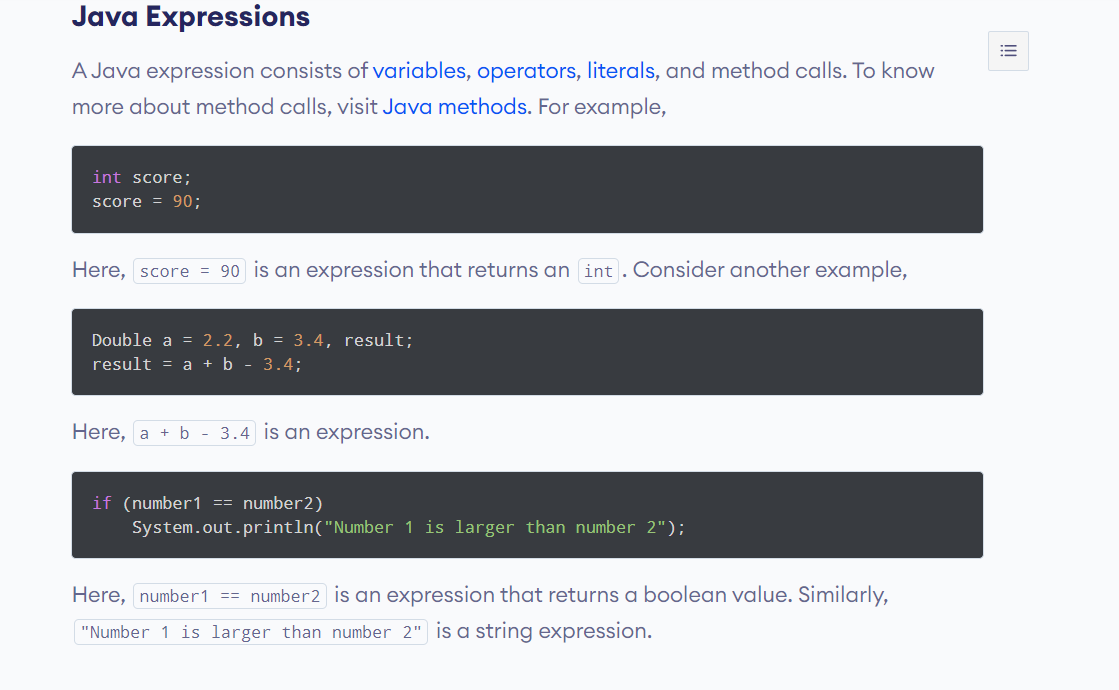
Description automatically generated

A diagram of clouds with text

Description automatically generated

A screenshot of a computer program

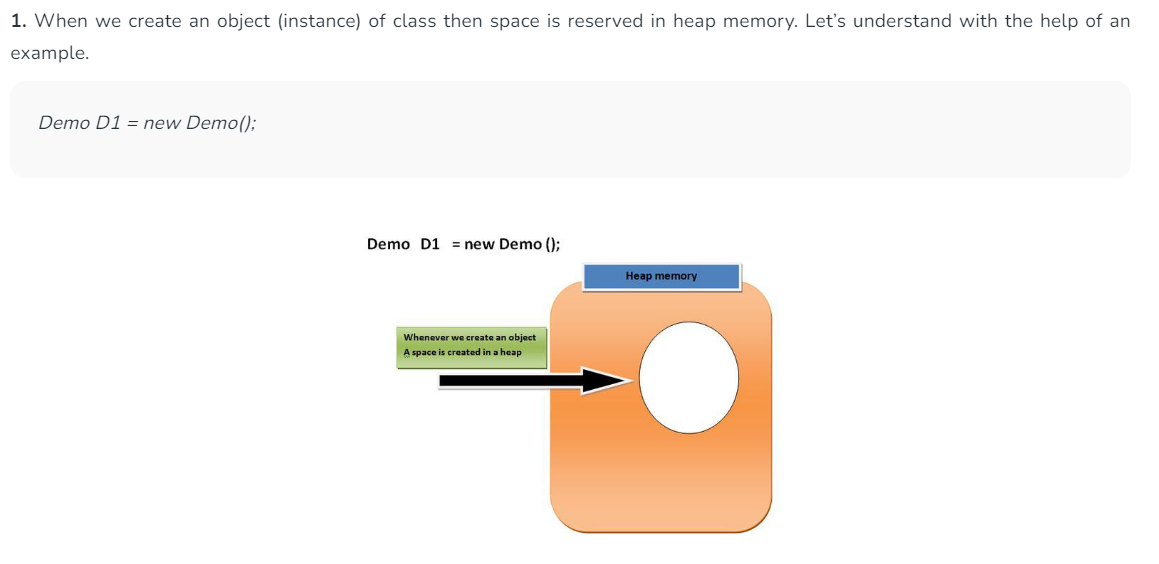
Description automatically generated



A screenshot of a computer

Description automatically generated

**Reference Variables in Java**

****

**A screenshot of a computer

Description automatically generated**

**Let us see what is actually happening step by step.**

**1.** When we create an object of demo class **new DEMO();**, the default constructor is called and returns a reference of the object, and simply this reference will be stored to the reference variable **D1** (As we know that associativity is Right-hand side to left-hand side).

**2.** The value of a reference variable is a reference. When we attempt to print the value of a reference variable, the output contains the name of the class which has been instantiated concatenated by @ and the hash code created for it by Java: the string **Demo@214c265e** tells us that the given variable is of type Name and its hexadecimal format of hash code is 214c265e.

**3.**At this point we will access the methods **display()** of the class demo using our custom reference variable that we created.

## What is Static Method in Java?

