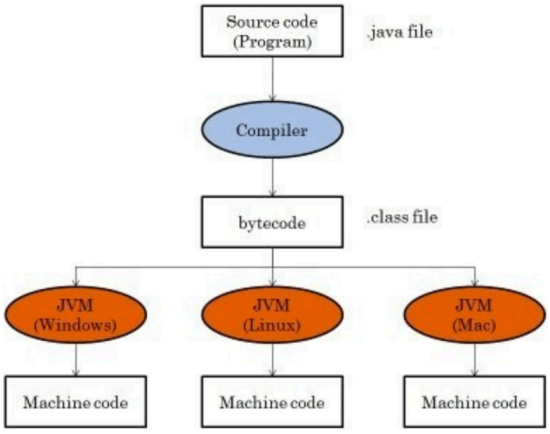
Introduction: Creation of Java, Byte code, Java Buzzwords, Object Oriented Programming, A simple program, Type conversion and casting, Arrays. **Classes:** Class fundamentals, declaring Objects, assigning object reference variables. **A Closer Look at Methods and Classes:** Introducing methods, constructors, this keyword, garbagecollection, the finalize () method.

Creation of Java

- Java is one of the most popular programming languages worldwide. It was created by James Gosling and Patrick Naughton, employees of Sun Microsystems, with support from Bill Joy, co-founder of Sun Microsystems.
- Sun officially presented the Java language at SunWorld on May 23, 1995. Then, in 2009, the Oracle company bought the Sun company.
- The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".
- Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

Byte code

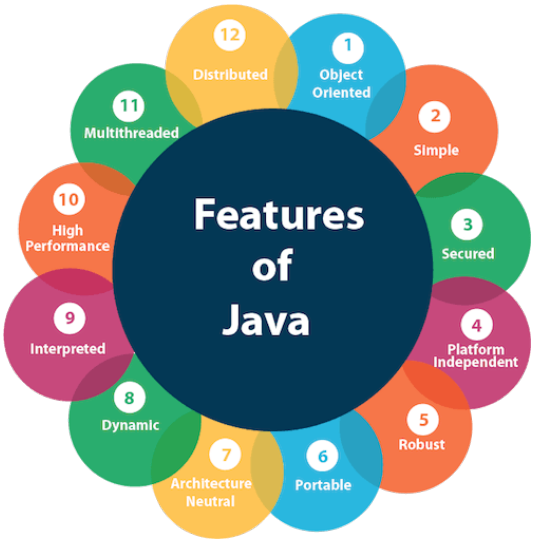
- Java bytecode is the instruction set for the Java Virtual Machine. It acts similar to an assembler which is an alias representation of a C++ code.
- As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we achieve platform independence in java.
- This essentially means that we only need to have basic java installation on any platforms that we want to run our code on.
- Resources required to run the bytecode are made available by the Java Virtual Machine, which calls the processor to allocate the required resources. JVM's are stack-based so they stack implementation to read the codes.



- Bytecode is essentially the machine level language which runs on the Java Virtual Machine.
- Whenever a class is loaded, it gets a stream of bytecode per method of the class. Whenever that method is called during the execution of a program, the bytecode for that method gets invoked.
- Javac not only compiles the program but also generates the bytecode for the program. Thus, we have realized that the bytecode implementation makes Java a **platform-independent** language.
- Portability ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more.

Java Buzzwords

- The primary objective of Java programming language creation was to make it portable, simple and secure programming language.
- Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.



Object-oriented- Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

Simple- Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

Secured- Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside a virtual machine sandbox
- Classloader
- Bytecode Verifier
- Security Manager

Platform Independent- Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

- There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

- The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:
    - Runtime Environment
    - API(Application Programming Interface)
- Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Robust- The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Portable- Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Architecture-neutral- Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Dynamic- Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Interpreted & High-performance- Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit

slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Multi-threaded- A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Distributed- Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
RMI = remote method invocation
EJB = enterprise java bean

Object Oriented Programming

- Object-oriented programming (OOP) is a fundamental programming paradigm based on the concept of "objects". These objects can contain data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods).
- OOP is at core of Java. In fact, all Java programs are to at least some extent objct-oriented.
- OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple programs.

