

Object Oriented Programming using Java

Programming Paradigm

- Paradigm means methodology
- A Programming Paradigm is a fundamental style of computer programming technique that defines how the structure and basic elements of a computer program is built
- While some programming languages strictly follow a single paradigm, others may draw concepts from more than one

Types of Programming Paradigm

- Monolithic Programming – emphasizes on finding a solution
- Structures Programming – focus on modules
- Procedure-oriented Programming – lays stress on algorithm
- Object-oriented Programming – emphasizes on class and objects

Monolithic Programming

- It indicates the program which contains a single function for a large program.
- A program is not divided into parts and hence is the name.
- When the program size increases it leads inconvenience and difficult to maintain.
- The program contains jump statements such as goto that transfer control to any statement as specified in it.
- The global data can be accessed and modified from any part of the program and hence posing a serious threat.
- It is suitable to develop simple and small application.
- Example : Basic

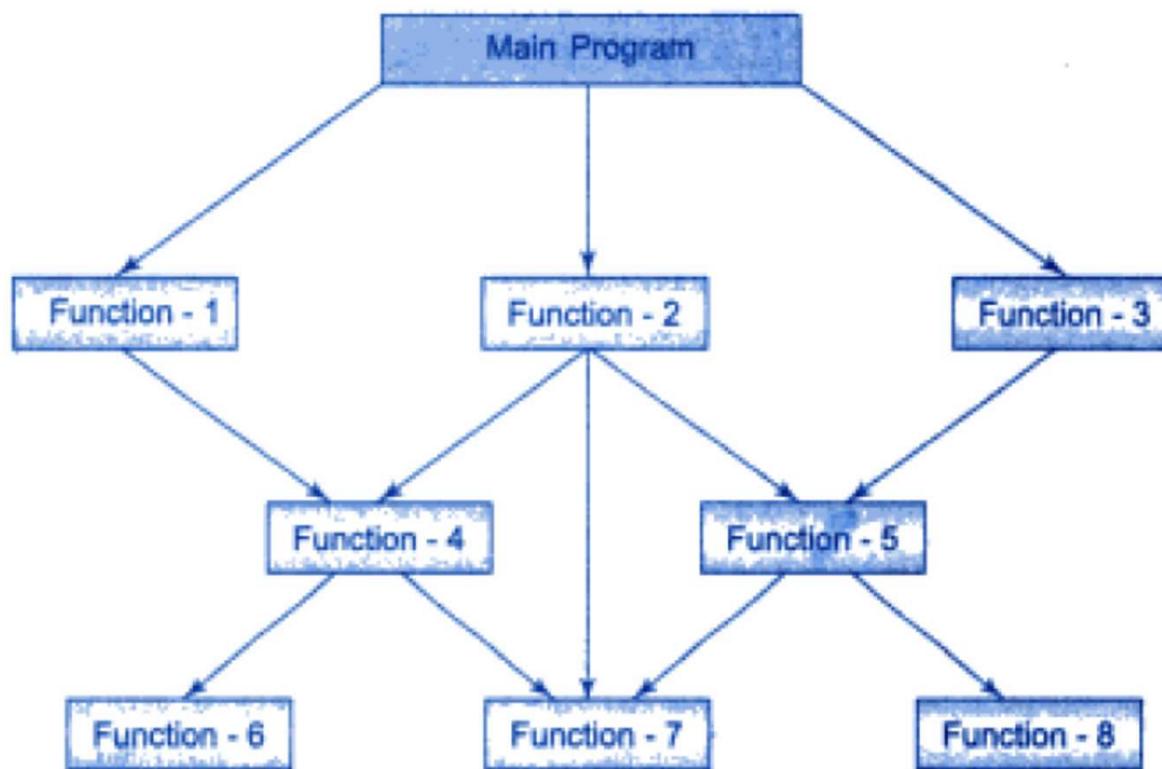
Structured Programming

- Programming tasks can be split into smaller sections known as functions or subroutines, which can be called whenever they are required.
- It is often associated with “Top-down” approach to design.
- It attempts to divide the problem into smaller blocks or procedures which interact with each other.
- Example : Pascal,Ada,C

Procedure-oriented Programming

- It basically consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as functions.
- The primary focus is on function rather than data.
- Examples : COBOL, Fortran

Structure of Procedure-oriented Programming

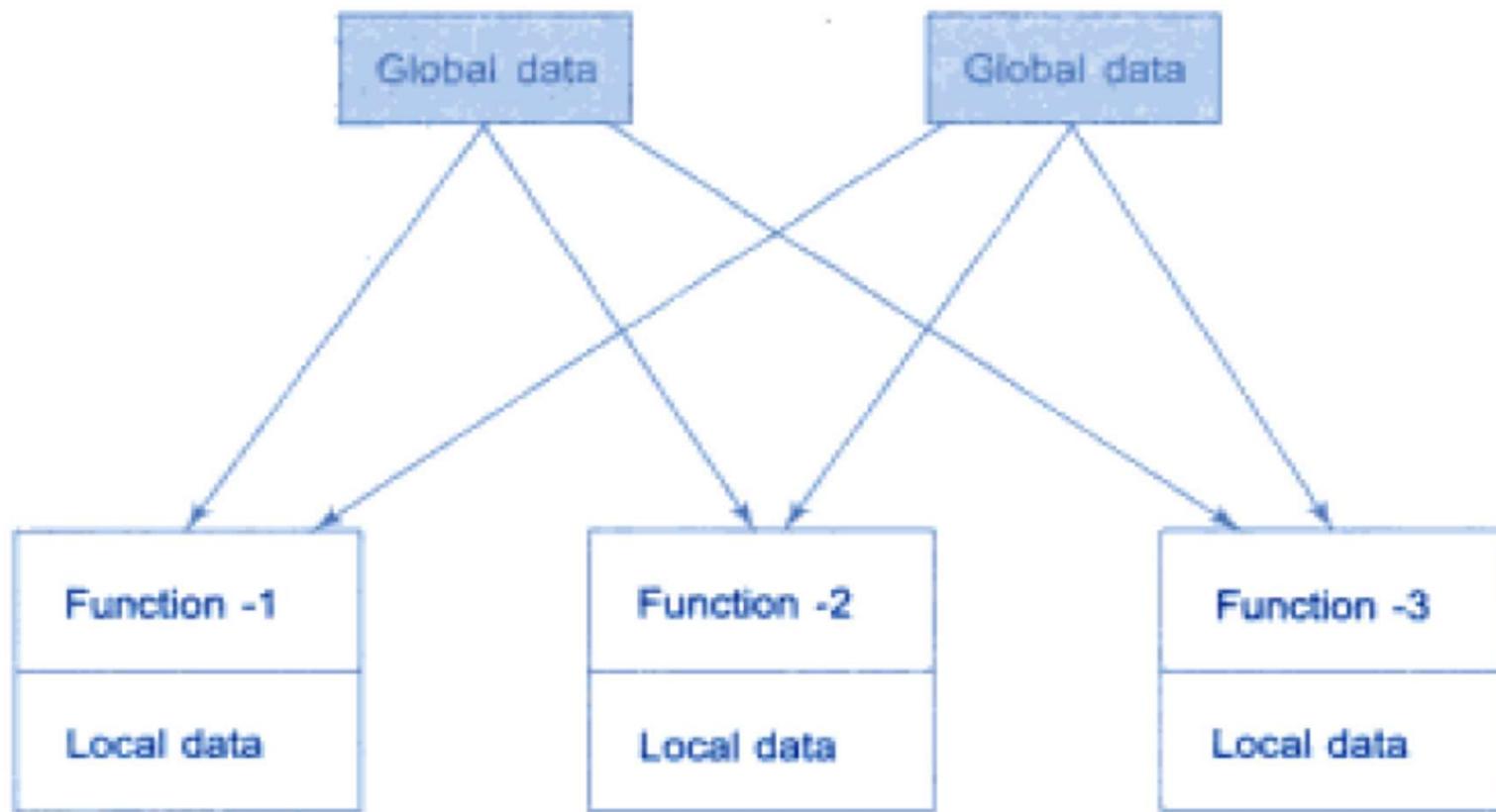


By Jagadish Sahoo

Characteristics of POP

- Emphasis is on doing things(Algorithms).
- Larger programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design

Relationship of Data and Functions in POP

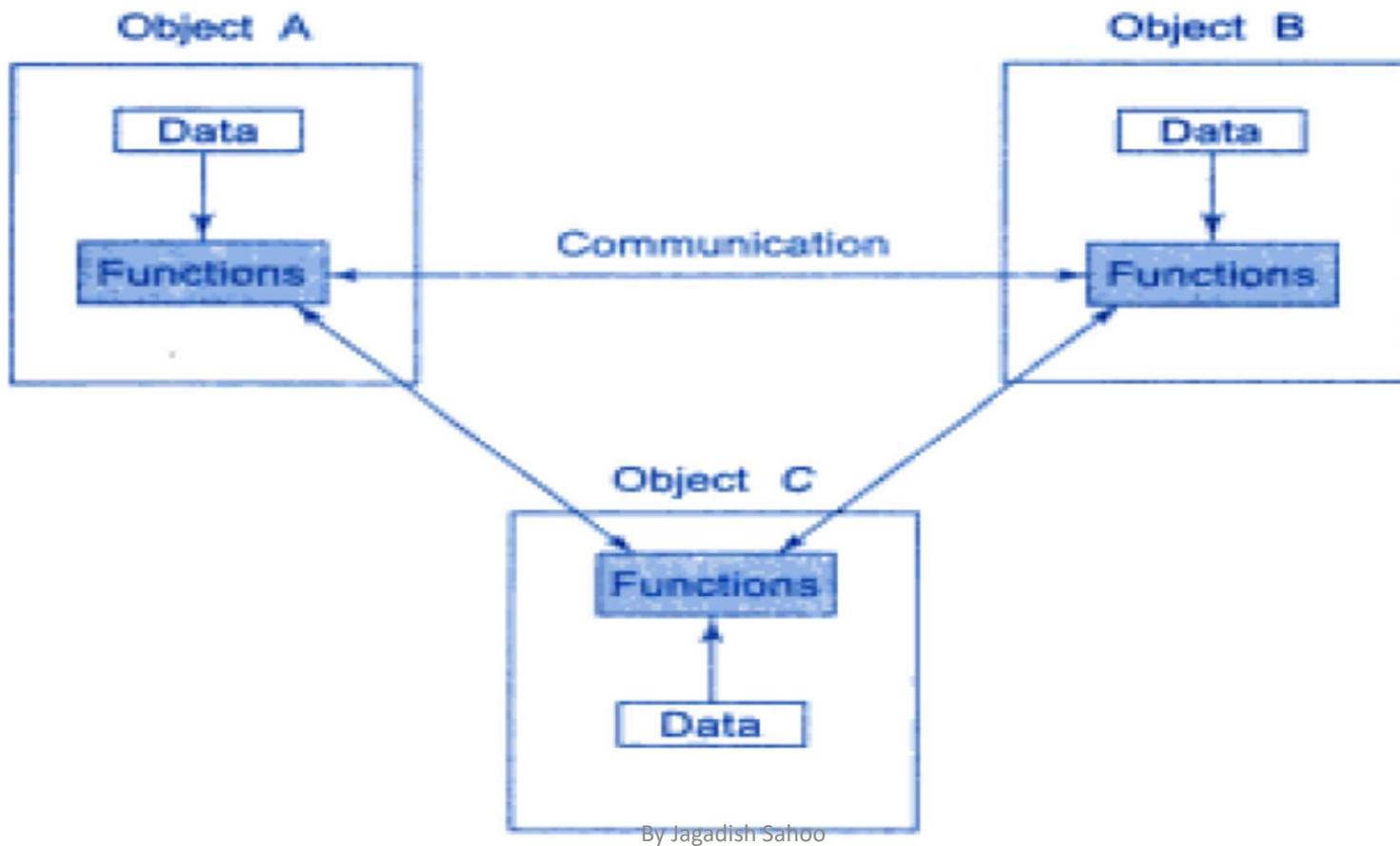


By Jagadish Sahoo

Object-oriented Programming

- It treats data as a critical element in program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions.
- OOP allows decomposition of a problem into number of entities called objects and then build data and functions around these objects.
- The data of an object can be accessed only by the functions associated with that object.
- Functions of one object can access the functions of another object.

Organization of data and function in OOP



Characteristics of OOP

- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data Structures are designed such that they characterize the objects.
- Functions that operate on data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.
- Follows bottom-up approach in program design.

Basic concepts of OOP

- Classes
- Objects
- Data abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Message Passing

By Jagadish Sahoo

Classes

- Classes are user-defined datatypes
- A class is a collection of Data member and member functions.
- Variables declared in a class are called data member and functions declared in class are called member functions or methods.
- Objects are variable of type class. Once a class has been defined, we can create any number of objects belonging to that class.
- Each object is associated with the data of type class with which they are created.
- If Fruit is a class, then apple, orange, banana etc. are the objects of the class Fruit.
- Class is a logical structure.

By Jagadish Sahoo

Object

- Basic run-time entities in an object-oriented system i.e. fundamental building blocks for designing a software.
- It is a collection of data members and associated member function(method).
- An object represents a particular instance of a class.
- An object has 3 characteristics:
 1. Name
 2. State
 3. Behavior
- Objects take up space in the memory and have associated address.
- When a program is executed, objects interact by sending messages to one another.
- Example : Book, Bank, Customer, Account etc.

By Jagadish Sahoo

Object: STUDENT

DATA

Name

Date-of-birth

Marks

FUNCTIONS

Total

Average

Display

By Jagadish Sahoo

Data Abstraction

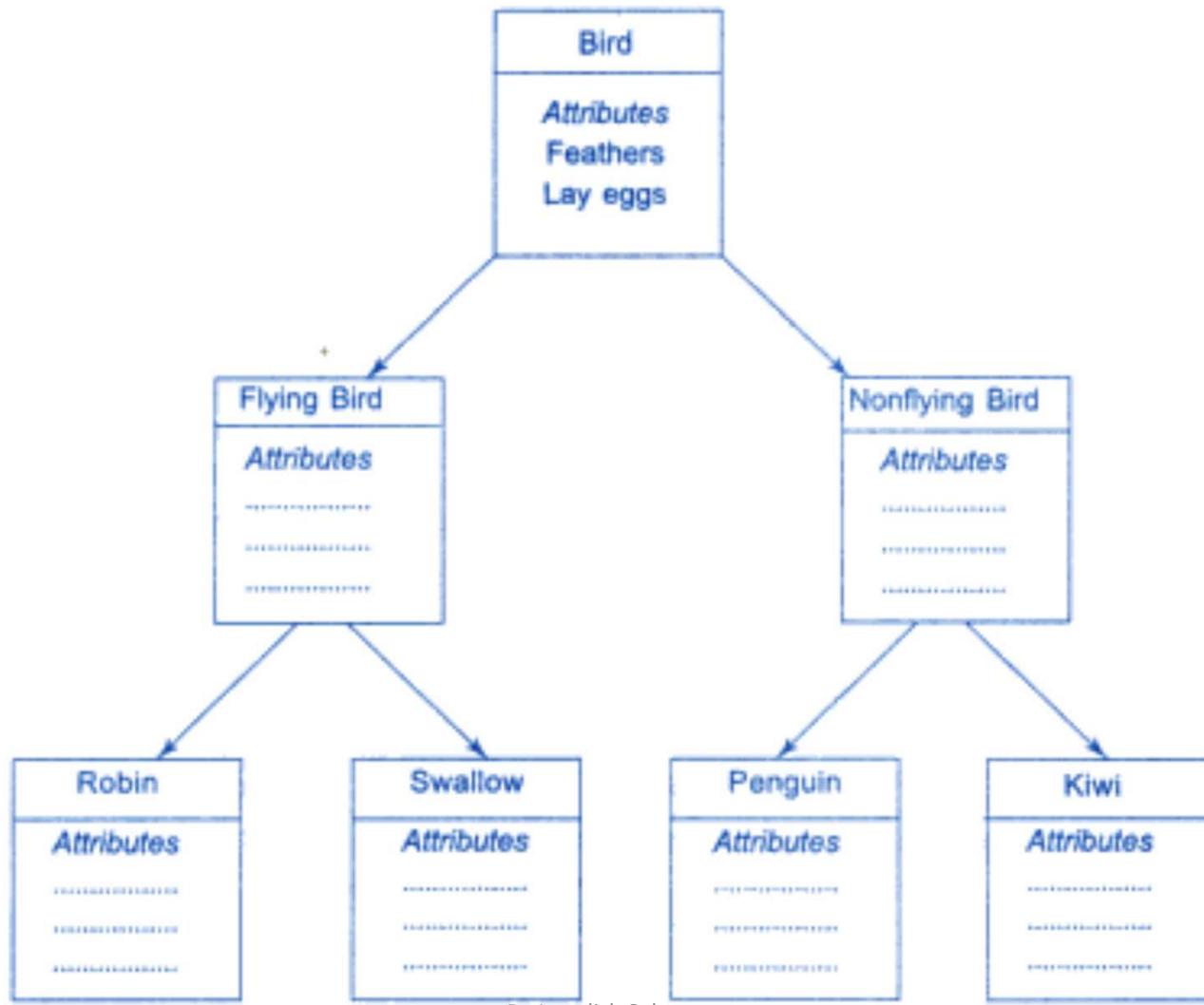
- Abstraction refers to the act of representing essential features without including the background details or explanation.
- Data abstraction is an encapsulation of object's state and behavior.
- Data abstraction increases the power of programming languages by creating user-defined data types.
- Classes uses the concept of abstraction and are defined as a list of abstract attributes(data members) and functions(methods).
- Since classes use the concept of data abstraction, they are also used as Abstract Data Type(ADT).

Encapsulation

- Data encapsulation combines data and functions into a single unit called class.
- When using data encapsulation, data is not accessed directly, it is only accessible through the methods present inside the class.
- Data encapsulation enables data hiding, which is an important concept of OOP.
- Example : Capsule is wrapped with different medicines.

Inheritance

- It is the process by which one object can acquire the properties of another.
- It allows the declaration and implementation of new class/classes from existing class.
- The existing class is known as base class/parent class/super class and the new class/classes is/are known as derived class/child class/sub class.
- It uses the concept of Reusability.



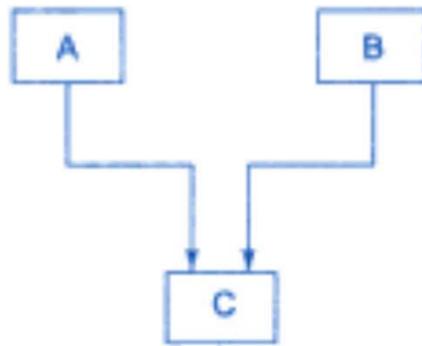
By Jagadish Sahoo

Types of Inheritance

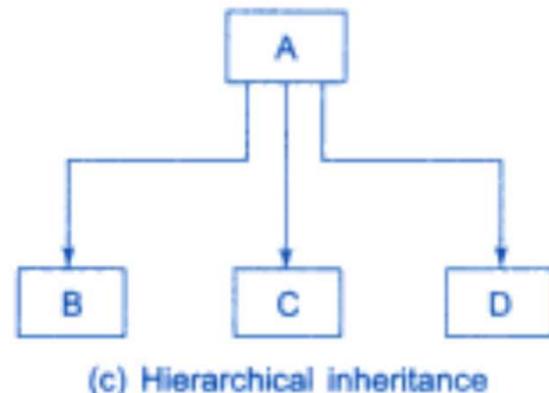
- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance



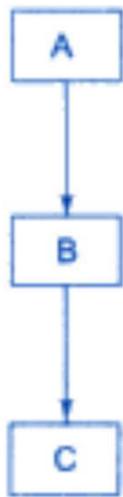
Single inheritance



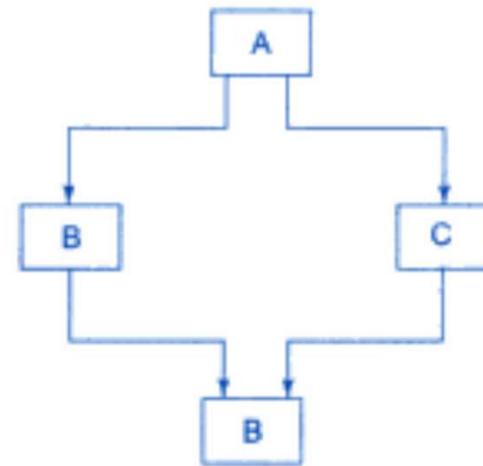
(b) Multiple inheritance



(c) Hierarchical inheritance



Multilevel inheritance



(e) Hybrid inheritance

By Jagadish Sahoo

Benefits of Inheritance

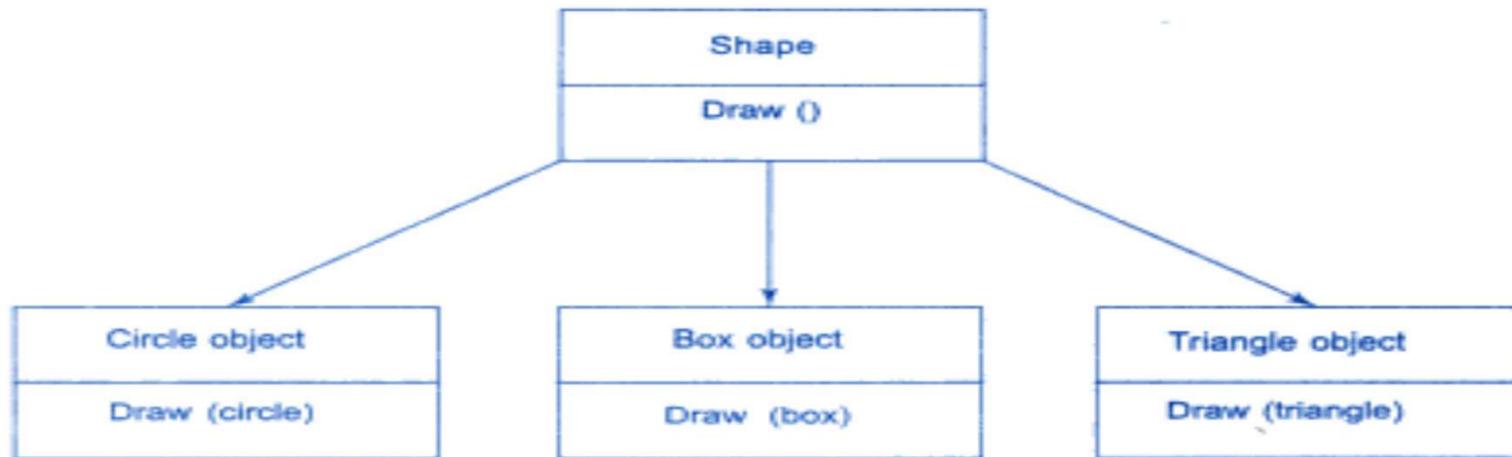
- It helps to reuse an existing part rather than hardcoding every time.
- It increases reliability and decreases maintenance cost.
- Development time is less & product can be generated more quickly.

Polymorphism

- The ability to take more than one form is known as Polymorphism.
- An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.
- + operator can be used to add 2 numbers and the result is sum of two numbers.
- Same + operators can be used to add 2 strings and the result is concatenation of 2 strings.
- This process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Polymorphism(Function Overloading)

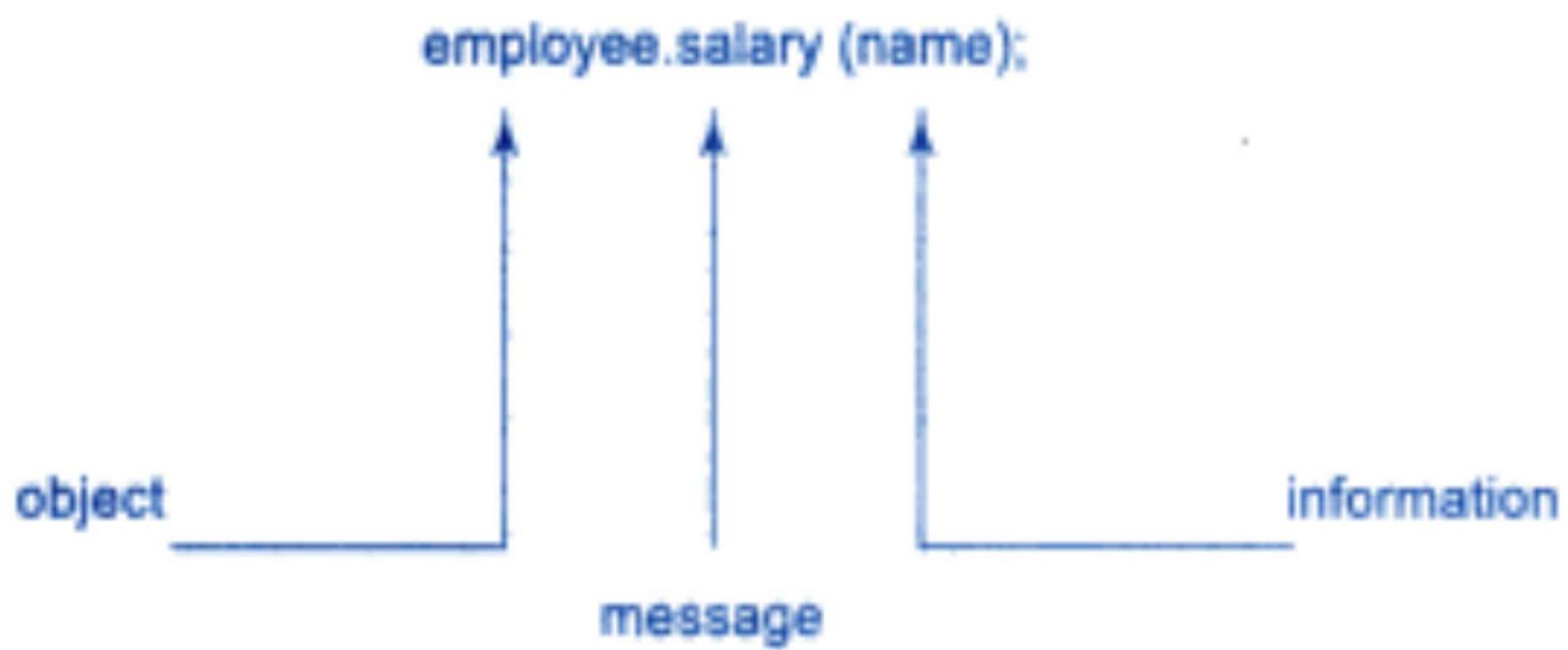
- A single function name can be used to handle different number and different types of arguments.
- Using a single function name to perform different types of tasks is known as function overloading.



By Jagadish Sahoo

Message Passing

- Any processing is accomplished by sending messages to objects.
- A message for an object is a request for execution of a procedure/function.
- It invoke a function on the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, name of the function(message) and information to be sent.



