

# OBJECT ORIENTED PROGRAMMING

## Lecture #1: Introduction

by

Mrs. Akhila Pragada

JAVA IS \_\_\_\_\_

simple programming language

one of the most popular programming language

open-source and free

has a huge community support (tens of millions of developers)

close to C & C++

it helps to create modular programs and reusable code

write a program once and then run this program on multiple operating  
systems

works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)

Java is a high-level, object-oriented programming language

## JAVA IS USED

Mobile applications (specially Android apps)

Desktop applications

Web applications

Gaming Applications

Cloud-based Applications

Software Tools

Database connection

And much, much more!



## COMPANIES THAT USE JAVA



## JAVA HISTORY

Developed by Sun Microsystems Inc in 1991

Initially named as Oak, later changed to Java

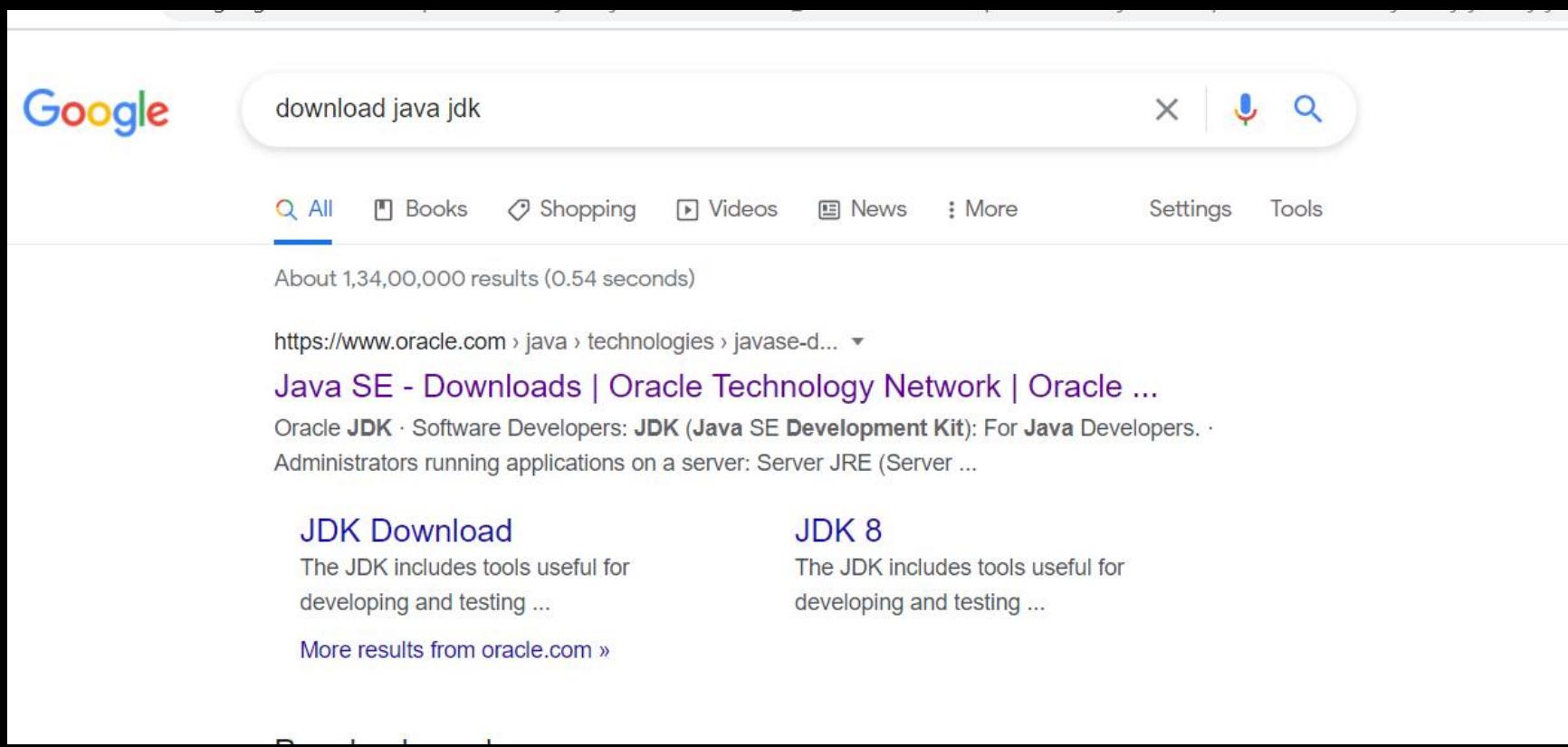


First publicly available version – Java 1.0 released in 1995

Oracle corporation acquired Sun Microsystems in 2010

The latest version of Java is Java 14 or **JDK** 14 released on March, 17th 2020

# INSTALLATION



A screenshot of a Google search results page. The search query "download java jdk" is entered in the search bar. The "All" tab is selected, showing approximately 1,34,00,000 results in 0.54 seconds. The top result is a link to Oracle's Java SE Downloads page, titled "Java SE - Downloads | Oracle Technology Network | Oracle ...". Below the link, a snippet of text from the page describes the Oracle JDK as a tool for Java Developers and Administrators. Two additional links are shown below the main result: "JDK Download" and "JDK 8", each with a brief description of what the JDK includes.

download java jdk

All Books Shopping Videos News More Settings Tools

About 1,34,00,000 results (0.54 seconds)

[https://www.oracle.com › java › technologies › javase-d...](https://www.oracle.com/java/technologies/javase-downloads.html)

**Java SE - Downloads | Oracle Technology Network | Oracle ...**

Oracle **JDK** · Software Developers: **JDK (Java SE Development Kit)**: For Java Developers. ·  
Administrators running applications on a server: Server JRE (Server ...)

**JDK Download**  
The JDK includes tools useful for developing and testing ...  
[More results from oracle.com »](#)

**JDK 8**  
The JDK includes tools useful for developing and testing ...

# INSTALLATION

**ORACLE**

Java / Technical Details / Java SE /  
Java SE Downloads

**Java SE Downloads**

Java Platform, Standard Edition

**Java SE 16**

Java SE 16 is the latest release for the Java SE Platform

- Documentation
- Installation Instructions
- Release Notes
- Oracle License
  - Binary License
  - Documentation License
- Java SE Licensing Information User Manual
  - Includes Third Party Licenses
- Certified System Configurations

**Oracle JDK**

 [JDK Download](#)

 [Documentation Download](#)

## Java SE Development Kit 16 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

### Important Oracle JDK License Update

**The Oracle JDK License has changed for releases starting April 16, 2019.**

The new [Oracle Technology Network License Agreement for Oracle Java SE](#) is substantially different from prior Oracle JDK licenses. The new license permits certain uses, such as personal use and development use, at no cost -- but other uses authorized under prior Oracle JDK licenses may no longer be available. Please review the terms carefully before downloading and using this product. An FAQ is available [here](#).

Commercial license and support is available with a low cost [Java SE Subscription](#).

Oracle also provides the latest OpenJDK release under the open source [GPL License](#) at [jdk.java.net](#).

See also:

[Java Developer Newsletter](#): From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.