An object consists of:

- **A unique identity:** Each object has a unique identity, even if the state is identical to that of another object.
- **State/Properties/Attributes:** State tells us how the object looks or what properties it has.
- **Behavior:** Behavior tells us what the object does.

Examples of object states and behaviors in Java:

Let's look at some real-life examples of the states and behaviors that objects can have.

Example 1:

- Object: car.
- State: color, brand, weight, model.
- Behavior: break, accelerate, turn, change gears.

Example 2:

- Object: house.
- State: address, color, location.
- Behavior: open door, close door, open blinds.

The three OOP Principles

All object-oriented programming language provides mechanisms that help you implement the object-oriented model. They are

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

- If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

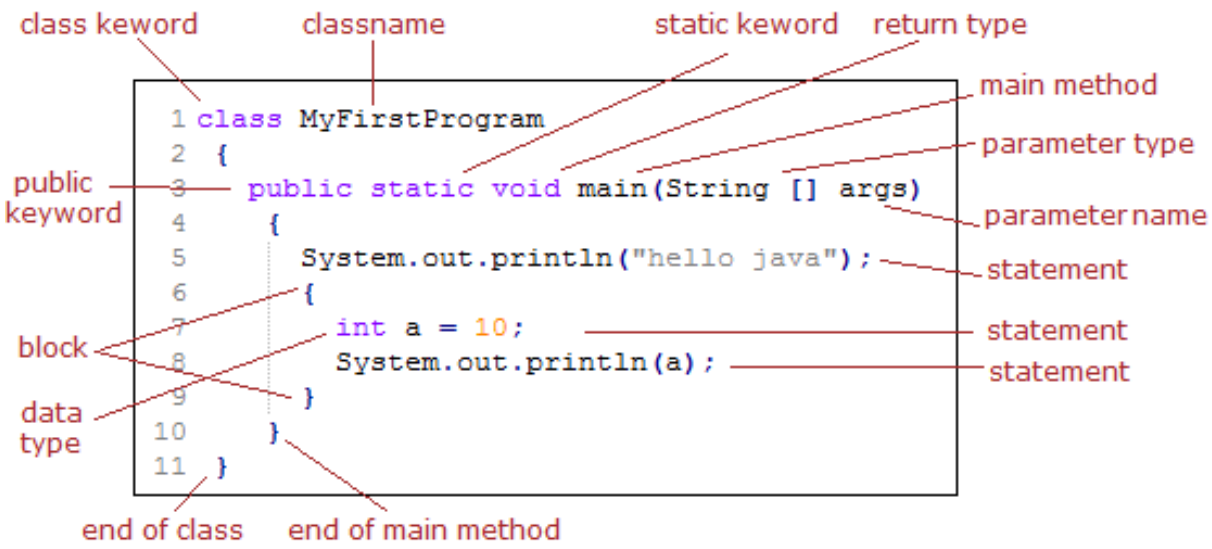
A simple program

```
class Simple
{
  public static void main(String args[])
{
    System.out.println("Hello Java");
  }
}
```

Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the Print Stream class, println() is a method of the Print Stream class. We will discuss the internal working of System.out.println() statement in the coming section.



Type Casting: In typing casting, a data type is converted into another data type by the programmer using the casting operator during the program design. In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called narrowing conversion.

Syntax/Declaration:-

destination_datatype = (target_datatype)variable;

(): is a casting operator.

Type conversion : In type conversion, a data type is automatically converted into another data type by a compiler at the compiler time. In type conversion, the destination data type cannot be smaller than the source data type, that's why it is also called widening conversion. One more important thing is that it can only be applied to compatible data types.

```
int x=30;
float y;
y=x;  // y==30.000000.
```