The static methods are the class methods and they don't belong to instances of the class. These are designed to be shared among all the instances of the same class and they are accessed by the class name of the particular class. A static keyword is used for declaring the static method in java

The static method in java cannot be overridden because static methods are linked to the class during the compilation time and we know very well that method overriding is based on the runtime polymorphism that's why the static methods cannot be overridden.

The static methods can be accessed by the outside environment very easily. We have already dicussed the main() method example in the above section.

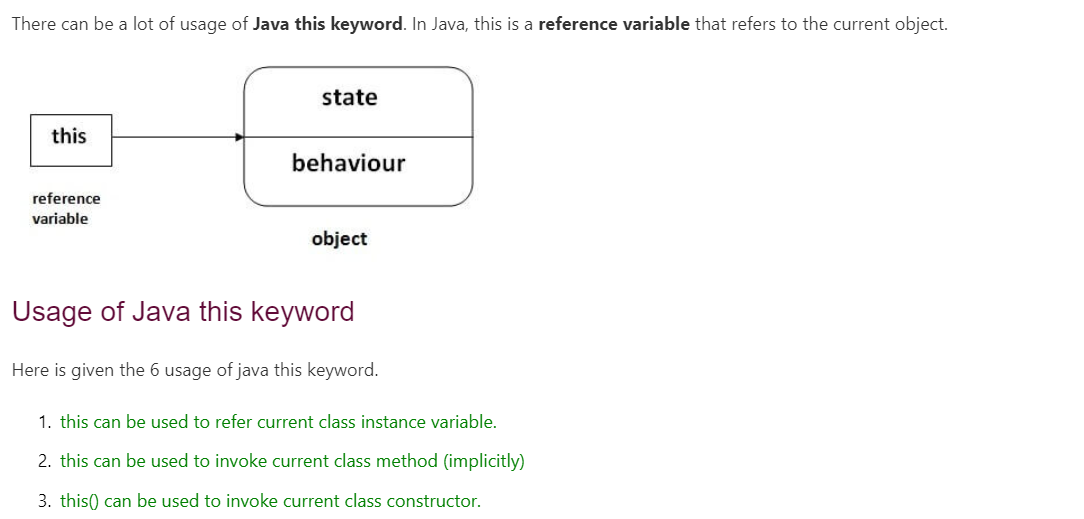
## Features of Static Method in Java

* Rather than being an instance of a class, a static method in Java is a method that is a part of the class itself.
* The method is available to each instance of the class.
* The method is available to each instance of the class. Without requiring the class's object, static methods have access to class variables (static variables) (instance).
* A static method can only access static data. It cannot access data that is dynamic (instance variables).
* Static methods can be accessed directly in non-static methods as well as static methods.