By Jagadish Sahoo

What is Java

- Java is a **programming language** and a **platform**.
- Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995.
- *James Gosling* is known as the father of Java.
- Before Java, its name was *Oak*.
- Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.
- **Platform:** Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform

Application

- Desktop Applications such as acrobat reader, media player, antivirus, etc.
- Web Applications such as irctc.co.in.
- Enterprise Applications such as banking applications.
- Mobile
- Embedded System
- Smart Card
- Robotics
- Games, etc.

Types of Java Applications

1) Standalone Application

- Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

- An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](#), [JSP](#), [Struts](#), [Spring](#), [Hibernate](#), [JSF](#), etc. technologies are used for creating web applications in Java.

3) Enterprise Application

- An application that is distributed in nature, such as banking applications, etc. is called enterprise application. In Java, [EJB](#) is used for creating enterprise applications.

4) Mobile Application

- An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

1) Java SE (Java Standard Edition)

- It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](#), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, etc.

2) Java EE (Java Enterprise Edition)

- It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB etc.

3) Java ME (Java Micro Edition)

- It is a micro platform which is mainly used to develop mobile applications.

4) JavaFX

- It is used to develop rich internet applications. It uses a light-weight user interface API.

History

- 1) **James Gosling, Mike Sheridan, and Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- 2) Initially designed for small, embedded systems in electronic appliances like set-top boxes.
- 3) Firstly, it was called "**Greentalk**" by James Gosling, and the file extension was .gt.
- 4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java named "Oak"?

- Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
- In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

- The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.
- According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.
- Java is an island of Indonesia where the first coffee was produced (called java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having coffee near his office.

- Java is just a name, not an acronym.
- Initially developed by James Gosling at [Sun Microsystems](#) (which is now a subsidiary of Oracle Corporation) and released in 1995.
- In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
- JDK 1.0 released in(January 23, 1996). After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications etc. Each new version adds the new features in Java.

Features of Java

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Simple

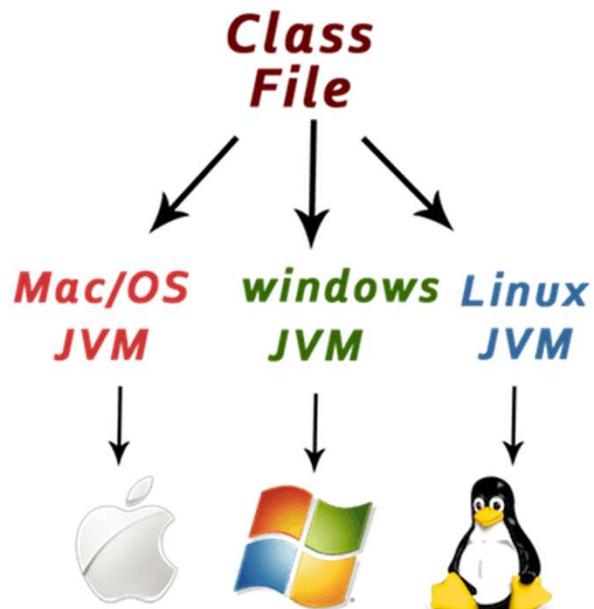
- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

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 incorporates both data and behavior.

Platform Independent

- Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines
- Java is a write once, run anywhere language.



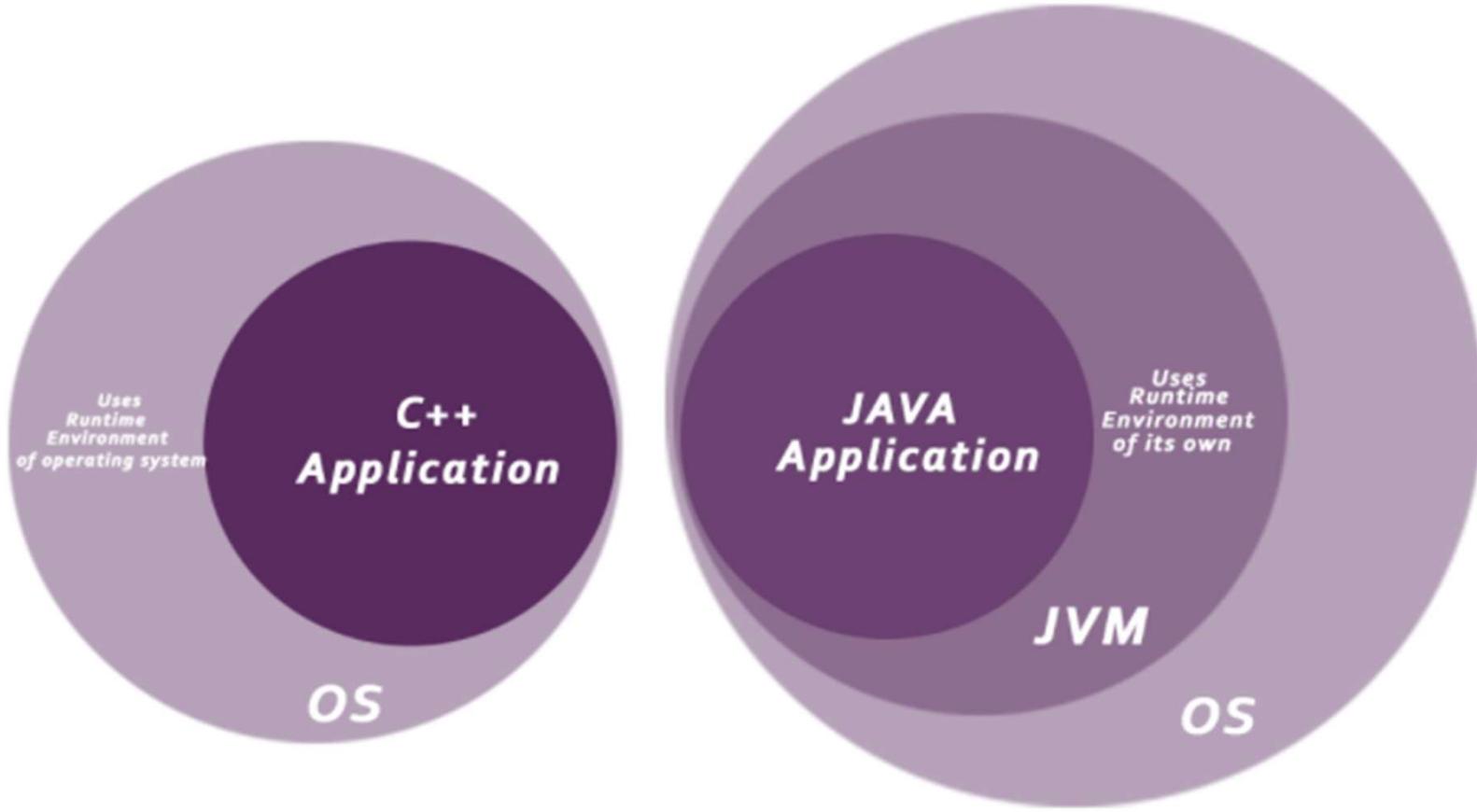
By Jagadish Sahoo

- A platform is the hardware or software environment in which a program runs.
- There are two types of platforms 1) software-based and 2) 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 the top of other hardware-based platforms.
- It has two components:
 1. Runtime Environment
 2. API(Application Programming Interface)

- Java code can be run 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).

Secured

- No explicit pointer
- Java Programs run inside a virtual machine sandbox
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.



By Jagadish Sahoo

Robust(Strong)

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java 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.

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.
- It occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

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

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.

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.

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.

Dynamic

- Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.
- Java supports dynamic compilation and automatic memory management (garbage collection).

Features	C++	Java
Platform Independence	Platform-dependent	Platform-independent
Interpreter and Compiler	Compiled programming language	Compiled and interpreted language
Portability	Not portable	Portable
Memory Management	Manual	System-controlled
Multiple Inheritance	Supports single inheritance and multiple inheritance	Only supports single inheritance
Overloading	Both operators and methods can be overloaded	Allows only method overloading
Compatibility with Other Programming Languages	Compatible with C	Not compatible with any language
Pointers	Supports pointers	Supports pointers with restrictions
Documentation Comment	Does not support documentation comments	Has built-in documentation comments support (<code>/**...**/</code>), allowing Java files to have their own documentation
Thread Support	Does not support thread	Has built-in thread support via the "thread" class

By Jagadish Sahoo

Requirements to write a java program

- Install the JDK
- Set path of the jdk/bin directory.
- Create the java program
- Compile and run the java program

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

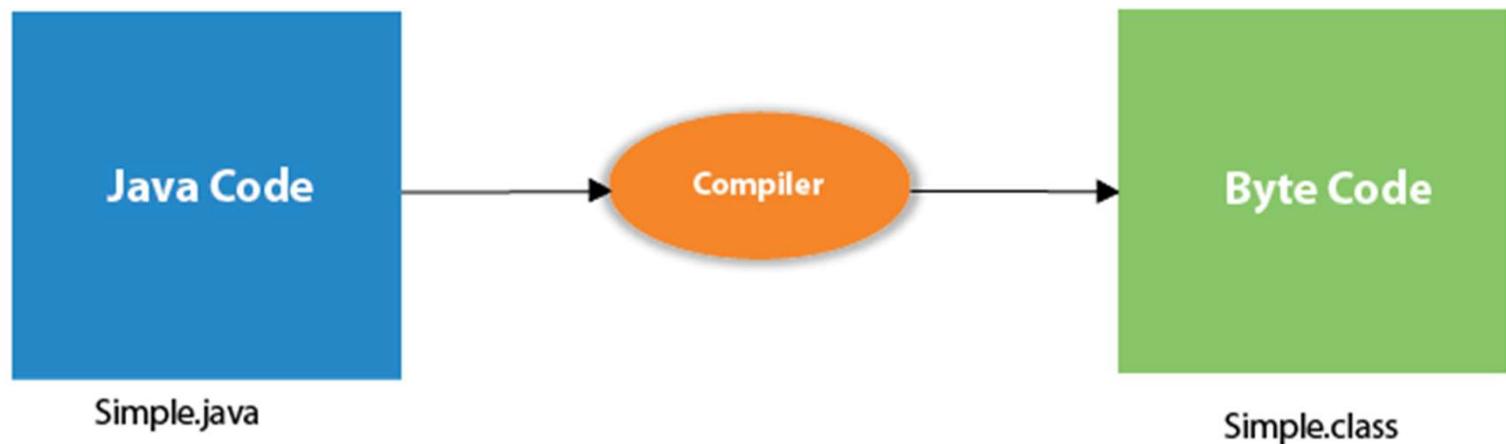
- save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Compilation Flow

- When we compile Java program using javac tool, java compiler converts the source code into byte code



By Jagadish Sahoo

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

- class keyword is used to declare a class in java.
- public keyword is an access modifier which 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 to create an object to invoke the main method. So it saves memory.

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

- 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 is used for command line argument.
- System.out.println() is used to print statement. Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class.

How many ways can we write a Java program

- By changing the sequence of the modifiers, method prototype is not changed in Java.

```
static public void main(String args[])
```

- The subscript notation in Java array can be used after type, before the variable or after the variable.

```
public static void main(String[] args)
```

```
public static void main(String []args)
```

```
public static void main(String args[])
```

- You can provide var-args support to the main method by passing 3 ellipses (dots)

```
public static void main(String... args)
```

- Having a semicolon at the end of class is optional in Java.

```
class A{  
    static public void main(String... args){  
        System.out.println("hello java4");  
    }  
};
```

Valid java main method signature

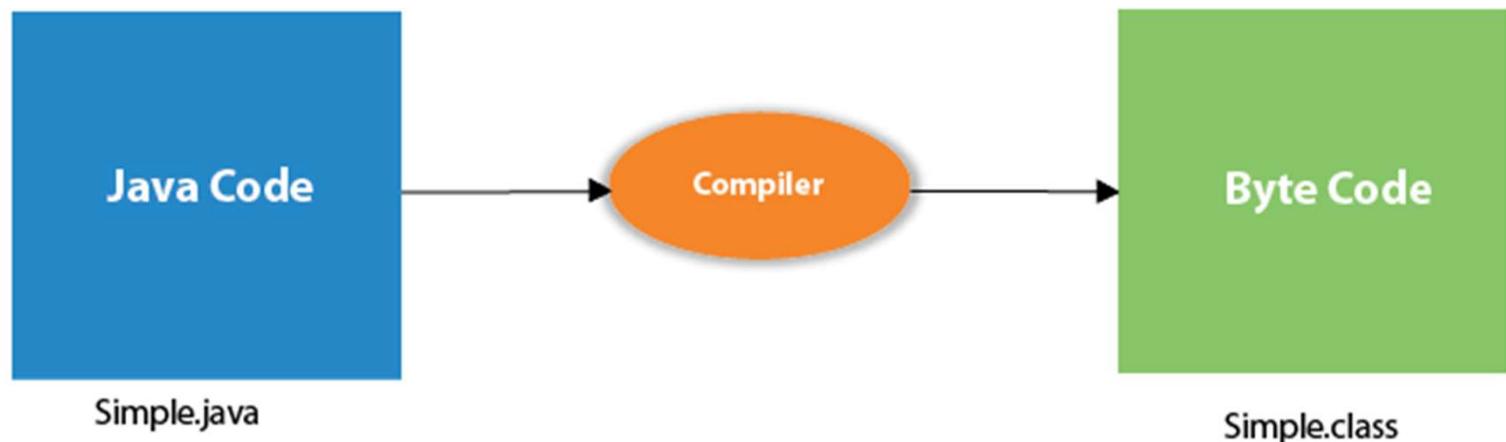
- **public static void** main(String[] args)
- **public static void** main(String []args)
- **public static void** main(String args[])
- **public static void** main(String... args)
- **static public void** main(String[] args)
- **public static final void** main(String[] args)
- **final public static void** main(String[] args)
- **final strictfp public static void** main(String[] args)

Invalid java main method signature

- **public void** main(String[] args)
- **static void** main(String[] args)
- **public void static** main(String[] args)
- **abstract public static void** main(String[] args)

What happens at compile time?

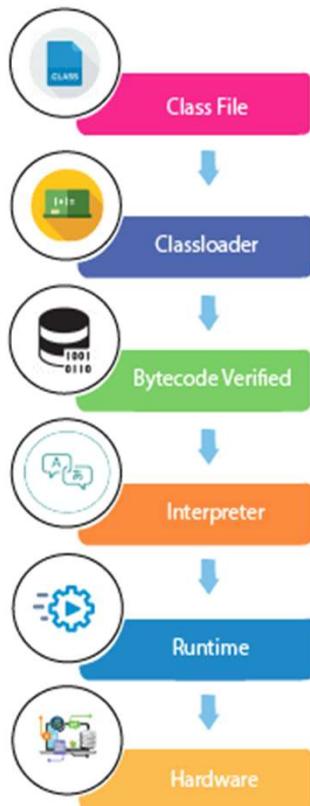
- At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.
-



By Jagadish Sahoo

What happens at runtime?

- At runtime, following steps are performed:



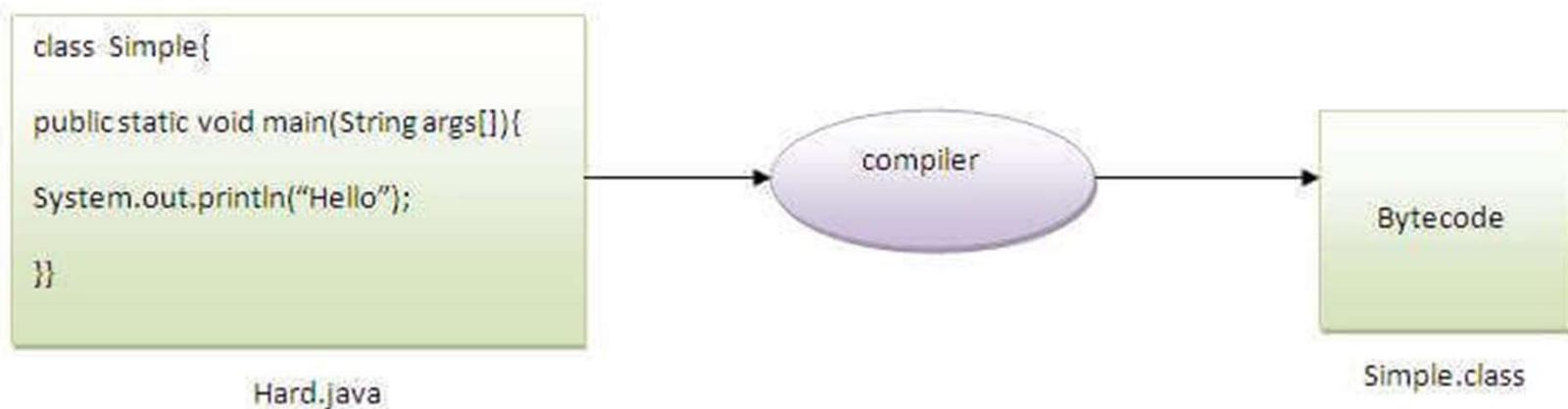
Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Can we save a java source file by other name than the class name?

- Yes, if the class is not public.



To compile:

javac Hard.java

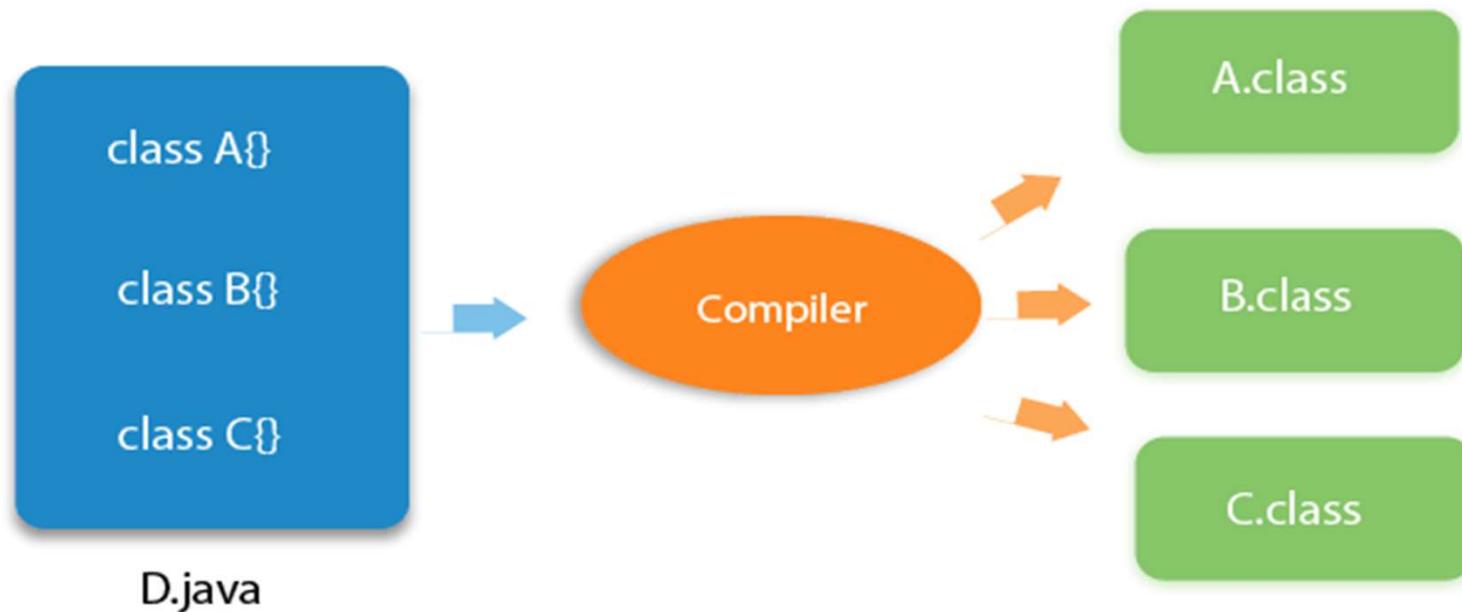
To execute:

java Simple

By Jagadish Sahoo

Can we have multiple classes in a java source file?

- Yes



JVM

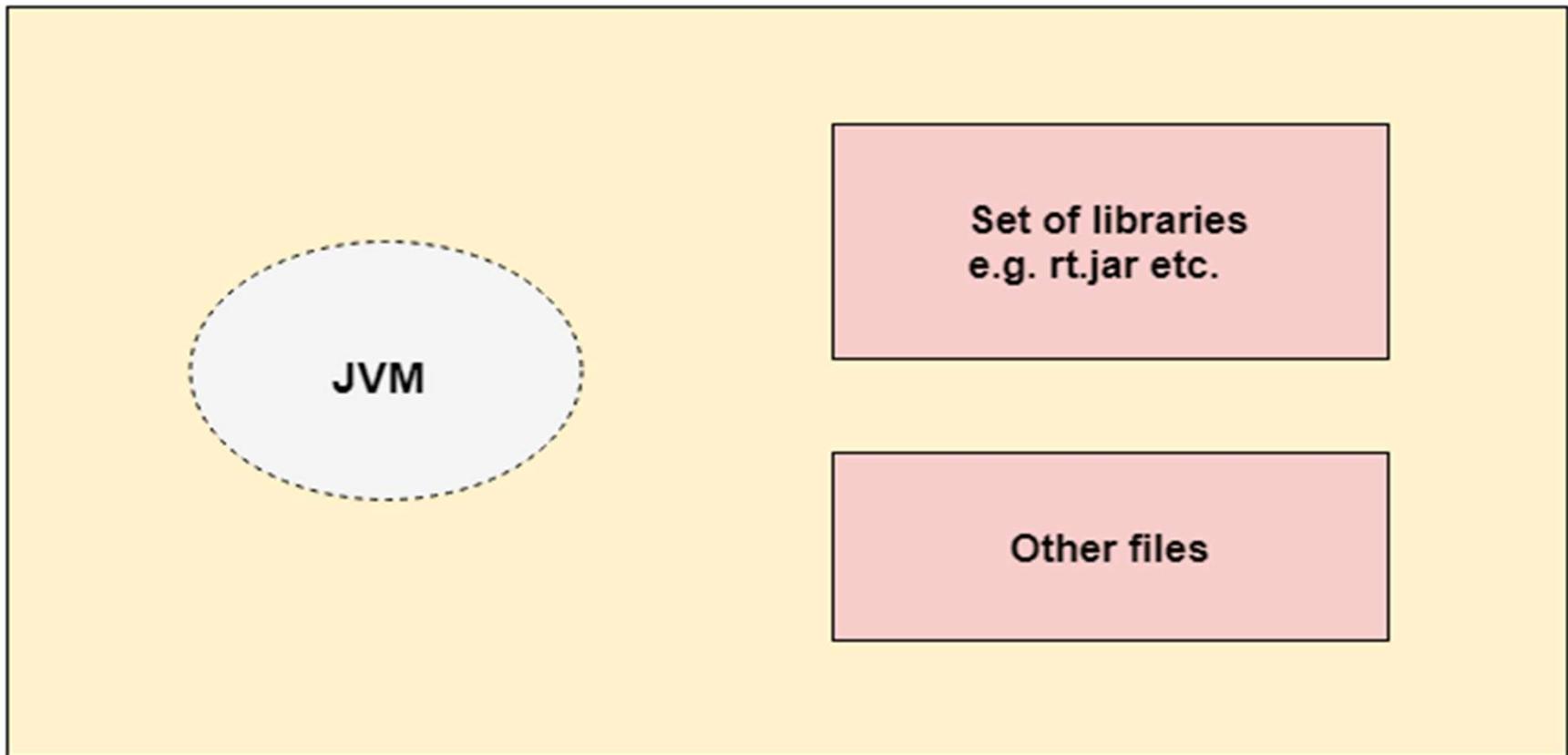
- JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist.
- It is a specification that provides a runtime environment in which Java bytecode can be executed.
- It can also run those programs which are written in other languages and compiled to Java bytecode.
- JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other.
- However, Java is platform independent.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

- JRE is an acronym for Java Runtime Environment.
- The Java Runtime Environment is a set of software tools which are used for developing Java applications.
- It is used to provide the runtime environment.
- It is the implementation of JVM.
- It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

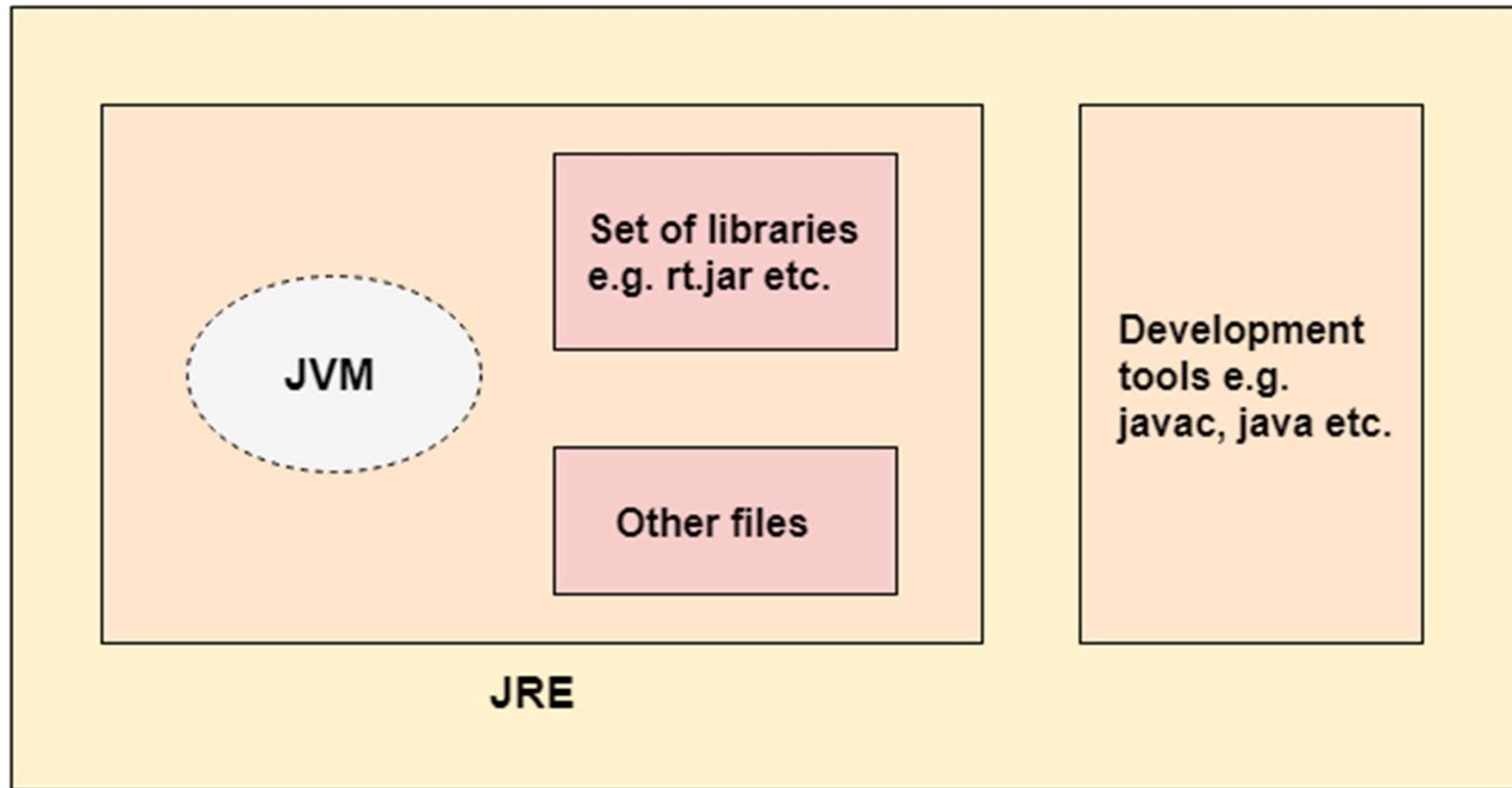


JRE

By Jagadish Sahoo

JDK

- JDK is an acronym for Java Development Kit.
- The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.
- It physically exists. It contains JRE + development tools.
- The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JDK