S.NO	TYPE CASTING	TYPE CONVERSION
1.	In type casting, a data type is converted into another data type by a programmer using casting operator.	Whereas in type conversion, a data type is converted into another data type by a compiler.
2.	Type casting can be applied to **compatible data types** as well as **incompatible data types**.	Whereas type conversion can only be applied to **compatible datatypes**.
3.	In type casting, casting operator is needed in order to cast a data type to another data type.	Whereas in type conversion, there is no need for a casting operator.
4.	In typing casting, the destination data type may be smaller than the source data type, when converting the data type to another data type.	Whereas in type conversion, the destination data type can't be smaller than source data type.
5.	Type casting takes place during the program design by programmer.	Whereas type conversion is done at the compile time.
6.	Type casting is also called narrowing conversion because in this, the destination data type may be smaller than the source data type.	Whereas type conversion is also called widening conversion because in this, the destination data type can not be smaller than the source data type.
7.	Type casting is often used in coding and competitive programming works.	Whereas type conversion is less used in coding and competitive programming as it might cause incorrect answer.
8.	Type casting is more efficient and reliable.	Whereas type conversion is less efficient and less reliable.

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
```

```
// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `length` property:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4
```

Classes: Class fundamentals

To create a class, use the keyword `class`:

Create a class named "Main" with a variable x:

```
public class Main
{
   int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

Create an object called "`Obj`" and print the value of `x`:

```
public class Main {

  int x = 5;

  public static void main(String[] args) {

    Main Obj = new Main();

    System.out.println(Obj.x);

  }

}
```

Multiple Objects

You can create multiple objects of one class:

Create two objects of `Main`:

```
public class Main {
  int x = 5;
  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

Declaring Objects

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

assigning object reference variables

Java, being an object-oriented programming language, allows the use of reference variables to work with objects and their data. In Java, objects are created dynamically on the heap memory, and reference variables are used to hold the memory address of these objects.

Understanding Reference Variables:

In Java, a reference variable is a variable that holds the memory address of an object rather than the actual object itself. It acts as a reference to the object and allows manipulation of its

data and methods. Reference variables are declared with a specific type, which determines the methods and fields that can be accessed through that variable.

When an object is created using the new keyword, memory is allocated on the heap to store the object's data. The reference variable is then used to refer to this memory location, making it possible to access and manipulate the object's properties and behaviors.

```
class Car {
    String brand;
    int year;
}
public class ReferenceVariableExample {
    public static void main(String[] args) {
        // Declare a reference variable of type Car
        Car myCar;
        // Create a new Car object and assign its reference to myCar
        myCar = new Car();
        // Access and modify the object's properties
        myCar.brand = "Toyota";
        myCar.year = 2021;
        // Use the reference variable to perform actions on the object
        System.out.println("Brand: " + myCar.brand);
        System.out.println("Year: " + myCar.year);
    }
}
```

Benefits and Usage of Reference Variables

- **Object Manipulation:** Reference variables allow programmers to work with objects, access their properties, and invoke their methods. They enable object-oriented programming principles such as encapsulation, inheritance, and polymorphism.
- **Memory Efficiency:** Reference variables only store the memory address of an object rather than the entire object itself. This approach helps conserve memory by avoiding unnecessary object duplication.
- **Object Passing:** Reference variables are often used when passing objects as arguments to methods or returning objects from methods. This allows for efficient memory usage and facilitates modular programming.

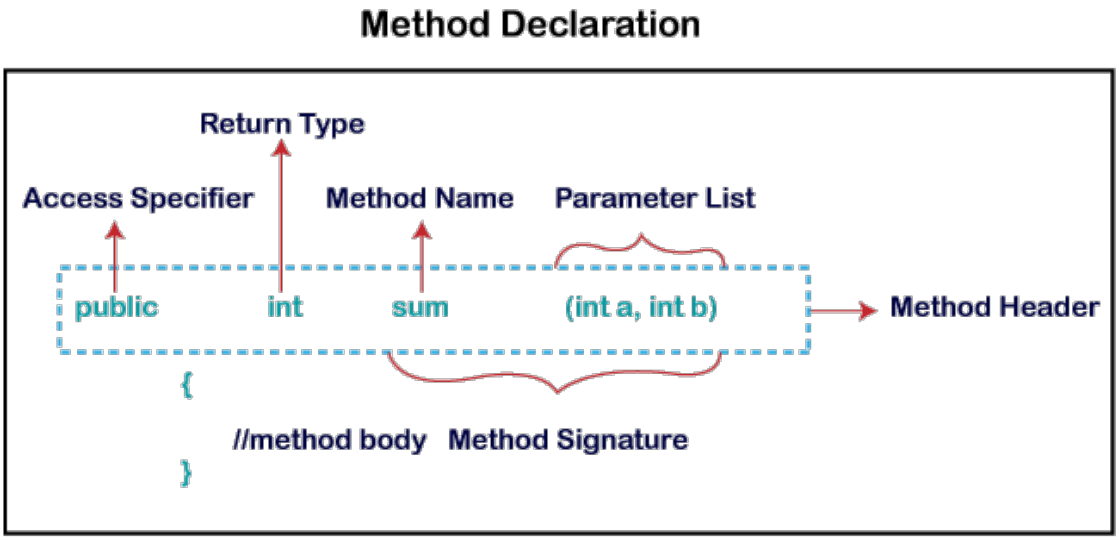
- **Dynamic Behavior:** Reference variables enable dynamic behavior in Java programs. Different objects can be assigned to the same reference variable, allowing flexibility in handling different types of objects at runtime.
- **Object Lifetime Control:** Using reference variables, developers can control the lifetime of objects dynamically. When a reference variable is no longer referencing an object, the object becomes eligible for garbage collection, freeing up memory resources.

Introducing methods

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.

- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the samepackage only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase**letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
    //creating Scanner class object
```

```
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
    //user defined method
    public static void findEvenOdd(int num)
    {
        //method body
        if(num%2==0)
        System.out.println(num+" is even");
        else
        System.out.println(num+" is odd");
    }
    }
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