## Use Static Method and Common Use for Static Methods

This question always arises in the mind of the programmer when to use a static method in the java programs. We can use the static method only in our class, when we know the proper use of the static method. Let's understand some common use for static methods so that we can efficiently use them.

* When the implementation of the particular method is *not dependent* on the instance variables and instance methods, In this case we can make that method to be static.
* If the implementation of the code of the method is not changed by the instances of the class. In this case, we can make the method a static method.
* If we want to define some utility functions such as user defined sorting method in the class, we can define them using static concepts. Because the utility function can easily be shared among all objects of the class due to the property of the static method.



### **1. Call-by-Value:**

In Call-by-value the copy of the value of the actual parameter is passed to the formal parameter of the method. Any of the modifications made to the formal parameter within the method do not affect the actual parameter.

**ALGORITHM:**

**Step 1:** Create a class named CallByValueExample.

**Step 2:** Inside the main method:

**Step 2.1:** Declare an integer variable num and assign it the value 10.

**Step 2.2:** Print the value of num before calling the method.

**Step 2.3:** Call the modifyValue method, passing num as the actual parameter.

**Step 2.4:** Print the value of num after calling the method.

**Step 3:** Define the modifyValue method that takes an integer parameter value:

**Step 3.1:** Modify the formal parameter value by assigning it the value 20.

**Step 3.2:** Print the value of value inside the method.

## Call-by-Reference:

call by reference" is a method of passing arguments to functions or methods where the memory address (or reference) of the variable is passed rather than the value itself. This means that changes made to the formal parameter within the function affect the actual parameter in the calling environment.

In "call by reference," when a reference to a variable is passed, any modifications made to the parameter inside the function are transmitted back to the caller. This is because the formal parameter receives a reference (or pointer) to the actual data.

### **ALGORITHM:**

**Step 1:** Start

**Step 2:** Define the class "CallByReference"

**Step 2.1:** Declare instance variables: a (int) and b (int)

**Step 2.1:** Define a constructor to assign values to a and b

**Step 3:** Define the method "changeValue" inside the "CallByReference" class:

**Step 3.1:** Accept a parameter of type "CallByReference" called "obj"

**Step 3.2:** Add 10 to the value of "obj.a"

**Step 3.3:** Add 20 to the value of "obj.b"

**Step 4:** Define the class "Main"

**Step 4.1:** Define the main method

**Step 4.2:** Create an instance of "CallByReference" called "object" with values 10 and 20

**Step 4.3:** Print the values of "object.a" and "object.b"

**Step 4.4:** Call the "changeValue" method on "object" and pass "object" as an argument

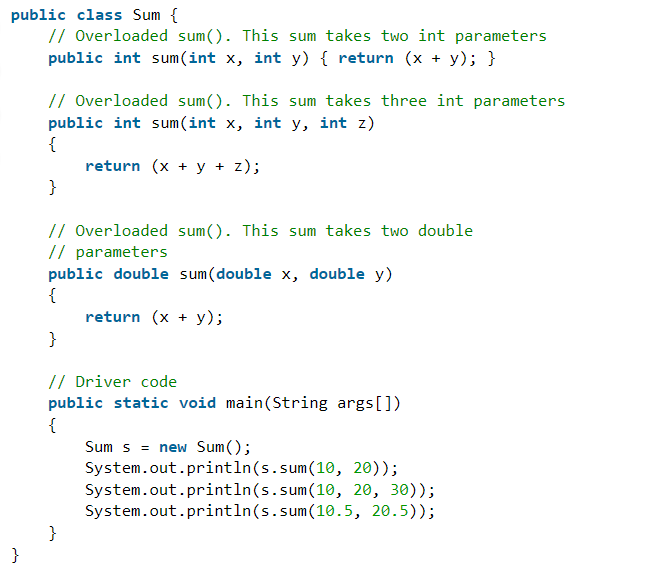
**Step 4.5:** Print the updated values of "object.a" and "object.b"