By Jagadish Sahoo

JVM (Java Virtual Machine) Architecture

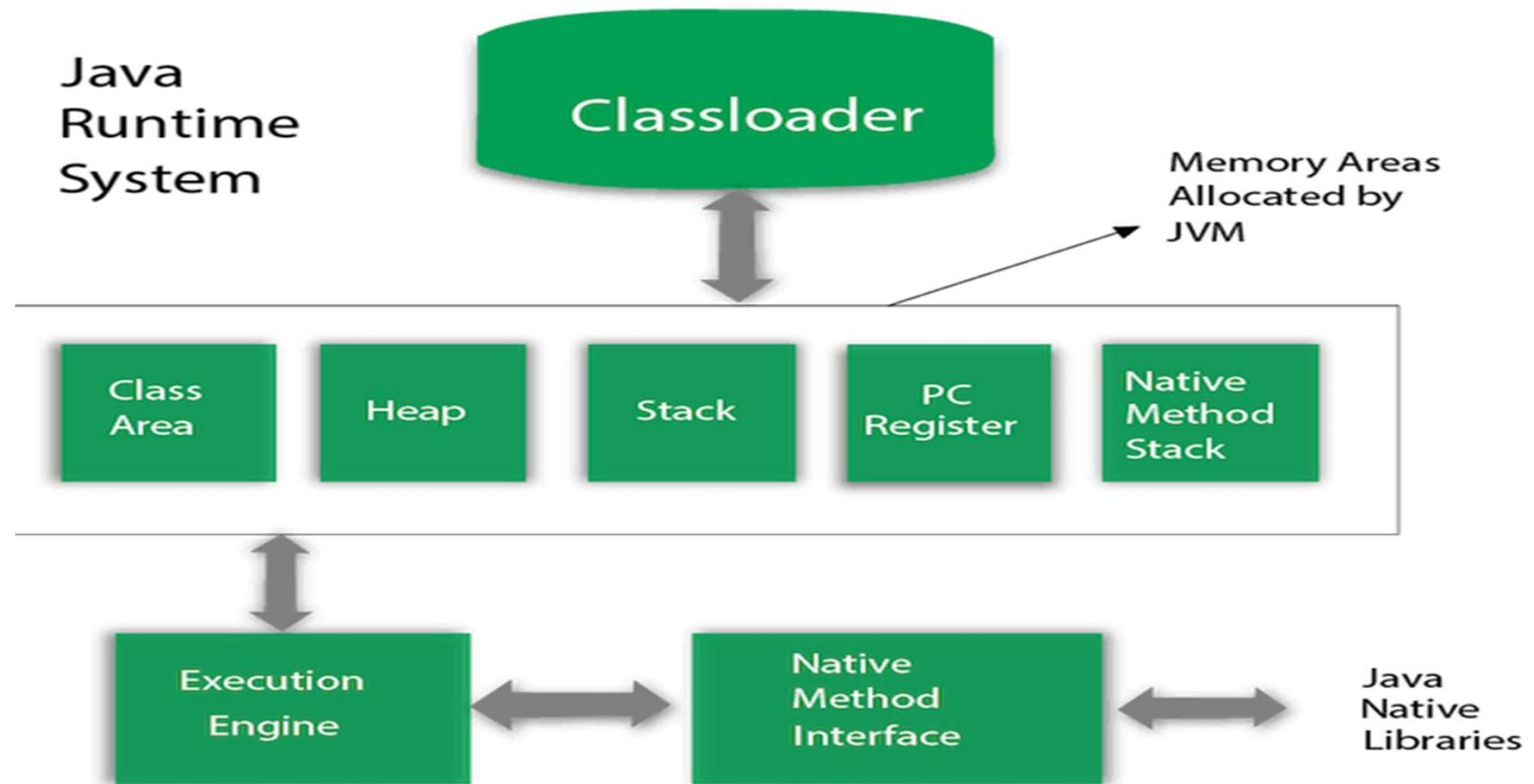
- VM (Java Virtual Machine) is an abstract machine.
- It is a specification that provides runtime environment in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What it does

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Architecture



By Jagadish Sahoo

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.

- **Bootstrap ClassLoader:** This is the first classloader which is the super class of Extension classloader. It loads the *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.
- **Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside `$JAVA_HOME/jre/lib/ext` directory.
- **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. It is also known as Application classloader.

- **Class(Method) Area**

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

- **Heap**

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

- **Stack**

1. Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.
2. Each thread has a private JVM stack, created at the same time as thread.
3. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

- **Program Counter Register**

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

- **Native Method Stack**

It contains all the native methods used in the application.

- **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.

- **Java Native Interface**

1. Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.
2. Java uses JNI framework to send output to the Console or interact with OS libraries.

Java Variables

- A variable is a container which holds the value while the [Java program](#) is executed. A variable is assigned with a data type.
- Variable is name of reserved area allocated in memory.
- Variable is a name of memory location. There are three types of variables in java: local, instance and static.
- There are two types of [data types in Java](#):
 1. primitive
 2. non-primitive.

Beginning Java programming

The process of Java programming can be simplified in three steps:

- Create the program by typing it into a text editor and saving it to a file
 - HelloWorld.java.
- Compile it by typing “javac HelloWorld.java” in the terminal window.
- Execute (or run) it by typing “java HelloWorld” in the terminal window.

```
/* This is a simple Java program.  
FileName : "HelloWorld.java". */  
class HelloWorld  
{  
    // Your program begins with a call to main().  
    // Prints "Hello, World" to the terminal window.  
    public static void main(String args[])  
    {  
        System.out.println("Hello, World");  
    }  
}
```

Class definition: This line uses the keyword **class** to declare that a new class is being defined.

```
class HelloWorld
```

HelloWorld is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace { and the closing curly brace } .

main method: In Java programming language, every application must contain a main method whose signature is:

```
public static void main(String[] args)
```

public: So that JVM can execute the method from anywhere.

static: Main method is to be called without object.

The modifiers public and static can be written in either order.

void: The main method doesn't return anything.

main(): Name configured in the JVM.

String[]: The main method accepts a single argument:

an array of elements of type String.

```
System.out.println("Hello, World");
```

This line outputs the string “Hello, World” followed by a new line on the screen. Output is actually accomplished by the built-in `println()` method. **System** is a predefined class that provides access to the system, and **out** is the variable of type output stream that is connected to the console.

Comments: They can either be multi-line or single line comments.

```
/* This is a simple Java program.  
Call this file "HelloWorld.java". */
```

This is a multiline comment. This type of comment must begin with `/*` and end with `*/`. For single line you may directly use `//` as in C/C++.

Important Points

- The name of the class defined by the program is HelloWorld, which is same as name of file(HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class and there is at most one public class which contain main() method.
- By convention, the name of the main class(class which contain main method) should match the name of the file that holds the program.

Compiling the program

After successfully setting up the environment, we can open terminal in both Windows/Unix and can go to directory where the file – HelloWorld.java is present.

Now, to compile the HelloWorld program, execute the compiler – javac , specifying the name of the source file on the command line, as shown:

```
javac HelloWorld.java
```

The compiler creates a file called HelloWorld.class (in present working directory) that contains the bytecode version of the program. Now, to execute our program, JVM(Java Virtual Machine) needs to be called using java, specifying the name of the class file on the command line, as shown:
`java HelloWorld`

This will print “Hello World” to the terminal screen.

Java Identifiers

- In programming languages, identifiers are used for identification purposes. In Java, an identifier can be a class name, method name, variable name, or label.

Rules for defining Java Identifiers

- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), ‘\$’(dollar sign) and ‘_’ (underscore).For example “roll@” is not a valid java identifier as it contain ‘@’ special character.
- Identifiers should **not** start with digits([0-9]). For example “123name” is a not a valid java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- **Reserved Words** can’t be used as an identifier. For example “int while = 20;” is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Reserved Words

- Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words.
- They can be briefly categorized into two parts : **keywords(50)** and **literals(3)**.
- Keywords define functionalities and literals define a value.

Data types in Java

There are majorly two types of languages.

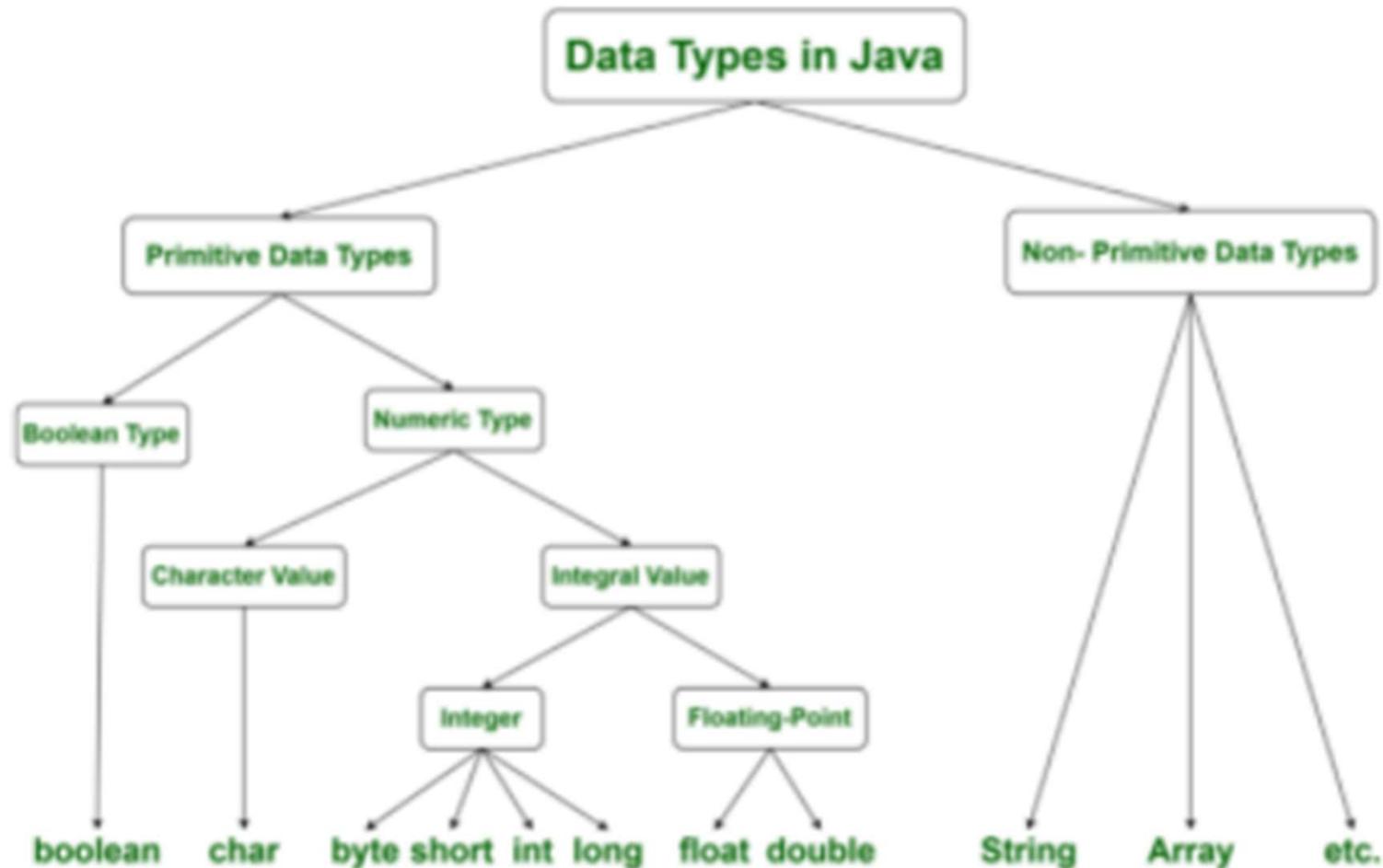
- **Statically typed language** where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types.

Example: C, C++, Java.

- The other is **Dynamically typed languages**. These languages can receive different data types over time.

Example: Ruby, Python

- Java is **statically typed** and also a **strongly typed language** because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.



Primitive Data Type

- Primitive data are only single values and have no special capabilities.

Type	Description	Default	Size	Example Literals	Range of Values
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\u0011', '\u0011', '\u0011', '\u0011'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

By Jagadish Sahoo

boolean

- boolean data type represents only one bit of information **either true or false**, but the size of the boolean data type is **virtual machine-dependent**.
- Values of type boolean are not converted implicitly or explicitly (with casts) to any other type.
- **Syntax :** boolean booleanvariable;
- **Size :** Virtual Machine dependent
- **Values:** true or false
- **Default Value:** false

byte

- The byte data type is an 8-bit signed integer. The byte data type is useful for saving memory in large arrays.
- **Syntax:** byte variablename;
- **Size:** 1 byte(8 bits)
- **Values:** -128 to +127
- **Default Value:** 0

short

- The short data type is a 16-bit signed integer.
- Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.
- **Syntax:** short variablename;
- **Size:** 2 bytes(16 bits)
- **Values:** -32, 768 to 32, 767 (inclusive)
- **Default Value:** 0

`int`

- It is a 32-bit signed integer.
- **Syntax:** `int variableName;`
- **Size:** 4 byte (32 bits)
- **Values:** -2, 147, 483, 648 to 2, 147, 483, 647 (inclusive)
- **Default Value:** 0

long

- The long data type is a 64-bit signed integer.
- **Syntax:** long variablename;
- **Size:** 8 byte (64 bits)
- **Values:** -9, 223, 372, 036, 854, 775, 808

to

9, 223, 372, 036, 854, 775, 807(inclusive)

Default Value: 0

float

- The float data type is a 32-bit floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers.
- **Syntax:** float variablename;
- **Size:** 4 byte (32 bits)
- **Values:** upto 7 decimal digits
- **Default Value:** 0.0

double

- The double data type is a 64-bit floating-point. For decimal values, this data type is generally the default choice.
- **Syntax:** double variablename;
- **Size:** 8 byte (64 bits)
- **Values:** upto 16 decimal digits
- **Default Value:** 0.0

char

- The char data type is a single 16-bit Unicode character.
- **Syntax:** char variablename;
- **Size:** 2 byte (16 bits)
- **Values:** '\u0000' (0) to '\uffff' (65535)
- **Default Value:** '\u0000'
- Unicode defines a fully international character set that can represent most of the world's written languages. It is a unification of dozens of character sets, such as Latin, Greeks, Cyrillic, Katakana, Arabic, and many more.

Why is the size of char is 2 byte in java..?

- In other languages like C/C++ uses only ASCII characters and to represent all ASCII characters 8-bits is enough, But java uses the Unicode system not the ASCII code system and to represent Unicode system 8 bit is not enough to represent all characters so java uses 2 bytes for characters.

Operators

- Arithmetic Operators
- Unary Operators
- Assignment Operator
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators
- Shift Operators
- instance of operator

Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

* : Multiplication

/ : Division

% : Modulo

+ : Addition

- : Subtraction

Unary Operators

Unary operators need only one operand. They are used to increment, decrement or negate a value.

- **- :Unary minus**, used for negating the values.
- **+ :Unary plus**, indicates positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is byte, char, or short. This is called unary numeric promotion.
- **++ :Increment operator**, used for incrementing the value by 1.
- **-- : Decrement operator**, used for decrementing the value by 1.

There are two varieties of increment operator.

- **Post-Increment** : Value is first used for computing the result and then incremented.
- **Pre-Increment** : Value is incremented first and then result is computed.

There are two varieties of decrement operator.

- **Post-decrement** : Value is first used for computing the result and then decremented.
 - **Pre-Decrement** : Value is decremented first and then result is computed.
-
- **! : Logical not operator**, used for inverting a boolean value.

Assignment Operator

- '=' Assignment operator is used to assign a value to any variable.
- It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.
- Syntax :

variable = value;

- In many cases assignment operator can be combined with other operators to build a shorter version of statement called Compound Statement.
- For example, instead of $a = a + 5$, we can write $a += 5$.
- $+=$, for adding left operand with right operand and then assigning it to variable on the left.
- $-=$, for subtracting left operand with right operand and then assigning it to variable on the left.
- $*=$, for multiplying left operand with right operand and then assigning it to variable on the left.
- $/=$, for dividing left operand with right operand and then assigning it to variable on the left.
- $\%=$, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

Relational Operators

- These operators are used to check for relations like equality, greater than, less than.
- They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements.
- Syntax :

variable relation_operator value

- **==, Equal to** : returns true if left hand side is equal to right hand side.
- **!=, Not Equal to** : returns true if left hand side is not equal to right hand side.
- **<, less than** : returns true if left hand side is less than right hand side.
- **<=, less than or equal to** : returns true if left hand side is less than or equal to right hand side.
- **>, Greater than** : returns true if left hand side is greater than right hand side.
- **>=, Greater than or equal to**: returns true if left hand side is greater than or equal to right hand side.

Logical Operators

- These operators are used to perform “logical AND” and “logical OR” operation, i.e. the function similar to AND gate and OR gate in digital electronics.
- The second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision.
- Conditional operators are-
 1. **&&, Logical AND** : returns true when both conditions are true.
 2. **||, Logical OR** : returns true if at least one condition is true.

Ternary operator

- Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary.
- Syntax :

condition ? if true : if false

```
public class operators {  
    public static void main(String[] args)  
    {  
        int a = 20, b = 10, c = 30, result;  
  
        // result holds max of three  
        // numbers  
        result = ((a > b)? (a > c)? a:c (b > c)? b: c);  
        System.out.println("Max of three numbers = "  
                           + result);  
    }  
}
```

Bitwise Operators

These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.

Shift Operators

- These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively.
- They can be used when we have to multiply or divide a number by two.
- Syntax

```
number shift_op number_of_places_to_shift;
```

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

instance of operator

- Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface.
- Syntax :

object instance of class/subclass/interface

```
class operators {
    public static void main(String[] args)
    {
        Person obj1 = new Person();
        Person obj2 = new Boy();
        System.out.println("obj1 instanceof Person: "
            + (obj1 instanceof Person));
        System.out.println("obj1 instanceof Boy: "
            + (obj1 instanceof Boy));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Person: "
            + (obj2 instanceof Person));
        System.out.println("obj2 instanceof Boy: "
            + (obj2 instanceof Boy));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}
```

```
class Person {  
}
```

```
class Boy extends Person implements MyInterface {  
}
```

```
interface MyInterface {  
}
```

Non-Primitive Data Type or Reference Data Types

- The Reference Data Types will contain a memory address of variable value because the reference types won't store the variable value directly in memory.
- Example: strings, objects, arrays, etc.
- String: Strings are defined as an array of characters. The difference between a character array and a string in Java is, the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char type entities.
- Unlike C/C++, Java strings are not terminated with a null character.

- Syntax :

```
<String_Type> <string_variable> = “<sequence_of_string>”;
```

Example :

```
// Declare String without using new operator
```

```
String s = “Jagadish”;
```

```
// Declare String using new operator
```

```
String s1 = new String(“Jagadish”);
```

- **Class:** A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:
- **Modifiers:** A class can be public or has default access
- **Class name:** The name should begin with a initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

- Object: It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :
 1. State: It is represented by attributes of an object. It also reflects the properties of an object.
 2. Behavior: It is represented by methods of an object. It also reflects the response of an object with other objects.
 3. Identity: It gives a unique name to an object and enables one object to interact with other objects.

- Interface: Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, nobody).
- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

- **Array:** An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays. In Java, all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using function `length`. This is different from C/C++ where we find length using `size`.
- A Java array variable can also be declared like other variables with `[]` after the data type.
- The variables in the array are ordered and each has an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is [Object](#).

Ways to read input from console in Java

- In Java, there are four different ways for reading input from the user in the command line environment(console).

- 1.Using Buffered Reader Class
2. Using Scanner Class
3. Using Console Class
4. Using Command line argument

Using Buffered Reader Class

- This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.
- The input is buffered for efficient reading.
- The wrapping code is hard to remember.
- To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test {
    public static void main(String[] args) throws IOException
    {
        // Enter data using BufferedReader
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        // Reading data using readLine
        String name = reader.readLine();

        // Printing the read line
        System.out.println(name);
    }
}
```

By Jagadish Sahoo

Using Scanner Class

- This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however, it is also can be used to read input from the user in the command line.
- Convenient methods for parsing primitives (`nextInt()`, `nextFloat()`, ...) from the tokenized input.
- Regular expressions can be used to find tokens.

```
import java.util.Scanner;
class GetInputFromUser {
    public static void main(String args[])
    {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);
        String s = in.nextLine();
        System.out.println("You entered string " + s);
        int a = in.nextInt();
        System.out.println("You entered integer " + a);
        float b = in.nextFloat();
        System.out.println("You entered float " + b);
        // closing scanner
        in.close();
    }
}
```

By Jagadish Sahoo

Difference between Scanner and BufferedReader Class in Java

java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

Issue with Scanner when nextLine() is used after nextXXX()

```
import java.util.Scanner;
class Differ
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter an integer");
        int a = scn.nextInt();
        System.out.println("Enter a String");
        String b = scn.nextLine();
        System.out.printf("You have entered:- "
            + a + " " + "and name as " + b);
    }
}
```

```
import java.io.*;
class Differ
{
    public static void main(String args[])
        throws IOException
    {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Enter an integer");
        int a = Integer.parseInt(br.readLine());
        System.out.println("Enter a String");
        String b = br.readLine();
        System.out.printf("You have entered:- " + a +
            " and name as " + b);
    }
}
```

By Jagadish Sahoo

- In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() does not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().
- In BufferedReader class there is no such type of problem. This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads till first newline character. If we use one more call of nextLine() method between nextXXX() and nextLine(), then this problem will not occur because nextLine() will consume the newline character. This problem is same as scanf() followed by gets() in C/C++.

- This problem can also be solved by using next() instead of nextLine() for taking input of strings

```
import java.util.Scanner;
class Differ {
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter an integer");
        int a = scn.nextInt();
        scn.nextLine();
        System.out.println("Enter a String");
        String b = scn.next();
        String c = scn.next();
        int d = scn.nextInt();
        System.out.printf("You have entered:- " + a + " " + "and name as " + b + " and " + c + d);
    }
}
```

By Jagadish Sahoo

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- BufferedReader has significantly larger buffer memory than Scanner.
- The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

Using Console Class

- It has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like `System.out.printf()`).
- Reading password without echoing the entered characters.
- Format string syntax can be used.
- Does not work in non-interactive environment (such as in an IDE)

```
public class Sample {  
    public static void main(String[] args)  
    {  
        // Using Console to input data from user  
        String name = System.console().readLine();  
  
        System.out.println("You entered string " + name);  
    }  
}
```

Using Command line argument

- Most used user input for competitive coding. The command-line arguments are stored in the String format. The parseInt method of the Integer class converts string argument into Integer.
- Similarly, for float and others during execution.
- The usage of args[] comes into existence in this input form. The passing of information takes place during the program run.
- The command line is given to args[]. These programs have to be run on cmd.

```
class Hello {  
    public static void main(String[] args)  
    {  
        // check if length of args array is  
        // greater than 0  
        if (args.length > 0) {  
            System.out.println(  
                "The command line arguments are:");  
  
            // iterating the args array and printing  
            // the command line arguments  
            for (String val : args)  
                System.out.println(val);  
        }  
        else  
            System.out.println("No command line "  
                + "arguments found.");  
    }  
}
```

By Jagadish Sahoo

```
// Java program to demonstrate working of split(regex,
// limit) with small limit.
public class GFG {
    public static void main(String args[])
    {
        String str = "jagadish@kumar@sahoo";
        String[] arrOfStr = str.split("@", 2);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

By Jagadish Sahoo

Flow Control

- **Decision Making in Java**
- Decision Making in programming is similar to decision making in real life. In programming also we face some situations where we want a certain block of code to be executed when some condition is fulfilled.
- A programming language uses control statements to control the flow of execution of program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

if: if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
    // Statements to execute if
    // condition is true
}
```

- Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.
- If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the immediate one statement to be inside its block.
- Example :

```
if(condition)
    statement1;
    statement2;
```

- if-else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

- nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements means an if statement inside an if statement. Java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}
```

- if-else-if ladder: Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
if (condition1)
```

```
    statement;
```

```
else if (condition2)
```

```
    statement;
```

```
.
```

```
.
```

```
else
```

```
    statement;
```

- switch-case The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
switch (expression)
```

```
{
```

```
    case value1:
```

```
        statement1;
```

```
        break;
```

```
    case value2:
```

```
        statement2;
```

```
        break;
```

```
.
```

```
.
```

```
    case valueN:
```

```
        statementN;
```

```
        break;
```

```
    default:
```

```
        statementDefault;
```

```
}
```

- Expression can be of type byte, short, int char or an enumeration. Beginning with JDK7, expression can also be of type String.
- Duplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.

- jump: Java supports three jump statement: break, continue and return. These three statements transfer control to other part of the program.
- Break: In Java, break is majorly used for:
 1. Terminate a sequence in a switch statement
 2. To exit a loop.
 3. Used as a “civilized” form of goto.

Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

Note: Break, when used inside a set of nested loops, will only break out of the innermost loop.

```
class BreakLoopDemo
{
    public static void main(String args[])
    {
        // Initially loop is set to run from 0-9
        for (int i = 0; i < 10; i++)
        {
            // terminate loop when i is 5.
            if (i == 5)
                break;

            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

By Jagadish Sahoo

- Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. Java uses label. A Label is used to identify a block of code.

label:

```
{  
    statement1;  
    statement2;  
    statement3;  
    .  
    .  
}
```

- break statement can be used to jump out of target block.
- Note: You cannot break to any label which is not defined for an enclosing block.

break label;

Continue

- Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

```
class ContinueDemo
{
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            // If the number is even
            // skip and continue
            if (i%2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```

By Jagadish Sahoo

Return

- The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.
- class Return
- {
- public static void main(String args[])
- {
- boolean t = true;
- System.out.println("Before the return.");
-
- if (t)
- return;
-
- // Compiler will bypass every statement
- // after return
- System.out.println("This won't execute.");
- }
- }

Loops in Java

- Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.
- Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.
- while loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

```
while (boolean condition)
```

```
{
```

```
    loop statements...
```

```
}
```

- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called Entry control loop
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

for loop

- for loop: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

for (initialization condition; testing condition;

increment/decrement)

{

statement(s)

}

- Initialization condition: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- Testing Condition: It is used for testing the exit condition for a loop. It must return a boolean value. It is also an Entry Control Loop as the condition is checked prior to the execution of the loop statements.
- Statement execution: Once the condition is evaluated to true, the statements in the loop body are executed.
- Increment/ Decrement: It is used for updating the variable for next iteration.
- Loop termination: When the condition becomes false, the loop terminates marking the end of its life cycle.