```
    public class Display
    {
      public static void main(String[] args)
    {
    show();
    }
      static void show()
    {
```

```
    System.out.println("It is an example of static method.");
  }
  }
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

```
public class InstanceMethodExample
{
  public static void main(String [] args)
{
  //Creating an object of the class
  InstanceMethodExample obj = new InstanceMethodExample();
  //invoking instance method
  System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
  s = a+b;
  //returning the sum
  return s;
}
}
```

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Syntax

```
abstract void method_name();
```

```
abstract class Demo //abstract class
{
    //abstract method declaration
    abstract void display();
}
public class MyClass extends Demo
{
    //method impelmentation
void display()
{
    System.out.println("Abstract method?");
}
public static void main(String args[])
{
    //creating object of abstract class
    Demo obj = new MyClass();
    //invoking abstract method
    obj.display();
}
}
```

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:

```
// Create a Main class
public class Main {
  int x;  // Create a class attribute
  // Create a class constructor for the Main class
  public Main() {
```

```
    x = 5;  // Set the initial value for the class attribute x
  }
  public static void main(String[] args) {
    Main myObj = new Main(); // Create an object of class Main
(This will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}
// Outputs 5
```

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

```
public class Main {
  int x;
  public Main(int y) {
    x = y;
  }
  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println(myObj.x);
  }
}
// Outputs 5
```

this keyword

The this keyword refers to the current object in a method or constructor.

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

this can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call

```
public class Main {
  int x;
  // Constructor with a parameter
  public Main(int x) {
    this.x = x;
  }
  // Call the constructor
  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println("Value of x = " + myObj.x);
  }
}
```

garbagecollection

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program.

How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part

of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program.

the finalize () method. To be continued in second module.

Finalize() is the method of Object class. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

Syntax

```
protected void finalize() throws Throwable
```

Protected method: protected is an access specifier for variables and methods in Java. When a variable or method is protected, it can be accessed within the class where it's declared and other derived classes of that class.

all classes inherit the Object class directly or indirectly in Java. The finalize() method is protected in the Object class so that all classes in Java can override and use it.

When to Use finalize() Method in Java?

Garbage collection is done automatically in Java, which the JVM handles. Java uses the finalize method to release the resources of the object that has to be destroyed.

GC call the finalize method only once, if an exception is thrown by finalizing method or the object revives itself from finalize(), the garbage collector will not call the finalize() method again.

It's not guaranteed whether or when the finalized method will be called. Relying entirely on finalization to release resources is not recommended.