**Step 5:** End

METHOD OVERLOADING IN JAVA

In Java, Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.

Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding. In Method overloading compared to the parent argument, the child argument will get the highest priority.



## Different Ways of Method Overloading in Java

* Changing the Number of Parameters.
* Changing Data Types of the Arguments.
* Changing the Order of the Parameters of Methods

## Advantages of Method Overloading

* Method overloading improves the Readability and reusability of the program.
* Method overloading reduces the complexity of the program.
* Using method overloading, programmers can perform a task efficiently and effectively.
* Using method overloading, it is possible to access methods performing related functions with slightly different arguments and types.
* Objects of a class can also be initialized in different ways using the constructors.

# **Java Garbage Collection**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

**Advantage of Garbage Collection**

It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.

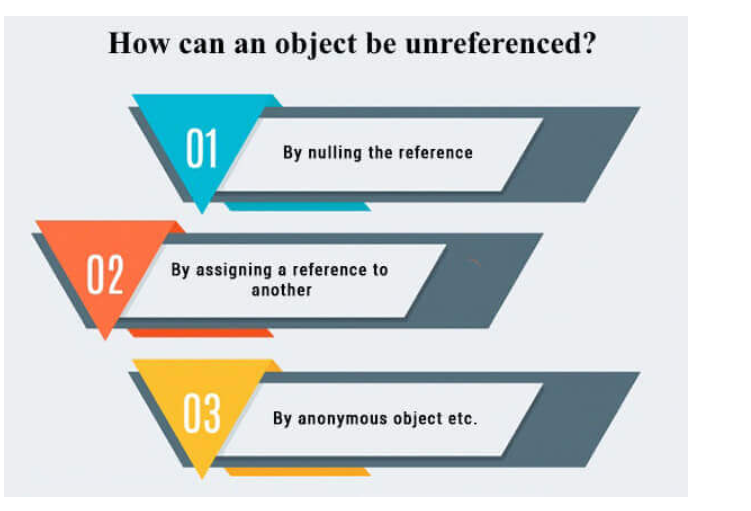
It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

**How can an object be unreferenced? There are many ways:**

nulling the reference

By assigning a reference to another

By anonymous object etc.



**Finalize method in java**

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

# **Inheritance in Java**

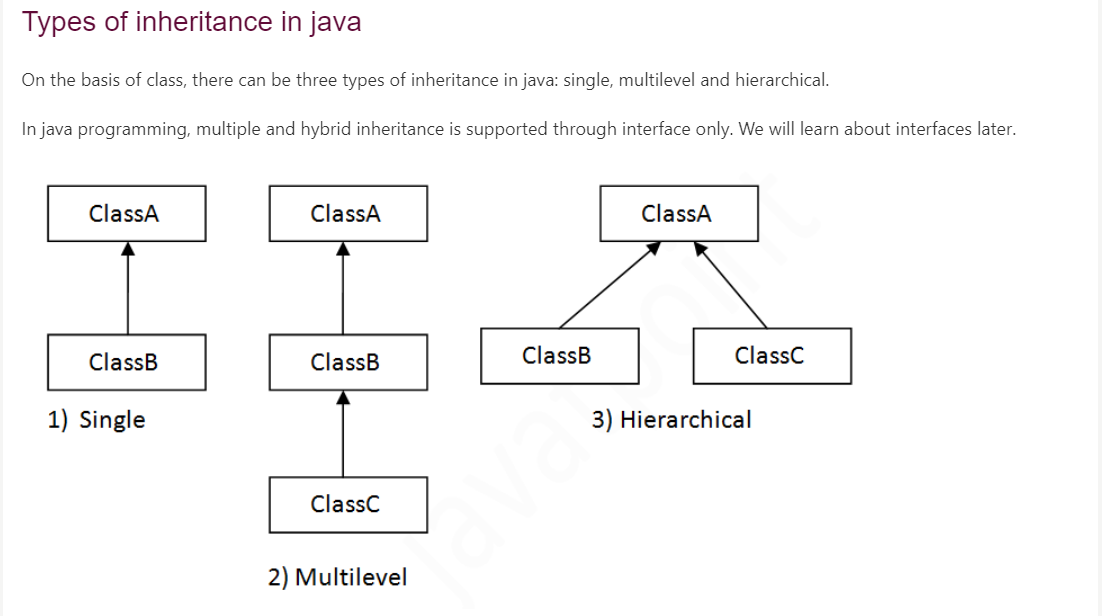
**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.