Enhanced For loop

- Java also includes another version of for loop introduced in Java 5. Enhanced for loop provides a simpler way to iterate through the elements of a collection or array. It is inflexible and should be used only when there is a need to iterate through the elements in sequential manner without knowing the index of currently processed element.
- The object/variable is immutable when enhanced for loop is used i.e it ensures that the values in the array can not be modified, so it can be said as read only loop where you can't update the values as opposite to other loops where values can be modified.

```
for (T element:Collection obj/array)
{
    statement(s)
}
```

```
public class enhancedforloop
{
    public static void main(String args[])
    {
        String array[] = {"Ron", "Harry", "Hermoine"};

        //enhanced for loop
        for (String x:array)
        {
            System.out.println(x);
        }

        /* for loop for same function
        for (int i = 0; i < array.length; i++)
        {
            System.out.println(array[i]);
        }
        */
    }
}
```

By Jagadish Sahoo

do-while

- do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

```
do
{
    statements..
}
while (condition);
```

- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

Pitfalls of Loops

- Infinite loop: One of the most common mistakes while implementing any sort of looping is that it may not ever exit, that is the loop runs for infinite time. This happens when the condition fails for some reason.

```
for (int i = 5; i != 0; i -= 2)
```

```
{
```

```
    System.out.println(i);
```

```
}
```

- Another pitfall is that you might be adding something into your collection object through loop and you can run out of memory. If you try and execute the below program, after some time, out of memory exception will be thrown.

```
import java.util.ArrayList;
public class Integer1
{
    public static void main(String[] args)
    {
        ArrayList<Integer> ar = new ArrayList<>();
        for (int i = 0; i < Integer.MAX_VALUE; i++)
        {
            ar.add(i);
        }
    }
}
```

Arrays in Java

- An array is a group of like-typed variables that are referred to by a common name.
- Arrays in Java work differently than they do in C/C++.
- Following are some important points about Java arrays.

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int or short value and not long.
- The direct superclass of an array type is Object.

- Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array.
- In case of primitive data types, the actual values are stored in contiguous memory locations.

One-Dimensional Arrays :

The general form of a one-dimensional array declaration is

`type var-name[];`

OR

`type[] var-name;`

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

```
int intArray[]; or int[] intArray;  
  
byte byteArray[];  
short shortsArray[];  
boolean booleanArray[];  
long longArray[];  
float floatArray[];  
double doubleArray[];  
char charArray[];  
  
// an array of references to objects of  
// the class MyClass (a class created by  
// user)  
MyClass myClassArray[];  
  
Object[] ao, // array of Object
```

By Jagadish Sahoo

- Although the first declaration above establishes the fact that intArray is an array variable, no actual array exists. It merely tells the compiler that this variable (intArray) will hold an array of the integer type. To link intArray with an actual, physical array of integers, you must allocate one using new and assign it to intArray.

Instantiating an Array in Java

- When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array in the following manner

The general form of new as it applies to one-dimensional arrays appears as follows:

var-name = new type [size];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array. That is, to use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

```
int intArray[]; //declaring array
```

```
intArray = new int[20]; // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types).

Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Array Literal

- In a situation, where the size of the array and variables of array are already known, array literals can be used.

```
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java

Accessing Java Array Elements using for Loop

- Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
for (int i = 0; i < arr.length; i++)  
    System.out.println("Element at index " + i + " : " + arr[i]);
```

Arrays of Objects

- An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] arr = new Student[7]; //student is a user-defined class
```

The studentArray contains seven memory spaces each of size of student class in which the address of seven Student objects can be stored. The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements in the following way.

```
class Student
{
    public int roll_no;
    public String name;
    Student(int roll_no, String name)
    {
        this.roll_no = roll_no;
        this.name = name;
    }
}

public class GFG
{
    public static void main (String[] args)
    {
        Student[] arr;
        arr = new Student[5];
        arr[0] = new Student(1,"aman");
        arr[1] = new Student(2,"vaibhav");
        arr[2] = new Student(3,"shikar");
        arr[3] = new Student(4,"dharmesh");
    }
}
```

By Jagadish Sahoo

```
for (int i = 0; i < arr.length; i++)  
    System.out.println("Element at " + i + " : " +  
        arr[i].roll_no +" "+ arr[i].name);  
}  
}
```

What happens if we try to access element outside the array size?

- JVM throws **ArrayIndexOutOfBoundsException** to indicate that array has been accessed with an illegal index. The index is either negative or greater than or equal to size of array.

```
class GFG
```

```
{  
    public static void main (String[] args)  
    {  
        int[] arr = new int[2];  
        arr[0] = 10;  
        arr[1] = 20;  
  
        for (int i = 0; i <= arr.length; i++)  
            System.out.println(arr[i]);  
    }  
}
```

Multidimensional Arrays

- Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array.
- These are also known as Jagged Arrays.
- A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Example

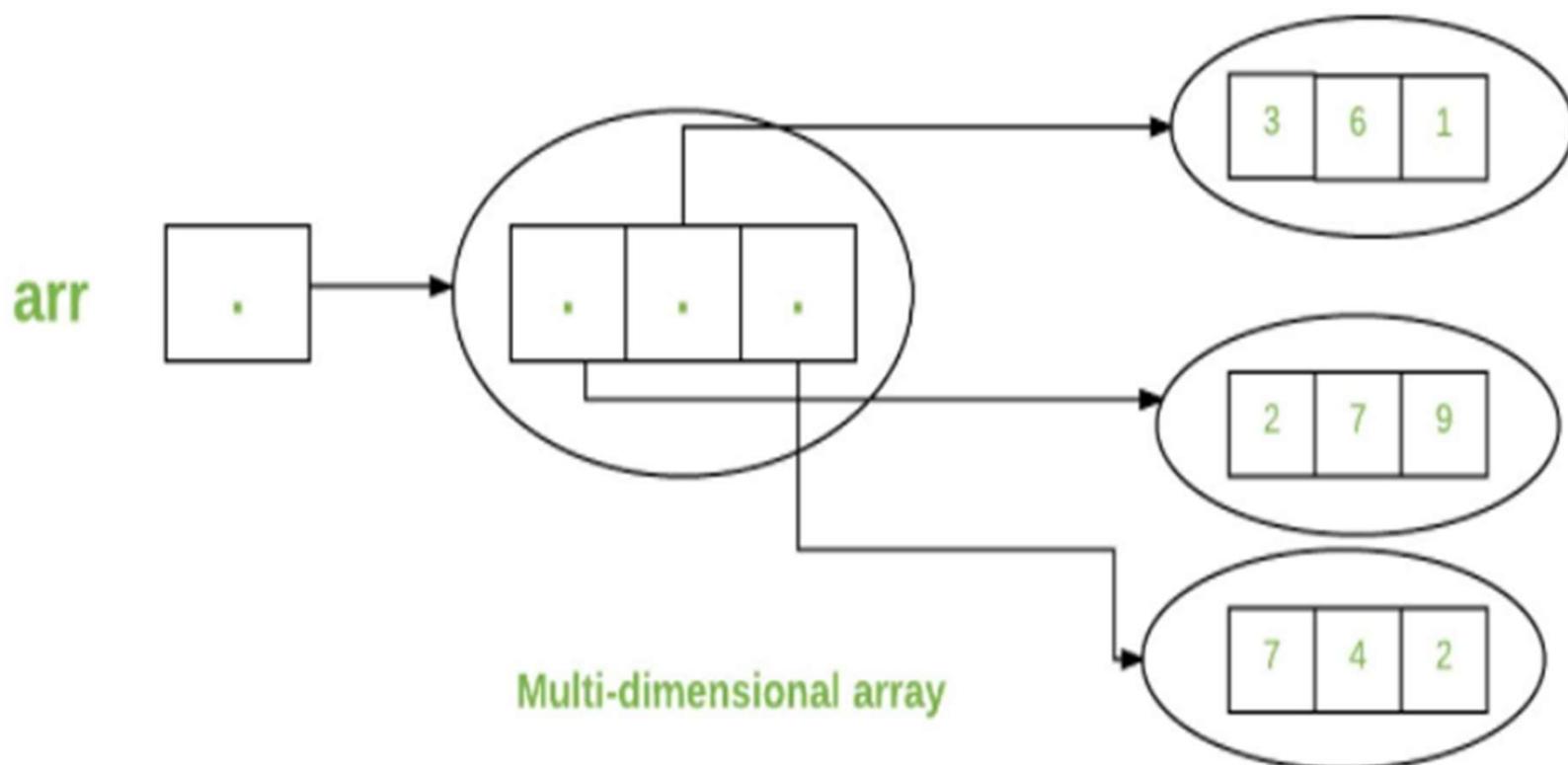
```
int[][] intArray = new int[10][20]; //a 2D array or matrix  
int[][][] intArray = new int[10][20][10]; //a 3D array
```

```
class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

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

By Jagadish Sahoo



By Jagadish Sahoo

Passing Arrays to Methods

- Like variables, we can also pass arrays to methods. For example, below program pass array to method *sum* for calculating sum of array's values.

```
class Test
{
public static void main(String args[])
{
    int arr[] = {3, 1, 2, 5, 4};
    sum(arr);
}

public static void sum(int[] arr)
{
    int sum = 0;
    for (int i = 0; i < arr.length; i++)
        sum+=arr[i];
    System.out.println("sum of array values : " + sum);
}
}
```

By Jagadish Sahoo

Returning Arrays from Methods

- As usual, a method can also return an array. For example, below program returns an array from method *m1*.

```
class Test
{
    public static void main(String args[])
    {
        int arr[] = m1();
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i]+" ");
    }
    public static int[] m1()
    {
        return new int[]{1,2,3};
    }
}
```

By Jagadish Sahoo

Class Objects for Arrays

- Every array has an associated Class object, shared with all other arrays with the same component type.

```
class Test
{
    public static void main(String args[])
    {
        int intArray[] = new int[3];
        byte byteArray[] = new byte[3];
        short shortsArray[] = new short[3];

        // array of Strings
        String[] strArray = new String[3];

        System.out.println(intArray.getClass());
        System.out.println(intArray.getClass().getSuperclass());
        System.out.println(byteArray.getClass());
        System.out.println(shortsArray.getClass());
        System.out.println(strArray.getClass());
    }
}
```

By Jagadish Sahoo

Output

```
class [I  
class java.lang.Object  
class [B  
class [S  
class [Ljava.lang.String;
```

- The string “[I” is the run-time type signature for the class object “array with component type int”.
- The only direct superclass of any array type is java.lang.Object.
- The string “[B” is the run-time type signature for the class object “array with component type byte”.
- The string “[S” is the run-time type signature for the class object “array with component type short”.
- The string “[L” is the run-time type signature for the class object “array with component type of a Class”. The Class name is then followed.

Array Members

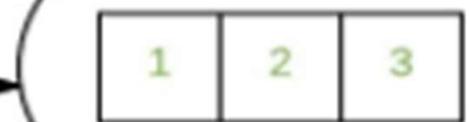
- arrays are object of a class and direct superclass of arrays is class `Object`.The members of an array type are all of the following:
 1. The public final field `length`, which contains the number of components of the array. `length` may be positive or zero.
 2. All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method.
 3. The public method `clone()`, which overrides `clone` method in class `Object` and throws no checked exceptions.

Cloning of arrays

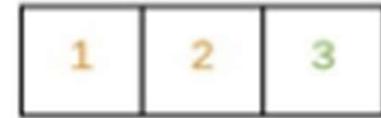
- When you clone a single dimensional array, such as Object[], a “deep copy” is performed with the new array containing copies of the original array’s elements as opposed to references.

```
class Test
{
    public static void main(String args[])
    {
        int intArray[] = {1,2,3};
        int cloneArray[] = intArray.clone();
        // will print false as deep copy is created
        // for one-dimensional array
        System.out.println(intArray == cloneArray);
        for (int i = 0; i < cloneArray.length; i++) {
            System.out.print(cloneArray[i] + " ");
        }
    }
}
```

intArray



cloneArray

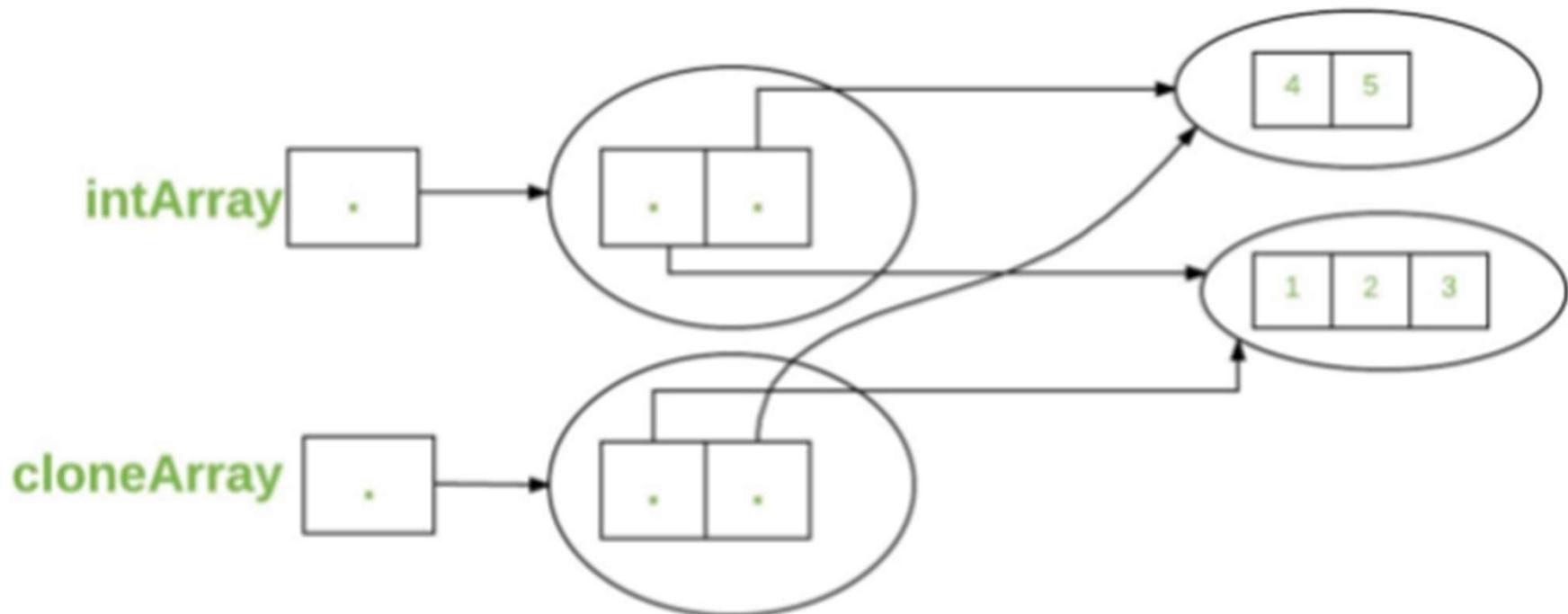


**Deep Copy is created for
one-dimensional array by `clone()`
method**

By Jagadish Sahoo

- A clone of a multi-dimensional array (like Object[][][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array, but subarrays are shared.

```
class Test
{
    public static void main(String args[])
    {
        int intArray[][] = {{1,2,3},{4,5}};
        int cloneArray[][] = intArray.clone();
        // will print false
        System.out.println(intArray == cloneArray);
        // will print true as shallow copy is created
        // i.e. sub-arrays are shared
        System.out.println(intArray[0] == cloneArray[0]);
        System.out.println(intArray[1] == cloneArray[1]);
    }
}
```



**Shallow Copy is created for
multi-dimensional array by `clone()`
method**

Classes and Objects in Java

- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:
- **Modifiers:** A class can be public or has default access.
- **class keyword:** class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

Access specifiers for classes or interfaces in Java

- In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.
 - 1) private (accessible within the class where defined)
 - 2) default or package private (when no access specifier is specified)
 - 3) protected
 - 4) public (accessible from any class)

The classes and interfaces themselves can have only two access specifiers when declared outside any other class.

- 1) Public
- 2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
protected class Test {}
```

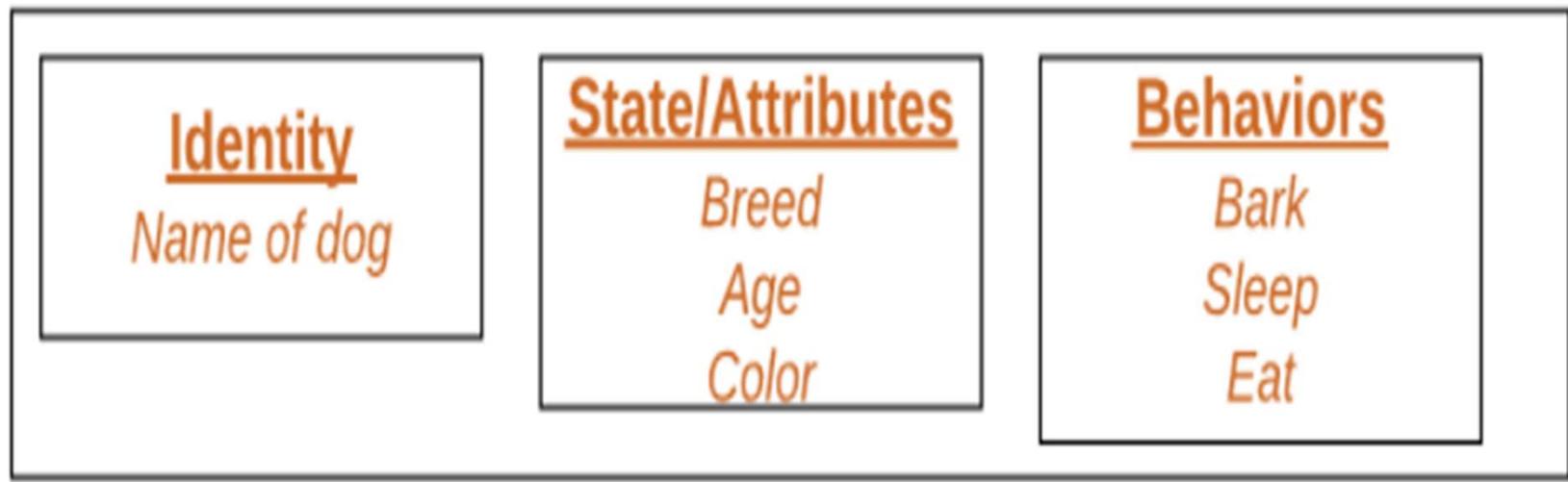
```
public class Main {  
    public static void main(String args[]) {  
    }  
}
```

Note : Nested interfaces and classes can have all access specifiers.

Object

- It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, interact by invoking methods. An object consists of :
- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

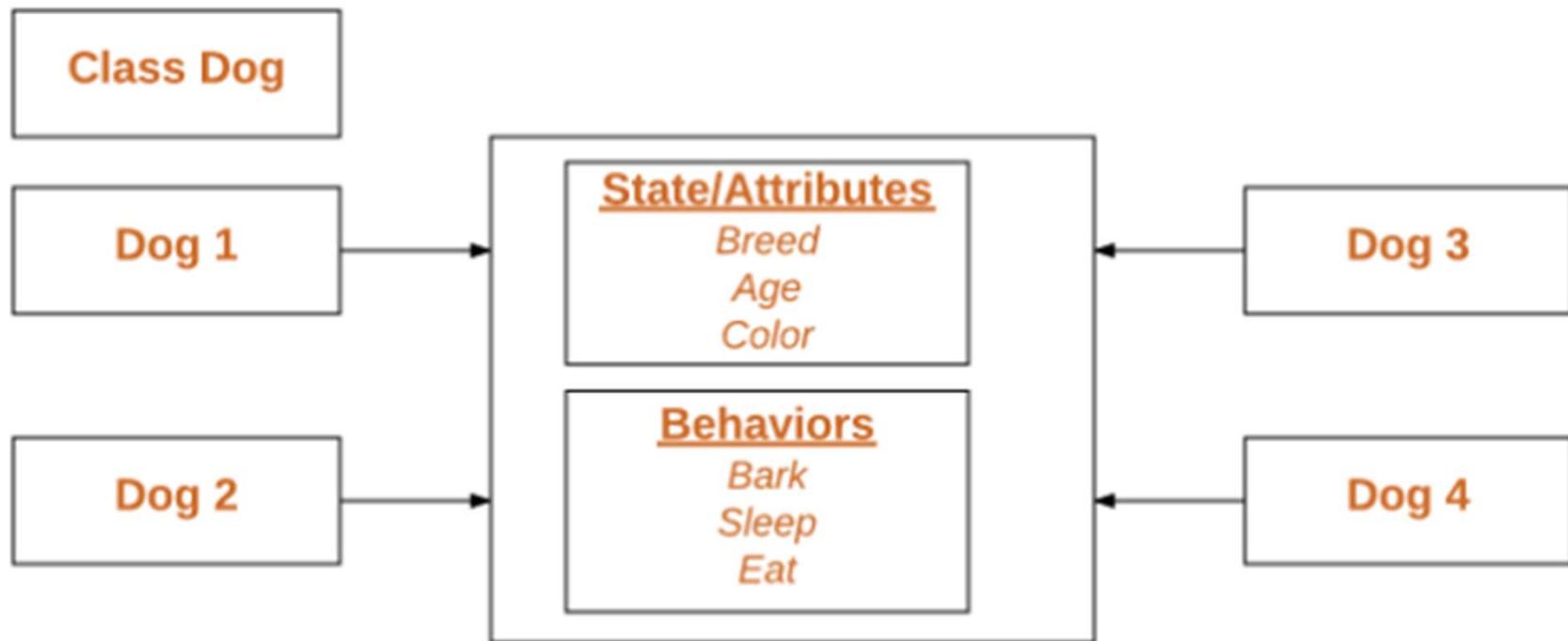
Example of an object: dog



- Objects correspond to things found in the real world.
- For example, a graphics program may have objects such as “circle”, “square”, “menu”.
- An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances.



By Jagadish Sahoo

- As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type.
- With a primitive variable, this declaration also reserves the proper amount of memory for the variable.
- So for reference variable, type must be strictly a concrete class name.
- In general, we can't create objects of an abstract class or an interface.

Example

Dog tuffy;

- if we declare reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing an object

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed, int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
}
```

By Jagadish Sahoo

```
public String getName()
{
    return name;
}
public String getBreed()
{
    return breed;
}
public int getAge()
{
    return age;
}
public String getColor()
{
    return color;
}
```

By Jagadish Sahoo

```
public String toString()
{
    return("Hi my name is "+ this.getName()+
        ".\nMy breed,age and color are " +
        this.getBreed()+"," + this.getAge()+
        ","+ this.getColor());
}

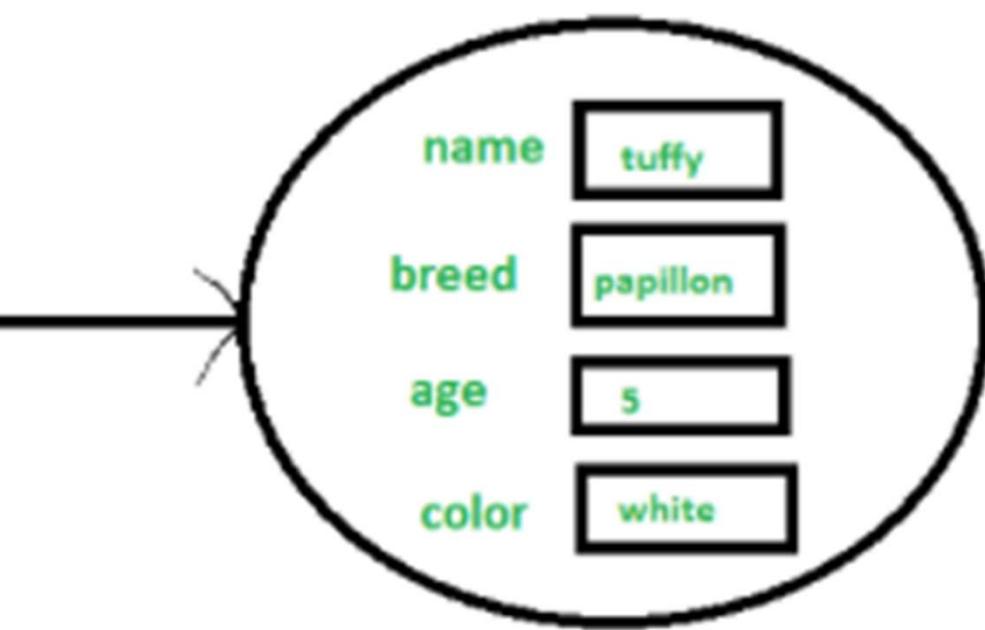
public static void main(String[] args)
{
    Dog tuffy = new Dog("tuffy","papillon", 5, "white");
    System.out.println(tuffy.toString());
}
```

- This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiates the constructors based on the number and the type of the arguments. The constructor in the Dog class takes four arguments. The following statement provides “tuffy”, “papillon”, 5, “white” as values for those arguments:

```
Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
```

Tuffy

an address(for
eg 100)



By Jagadish Sahoo

- All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor.
- This default constructor calls the class parent's no-argument constructor (as it contain only one statement i.e super());, or the Object class constructor if the class has no other parent (as Object class is parent of all classes either directly or indirectly).

Ways to create object of a class

- There are four ways to create objects in java.
- Strictly speaking there is only one way(by using new keyword),and the rest internally use new keyword.

Using new keyword

- It is the most common and general way to create object in java.

Example:

```
// creating object of class Test
```

```
Test t = new Test();
```

Using Class.forName(String className) method:

- There is a pre-defined class in java.lang package with name Class. The `forName(String className)` method returns the Class object associated with the class with the given string name.
- We have to give the fully qualified name for a class.
- On calling `new Instance()` method on this Class object returns new instance of the class with the given string name.
- // creating object of public class Test
- // consider class Test present in com.p1 package
- `Test obj = (Test)Class.forName("com.p1.Test").newInstance();`

Using clone() method

- `clone()` method is present in `Object` class. It creates and returns a copy of the object.

```
// creating object of class Test
```

```
Test t1 = new Test();
```

```
// creating clone of above object
```

```
Test t2 = (Test)t1.clone();
```

Deserialization

- De-serialization is technique of reading an object from the saved state in a file.