There are other ways to release the resources used in Java, like the close() method for file handling or the destroy() method. But, the issue with these methods is they don't work automatically, we have to call them manually every time.

In such cases, to improve the chances of performing clean-up activity, we can use the finalize() method in the final block. finally block will execute finalize() method even though the user has used close() method manually.

We can use the finalize() method to release all the resources used by the object.* In the final block, we can override the finalize() method.* An example of overriding the finalize method in Java is given in the next section of this article.

```
public class Student
{
   public static void main(String[] args)
   {
      Student s1 = new Student();
      s1 = null;
      System.gc();
      System.out.println("Garbage collector is called");
   }

   @Override
   protected void finalize()
   {
      System.out.println("Finalize method is called.");
   }
}
```

Explanation:

In this example, the s1 object is eligible for garbage collection. When System.gc(); invokes the garbage collector, it calls the object class's finalize() method. The overridden finalize() method of the Student class is called, and it prints Finalize method is called.

2. Explicit Call to finalize() Method

When we call the finalize() method explicitly, the JVM treats it as a normal method; it cannot remove the object from memory. The finalize() method can release memory and resources related to an object only when a Garbage collector calls it. Compiler will ignore the finalize() method if it's called explicitly and not invoked by the Garbage collector. Let's understand this practically:

```
public class Demo
{
   public static void main(String[] args)
   {
      Demo demo1 = new Demo();
      Demo demo2 = new Demo();
      demo1 = demo2;
      demo1.finalize(); // Explicit call to finalize method
      System.out.println("Garbage collector is called");
      System.gc(); // Implicit call to finalize() method
   }

   @Override
   protected void finalize()
   {
      System.out.println("Finalize() method is called");
   }
}
```

Best Practices to Use finalize() Method Correctly

Try to avoid the use of the finalize method. In case you need to use it, the following are points to remember while using the finalize() method:

Do not use the finalize() method to execute time-critical application logic, as the execution of the finalize method cannot be predicted.

Call the super.finalize() method in the finally block; it will ensure that the finalize method will be executed even if any exception is caught. Refer to the following finalize() method template:

```
@Override
protected void finalize() throws Throwable
{
  try{
    //release resources here
  }catch(Throwable t){
    throw t;
  }finally{
    super.finalize();
  }
}
```

Conclusion

- finalize() is an Object class method in Java, and it can be used in all other classes by overriding it.
- The garbage collector uses the finalize() method to complete the clean-up activity before destroying the object.
- JVM allows invoking of the finalize() method in Java only once per object.
- Garbage collection is automated in Java and controlled by JVM.
- Garbage Collector calls the finalize method in java of that class whose object is eligible for Garbage collection.

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder
{
  static int add(int a,int b)
  {
     return a+b;}
    static int add(int a,int b,int c)
     {
        return a+b+c;
      }
```

```
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

Constructor overloading in Java

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Consider the following Java program, in which we have used different constructors in the class.

```
public class Student {
//instance variables of the class
int id;
String name;

Student(){
System.out.println("this a default constructor");
}

Student(int i, String n){
id = i;
name = n;
}

public static void main(String[] args) {
//object creation
Student s = new Student();
System.out.println("\nDefault Constructor values: \n");
```

```
System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

System.out.println("\nParameterized Constructor values: \n");
Student student = new Student(10, "David");
System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.nam
e);
}
}
```

In the above example, the Student class constructor is overloaded with two different constructors, I.e., default and parameterized.

Method Overloading and Type Promotion

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers**and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}


//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}


//save by B.java
```

```
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
```

```
    String college="DSCE";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="DSCE";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="DSCE";
 s1.display();
 s2.display();
 }
}
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "DSCE";
    //static method to change the value of static variable
    static void change(){
    college = "DSU";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
```

```
    s2.display();
    s3.display();
    }
}
```

Output:111 Karan DSU
222 Aryan DSU
333 Sonoo DSU

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static
{
System.out.println("static block is invoked");
}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output: static block is invoked
Hello main

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.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//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

```
class Bike{
  final void run()
{
```

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

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[])
  {
  Honda honda= new Honda();
  honda.run();
  }
}
```

Output: Compile Time Error

3) Java final class

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

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error