[Java Platform, Standard Edition](#) | [Java Platform, Enterprise Edition](#)

# INSTALLATION

Linux ARM 64 RPM Package	144.84 MB	 <a href="#">jdk-16_linux-aarch64_bin.rpm</a>
Linux ARM 64 Compressed Archive	160.69 MB	 <a href="#">jdk-16_linux-aarch64_bin.tar.gz</a>
Linux x64 Debian Package	146.14 MB	 <a href="#">jdk-16_linux-x64_bin.deb</a>
Linux x64 RPM Package	152.96 MB	 <a href="#">jdk-16_linux-x64_bin.rpm</a>
Linux x64 Compressed Archive	170 MB	 <a href="#">jdk-16_linux-x64_bin.tar.gz</a>
macOS Installer	166.56 MB	 <a href="#">jdk-16_osx-x64_bin.dmg</a>
macOS Compressed Archive	167.16 MB	 <a href="#">jdk-16_osx-x64_bin.tar.gz</a>
Windows x64 Installer	150.55 MB	 <a href="#">jdk-16_windows-x64_bin.exe</a>
Windows x64 Compressed Archive	168.74 MB	 <a href="#">jdk-16_windows-x64_bin.zip</a>

## INSTALLATION

X

You must accept the [Oracle Technology Network License Agreement](#) for Oracle Java SE to download this software.

- I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE

[Download jdk-16\\_windows-x64\\_bin.exe](#) 

# OBJECT ORIENTED PROGRAMMING

## Lecture #2: POP vs OOP

by

Mrs. Akhila Pragada

## **PROGRAMMING LANGUAGES**

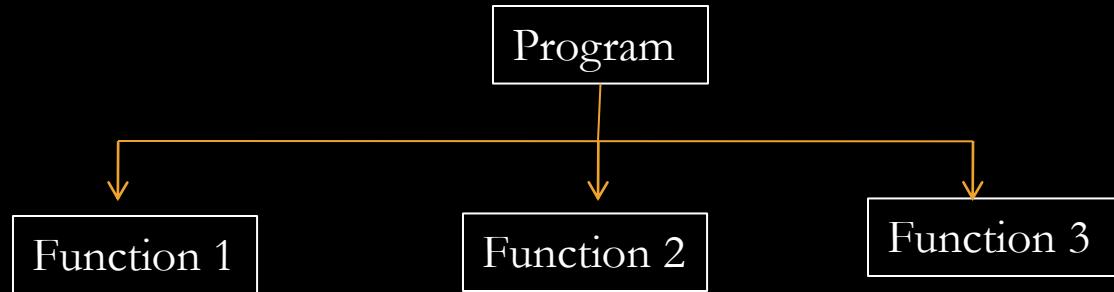
### Programming languages

Procedure Oriented languages

Object Oriented languages

## PROCEDURE ORIENTED PROGRAMMING (POP)

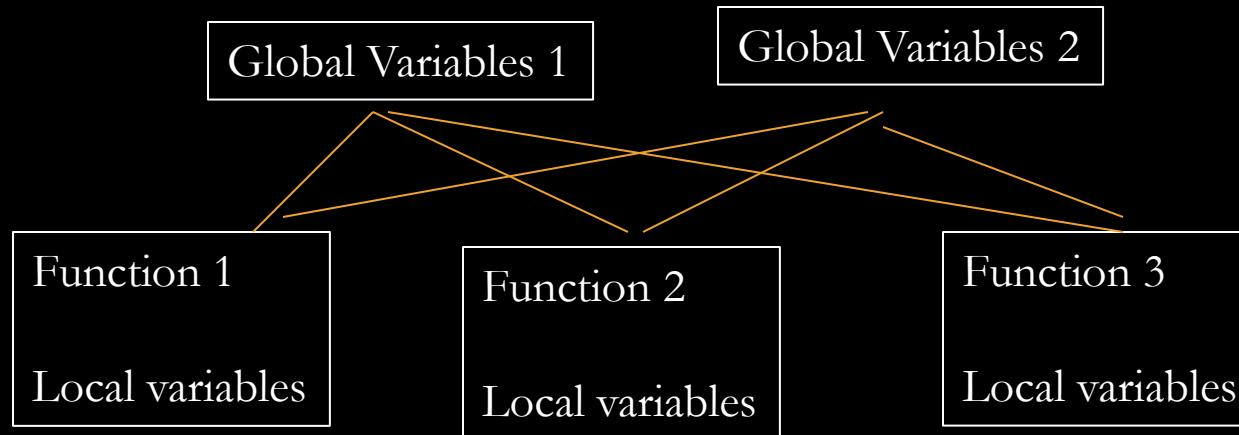
- Large Programs are divided into sequence of functions (Procedures)



- Each procedure can be called by another procedure during execution by calling it using its name
- In POP main focus is on procedures only not on data
- Follows “Top-Down Approach”.

## PROCEDURE ORIENTED PROGRAMMING (POP)

- Global variables can be accessed by all functions

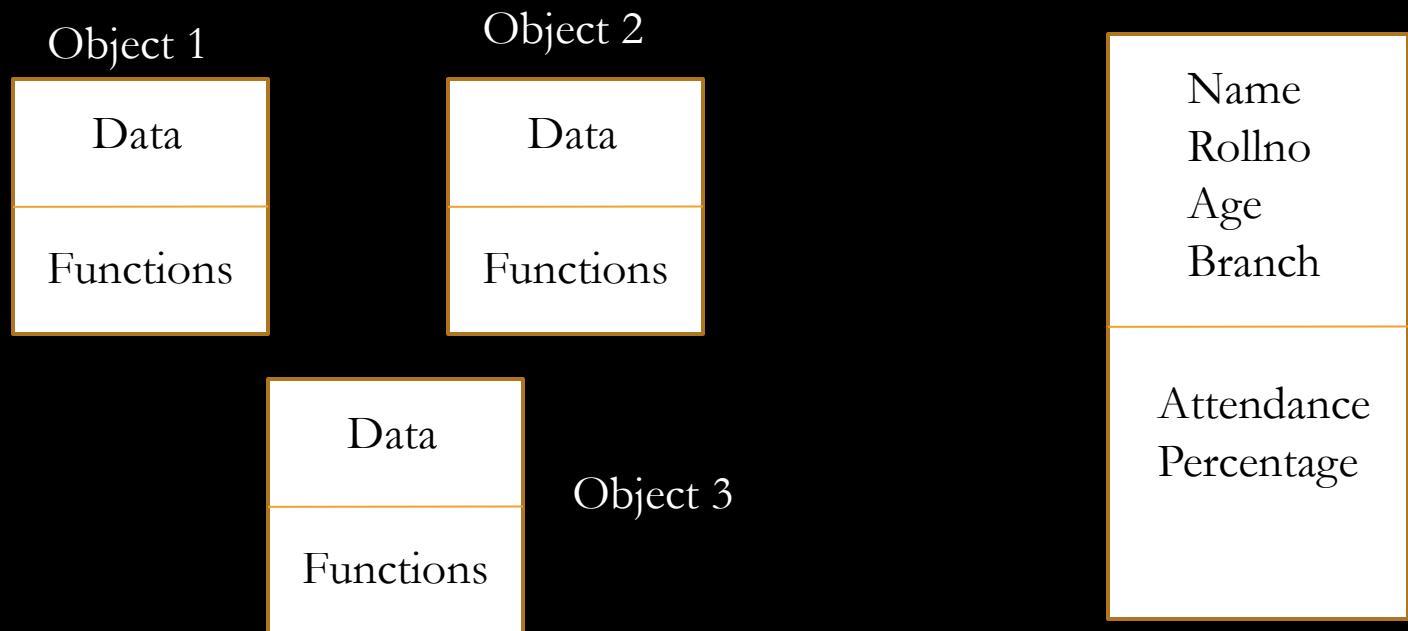


- No security to data - Data can be freely moved from one procedure to another procedure
- If we need to change the type of data of global variable then we need to revise all the functions that are using this global variables. Failing this may result in errors
- POP does not model real world problems very well. Because functions are action oriented
- Examples: C, PASCAL, COBOL, BASIC, FORTRAN