```
FileInputStream file = new FileInputStream(filename);
```

```
ObjectInputStream in = new ObjectInputStream(file);
```

```
Object obj = in.readObject();
```

Creating multiple objects by one type only

- In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java.
- Example:

```
Test test = new Test();
```

```
test = new Test();
```

- In inheritance system, we use parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using same referenced variable. Example:

```
class Animal {}
```

```
class Dog extends Animal {}
```

```
class Cat extends Animal {}
```

```
public class Test
```

```
{
```

```
    // using Dog object
```

```
    Animal obj = new Dog();
```

```
    // using Cat object
```

```
    obj = new Cat();
```

```
}
```

Anonymous objects

- Anonymous objects are the objects that are instantiated but are not stored in a reference variable.
1. They are used for immediate method calling.
 2. They will be destroyed after method calling.
 3. They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).

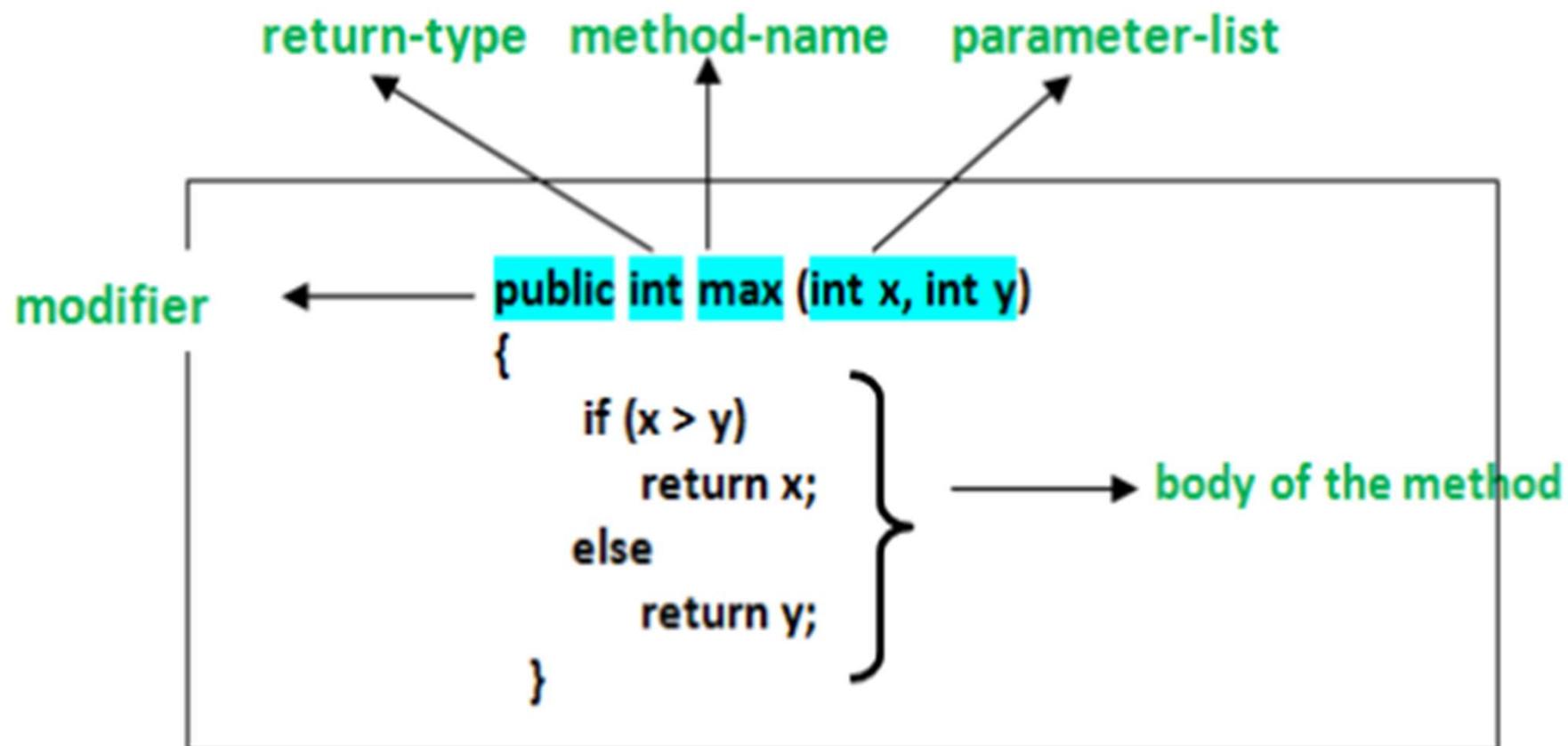
Methods in Java

- A method is a collection of statements that perform some specific task and return the result to the caller.
- A method can perform some specific task without returning anything.
- Methods allow us to reuse the code without retyping the code.
- In Java, every method must be part of some class which is different from languages like C, C++, and Python.

Method Declaration

- In general, method declarations has six components :
- 1. Modifier:- Defines access type of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
- public: accessible in all class in your application.
- protected: accessible within the class in which it is defined and in its subclass(es)
- private: accessible only within the class in which it is defined.
- default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.

2. The return type : The data type of the value returned by the method or void if does not return a value.
3. Method Name : the rules for field names apply to method names as well, but the convention is a little different.
4. Parameter list : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses () .
5. Exception list : The exceptions you expect by the method can throw, you can specify these exception(s).
6. Method body : it is enclosed between braces. The code you need to be executed to perform your intended operations.



- **Method signature:** It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.

max(int x, int y)

How to name a Method?: A method name is typically a single word that should be a **verb** in lowercase or multi-word, that begins with a **verb** in lowercase followed by **adjective, noun.....** After the first word, first letter of each word should be capitalized. For example, findSum, computeMax, setX and getX

Calling a method

- The method needs to be called for using its functionality.
- A method returns to the code that invoked it when:
 1. It completes all the statements in the method
 2. It reaches a return statement
 3. Throws an exception

Call by Value

- Pass By Value: Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as call by value.
- Java in fact is strictly call by value.

```
class CallByValue {  
    public static void Example(int x, int y)  
    {  
        x++;  
        y++;  
    }  
}  
  
public class Main {  
    public static void main(String[] args)  
    {  
        int a = 10;  
        int b = 20;  
        CallByValue object = new CallByValue();  
        System.out.println("Value of a: " + a + " & b: " + b);  
        object.Example(a, b);  
        System.out.println("Value of a: " + a + " & b: " + b);  
    }  
}
```

Call by reference(aliasing)

- Changes made to formal parameter do get transmitted back to the caller through parameter passing.
- Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.
- This method is also called as call by reference. This method is efficient in both time and space.

```
class CallByReference {  
    int a, b;  
    CallByReference(int x, int y)  
    {  
        a = x;  
        b = y;  
    }  
    void ChangeValue(CallByReference obj)  
    {  
        obj.a += 10;  
        obj.b += 20;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args)  
    {  
        CallByReference object      = new CallByReference(10, 20);  
        System.out.println("Value of a: " + object.a + " & b: " + object.b);  
        object.ChangeValue(object);  
        System.out.println("Value of a: " + object.a + " & b: " + object.b);  
    }  
}
```

- when we pass a reference, a new reference variable to the same object is created.
- So we can only change members of the object whose reference is passed.
- We cannot change the reference to refer to some other object as the received reference is a copy of the original reference.

- Methods calls are implemented through stack.
- Whenever a method is called a stack frame is created within the stack area and after that the arguments passed to and the local variables and value to be returned by this calling method are stored in this stack frame and when execution of the called method is finished, the allocated stack frame would be deleted.
- There is a stack pointer register that tracks the top of the stack which is adjusted accordingly.

Returning Multiple values in Java

```
class Test {  
    static int[] getSumAndSub(int a, int b)  
    {  
        int[] ans = new int[2];  
        ans[0] = a + b;  
        ans[1] = a - b;  
        return ans;  
    }  
    public static void main(String[] args)  
    {  
        int[] ans = getSumAndSub(100, 50);  
        System.out.println("Sum = " + ans[0]);  
        System.out.println("Sub = " + ans[1]);  
    }  
}
```

By Jagadish Sahoo

If returned elements are of different types

- Using Pair (If there are only two returned values)

```
import javafx.util.Pair;  
class GfG {  
    public static Pair<Integer, String> getTwo()  
    {  
        return new Pair<Integer, String>(10, "Jagadish Sahoo");  
    }  
    // Return multiple values from a method in Java 8  
    public static void main(String[] args)  
    {  
        Pair<Integer, String> p = getTwo();  
        System.out.println(p.getKey() + " " + p.getValue());  
    }  
}
```

If there are more than two returned values

- We can encapsulate all returned types into a class and then return an object of that class.

```
class MultiDivAdd {  
    int mul; // To store multiplication  
    double div; // To store division  
    int add; // To store addition  
    MultiDivAdd(int m, double d, int a)  
    {  
        mul = m;  
        div = d;  
        add = a;  
    }  
}
```

```
class Test {  
    static MultiDivAdd getMultDivAdd(int a, int b)  
    {  
        // Returning multiple values of different  
        // types by returning an object  
        return new MultiDivAdd(a * b, (double)a / b, (a + b));  
    }  
    public static void main(String[] args)  
    {  
        MultiDivAdd ans = getMultDivAdd(10, 20);  
        System.out.println("Multiplication = " + ans.mul);  
        System.out.println("Division = " + ans.div);  
        System.out.println("Addition = " + ans.add);  
    }  
}
```

By Jagadish Sahoo

Returning list of Object Class

```
import java.util.*;
class GfG {
    public static List<Object> getDetails()
    {
        String name = "Geek";
        int age = 35;
        char gender = 'M';
        return Arrays.asList(name, age, gender);
    }
    public static void main(String[] args)
    {
        List<Object> person = getDetails();
        System.out.println(person);
    }
}
```

By Jagadish Sahoo

Method Overloading in Java

- Java can distinguish the methods with different method signatures.
i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, the order of the parameters, and data types of the parameters) within the same class.
- Overloaded methods are differentiated based on the number and type of the parameters passed as an argument to the methods.
- You can not define more than one method with the same name, Order and the type of the arguments. It would be a compiler error.
- The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error.

Why do we need Method Overloading?

- if we need to do some kind of the operation with different ways i.e. for different inputs.
- For example , we are doing the addition operation for different inputs. It is hard to find many meaningful names for a single action.

Different ways of doing overloading methods

Method overloading can be done by changing:

- The number of parameters in methods.
- The data types of the parameters of methods.
- The Order of the parameters of methods.

Method 1: By changing the number of parameters.

```
import java.io.*;
class Addition {
    public int add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }
    public int add(int a, int b, int c)
    {
        int sum = a + b + c;
        return sum;
    }
}
class GFG {
    public static void main(String[] args)
    {
        Addition ob = new Addition();
        int sum1 = ob.add(1, 2);
        System.out.println("sum of the two integer value :" + sum1);
        int sum2 = ob.add(1, 2, 3);
        System.out.println( "sum of the three integer value :" + sum2);
    }
}
```

Method 2: By changing the Data types of the parameters

```
import java.io.*;
class Addition {
public int add(int a, int b, int c)
{
int sum = a + b + c;
    return sum;
}
public double add(double a, double b, double c)
{
double sum = a + b + c;
    return sum;
}
class GFG {
    public static void main(String[] args)
    {
        Addition ob = new Addition();
        int sum2 = ob.add(1, 2, 3);
        System.out.println("sum of the three integer value :" + sum2);
        double sum3 = ob.add(1.0, 2.0, 3.0);
        System.out.println("sum of the three double value :" + sum3);
    }
}
```

By Jagadish Sahoo

Method 3: By changing the Order of the parameters

```
import java.io.*;
class Para {
    public void getIdentity(String name, int id)
    {
        System.out.println("Name :" + name + " " + "Id :" + id);
    }
    public void getIdentity(int id, String name)
    {
        System.out.println("Id :" + id + " " + "Name :" + name);
    }
}
class GFG {
    public static void main(String[] args)
    {
        Para p = new Para();
        p.getIdentity("Mohit", 1);
        p.getIdentity(2, "shubham");
    }
}
```

By Jagadish Sahoo

What happens when method signature is the same and the return type is different?

- The compiler will give an error as the return value alone is not sufficient for the compiler to figure out which function it has to call.
- Only if both methods have different parameter types (so, they have a different signature), then Method overloading is possible.

```
import java.io.*;
class Addition {
    public int add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }
    public double add(int a, int b)
    {
        double sum = a + b + 0.0;
        return sum;
    }
}
class GFG {
    public static void main(String[] args)
    {
        try {
            Addition ob = new Addition();
            int sum1 = ob.add(1, 2);
            System.out.println("sum of the two integer value :" + sum1);
            int sum2 = ob.add(1, 2);
            System.out.println("sum of the three integer value :" + sum2);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

By Jagadish Sahoo

Scope of Variables In Java

- Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e. scope of a variable can be determined at compile time.
- Java programs are organized in the form of classes. Every class is part of some package.

Scope rules

- Member Variables (Class Level Scope)
- Local Variables (Method Level Scope)
- Loop Variables (Block Scope)

Member Variables (Class Level Scope)

- These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class.

```
public class Test
{
    // All variables defined directly inside a class
    // are member variables
    int a;
    private String b;
    void method1() {....}
    int method2() {....}
    char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't affect scope of them within a class.
- Member variables can be accessed outside a class with following rules

Modifier	Package	Subclass	World
<code>public</code>	Yes	Yes	Yes
<code>protected</code>	Yes	Yes	No
<code>Default (no modifier)</code>	Yes	No	No
<code>private</code>	No	No	No

By Jagadish Sahoo

Local Variables (Method Level Scope)

- Variables declared inside a method have method level scope and can't be accessed outside the method.
- Local variables don't exist after method's execution is over.

```
public class Test
{
    void method1()
    {
        // Local variable (Method level scope)
        int x;
    }
}
```

```
class Test
{
    private int x;
    public void setX(int x)
    {
        this.x = x;
    }
}
```

this keyword to differentiate between the local and class variables.

```
public class Test
{
    static int x = 11;
    private int y = 33;
    public void method1(int x)
    {
        Test t = new Test();
        this.x = 22;
        y = 44;
        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }

    public static void main(String args[])
    {
        Test t = new Test();
        t.method1(5);
    }
}
```

By Jagadish Sahoo

Loop Variables (Block Scope)

- A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

```
public class Test
{
    public static void main(String args[])
    {
        {
            int x = 10;
            System.out.println(x);
        }
        // Uncommenting below line would produce error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

By Jagadish Sahoo

Predict the O/P

```
class Test
{
    public static void main(String args[])
    {
        int a = 5;
        for (int a = 0; a < 5; a++)
        {
            System.out.println(a);
        }
    }
}
```

In C++, it will run. But in java it is an error because in java, the name of the variable of inner and outer loop must be different.

Important Points about Variable scope in Java

- In general, a set of curly brackets { } defines a scope.
- In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
- Any variable defined in a class outside of any method can be used by all member methods.
- When a method has the same local variable as a member, “this” keyword can be used to reference the current class variable.
- For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

Constructors in Java

- Constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.

Need of Constructor

Suppose there is a class with the name Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer's memory), then can a box be there with no value defined for its dimensions. The answer is no.

So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

When is a Constructor called?

- Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members** of the same class.
- A constructor is invoked at the time of object or instance creation. For Example:

```
class Hello
{
    .....
    // A Constructor
    Hello() {}

    .....
}

// We can create an object of the above class
// using the below statement. This statement
// calls above constructor.

Hello obj = new Hello();
```

Rules for writing Constructor:

- Constructor(s) of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of constructor

1. No-argument constructor(Default constructor)

- A constructor that has no parameter is known as the default constructor.
- If we don't define a constructor in a class, then the compiler creates default constructor(with no arguments) for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
- Default constructor provides the default values to the object like 0, null, etc. depending on the type.

```
import java.io.*;
class Myclass
{
    int num;
    String name;
    // this would be invoked while an object
    // of that class is created.
    Myclass()
    {
        System.out.println("Constructor called");
    }
}
class GFG
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        Myclass obj = new Myclass();

        // Default constructor provides the default
        // values to the object like 0, null
        System.out.println(obj.name);
        System.out.println(obj.num);
    }
}
```

By Jagadish Sahoo

2. Parameterized Constructor:

- A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.
-

```
import java.io.*;
class Student
{
    // data members of the class.
    String name;
    int id;
    // constructor would initialize data members
    // with the values of passed arguments while
    // object of that class created.
    Student(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
class GFG
{
    public static void main (String[] args)
    {
        // this would invoke the parameterized constructor.
        Student obj = new Student("adam", 1);
        System.out.println("Name :" + obj.name + " and Id :" + obj.id);
    }
}
```

By Jagadish Sahoo

Does constructor return any value?

- There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.

Constructor Overloading

- Like methods, we can overload constructors for creating objects in different ways.
- Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

```
import java.io.*;
class Paracon
{
Paracon(String name)
{
    System.out.println("Constructor with one " +"argument - String :" + name);
}
Paracon(String name, int age)
{
    System.out.println("Constructor with two arguments : " + " String and Integer
: " + name + " + age);
}
Paracon(long id)
{
    System.out.println("Constructor with one argument : " + "Long :" + id);
}
}
```

```
class GFG
{
    public static void main(String[] args)
    {
        Paracon obj1 = new Paracon("Jagadish");
        Paracon obj2 = new Paracon("Aditya", 15);
        Paracon obj3 = new Paracon(325114552);
    }
}
```

By Jagadish Sahoo

How constructors are different from methods in Java?

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

Copy Constructor in Java

- Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

```
class Complex {
```

```
    private double re, im;
```

```
    // A normal parameterized constructor
```

```
    public Complex(double re, double im) {
```

```
        this.re = re;
```

```
        this.im = im;
```

```
}
```

```
    // copy constructor
```

```
    Complex(Complex c) {
```

```
        System.out.println("Copy constructor called");
```

```
        re = c.re;
```

```
        im = c.im;
```

```
}
```

```
    // Overriding the toString of Object class
```

```
    @Override
```

```
    public String toString() {
```

```
        return "(" + re + " + " + im + "i)";
```

```
}
```

```
}
```

By Jagadish Sahoo

Constructor Overloading in Java

```
class Box
{
    double width, height, depth;
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    Box()
    {
        width = height = depth = 0;
    }
    Box(double len)
    {
        width = height = depth = len;
    }
    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}
```

By Jagadish Sahoo

```
public class Test
{
    public static void main(String args[])
    {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println(" Volume of mycube is " + vol);
    }
}
```

By Jagadish Sahoo

Using this() in constructor overloading

- `this()` reference can be used during constructor overloading to call default constructor implicitly from parameterized constructor.
- `this()` should be the first statement inside a constructor.

```
public class Main {  
  
    public static void main(String[] args) {  
        Complex c1 = new Complex(10, 15);  
  
        // Following involves a copy constructor call  
        Complex c2 = new Complex(c1);  
  
        // Note that following doesn't involve a copy constructor call as  
        // non-primitive variables are just references.  
        Complex c3 = c2;  
  
        System.out.println( c2.re + " + " + c2.im );  
        System.out.println( c2.re + " + " + c2.im );  
    }  
}
```

By Jagadish Sahoo

```
class Box
{
    double width, height, depth;
    int boxNo;
Box(double w, double h, double d, int num)
{
    width = w;
    height = h;
    depth = d;
    boxNo = num;
}
Box()
{
width = height = depth = 0;
}
Box(int num)
{
this();
boxNo = num;
}
```

By Jagadish Sahoo

```
public static void main(String[] args)
{
    Box box1 = new Box(1);
    System.out.println(box1.width);
}
```

- we called `Box(int num)` constructor during object creation using only box number. By using `this()` statement inside it, the default constructor(`Box()`) is implicitly called from it which will initialize dimension of Box with 0.

Important points to be taken care while doing Constructor Overloading

- Constructor calling must be the **first** statement of constructor in Java.
- If we have defined any parameterized constructor, then compiler will not create default constructor. and vice versa if we don't define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
- Recursive constructor calling is invalid in java.

Constructors overloading vs Method overloading

- constructor overloading is somewhat similar to method overloading.
If we want to have different ways of initializing an object using different number of parameters, then we must do constructor overloading as we do method overloading when we want different definitions of a method based on different parameters.

Constructor Chaining

- Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining can be done in two ways:

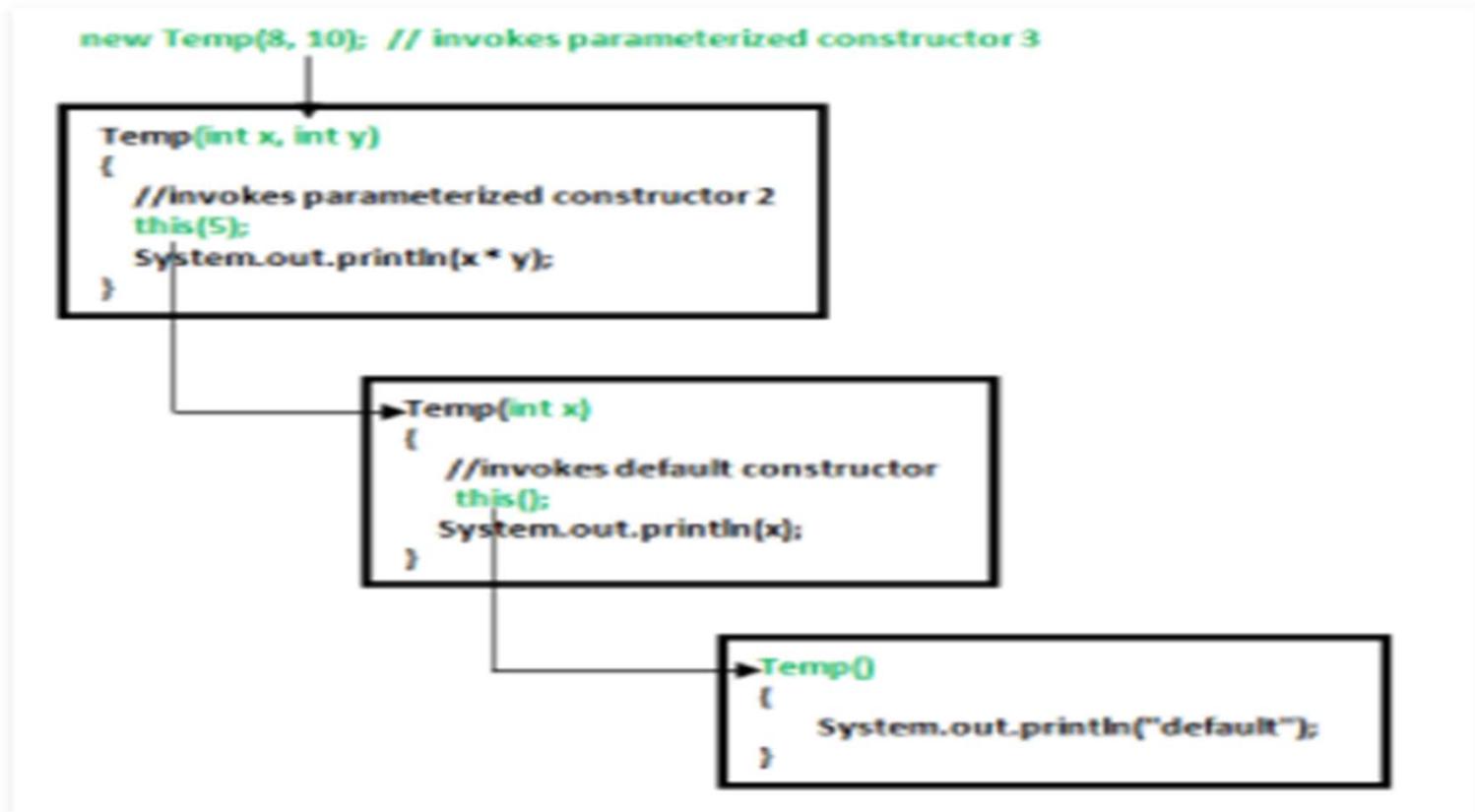
- **Within same class:** It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class.

- Constructor chaining occurs through **inheritance**.
- A sub class constructor's task is to call super class's constructor first.
- This ensures that creation of sub class's object starts with the initialization of the data members of the super class.
- There could be any numbers of classes in inheritance chain.
- Every constructor calls up the chain till class at the top is reached.

Why do we need constructor chaining ?

- This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Constructor Chaining within same class using this() keyword



By Jagadish Sahoo

```
class Temp
{
Temp()
{
this(5);
    System.out.println("The Default constructor");
}
Temp(int x)
{
this(5, 15);
    System.out.println(x);
}
Temp(int x, int y)
{
    System.out.println(x * y);
}

public static void main(String args[])
{
    new Temp();
}
```

By Jagadish Sahoo

Rules of constructor chaining :

- The **this()** expression should always be the first line of the constructor.
- There should be at-least be one constructor without the this() keyword (Temp(int x, int y) in previous example).
- Constructor chaining can be achieved in any order.

What happens if we change the order of constructors?

Nothing, Constructor chaining can be achieved in any order

Constructor Chaining to other class using super() keyword

```
class Base
{
    String name;
    Base()
    {
        this("");
        System.out.println("No-argument constructor of" + " base class");
    }
    Base(String name)
    {
        this.name = name;
        System.out.println("Calling parameterized constructor" + " of base");
    }
}

class Derived extends Base
{
    Derived()
    {
        System.out.println("No-argument constructor " + "of derived");
    }

    Derived(String name)
    {
        super(name);
        System.out.println("Calling parameterized " + "constructor of derived");
    }
}
```

By Jagadish Sahoo

```
public static void main(String args[])
{
    Derived obj = new Derived("test");
}
}
```

By Jagadish Sahoo

Strings in Java

- Strings in Java are Objects that are backed internally by a char array.
- Since arrays are immutable(cannot grow), Strings are immutable as well.
- Whenever a change to a String is made, an entirely new String is created.

Syntax:

<String_Type> <string_variable> = "<sequence_of_string>";

Example:

```
String str = "Jagadish";
```

Memory allotment of String

- Whenever a String Object is created as a literal, the object will be created in String constant pool. This allows JVM to optimize the initialization of String literal.

For example:

```
String str = "Jagadish";
```

- The string can also be declared using **new** operator i.e. dynamically allocated. In case of String are dynamically allocated they are assigned a new memory location in heap. This string will not be added to String constant pool.