## Introduction to Java Abstract Class

Hiding internal details and showing only the functionality is known as abstraction. Abstraction is one of the key concepts in Object Oriented Programming Paradigm.

**In order to implement Abstraction, we use Abstract Classes or Interfaces**. Abstract class in Java is a collection of abstract and non-abstract methods. Abstract methods do not have a body or implementation. The implementation of the abstract methods is provided by the child classes which extend the abstract class.

Basically, an Abstract Class is nothing but a blueprint for the child class. Abstract classes justify the layout of your idea and do not really implement them.

### Real-life Example

Let's understand using a real-life example. Suppose you want to create a **Calculator app**.

* Before actually writing code for that, you need to have a blueprint of the things you need for this project. Let's say you need functions like:
  + add() for addition
  + sub() for subtraction
  + mul() for multiplication
  + div() for division
* All these methods will be treated as abstract methods because we are yet not thinking about their implementations. We are just collecting the basic functions we think we need in order to create a calculator app.
* All these functions will be declared as Abstract methods inside the abstract class. These methods will not be implemented inside the abstract class. We will just be declaring them using the keyword "abstract".
* The definition will happen inside the child class once the child class will extend to the abstract class.

## Rules for Using Abstract Class in Java

Let's recap the rules to declare an abstract class -

* You have to use the keyword abstract.
* You cannot instantiate an abstract class.
* An abstract class can contain both abstract and non-abstract methods.
* You can include non-abstract final methods (a method that cannot be overridden) as well in your abstract class.
* Final methods in abstract classes can not be abstract. They must be implemented in the abstract class itself.
* You can also include constructors and non-abstract static methods in your abstract class.

# **Final Keyword In Java**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

## 1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

### **Example of final variable**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

1. **class** Bike9{
2. **final** **int** speedlimit=90;//final variable
3. **void** run(){
4. speedlimit=400;
5. }
6. **public** **static** **void** main(String args[]){
7. Bike9 obj=**new**  Bike9();
8. obj.run();
9. }
10. }//end of class

Output:Compile Time Error

## 2) Java final method

If you make any method as final, you cannot override it.

### **Example of final method**

1. **class** Bike{
2. **final** **void** run(){System.out.println("running");}
3. }
5. **class** Honda **extends** Bike{
6. **void** run(){System.out.println("running safely with 100kmph");}
8. **public** **static** **void** main(String args[]){
9. Honda honda= **new** Honda();
10. honda.run();
11. }
12. }

Output:Compile Time Error

## 3) Java final class

If you make any class as final, you cannot extend it.

### **Example of final class**

1. **final** **class** Bike{}
3. **class** Honda1 **extends** Bike{
4. **void** run(){System.out.println("running safely with 100kmph");}
6. **public** **static** **void** main(String args[]){
7. Honda1 honda= **new** Honda1();
8. honda.run();
9. }
10. }

Output:Compile Time Error