# OBJECT ORIENTED PROGRAMMING (OOP)

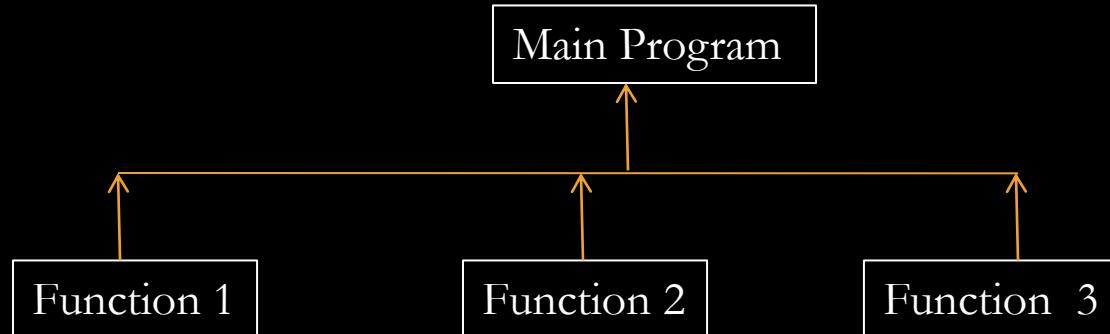
- In OOP data is secured -does not allow it to flow freely among functions
- Program is divided into objects and then built functions and data around these objects.

Student Object



## OBJECT ORIENTED PROGRAMMING (OOP)

- Data of an object is accessed only by the functions associated by that object
- Objects communicate with each other through functions
- In OOP emphasis is given to data rather than procedures
- OOP follows “Bottom-Up Approach”



Examples: Java, VB.Net, C#

## POP VS OOP

### Procedure Oriented Programming

- Program is divided into smaller programs called Procedures
- Top-Down Approach
- Importance is given to procedures rather than data
- No security to data
- Does not model real-world problems very well
- Suitable for developing medium size projects

### Object Oriented Programming

- Program is divided into entities called Objects
- Bottom-Up Approach
- Importance is given to data rather than Procedures/functions
- Data is Secure
- Model real-world problems
- Suitable to developing large size projects

## C VS JAVA

C

- Procedure Oriented Programming Language
- Complied
- Supports Preprocessor Directives
- User takes care of memory management
- Uses pointers
- Platform dependent
- Variable declaration at the beginning of the block
- Uses global variable

Java

- Object Oriented Programming Language
- Complied & Interpreted
- Does not support preprocessor directives
- Internally manages the memory
- Does not use pointers
- Platform independent
- Variable can be declared anywhere
- No global variable

# OBJECT ORIENTED PROGRAMMING

## Lecture #3: Features of Java

## CHARACTERISTICS OF OOP

- Object
- Class
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

## CLASS & OBJECT

A **class** is a data type that allows programmers to create objects. A class provides a definition for an object, describing an object's attributes (data) and methods (operations).

An object is an *instance* of a class. With one class, you can have as many objects as required.

This is analogous to a variable and a data type, the class is the data type and the object is the variable.

## DATA ABSTRACTION

**Data abstraction** is the process of hiding certain details and showing only essential information to the user.

## ABSTRACTION

### Abstraction

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only important things to the user and hides the internal details.
- For example sending SMS, you just type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

## ENCAPSULATION

**Encapsulation** is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

## ENCAPSULATION

### Encapsulation

- Encapsulation in java is a *process of wrapping code and data together into a single unit*, for example capsule i.e. mixed of several medicines.
- We can create a fully encapsulated class in java by making all the data members of the class private.
- Advantages:
  - By providing only setter or getter method, you can make the class read-only or write-only.
  - It provides you the control over the data.
- The Java Bean class is the example of fully encapsulated class.

## INHERITANCE

**Inheritance** can be defined as the process where one class acquires the properties (methods and fields) of another class.

# INHERITANCE..

- ❖ As the name suggests, inheritance means to take something that is already made.
- ❖ It is one of the most important feature of Object Oriented Programming.
- ❖ It is the concept that is used for reusability purpose.
- ❖ Inheritance is the mechanism through which we can derived classes from other classes.
- ❖ The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class.

## POLYMORPHISM

An operation exhibits different behaviors in different instances

# What is Polymorphism?

- Generally, polymorphism refers to the ability to appear in many forms
- Polymorphism in a Java program
  - The ability of a reference variable to change behavior according to what object instance it is holding.
  - This allows multiple objects of different subclasses to be treated as objects of a single super class, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to



# Method Binding

- Determining the method to execute in response to a message.
- Binding can be accomplished either statically or dynamically.

## Static Binding –

- Also known as “*Early Binding*”.
- Resolved at compile time.
- Resolution based on static type of the object(s).

## Dynamic Binding –

- Also known as “*Late Binding*”.
- Resolved at run-time.
- Resolution based on the dynamic type of the object(s).
- Uses method dispatch table or Virtual function table.

## DYNAMIC BINDING

**Dynamic binding** also called **dynamic** dispatch is the process of linking procedure call to a specific sequence of code (method) at run-time.

It **means** that the code to be executed for a specific procedure call is not known until run-time.

**Dynamic binding** is also known as late **binding** or run-time **binding**.

## MESSAGE PASSING

**Message Passing** is nothing but sending and receiving of information by the objects same as people exchange information.

## **FEATURES OF JAVA**

- Object
- Class
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing
- Platform independent
- Compiled and interpreted
- Robust and secure
- Distributed
- Multi-threading
- Dynamic and extensible

## JAVA BUZZWORDS

Platform independent

“Write once and run anywhere”

Secure

Based on public key encryption and helps to design virus free systems

Architectural and Neutral

Ability of generating Architectural-neutral objects that can run on multiple processors

Portable

Uses a compiler that is written using ANSI C and POSIX sub suit making it portable

Robust

Eliminates the most number of errors in the compile time itself

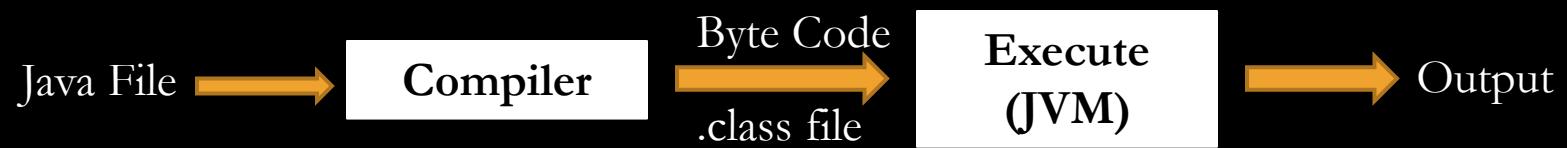
Multi-Threading

Capable to execute code simultaneously above multiple processors at a time.