For example:

```
String str = new String("Jagadish");
```

```
import java.io.*;
import java.lang.*;

class Test {
    public static void main(String[] args)
    {
        // Declare String without using new operator
        String s = "Jagadish";

        // Prints the String.
        System.out.println("String s = " + s);

        // Declare String using new operator
        String s1 = new String("Jagadish");

        // Prints the String.
        System.out.println("String s1 = " + s1);
    }
}
```

By Jagadish Sahoo

StringBuffer

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

Syntax:

```
StringBuffer s = new StringBuffer("Jagadish");
```

StringBuilder

- The StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

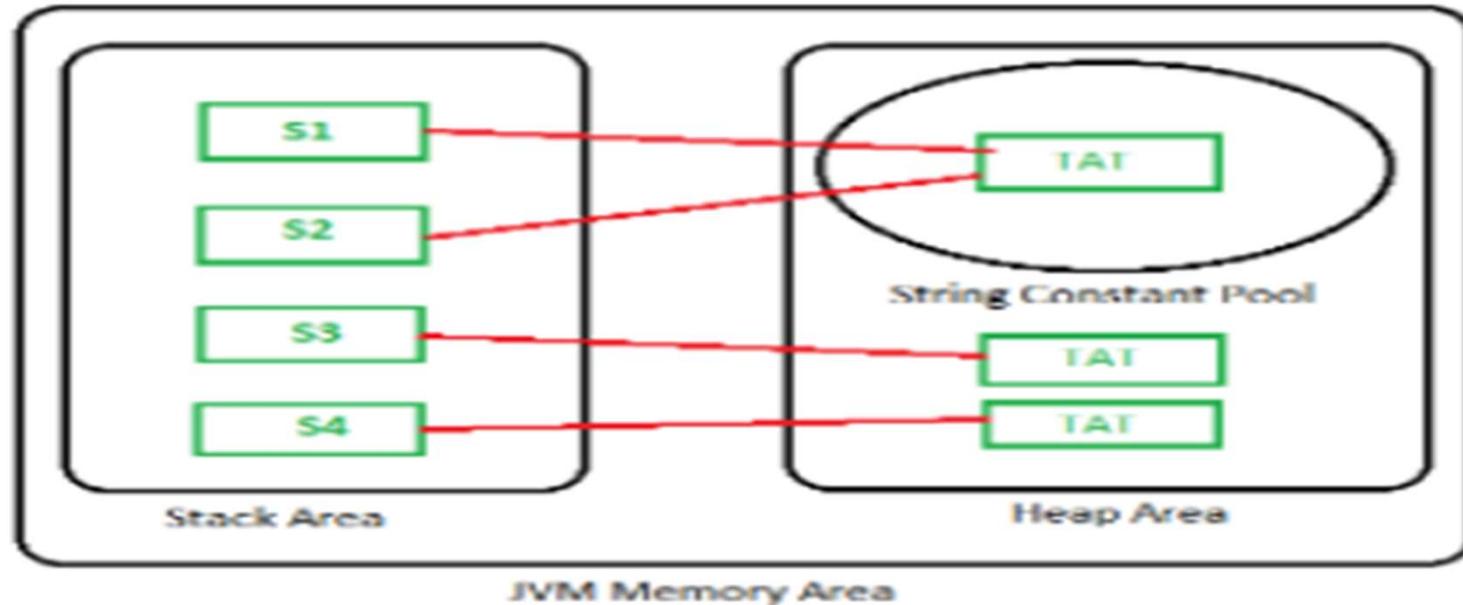
Syntax:

```
StringBuilder str = new StringBuilder();  
str.append("Hello");
```

StringTokenizer

- StringTokenizer class in Java is used to break a string into tokens.

```
class StringStorage {  
    public static void main(String args[])  
    {  
        String s1 = "TAT";  
        String s2 = "TAT";  
        String s3 = new String("TAT");  
        String s4 = new String("TAT");  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println(s3);  
        System.out.println(s4);  
    }  
}
```



Note: All objects in Java are stored in a heap. The reference variable is to the object stored in the stack area or they can be contained in other objects which puts them in the heap area also.

String class in Java

- String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

- There are two ways to create string in Java:

- **String literal**

```
String s = "Jagadish";
```

- **Using new keyword**

```
String s = new String ("Jagadish");
```

Constructors

1. String(byte[] byte_arr) – Construct a new String by decoding the *byte array*. It uses the platform's default character set for decoding.

- Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};
```

```
String s_byte =new String(b_arr);
```

2. String(byte[] byte_arr, Charset char_set)

- Construct a new String by decoding the byte array. It uses the char_set for decoding.

Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};  
Charset cs = Charset.defaultCharset();  
String s_byte_char = new String(b_arr, cs);
```

3. String(byte[] byte_arr, String char_set_name)

- Construct a new String by decoding the byte array. It uses the char_set_name for decoding.
- It looks similar to the previous constructs and they appear before similar functions but it takes the String(which contains char_set_name) as parameter while the previous constructor takes CharSet.

Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};  
String s = new String(b_arr, "US-ASCII");
```

4. String(byte[] byte_arr, int start_index, int length)

- Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).

Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};
```

```
String s = new String(b_arr, 1, 3);
```

5. String(byte[] byte_arr, int start_index, int length, Charset char_set)

- Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).
- Uses char_set_name for decoding.

Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};
```

```
Charset cs = Charset.defaultCharset();
```

```
String s = new String(b_arr, 1, 3, cs);
```

6. String(byte[] byte_arr, int start_index, int length, String char_set_name)

- Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location).Uses char_set_name for decoding.

Example:

```
byte[] b_arr = {72, 102, 102, 108, 115};  
String s = new String(b_arr, 1, 4, "US-ASCII");
```

7. String(char[] char_arr)

- Allocates a new String from the given Character array

Example:

```
char char_arr[] = {'H', 'e', 'l', 'l', 'o'};
```

```
String s = new String(char_arr); //Hello
```

8. String(char[] char_array, int start_index, int count)

- Allocates a String from a given character array but choose count characters from the start_index.

Example:

```
char char_arr[] = {'H', 'e', 'l', 'l', 'o'};
```

```
String s = new String(char_arr , 1, 3); //ell
```

9. String(int[] uni_code_points, int offset, int count)

Allocates a String from a uni_code_array but choose count characters from the start_index.

Example:

```
int[] uni_code = {72, 102, 102, 108, 115};
```

```
String s = new String(uni_code, 1, 3);
```

10. String(StringBuffer s_buffer)

Allocates a new string from the string in s_buffer

Example:

```
StringBuffer s_buffer = new StringBuffer("Jagadish");
```

```
String s = new String(s_buffer);
```

11. String(StringBuilder s_builder)

Allocates a new string from the string in s_builder

- Example:
- ```
StringBuilder s_builder = new StringBuilder("Jagadish");
```
- ```
String s = new String(s_builder);
```

String Methods

1. **int length();** Returns the number of characters in the String.

"jagadish".length(); // returns 13

2. **Char charAt(int i);** Returns the character at ith index.

" jagadish".charAt(3); // returns ‘a’

3. **String substring (int i);** Return the substring from the ith index character to end.

" jagadish".substring(3); // returns “adish”

4. **String substring (int i, int j);** Returns the substring from i to j-1 index.

"jagadish".substring(2, 5); // returns “gad”

5. String concat(String str); Concatenates specified string to the end of this string.

```
String s1 = "Jagadish";
```

```
String s2 = "Sahoo";
```

```
String output = s1.concat(s2); // returns "JagadishSahoo"
```

6. int indexOf (String s); Returns the index within the string of the first occurrence of the specified string.

```
String s = "Learn Share Learn";
```

```
int output = s.indexOf("Share"); // returns 6
```

7. int indexOf (String s, int i); Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";
```

```
int output = s.indexOf("ea",3); // returns 13
```

8. Int lastIndexOf(String s); Returns the index within the string of the last occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.lastIndexOf("a"); // returns 14
```

9. boolean equals(Object otherObj); Compares this string to the specified object.

```
Boolean out = "Jagadish".equals("Jagadish"); // returns true
Boolean out = "Jagadish".equals("jagadish"); // returns false
```

10. boolean equalsIgnoreCase (String anotherString); Compares string to another string, ignoring case considerations.

```
Boolean out = "Jagadish".equals("Jagadish"); // returns true
Boolean out = "Jagadish".equals("jagadish"); // returns true
```

11. int compareTo(String anotherString); Compares two string lexicographically.
int out = s1.compareTo(s2); // where s1 and s2 are strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2
out = 0 // s1 and s2 are equal.
out > 0 // s1 comes after s2.

12. int compareTolgnoreCase(String anotherString); Compares two string lexicographically, ignoring case considerations.

int out = s1.compareToIgnoreCase(s2);
// where s1 and s2 are strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2
out = 0 // s1 and s2 are equal.
out > 0 // s1 comes after s2.

13. String toLowerCase();

Converts all the characters in the String to lower case.

```
String word1 = "HeLLo";
```

```
String word3 = word1.toLowerCase(); // returns "hello"
```

14. String toUpperCase();

Converts all the characters in the String to upper case.

```
String word1 = "HeLLo";
```

```
String word2 = word1.toUpperCase(); // returns "HELLO"
```

15. String trim();

Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = " Learn Share Learn ";
```

```
String word2 = word1.trim(); // returns "Learn Share Learn"
```

16. String replace (char oldChar, char newChar);

Returns new string by replacing all occurrences of oldChar with newChar.

```
String s1 = "Hello";
```

```
String s2 = s1.replace('l','g'); // returns "Hego"
```

StringBuffer class in Java

- StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- `StringBuffer()`: It reserves room for 16 characters without reallocation.

```
StringBuffer s=new StringBuffer();
```

- `StringBuffer(int size)`It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer s=new StringBuffer(20);
```

- `StringBuffer(String str)`: It accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s=new StringBuffer("Jagadish");
```

Methods

- **length() and capacity()**: The length of a StringBuffer can be found by the length() method, while the total allocated capacity can be found by the capacity() method.

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("Jagadish");
        int p = s.length();
        int q = s.capacity();
        System.out.println("Length of string GeeksforGeeks=" + p);
        System.out.println("Capacity of string GeeksforGeeks=" + q);
    }
}
```

- `append()`: It is used to add text at the end of the existing text. Here are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("Jagadish");
        s.append("Sahoo");
        System.out.println(s); // returns JagadishSahoo
        s.append(123);
        System.out.println(s); // returns JagadishSahoo123
    }
}
```

- `insert()`: It is used to insert text at the specified index position.

`StringBuffer insert(int index, String str)`

`StringBuffer insert(int index, char ch)`

Here, `index` specifies the index at which point the string will be inserted into the invoking `StringBuffer` object.

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("JagadishSahoo");
        s.insert(8, "for");
        System.out.println(s); // returns JagadishforSahoo
        s.insert(0, 5);
        System.out.println(s); // returns 5JagadishforSahoo
        s.insert(3, true);
        System.out.println(s);
        s.insert(5, 41.35d);
        System.out.println(s);
        s.insert(8, 41.35f);
        System.out.println(s);
        char geeks_arr[] = { 'p', 'a', 'w', 'a', 'n' };
        // insert character array at offset 9
        s.insert(2, geeks_arr);
        System.out.println(s);
    }
}
```

By Jagadish Sahoo

- `reverse()`: It can reverse the characters within a `StringBuffer` object using `reverse()`. This method returns the reversed object on which it was called.

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("Jagadish");
        s.reverse();
        System.out.println(s);
    }
}
```

- `delete()` and `deleteCharAt()`: It can delete characters within a `StringBuffer` by using the methods `delete()` and `deleteCharAt()`. The `delete()` method deletes a sequence of characters from the invoking object. Here, `start Index` specifies the index of the first character to remove, and `end Index` specifies an index one past the last character to remove. Thus, the substring deleted runs from `start Index` to `endIndex-1`. The resulting `StringBuffer` object is returned. The `deleteCharAt()` method deletes the character at the index specified by `loc`. It returns the resulting `StringBuffer` object. These methods are shown here:
 - `StringBuffer delete(int startIndex, int endIndex)`
 - `StringBuffer deleteCharAt(int loc)`

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("JagadishSahoo");
        s.delete(0, 5);
        System.out.println(s);
        s.deleteCharAt(7);
        System.out.println(s);
    }
}
```

- `replace()`: It can replace one set of characters with another set inside a `StringBuffer` object by calling `replace()`. The substring being replaced is specified by the indexes `startIndex` and `endIndex`. Thus, the substring at `startIndex` through `endIndex-1` is replaced. The replacement string is passed in `str`. The resulting `StringBuffer` object is returned.
- `StringBuffer replace(int startIndex, int endIndex, String str)`

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        StringBuffer s = new StringBuffer("JagadishSahoo");
        s.replace(5, 8, "are");
        System.out.println(s);
    }
}
```

➤ `ensureCapacity()`: It is used to increase the capacity of a `StringBuffer` object. The new capacity will be set to either the value we specify or twice the current capacity plus two (i.e. `capacity+2`), whichever is larger. Here, `capacity` specifies the size of the buffer.

- `void ensureCapacity(int capacity)`

➤ `void setCharAt(int index, char ch)`: In this method, the character at the specified index is set to `ch`.

Syntax:

```
public void setCharAt(int index, char ch)
```

- `String substring(int start)`: This method returns a new `String` that contains a subsequence of characters currently contained in this character sequence.

Syntax:

```
public String substring(int start)
```

```
public String substring(int start, int end)
```

- `String toString()`: This method returns a string representing the data in this sequence.

Syntax:

```
public String toString()
```

StringBuilder Class in Java

- The StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.
- The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters.
- However the StringBuilder class differs from the StringBuffer class on the basis of synchronization.
- The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

- Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case).
- Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.
- Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

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.

Note : There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

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

By Jagadish Sahoo

```
Private  
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  
}  
}
```

- If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

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.

```
//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  
    }  
}
```

Protected

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

```
//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();  
}  
}
```

By Jagadish Sahoo

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();  
    }  
}
```

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- 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.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

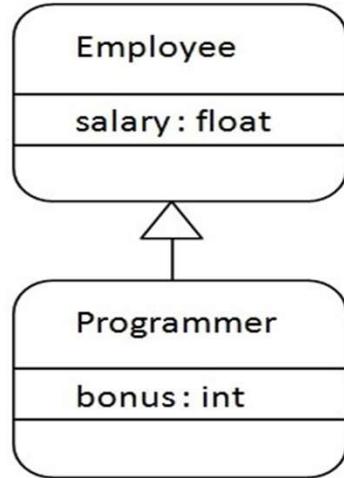
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent 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.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
```

```
{  
    //methods and fields  
}
```

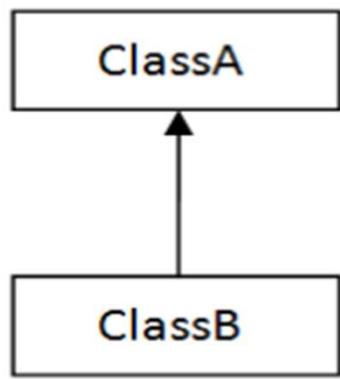
- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



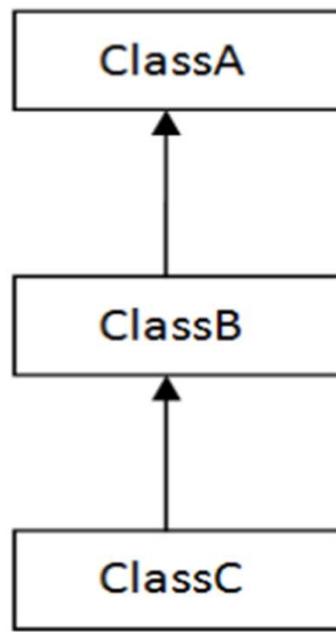
Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

```
class Employee{  
    float salary=40000;  
}  
  
class Programmer extends Employee{  
    int bonus=10000;  
    public static void main(String args[]){  
        Programmer p=new Programmer();  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
    }  
}
```

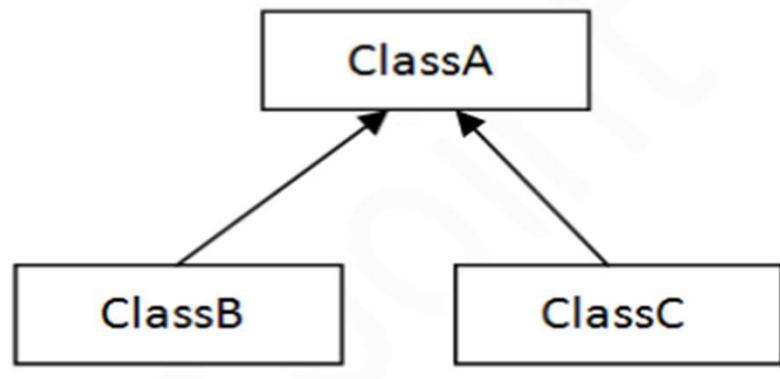
Types of inheritance in java



1) Single

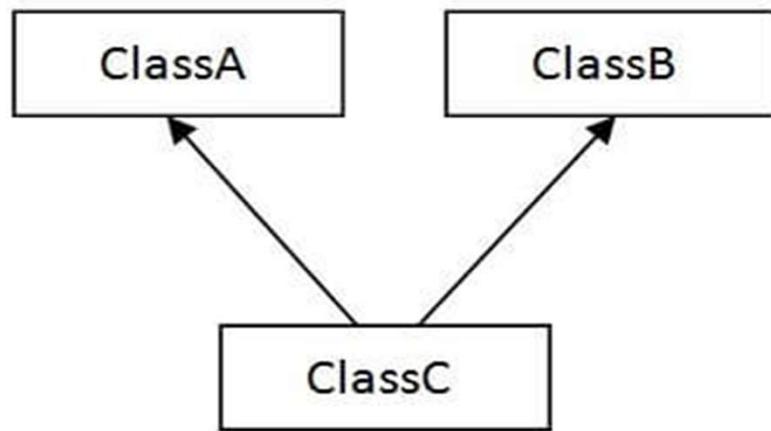


2) Multilevel

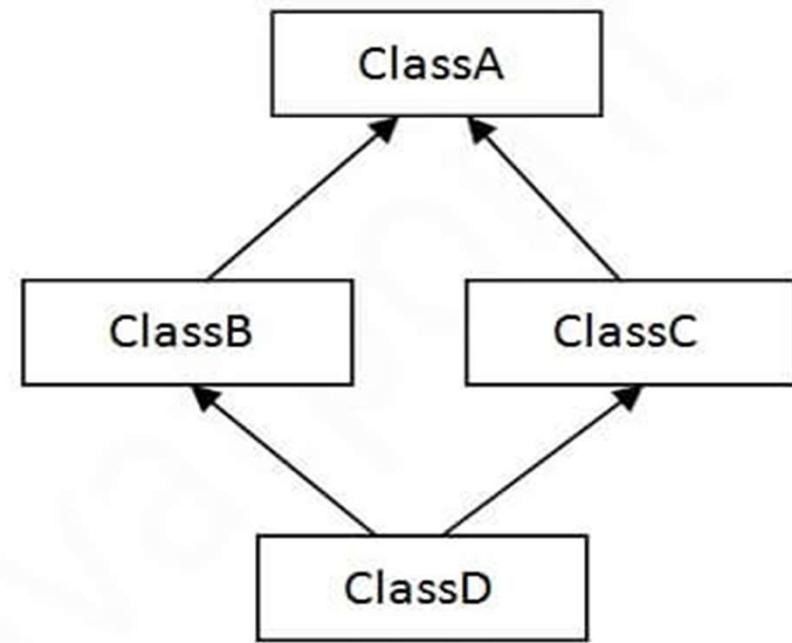


3) Hierarchical

- In java programming, multiple and hybrid inheritance is supported through interface only.



4) Multiple



5) Hybrid

Single Inheritance

- When a class inherits another class, it is known as a *single inheritance*.

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
  
class Dog extends Animal{  
void bark(){System.out.println("barking...");}  
}  
  
class TestInheritance{  
public static void main(String args[]){  
Dog d=new Dog();  
d.bark();  
d.eat();  
}}}
```

Multilevel Inheritance

- When there is a chain of inheritance, it is known as *multilevel inheritance*.

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
  
class Dog extends Animal{  
void bark(){System.out.println("barking...");}  
}  
  
class BabyDog extends Dog{  
void weep(){System.out.println("weeping...");}  
}  
  
class TestInheritance2{  
public static void main(String args[]){  
BabyDog d=new BabyDog();  
d.weep();  
d.bark();  
d.eat();  
}}
```

By Jagadish Sahoo

Hierarchical Inheritance

- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

```
class Animal{
void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
void bark(){System.out.println("barking...");}
}

class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}

class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark(); //C.T.Error
}}
}
```

Why multiple inheritance is not supported in java?

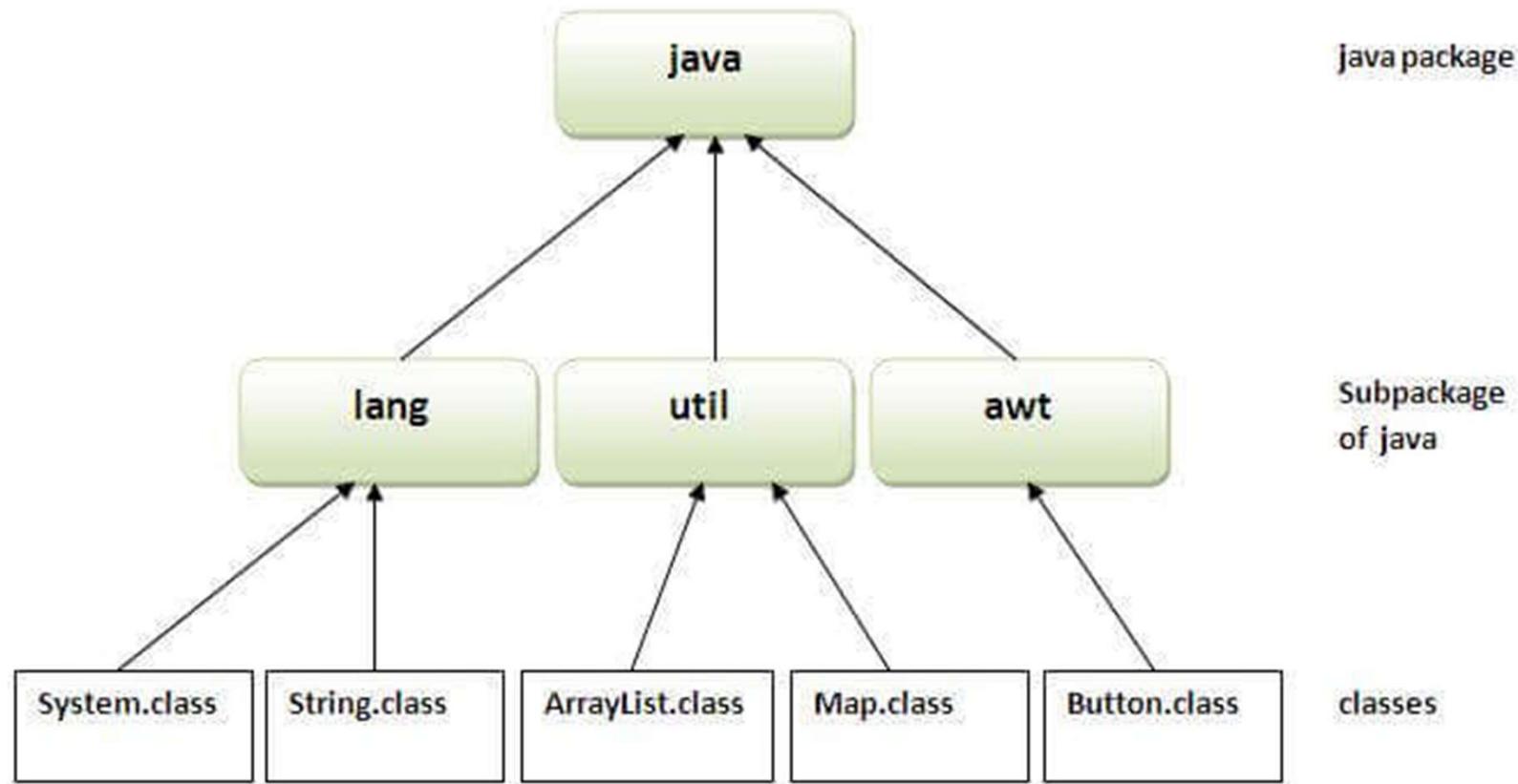
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{  
void msg(){System.out.println("Hello");}  
}  
class B{  
void msg(){System.out.println("Welcome");}  
}  
class C extends A,B{//suppose if it were
```

```
public static void main(String args[]){  
    C obj=new C();  
    obj.msg();//Now which msg() method would be invoked?  
}  
}
```

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



By Jagadish Sahoo

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Creating Package

- The **package keyword** is used to create a package in java.

```
//save as Simple.java
```

```
package mypack;  
public class Simple{  
    public static void main(String args[]){  
        System.out.println("Welcome to package");  
    }  
}
```

Compiling a Package

Syntax

```
javac -d directory javafilename
```

```
javac -d . Simple.java
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to run java package program

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

How to access package from another package?

- There are three ways to access the package from outside the package.
1. import package.*;
 2. import package.classname;
 3. fully qualified name.

Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java
```

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

```
//save by B.java
package mypack;
import pack1.*;

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

Using packagename.classname

- If you import package.classname then only declared class of this package will be accessible.

//save by A.java

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

```
//save by B.java
package mypack;
import pack.A;

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

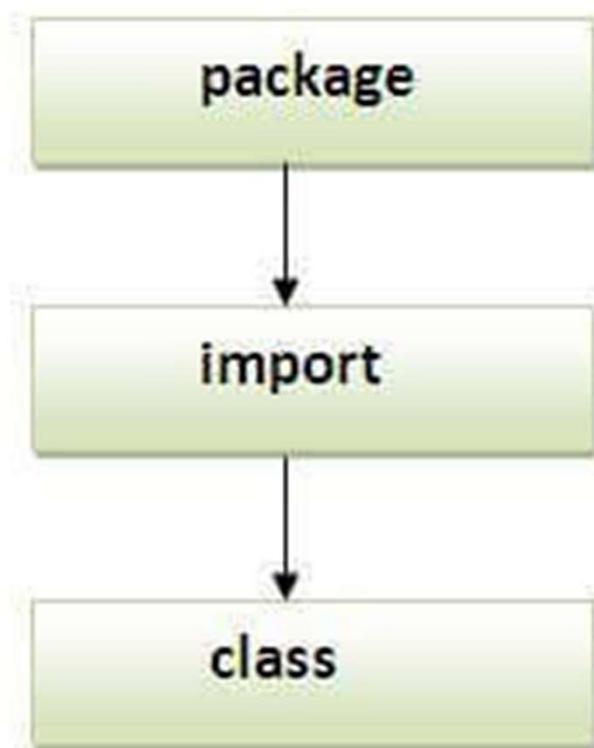
Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

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

//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

- If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.
- Note: Sequence of the program must be package then import then class.



By Jagadish Sahoo

Subpackage in java

- Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

```
package pack1.pack2;  
class Simple{  
    public static void main(String args[]){  
        System.out.println("Hello subpackage");  
    }  
}
```

- To Compile: javac -d . Simple.java
- To Run: java pack1.pack2.Simple

How to put two public classes in a package?

- If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
```

```
package javatpoint;
```

```
public class A{}
```

```
//save as B.java
```

```
package javatpoint;
```

```
public class B{}
```

Interface in Java

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

- There are mainly three reasons to use interface.

It is used to achieve abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

How to declare an interface?