# OBJECT ORIENTED PROGRAMMING

Lecture #4: First Java program

## JAVA VIRTUAL MACHINE (JVM)



## SIMPLE JAVA PROGRAM

```
class classname  
{  
    public static void main (String a[])  
    {  
        Block of statements  
    }  
}
```

```
class First  
{  
    public static void main (String a[])  
    {  
        System.out.print("First java program");  
    }  
}
```

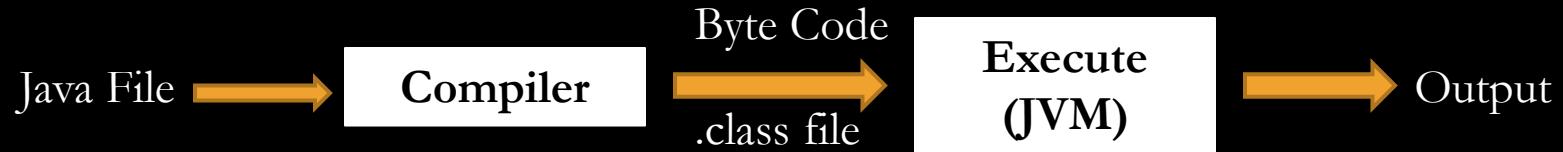
### Note:

- Java is case sensitive
- class name first letter should be capital
- Every statement should end with semicolon

### Keywords:

class, public, static, void

## COMPILING AND EXECUTING



**Saving:**

First.java

**Compiling:**

javac First.java

**Execution:**

java First

## OBJECT ORIENTED PROGRAMMING

Lecture #5: Compiling & Executing Java program  
(Setting classpath)

## SIMPLE JAVA PROGRAM

```
class First
{
    public static void main (String a[])
    {
        System.out.print("First java program");
    }
}
```

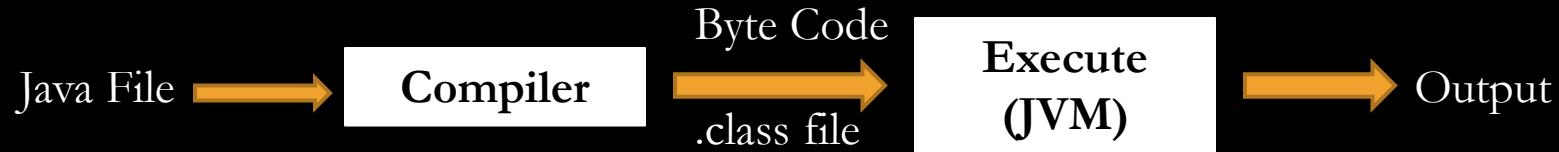
### Note:

- Java is case sensitive
- class name first letter should be capital
- Every statement should end with semicolon

### keywords:

class, public, static, void

## COMPILING AND EXECUTING



**Saving:**

First.java

**Compiling:**

javac First.java

**Execution:**

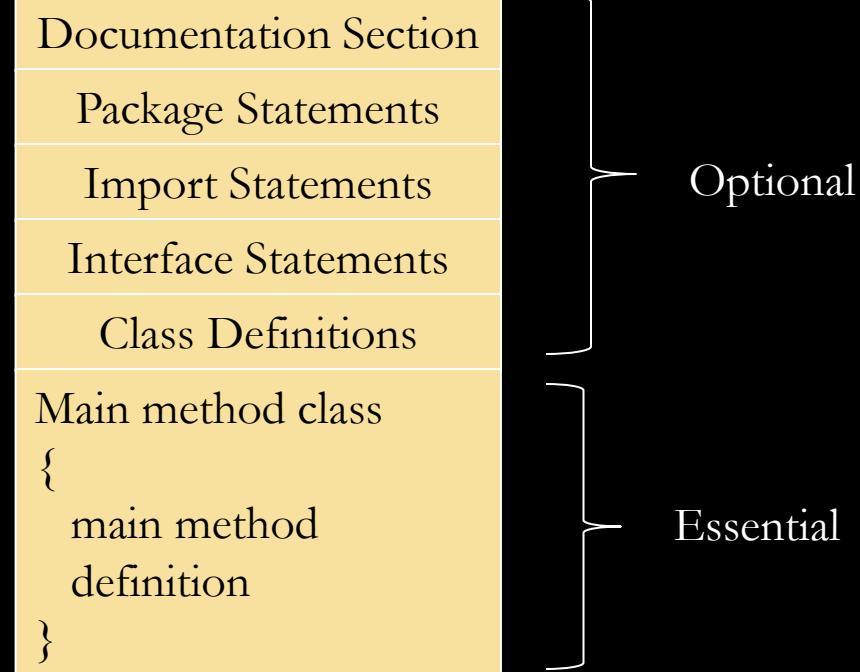
java First



# OBJECT ORIENTED PROGRAMMING

## Lecture #6: Structure of Java Program

# STRUCTURE OF JAVA PROGRAM



## DOCUMENTATION SECTION

- Used for writing comment lines
- Ignored by compiler

3 ways:

➤    //

This is for single line Ex: //Welcome to Java program

➤    /\*       \*/

This is for multiple lines Ex: /\* Welcome to  
Java Program \*/

➤    /\*\*      \*/

This is used to generate automatic comment

## PACKAGE STATEMENT

**Java packages** are a mechanism to group **Java** classes that are related to each other, into the same "group" (**package**)

Ex: package Employee

## IMPORT STATEMENTS

To **import java** package into a class, we need to use **java import** keyword which is used to access package and its classes into the **java** program

Can **import** built-in and user-defined packages

Ex: import java.lang.Math.\*;

Import java.util.\*;

## INTERFACE STATEMENTS

Used to achieve multiple inheritance

## CLASS DEFINITIONS

```
Class Definitions
Main method class
{
    main method
    definition
}
```

The diagram illustrates the structure of a Java class definition. It shows a yellow rectangular box containing the code: 'Class Definitions', 'Main method class', '{', 'main method', 'definition', and '}'. A brace on the right side of the box groups the first two lines ('Main method class' and '{') under the label 'Optional'. A larger brace groups the entire block from 'Main method class' to '}' under the label 'Essential'.

Any number of classes can be defined

All the objects of class should be created in main method.

One class should have main method—Main method class

# OBJECT ORIENTED PROGRAMMING

## Lecture #7: Variables & Datatypes

## VARIABLE

A **Java variable** is a piece of memory that can contain a data value.

**Variables** are typically used to store information

### Variable Declaration:

Syntax:

datatype variable name;

Example:

int a;

### Variable Assignment:

Assigning a value to a variable

Syntax:

variable name=value;

Example:

a=10;

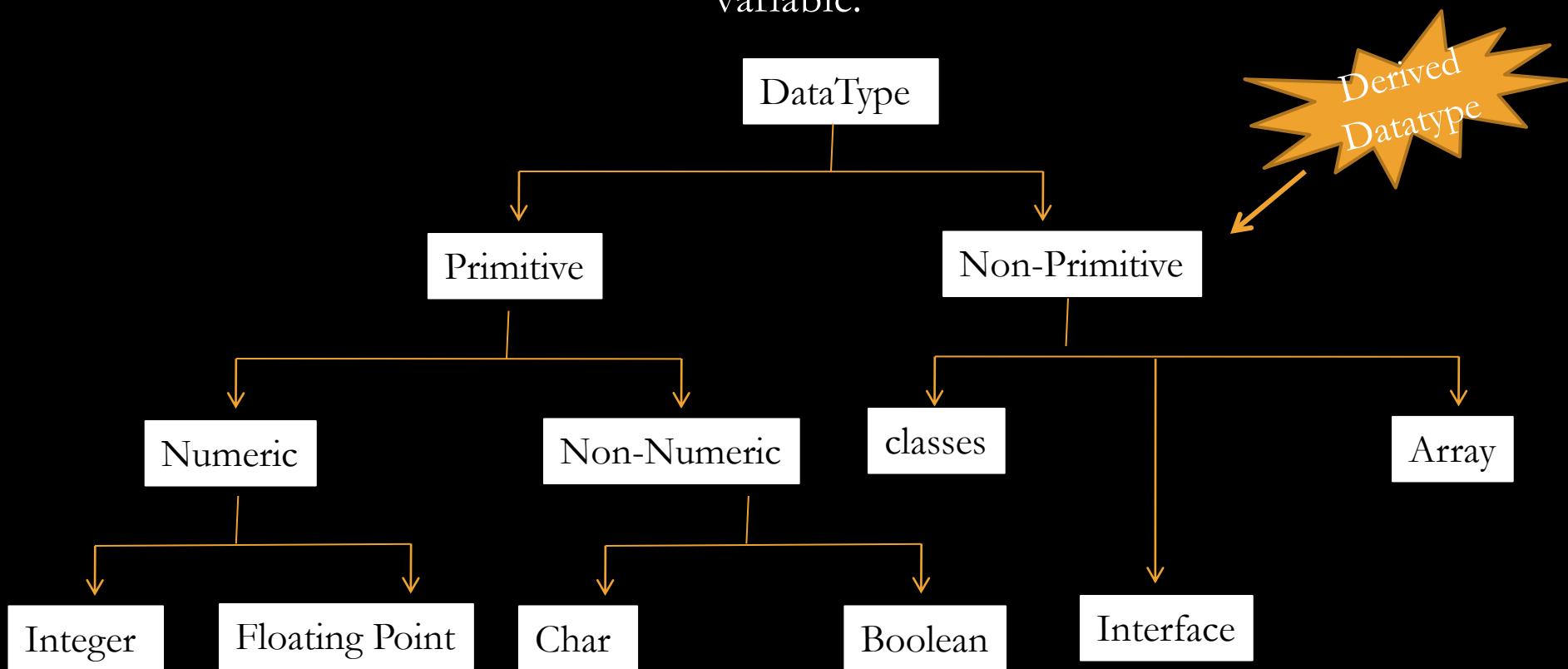
## VARIABLE

### Variable Naming Conventions:

- Java variable names are case sensitive. The variable name apple is not the same as Apple or APPLE.
- Java variable names must start with a letter, or the \$ or \_ character.
- After the first character in a Java variable name, the name can also contain numbers (in addition to letters, the \$, and the \_ character).
- Variable names cannot be equal to reserved key words in Java. For instance, the words int or for are reserved words in Java. Therefore you cannot name your variables int or fo

## DATATYPES IN JAVA

**Data type** specifies the size and type of values that can be stored in an variable.



## INTEGER

Integer types can hold whole numbers such as 123 and -96. The size of the values that can be stored depends on the integer type that we choose.

Type	Size	Range of values that can be stored
byte	1 byte	-128 to 127
short	2 bytes	-32768 to 32767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

1 byte = 8 bits

Example:  
int a;  
byte b;  
a=10;

The range of values is calculated as  $-(2^{n-1})$  to  $(2^{n-1})-1$ ; where n is the number of bits required.

## FLOATING POINT

Floating point data types are used to represent numbers with a fractional part.

There are two subtypes:

Type	Size	Range of values that can be stored
<b>float</b>	4 byte	3.4e-038 to 3.4e+038
<b>double</b>	8 bytes	1.7e-308 to 1.7e+038

Example:

```
float a;  
double b;  
a=4.7;
```

## CHARACTER

It stores character constants in the memory

It assumes a size of 2 bytes

Basically it can hold only a single character

Keyword: char

Keep the value in single quote.

Example:

```
char a;  
char b;  
a='k';  
b='2';
```

## **BOOLEAN**

Boolean data types are used to store values with two states: true or false.