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

Syntax:

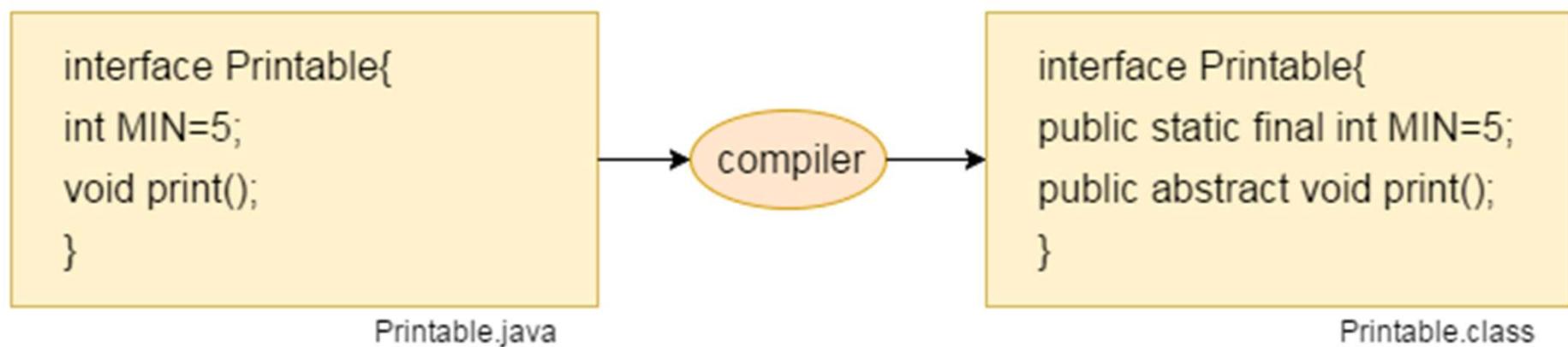
```
interface <interface_name>{
```

```
    // declare constant fields
```

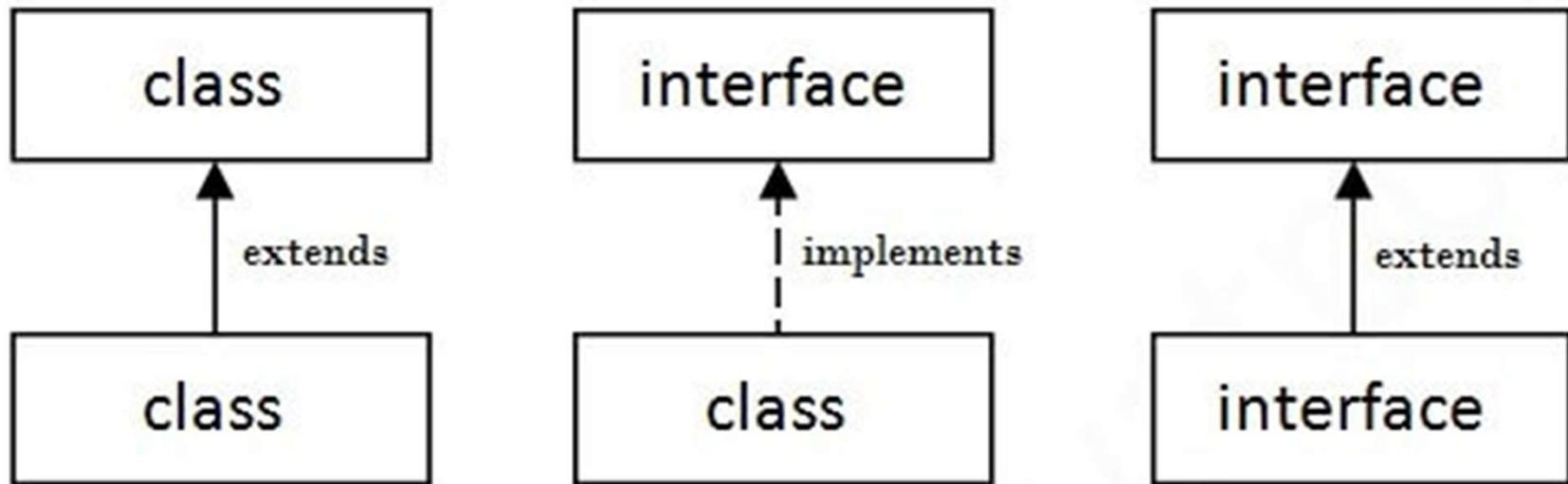
```
    // declare methods that abstract by default.
```

```
}
```

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.
- In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces



```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}
}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

- In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes.
- In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers.
- Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

```
//Interface declaration: by first user
interface Drawable{
void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle(); //In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
}
```

```
interface Bank{
    float rateOfInterest();
}

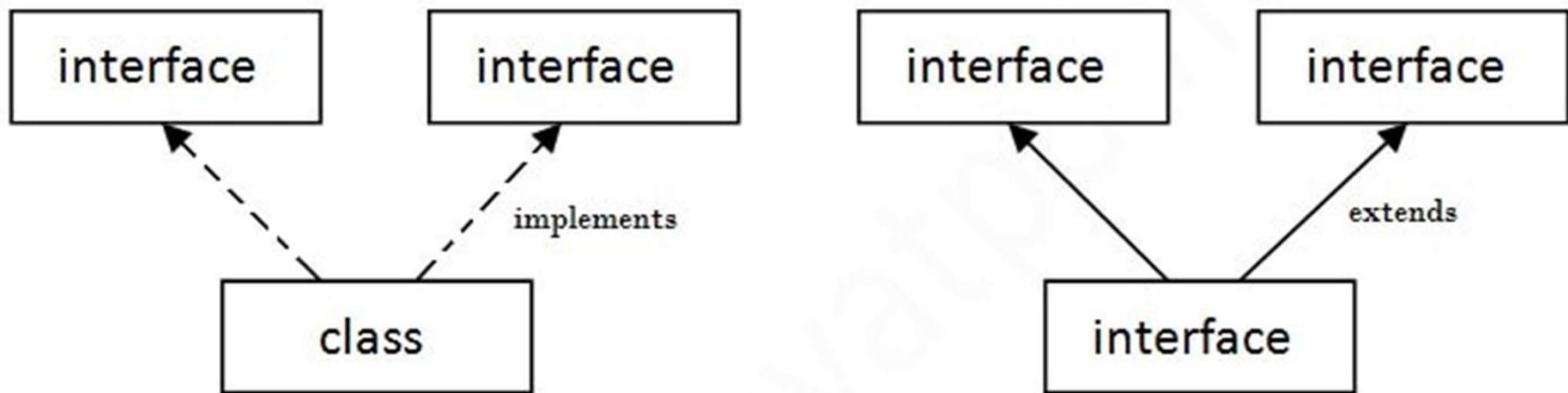
class SBI implements Bank{
    public float rateOfInterest(){return 9.15f;}
}

class PNB implements Bank{
    public float rateOfInterest(){return 9.7f;}
}

class TestInterface2{
    public static void main(String[] args){
        Bank b=new SBI();
        System.out.println("ROI: "+b.rateOfInterest());
    }
}
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

By Jagadish Sahoo

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- Multiple inheritance is not supported in the case of class because of ambiguity.
- However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}
```

Interface inheritance

- A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}

interface Showable extends Printable{
void show();
}

class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome")}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
}
}
```

By Jagadish Sahoo

Java 8 Default Method in Interface

- Since Java 8, we can have method body in interface. But we need to make it default method.

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}

class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
}
```

Java 8 Static Method in Interface

- Since Java 8, we can have static method in interface.

```
interface Drawable{  
void draw();  
static int cube(int x){return x*x*x;}  
}  
class Rectangle implements Drawable{  
public void draw(){System.out.println("drawing rectangle");}  
}
```

```
class TestInterfaceStatic{  
public static void main(String args[]){  
Drawable d=new Rectangle();  
d.draw();  
System.out.println(Drawable(cube(3));  
}}
```

By Jagadish Sahoo

What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

Nested Interface in Java

An interface can have another interface which is known as a nested interface.

```
interface printable{  
    void print();  
  
interface MessagePrintable{  
    void msg();  
}  
}
```

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.

By Jagadish Sahoo

7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Wrapper classes in Java

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive.*
- Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

- Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- Synchronization: Java synchronization works with objects in Multithreading.
- java.util package: The java.util package provides the utility classes to deal with objects.
- Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

- The eight classes of the `java.lang` package are known as wrapper classes in Java.

Primitive Type	Wrapper class
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

By Jagadish Sahoo

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,
- For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}}
```

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
//Java program to convert object into primitives  
//Unboxing example of Integer to int  
public class WrapperExample2{  
public static void main(String args[]){  
    //Converting Integer to int  
    Integer a=new Integer(3);  
    int i=a.intValue();//converting Integer to int explicitly  
    int j=a;//unboxing, now compiler will write a.intValue() internally  
  
    System.out.println(a+" "+i+" "+j);  
}}
```

By Jagadish Sahoo

Utility methods of Wrapper classes

- The objective of Wrapper class is to define several utility methods which are required for the primitive types.
- There are 4 utility methods for primitive type which is defined by Wrapper class:
 1. **valueOf()** method
 2. **xxxValue()** method
 3. **parseXxx()** method
 4. **toString()** method

`valueOf()` method

- We can use `valueOf()` method to create Wrapper object for given primitive or String.
- There are 3 types of `valueOf()` methods:
 1. **Wrapper valueOf(String s)**
 2. **Wrapper valueOf(String s, int radix)**
 3. **Wrapper valueOf(primitive p)**

Wrapper valueOf(String s)

- Every wrapper class except Character class contains a static valueOf() method to create Wrapper class object for given String.

Syntax:

```
public static Wrapper valueOf(String s);
class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf("10");
        System.out.println(I);
        Double D = Double.valueOf("10.0");
        System.out.println(D);
        Boolean B = Boolean.valueOf("true");
        System.out.println(B);

        // Here we will get RuntimeException
        Integer I1 = Integer.valueOf("ten");
    }
}
```

By Jagadish Sahoo

Wrapper valueOf(String s, int radix)

- Every Integral Wrapper class Byte, Short, Integer, Long) contains the following valueOf() method to create a Wrapper object for the given String with specified radix. The range of the radix is 2 to 36.

Syntax:

```
public static Wrapper valueOf(String s, int radix)
class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf("1111", 2);
        System.out.println(I);
        Integer I1 = Integer.valueOf("1111", 4);
        System.out.println(I1);
    }
}
```

Wrapper valueOf(primitive p)

Every Wrapper class including Character class contains the following method to create a Wrapper object for the given primitive type.

Syntax:

```
public static Wrapper valueOf(primitive p);
class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf(10);
        Double D = Double.valueOf(10.5);
        Character C = Character.valueOf('a');
        System.out.println(I);
        System.out.println(D);
        System.out.println(C);
    }
}
```

xxxValue() method

- We can use xxxValue() methods to get the primitive for the given Wrapper Object. Every number type Wrapper class(Byte, Short, Integer, Long, Float, Double) contains the following 6 methods to get primitive for the given Wrapper object:
 1. public byte byteValue()
 2. public short shortValue()
 3. public int intValue()
 4. public long longValue()
 5. public float floatValue()
 6. public float doubleValue()

```
class GFG {  
    public static void main(String[] args)  
    {  
        Integer l = new Integer(130);  
        System.out.println(l.byteValue());  
        System.out.println(l.shortValue());  
        System.out.println(l.intValue());  
        System.out.println(l.longValue());  
        System.out.println(l.floatValue());  
        System.out.println(l.doubleValue());  
    }  
}
```

parseXxx() method

- We can use parseXxx() methods to convert String to primitive.
- There are two types of parseXxx() methods:
 1. **primitive parseXxx(String s)**
 2. **parseXxx(String s, int radix)**

- **primitive parseXxx(String s)** : Every Wrapper class except character class contains the following parseXxx() method to find primitive for the given String object.

Syntax:

```
public static primitive parseXxx(String s);
class GFG {
    public static void main(String[] args)
    {
        int i = Integer.parseInt("10");
        double d = Double.parseDouble("10.5");
        boolean b = Boolean.parseBoolean("true");
        System.out.println(i);
        System.out.println(d);
        System.out.println(b);
    }
}
```

By Jagadish Sahoo

- **parseXxx(String s, int radix)** : Every Integral type Wrapper class (Byte, Short, Integer, Long) contains the following parseXxx() method to convert specified radix String to primitive.

Syntax:

```
public static primitive parseXxx(String s, int radix);
```

```
class GFG {
```

```
    public static void main(String[] args)
```

```
{
```

```
    int i = Integer.parseInt("1000", 2);
```

```
    long l = Long.parseLong("1111", 4);
```

```
    System.out.println(i);
```

```
    System.out.println(l);
```

```
}
```

```
}
```

toString() method

- We can use `toString()` method to convert Wrapper object or primitive to String.
There are few forms of `toString()` method:
- **public String toString()** : Every wrapper class contains the following `toString()` method to convert Wrapper Object to String type.

Syntax:

```
public String toString();
class GFG {
    public static void main(String[] args)
    {
        Integer I = new Integer(10);
        String s = I.toString();
        System.out.println(s);
    }
}
```

- **toString(primitive p)** : Every Wrapper class including Character class contains the following static `toString()` method to convert primitive to String.

Syntax:

```
public static String toString(primitive p);
class GFG {
    public static void main(String[] args)
    {
        String s = Integer.toString(10);
        System.out.println(s);
        String s1 = Character.toString('a');
        System.out.println(s1);
    }
}
```

- **toString(primitive p, int radix)** : Integer and Long classes contains the following `toString()` method to convert primitive to specified radix String.

Syntax:

```
public static String toString(primitive p, int radix);
```

```
class GFG {
```

```
    public static void main(String[] args)
```

```
    {
```

```
        String s = Integer.toString(15, 2);
```

```
        System.out.println(s);
```

```
        String s1 = Long.toString(11110000, 4);
```

```
        System.out.println(s1);
```

```
    }
```

```
}
```

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

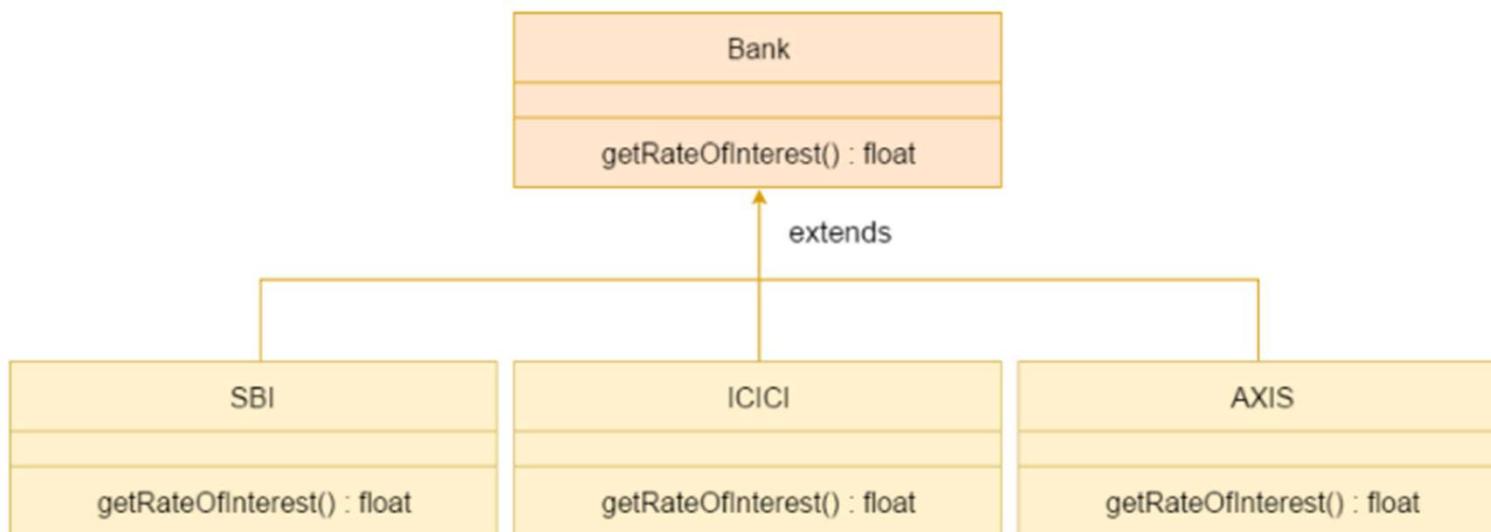
```
class Vehicle{  
    void run(){System.out.println("Vehicle is running");}  
}  
//Creating a child class  
class Bike extends Vehicle{  
    public static void main(String args[]){  
        //creating an instance of child class  
        Bike obj = new Bike();  
        //calling the method with child class instance  
        obj.run();  
    }  
}
```

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run()
{
System.out.println("Bike is running safely");
}
public static void main(String args[]){
Bike2 obj = new Bike2();//creating object
obj.run();//calling method
}
}
```

By Jagadish Sahoo

A real example of Java Method Overriding

- Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



By Jagadish Sahoo

```
class Bank{
int getRateOfInterest(){return 0;}
}

class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}

class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}

class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

By Jagadish Sahoo

- Can we override static method?

No, a static method cannot be overridden.

- Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

- No, because the main is a static method.

Difference between method overloading and method overriding in java

No.	Method Overloading	Method Overriding
1)	Method overloading is used to <i>increase the readability</i> of the program.	Method overriding is used to <i>provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs in <i>two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding. By Jagadish Sahoo

Abstract class in Java

- A class which is declared with the `abstract` keyword is known as an abstract class.
- It can have abstract and non-abstract methods (method with the body).
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

- There are two ways to achieve abstraction in java
 1. Abstract class (0 to 100%)
 2. Interface (100%)

Abstract class in Java

- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented. It cannot be instantiated.

Rules for Java Abstract class

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- If there is an abstract method in a class, that class must be abstract.

Super Keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

super is used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}

class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}

class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

By Jagadish Sahoo

super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}

class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
}
```

By Jagadish Sahoo

super is used to invoke parent class constructor

```
class Animal{  
Animal(){System.out.println("animal is created");}  
}  
  
class Dog extends Animal{  
Dog(){  
super();  
System.out.println("dog is created");  
}  
}  
  
class TestSuper3{  
public static void main(String args[]){  
Dog d=new Dog();  
}}
```

- Example of abstract class

```
abstract class A{}
```

- Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

- Example of abstract method

```
abstract void printStatus(); //no method body and abstract
```

Example of Abstract class that has an abstract method

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){System.out.println("running safely");}  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

By Jagadish Sahoo

```
abstract class Shape{
abstract void draw();
}

//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}

class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}

//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1(); //In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
}
```

By Jagadish Sahoo

```
abstract class Bank{
    abstract int getRateOfInterest();
}

class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}

class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}

class TestBank{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
        b=new PNB();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    }
}
```

By Jagadish Sahoo

Abstract class having constructor, data member and methods

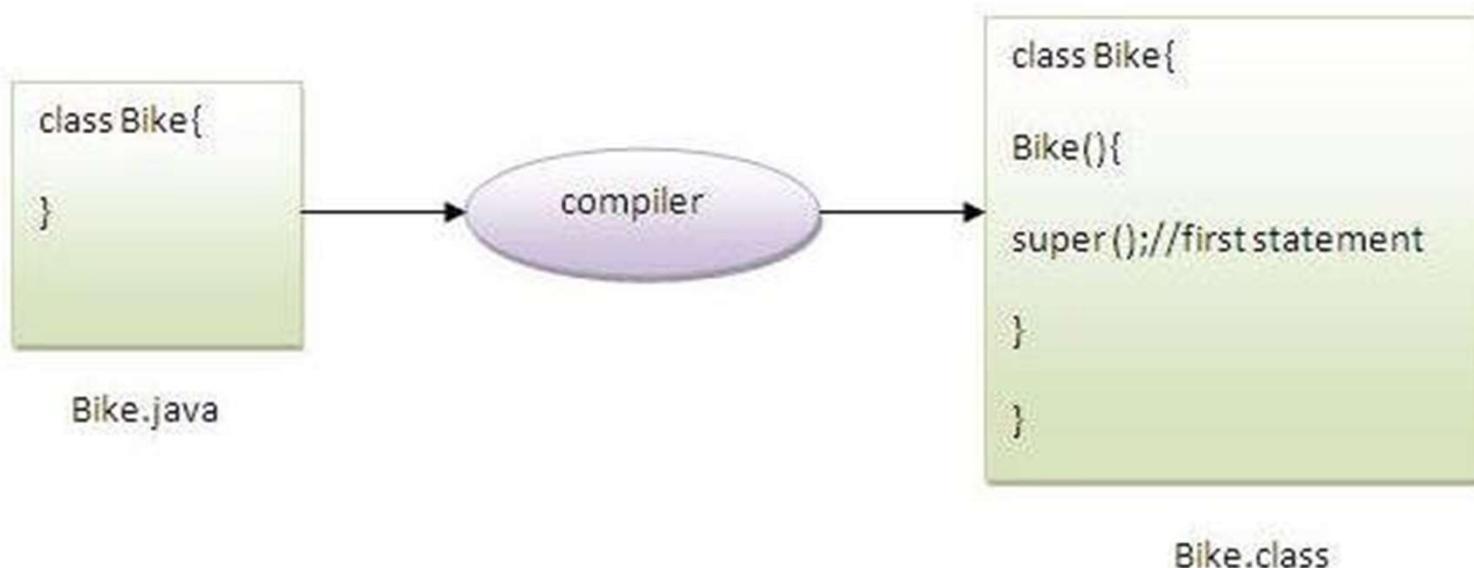
```
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

By Jagadish Sahoo

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



```
class Animal{
Animal(){System.out.println("animal is created");}
}

class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}

class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
}
```

Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{  
    int id;  
    String name;  
    Person(int id, String name){  
        this.id=id;  
        this.name=name;  
    }  
    }  
  
    class Emp extends Person{  
        float salary;  
        Emp(int id, String name, float salary){  
            super(id, name); //reusing parent constructor  
            this.salary=salary;  
        }  
        void display(){System.out.println(id+" "+name+" "+salary);}  
    }
```

By Jagadish Sahoo

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.

Java final variable

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

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

Output:Compile Time Error

Java final method

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

```
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

Java final class

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

```
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

By Jagadish Sahoo

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it.

```
class Bike{  
    final void run(){System.out.println("running...");}
```

```
}
```

```
class Honda2 extends Bike{  
    public static void main(String args[]){  
        new Honda2().run();  
    }  
}
```

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

```
class Student{  
    int id;  
    String name;  
    final String PAN_CARD_NUMBER;  
    ...  
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor.

```
class Bike10{  
    final int speedlimit;//blank final variable
```

```
Bike10(){  
    speedlimit=70;  
    System.out.println(speedlimit);  
}
```

```
public static void main(String args[]){  
    new Bike10();  
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{  
    int cube(final int n){  
        n=n+2;//can't be changed as n is final  
        n*n*n;  
    }  
    public static void main(String args[]){  
        Bike11 b=new Bike11();  
        b.cube(5);  
    }  
}
```

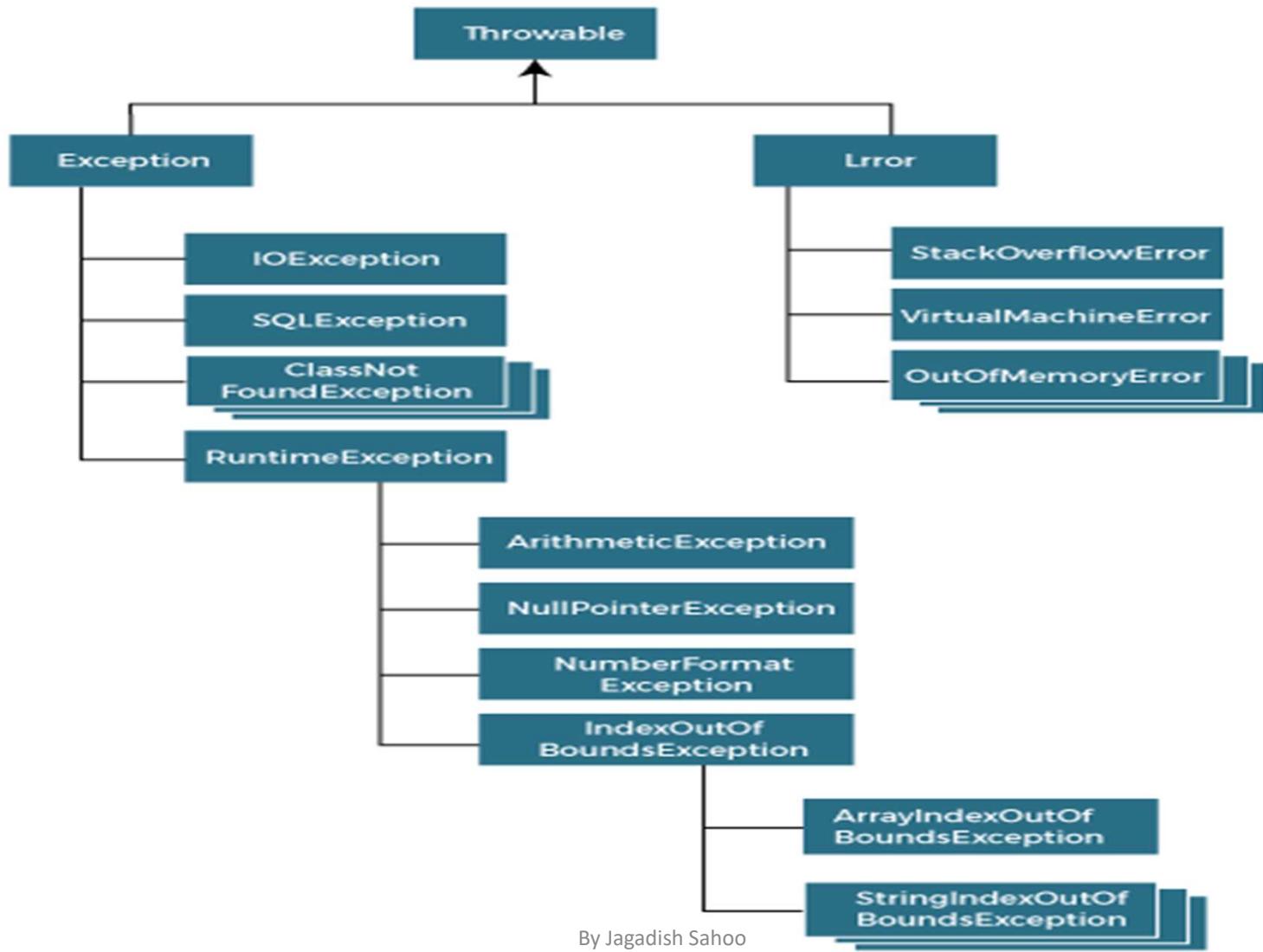
```
class TestSuper5{
    public static void main(String[] args){
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
    }
}
```

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.
- Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Hierarchy of Java Exception classes

- The `java.lang.Throwable` class is the root class of Java Exception hierarchy inherited by two subclasses:
- `Exception` and `Error`.



By Jagadish Sahoo

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

- Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed.
- However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked.
- An error is considered as the unchecked exception.
- However, according to Oracle, there are three types of exceptions namely:
 1. Checked Exception
 2. Unchecked Exception
 3. Error

Difference between Checked and Unchecked Exceptions

- 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

- 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

- 3) Error

Error is irrecoverable. Some examples of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

```
public class JavaExceptionExample{
    public static void main(String args[]){
        try{
            //code that may raise exception
            int data=100/0;
        }catch(ArithmetcException e){System.out.println(e);}
        //rest code of the program
        System.out.println("rest of the code...");
    }
}
```

```
Exception in thread main java.lang.ArithmetcException:/ by zero
rest of the code...
```

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
```

```
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";  
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to its size, the ArrayIndexOutOfBoundsException occurs. There may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];  
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{  
//code that may throw an exception  
}  
catch(Exception_class_Name ref){}
```

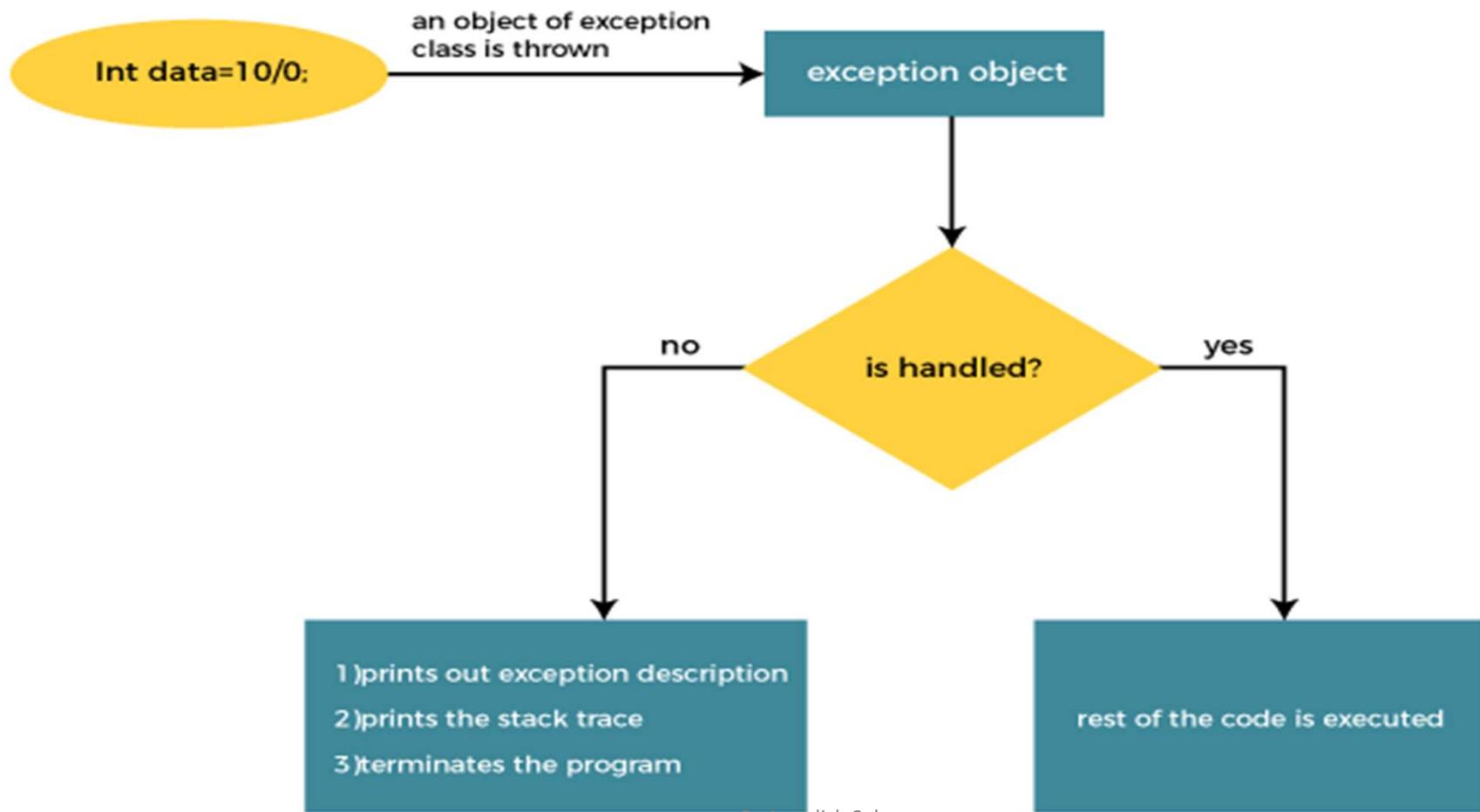
Syntax of try-finally block

```
try{  
//code that may throw an exception  
}  
finally{}
```

Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block



By Jagadish Sahoo

- The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
 1. Prints out exception description.
 2. Prints the stack trace (Hierarchy of methods where the exception occurred).
 3. Causes the program to terminate.
- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

```
public class TryCatchExample1 {  
    public static void main(String[] args) {  
        int data=50/0; //may throw exception  
        System.out.println("rest of the code");  
    }  
}
```

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
```

the rest of the code is not executed (in such case, the rest of the code statement is not printed).

Solution by exception handling

```
public class TryCatchExample2 {  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        //handling the exception  
        catch(ArithmetricException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

```
java.lang.ArithmetricException: / by zero  
rest of the code
```

```
public class TryCatchExample3 {  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
                // if exception occurs, the remaining statement will not execute  
            System.out.println("rest of the code");  
        }  
        // handling the exception  
        catch(ArithmetcException e)  
        {  
            System.out.println(e);  
        }  
    }  
}
```

java.lang.ArithmetcException: / by zero

By Jagadish Sahoo

Handle the exception using the parent class exception.

```
public class TryCatchExample4 {  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        // handling the exception by using Exception class  
        catch(Exception e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

```
java.lang.ArithmcticException: / by zero  
rest of the code
```

By Jagadish Sahoo

Print a custom message on exception

```
public class TryCatchExample5 {  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // displaying the custom message  
            System.out.println("Can't divided by zero");  
        }  
    }  
}
```

Can't divided by zero

Resolve the exception in a catch block

```
public class TryCatchExample6 {  
    public static void main(String[] args) {  
        int i=50;  
        int j=0;  
        int data;  
        try  
        {  
            data=i/j; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // resolving the exception in catch block  
            System.out.println(i/(j+2));  
        }  
    }  
}
```

```
public class TryCatchExample9 {  
    public static void main(String[] args) {  
        try  
        {  
            int arr[] = {1,3,5,7};  
            System.out.println(arr[10]); //may throw exception  
        }  
        // handling the array exception  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```

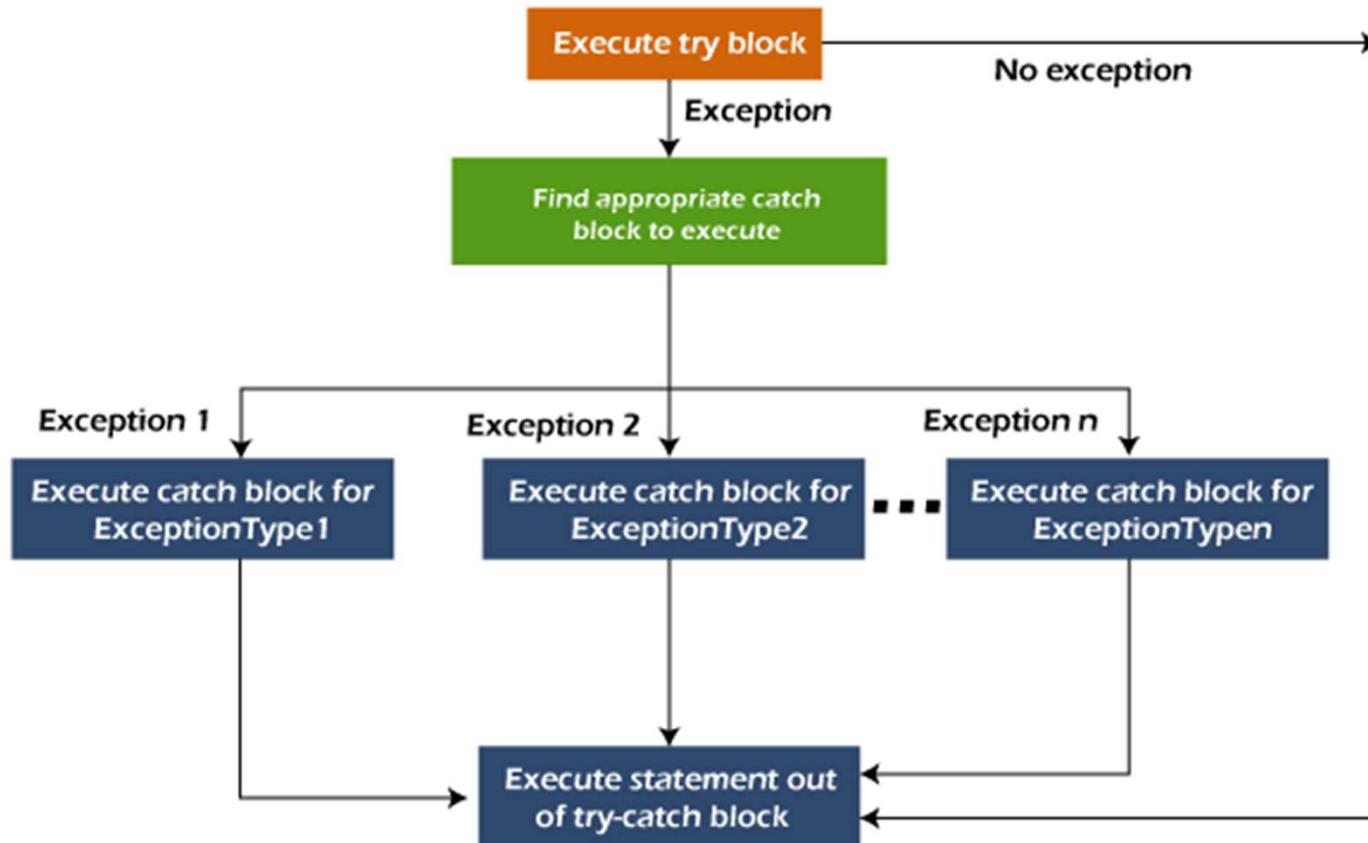
```
java.lang.ArrayIndexOutOfBoundsException: 10  
rest of the code
```

By Jagadish Sahoo

Java Catch Multiple Exceptions

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler.
- So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block



By Jagadish Sahoo

```
public class MultipleCatchBlock1 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Arithmetic Exception occurs
rest of the code

By Jagadish Sahoo

```
public class MultipleCatchBlock2 {  
    public static void main(String[] args) {  
        try{  
            int a[]={};  
  
            System.out.println(a[10]);  
        }  
        catch(ArithmaticException e)  
        {  
            System.out.println("Arithmatic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

```
ArrayIndexOutOfBoundsException Exception occurs  
rest of the code
```

By Jagadish Sahoo

```
public class MultipleCatchBlock3 {  
    public static void main(String[] args) {  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
            System.out.println(a[10]);  
        }  
        catch(ArithmetricException e)  
        {  
            System.out.println("Arithmetric Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Arithmetric Exception occurs
rest of the code

By Jagadish Sahoo

```
public class MultipleCatchBlock4 {  
    public static void main(String[] args) {  
        try{  
            String s=null;  
            System.out.println(s.length());  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmatic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Parent Exception occurs
rest of the code

```
class MultipleCatchBlock5{
    public static void main(String args[]){
        try{
            int a[]={};
            a[5]=30/0;
        }
        catch(Exception e){System.out.println("common task completed");}
        catch(ArithmetricException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2
completed");}
        System.out.println("rest of the code...");
    }
}
```

Compile-time error

By Jagadish Sahoo

Java Nested try block

- In Java, using a try block inside another try block is permitted. It is called as nested try block.
- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
try
{
    statement 1;
    statement 2;
try
{
    statement 3;
    statement 4;
try
{
    statement 5;
    statement 6;
}
catch(Exception e2)
{
}
}

catch(Exception e1)
{
}
}
//catch block of parent (outer) try block
catch(Exception e3)
{
}
....
```

By Jagadish Sahoo

```
public class NestedTryBlock{
    public static void main(String args[]){
        //outer try block
        try{
            //inner try block 1
            try{
                System.out.println("going to divide by 0");
                int b =39/0;
            }
            //catch block of inner try block 1
            catch(ArithmeticException e)
            {
                System.out.println(e);
            }
        }
    }
}
```

```
//inner try block 2
try{
    int a[] = new int[5];
    //assigning the value out of array bounds
    a[5] = 4;
}
//catch block of inner try block 2
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e);
}
System.out.println("other statement");
}
//catch block of outer try block
catch(Exception e)
{
    System.out.println("handled the exception (outer catch)");
}
System.out.println("normal flow..");
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmetricException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

By Jagadish Sahoo

- When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.
- If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example

- Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

```
public class NestedTryBlock2 {  
    public static void main(String args[])  
    {  
        // outer (main) try block  
        try {  
            //inner try block 1  
            try {  
                // inner try block 2  
                try {  
                    int arr[] = { 1, 2, 3, 4 };  
  
                    //printing the array element out of its bounds  
                    System.out.println(arr[10]);  
                }  
            }  
        }  
    }  
}
```

By Jagadish Sahoo

```
// to handles ArithmeticException
    catch (ArithmaticException e) {
        System.out.println("Arithmatic exception");
        System.out.println(" inner try block 2");
    }
}

// to handle ArithmaticException
catch (ArithmaticException e) {
    System.out.println("Arithmatic exception");
    System.out.println("inner try block 1");
}

// to handle ArrayIndexOutOfBoundsException
catch (ArrayIndexOutOfBoundsException e4) {
    System.out.print(e4);
    System.out.println(" outer (main) try block");
}

catch (Exception e5) {
    System.out.print("Exception");
    System.out.println(" handled in main try-block");
}

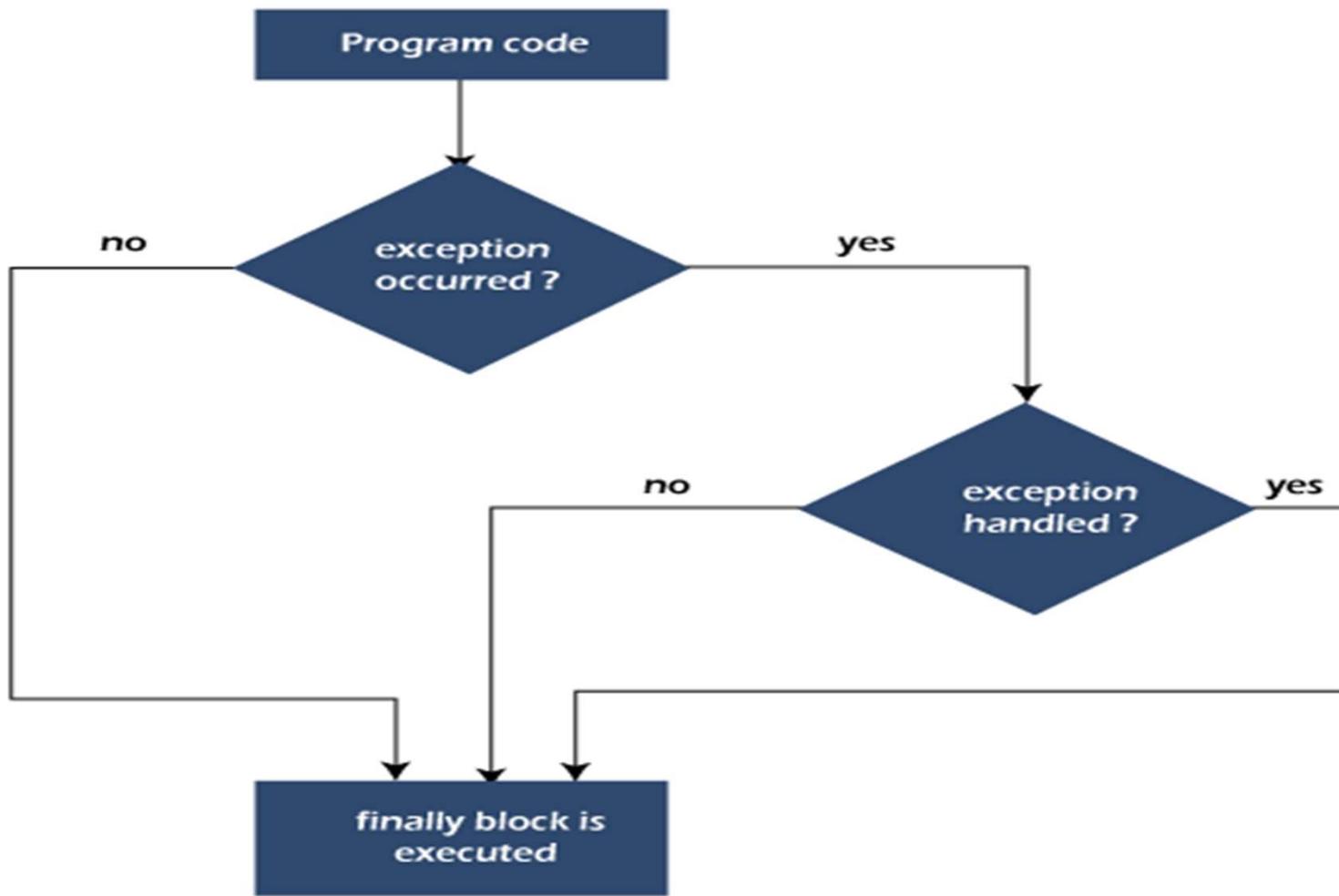
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
(main) try block
```

By Jagadish Sahoo

Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.



By Jagadish Sahoo

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Case 1: When an exception does not occur

```
class TestFinallyBlock {  
    public static void main(String args[]){  
        try{  
            //below code do not throw any exception  
            int data=25/5;  
            System.out.println(data);  
        }  
        //catch won't be executed  
        catch(NullPointerException e){  
            System.out.println(e);  
        }  
        //executed regardless of exception occurred or not  
        finally {  
            System.out.println("finally block is always executed");  
        }  
  
        System.out.println("rest of phe code...");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock  
5  
finally block is always executed  
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

```
public class TestFinallyBlock1{  
    public static void main(String args[]){  
        try {  
            System.out.println("Inside the try block");  
            //below code throws divide by zero exception  
            int data=25/0;  
            System.out.println(data);  
        }  
        //cannot handle Arithmetic type exception  
        //can only accept Null Pointer type exception  
        catch(NullPointerException e){  
            System.out.println(e);  
        }  
        //executes regardless of exception occurred or not  
        finally {  
            System.out.println("finally block is always executed");  
        }  
        System.out.println("rest of the code...");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1  
Inside the try block  
finally block is always executed  
Exception in thread "main" java.lang.ArithmetricException: / by zero  
at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block

```
public class TestFinallyBlock2{  
    public static void main(String args[]){  
        try {  
            System.out.println("Inside try block");  
            //below code throws divide by zero exception  
            int data=25/0;  
            System.out.println(data);  
        }  
        //handles the Arithmetic Exception / Divide by zero exception  
        catch(ArithmaticException e){  
            System.out.println("Exception handled");  
            System.out.println(e);  
        }  
        //executes regardless of exception occurred or not  
        finally {  
            System.out.println("finally block is always executed");  
        }  
        System.out.println("rest of the code...");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2  
Inside try block  
Exception handled  
java.lang.ArithmaticException: / by zero  
finally block is always executed  
rest of the code...
```

Java throw Exception

- In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.
- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

- We can also define our own set of conditions and throw an exception explicitly using throw keyword.
- For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
- The syntax of the Java throw keyword is

```
throw new exception_class("error message");  
throw new IOException("sorry device error");
```

Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {  
    //function to check if person is eligible to vote or not  
    public static void validate(int age) {  
        if(age<18) {  
            //throw Arithmetic exception if not eligible to vote  
            throw new ArithmeticException("Person is not eligible to vote");  
        }  
        else {  
            System.out.println("Person is eligible to vote!!");  
        }  
    }  
}
```

By Jagadish Sahoo

```
//main method  
public static void main(String args[]){  
    //calling the function  
    validate(13);  
    System.out.println("rest of the code...");  
}  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1  
Exception in thread "main" java.lang.ArithmetricException: Person is not eligible to  
vote  
    at TestThrow1.validate(TestThrow1.java:8)  
    at TestThrow1.main(TestThrow1.java:18) By Jagadish Sahoo
```

Example 2: Throwing Checked Exception

```
import java.io.*;
public class TestThrow2 {
    public static void method() throws FileNotFoundException {
        FileReader file = new FileReader("C:\\\\Users\\\\Anurati\\\\Desktop\\\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);
        throw new FileNotFoundException();
    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
    at TestThrow2.method(TestThrow2.java:12)
    at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

By Jagadish Sahoo

Example 3: Throwing User-defined Exception

- // class represents user-defined exception

```
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
// Class that uses above MyException
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        }
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

By Jagadish Sahoo

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{  
//method code  
}
```

Which exception should be declared?

Ans: Checked exception only, because:

unchecked exception: under our control so we can correct our code.

error: beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

2 cases of throws

Case 1: We have caught the exception i.e. we have handled the exception using try/catch block.

Case 2: We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

```
import java.io.*;  
class M{  
    void method()throws IOException{  
        throw new IOException("device error");  
    }  
}  
  
public class Testthrows2{  
    public static void main(String args[]){  
        try{  
            M m=new M();  
            m.method();  
        }catch(Exception e){System.out.println("exception handled");}  
  
        System.out.println("normal flow...");  
    }  
}
```

```
exception handled  
normal flow...
```

By Jagadish Sahoo

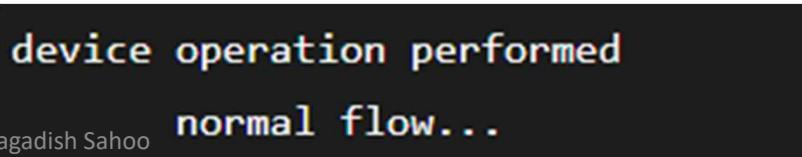
Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

A) If exception does not occur

```
import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
```



device operation performed
normal flow...

By Jagadish Sahoo

B) If exception occurs

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
```

Exception in thread "main" java.io.IOException: device error
at M.method(Testthrows4.java:4)
at Testthrows4.main(Testthrows4.java:10)

By Jagadish Sahoo

Difference between throw and throws

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used to throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	

By Jagadish Sahoo

3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

```
public class TestThrow {  
    //defining a method  
    public static void checkNum(int num) {  
        if (num < 1) {  
            throw new ArithmeticException("\nNumber is negative, cannot calculate square");  
        }  
        else {  
            System.out.println("Square of " + num + " is " + (num*num));  
        }  
    }  
    //main method  
    public static void main(String[] args) {  
        TestThrow obj = new TestThrow();  
        obj.checkNum(-3);  
        System.out.println("Rest of the code..");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow  
Exception in thread "main" java.lang.ArithmetricException:  
Number is negative, cannot calculate square  
    at TestThrow.checkNum(TestThrow.java:6)  
    at TestThrow.main(TestThrow.java:16)
```

By Jagadish Sahoo

Java throws Example

```
public class TestThrows {  
    //defining a method  
    public static int divideNum(int m, int n) throws ArithmeticException {  
        int div = m / n;  
        return div;  
    }  
    //main method  
    public static void main(String[] args) {  
        TestThrows obj = new TestThrows();  
        try {  
            System.out.println(obj.divideNum(45, 0));  
        }  
        catch (ArithmeticException e){  
            System.out.println("\nNumber cannot be divided by 0");  
        }  
        System.out.println("Rest of the code..");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrows  
Number cannot be divided by 0  
Rest of the code..
```

By Jagadish Sahoo

Java throw and throws Example

```
public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmeticException
    static void method() throws ArithmeticException
    {
        System.out.println("Inside the method()");
        throw new ArithmeticException("throwing ArithmeticException");
    }
    //main method
    public static void main(String args[])
    {
        try
        {
            method();
        }
        catch(ArithmetricException e)
        {
            System.out.println("caught in main() method");
        }
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

By Jagadish Sahoo

Difference between final, finally and finalize

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.

3.	Functionality	<p>(1) Once declared, final variable becomes constant and cannot be modified.</p> <p>(2) final method cannot be overridden by subclass.</p> <p>(3) final class cannot be inherited.</p>	<p>(1) finally block runs the important code even if exception occurs or not.</p> <p>(2) finally block cleans up all the resources used in try block</p>	<p>finalize method performs the cleaning activities with respect to the object before its destruction.</p>
4.	Execution	<p>Final method is executed only when we call it.</p>	<p>Finally block is executed as soon as the try-catch block is executed.</p> <p>It's execution is not dependant on the exception.</p>	<p>finalize method is executed just before the object is destroyed.</p>

By Jagadish Sahoo

Java finalize Example

```
public class FinalizeExample {  
    public static void main(String[] args)  
    {  
        FinalizeExample obj = new FinalizeExample();  
        // printing the hashCode  
        System.out.println("HashCode is: " + obj.hashCode());  
        obj = null;  
        // calling the garbage collector using gc()  
        System.gc();  
        System.out.println("End of the garbage collection");  
    }  
    // defining the finalize method  
    protected void finalize()  
    {  
        System.out.println("Called the finalize() method");  
    }  
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java  
Note: FinalizeExample.java uses or overrides a deprecated API.  
Note: Recompile with -Xlint:deprecation for details.  
  
C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample  
HashCode is: 746292446  
End of the garbage collection  
Called the finalize() method
```

By Jagadish Sahoo

Java Custom Exception

- In Java, we can create our own exceptions that are derived classes of the Exception class.
- Creating our own Exception is known as custom exception or user-defined exception.
- Basically, Java custom exceptions are used to customize the exception according to user need.
- Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

Why use custom exceptions?

- Java exceptions cover almost all the general type of exceptions that may occur in the programming.
- However, we sometimes need to create custom exceptions.
 1. To catch and provide specific treatment to a subset of existing Java exceptions.
 2. Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.
- In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

```
public class WrongFileNameException extends Exception {  
    public WrongFileNameException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

```
// class representing custom exception
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{
    // method to check the age
    static void validate (int age) throws InvalidAgeException{
        if(age < 18){
            // throw an object of user defined exception
            throw new InvalidAgeException("age is not valid to vote");
        }
        else {
            System.out.println("welcome to vote");
        }
    }
}
```

By Jagadish Sahoo

```
// main method
public static void main(String args[])
{
    try
    {
        // calling the method
        validate(13);
    }
    catch (InvalidAgeException ex)
    {
        System.out.println("Caught the exception");
        // printing the message from InvalidAgeException object
        System.out.println("Exception occurred: " + ex);
    }
}
```

```
System.out.println("rest of the code...");
```

```
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java
C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occurred: InvalidAgeException: age is not valid to vote
rest of the code...
```

By Jagadish Sahoo

```
// class representing custom exception
class MyCustomException extends Exception
{
}

// class that uses custom exception MyCustomException
public class TestCustomException2
{
    // main method
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new MyCustomException();
        }
        catch (MyCustomException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
        System.out.println("rest of the code...");
    }
}
```

The constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java
C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
By Jagadish Sahoo
```