Its size is 1 bit

Keyword: boolean

Example:

```
boolean a;
```

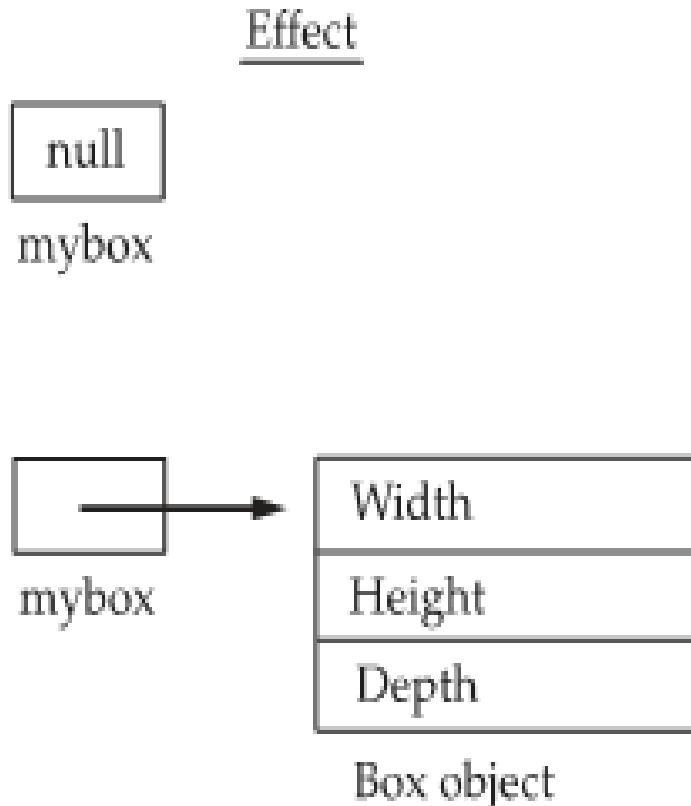
## OBJECT DECLARATION

**FIGURE 6-1**  
Declaring an object  
of type Box

Statement

```
Box mybox;
```

```
mybox = new Box();
```

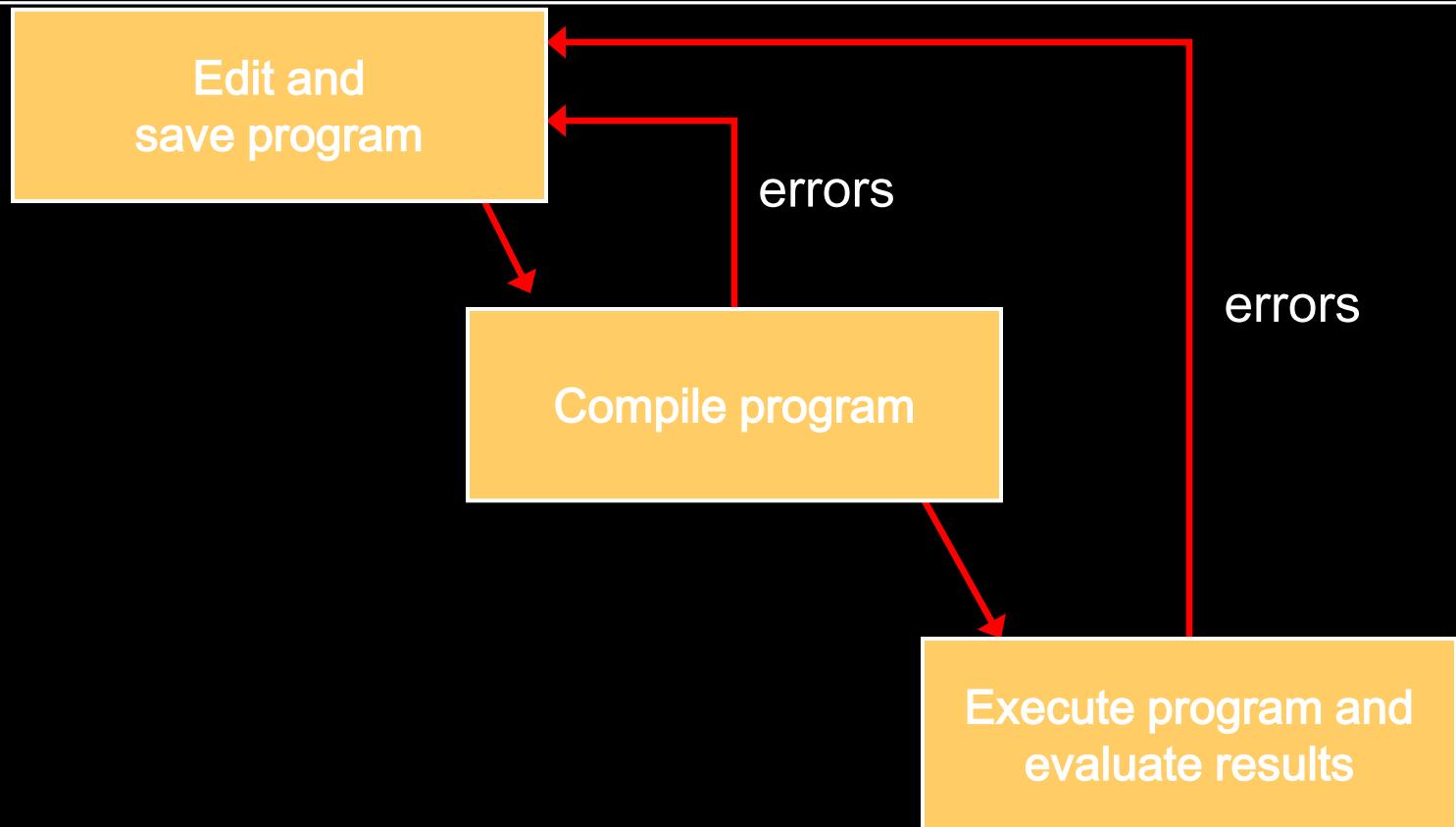


## EXAMPLE PROGRAM

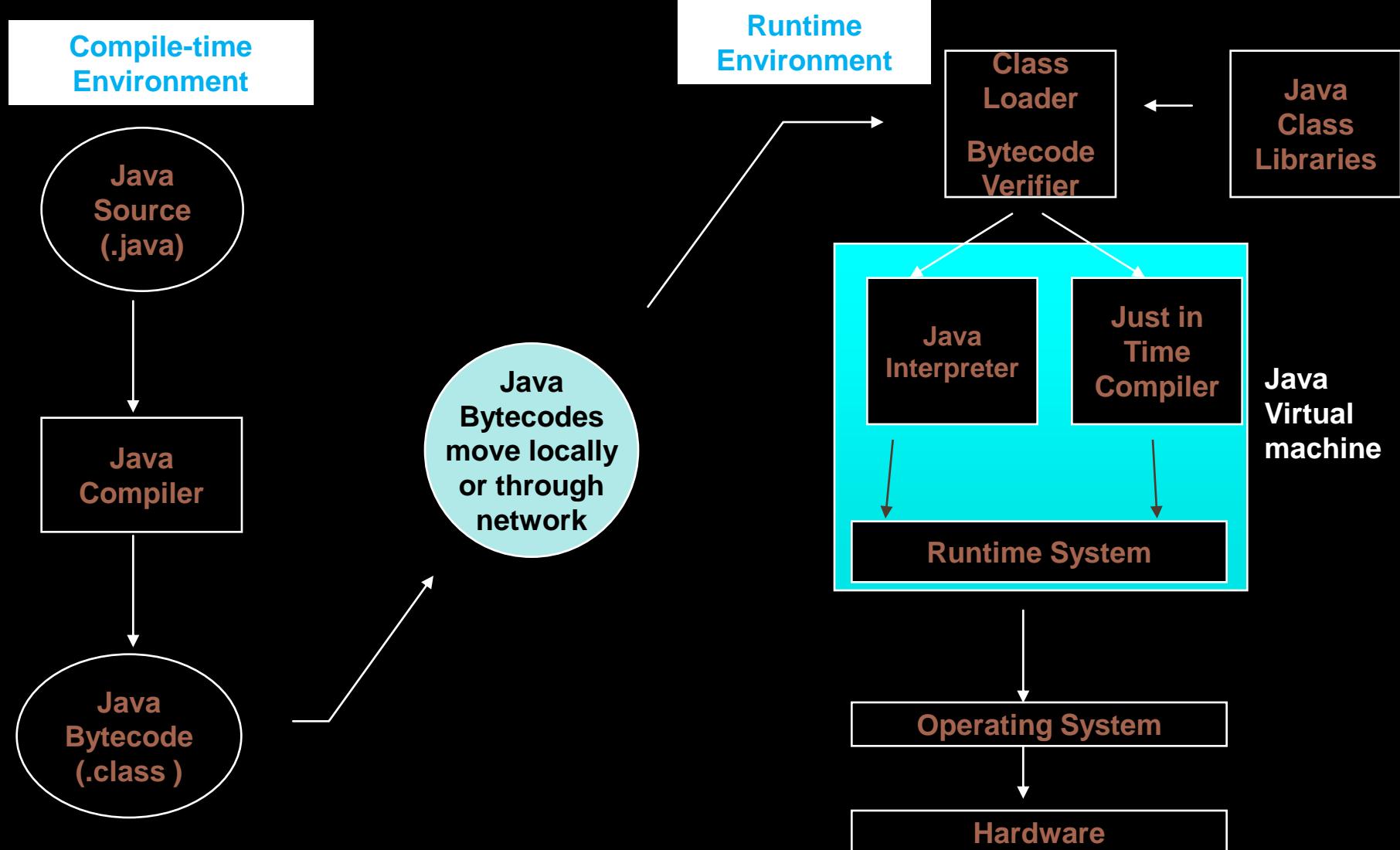
### Example of Class

```
class Student{  
    int age;  
    String name;  
    void printData(){  
        System.out.println("Name:"+name+"\nAge:"+age);  
    }  
    public static void main(String args[]){  
        Student st=new Student();  
        st.name="John";  
        st.age=22;  
        st.printData();  
    }  
}
```

## BASIC PROGRAM DEVELOPMENT

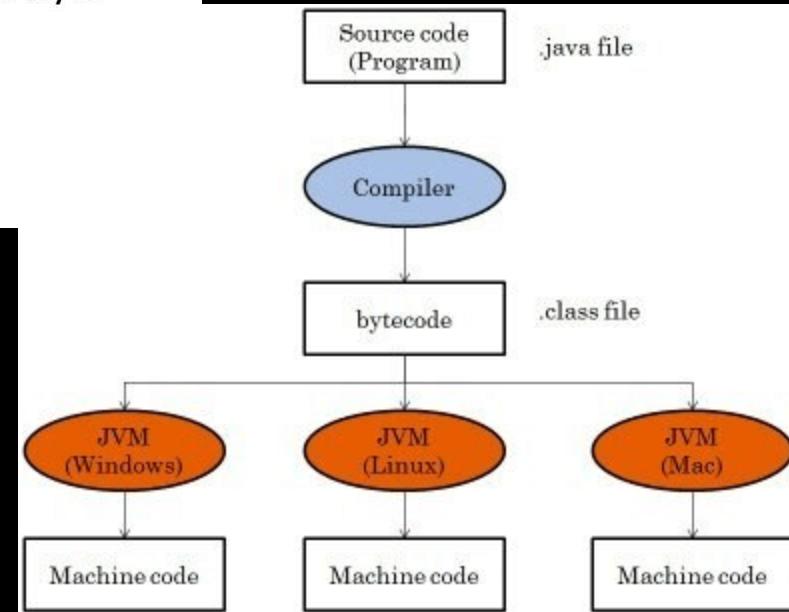


# Java Environment/ Life Cycle of Java Code

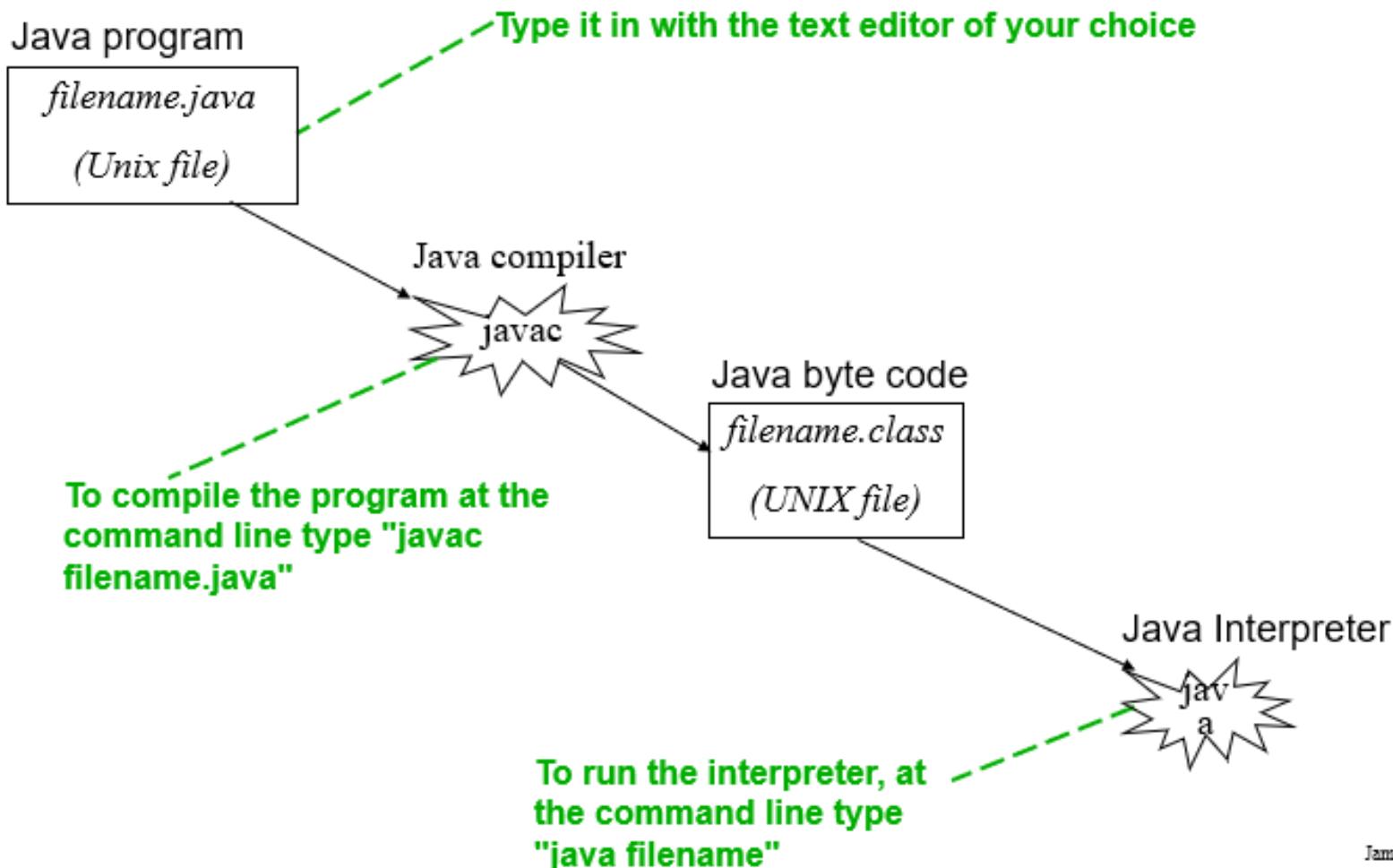


## Java Bytecode

- `javac` compiles your java source code into an intermediate set of instruction called the bytecode.
- JVM (Java Virtual Machine) is a machine that interfaces with the hardware to run the byte code instructions.



# CREATING, COMPILING AND RUNNING JAVA PROGRAMS



## Naming Conventions

**Java Naming Convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.**

Name	Conventions
Class Name	Should start with uppercase letter and be a noun e.g. String, Colour, Button, Thread, etc.
Interface Name	Should start with uppercase letter and be an adjective e.g. Runnable, ActionListener, etc.
Method Name	Should start with lowercase letter and be a verb e.g. actionPerformed(), main(), println(), etc.
Variable Name	Should start with lowercase letter e.g. forstName, orderNumber, etc.
Package Name	Should be in lowercase letter e.g. java, lang, sql, util, etc.
Constant Name	Should be in uppercase letter e.g. RED, YELLOW, etc.

## JAVA KEYWORDS

abstract	else	interface	super
boolean	extends	long	switch
break	false <sup>a</sup>	native	synchronized
byte	final	new	this
case	finally	null <sup>a</sup>	throw
catch	float	package	throws
char	for	private	transient
class	goto <sup>b</sup>	protected	true <sup>a</sup>
const <sup>b</sup>	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while
double	int	strictfp <sup>c</sup>	

**Note.** <sup>a</sup> true, false, and null are reserved words.

<sup>b</sup>indicates a keyword that is not currently used. <sup>c</sup>indicates a keyword that was added for Java 2

## OPERATOR EVALUATION

Parentheses have the highest precedence and it can be used to force an expression to evaluate in order you want.

(3-1) is 2 and  $(1+1)^{**}(5-2)$  is 8. you can also use parentheses to make an expression easier to read, as in  $(\text{min}^{*}100)/60$ , even though it does not change the result.

Exponentiation has the next highest precedence, so  $2^{**}1+1$  is 3 and not 4 and  $3*1^{**}3$  is 3 and not 27.

Multiplication and division have the same precedence which is higher than addition and subtraction which also has the same precedence. So  $2*3-1=5$  and  $2/3-1=-1$ (integer division,  $2/3=0$ ).

Operators with the same precedence are evaluated from left to right. So in the expression  $\text{min}^{*}100/60$ , the multiplication operation takes first place, yielding  $5900/60$  which in turn yields 98.

If the operations had been evaluated from right to left, the result would have been  $59*1$  that produce the result as 59 which is wrong

## EXAMPLE:

$12+20*2-4*2/2.0 \ # 48.0$

1.  $12+40-4*2/2.0$

2.  $12+40-8/2.0$

3.  $12+40-4.0$

4.  $52-4.0$

5.  $48.0$

$12+(20*2)-4*(2/2.0) \ # 48.0$

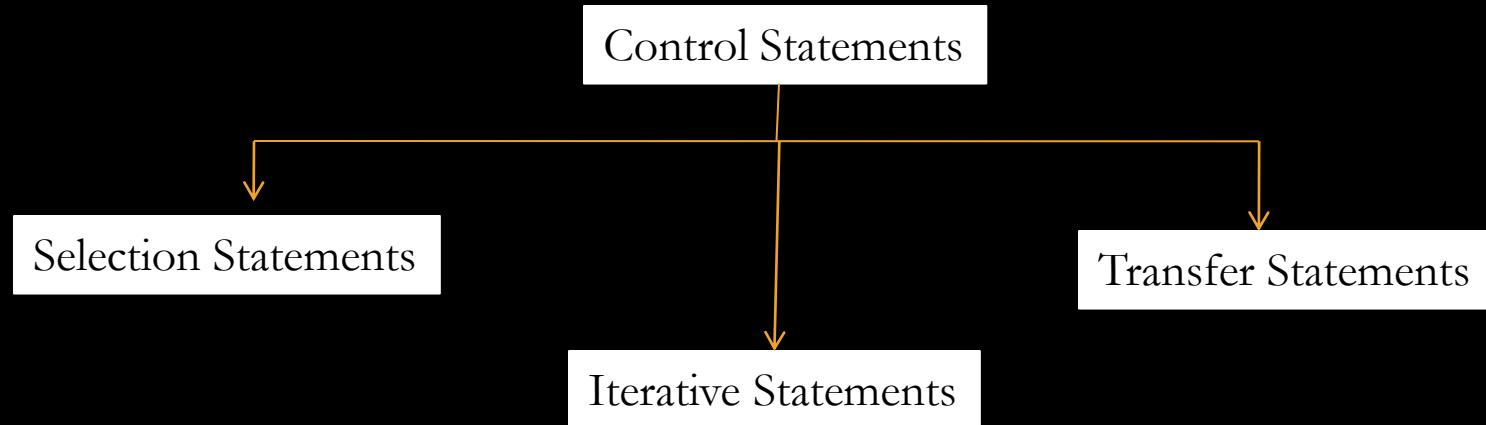
$12+(20*2)-(4*2)/2.0 \ # 48.0$

expression is evaluated from left -> right. BODMAS(bracket of division multiplication) rule is followed while evaluating the expression.

## **OBJECT ORIENTED PROGRAMMING**

Lecture #8: Control Statements in Java  
(Part-1)

# CONTROL STATEMENTS



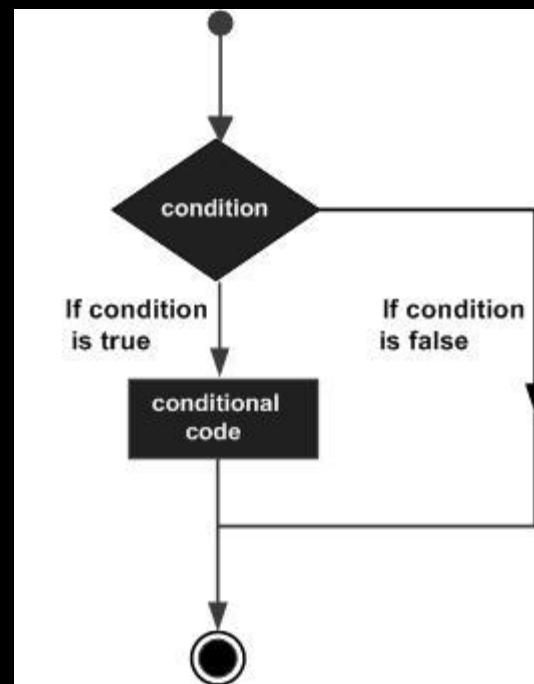
## SELECTION STATEMENTS (if)

### “Simple if”

Syntax:

```
if(condition)
{
    block1- stmt1
    block1-stmt2
}
```

Flow Chart



Example

```
//checking the age
if(age>18)
    System.out.print("Age > 18");
```

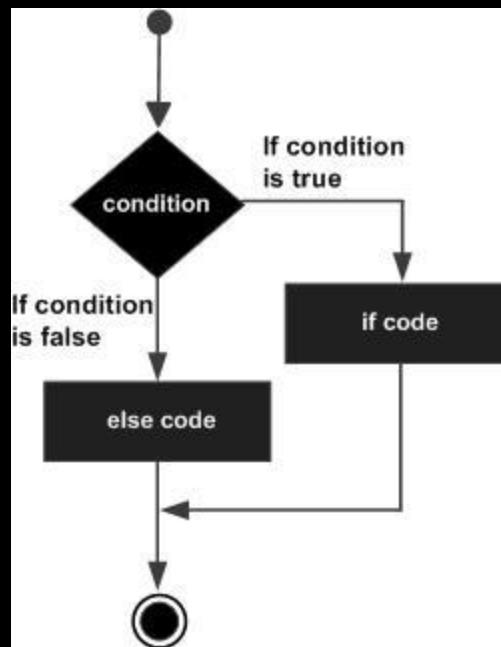
## SELECTION STATEMENTS (if)

### “if-else Statement”

Syntax:

```
if(condition)
{
    block1-stmt1
    block1-stmt2
}
else
{
    block2-stmt1
    block2-stmt2
}
```

Flow Chart



Example

```
if( x < 20 )
{
    System.out.print("Yes");
}
else
{
    System.out.print("NO");
}
```

## SELECTION STATEMENTS (if)

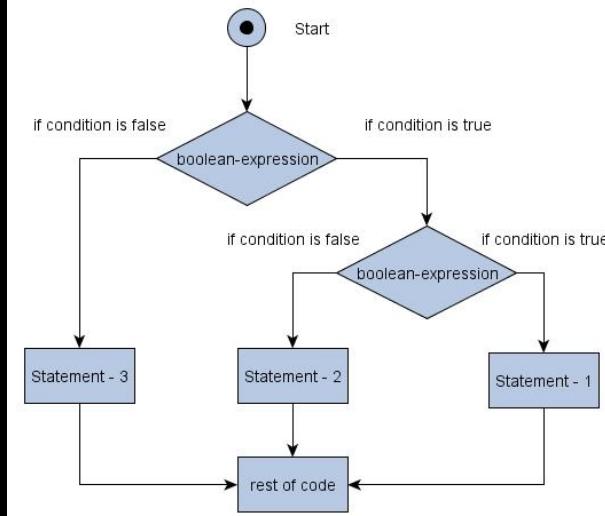
### “Nested if-else Statement”

Syntax:

```
if(condition)
{
    if(condition)
    {
        block1-stmt1
        block1-stmt2
    }
    else
    {
        block2-stmt1
        block2-stmt2
    }
}

else
{
    block3-stmt1
    block3-stmt2
}
```

Flow Chart



Example

```
if( a>b)
{
    if(a >c)
        System.out.print("A is big");
    else
        System.out.print("b is big");
}
else
{
    if(b >c)
        System.out.print("B is big");
    else
        System.out.print("b is big");
}
```

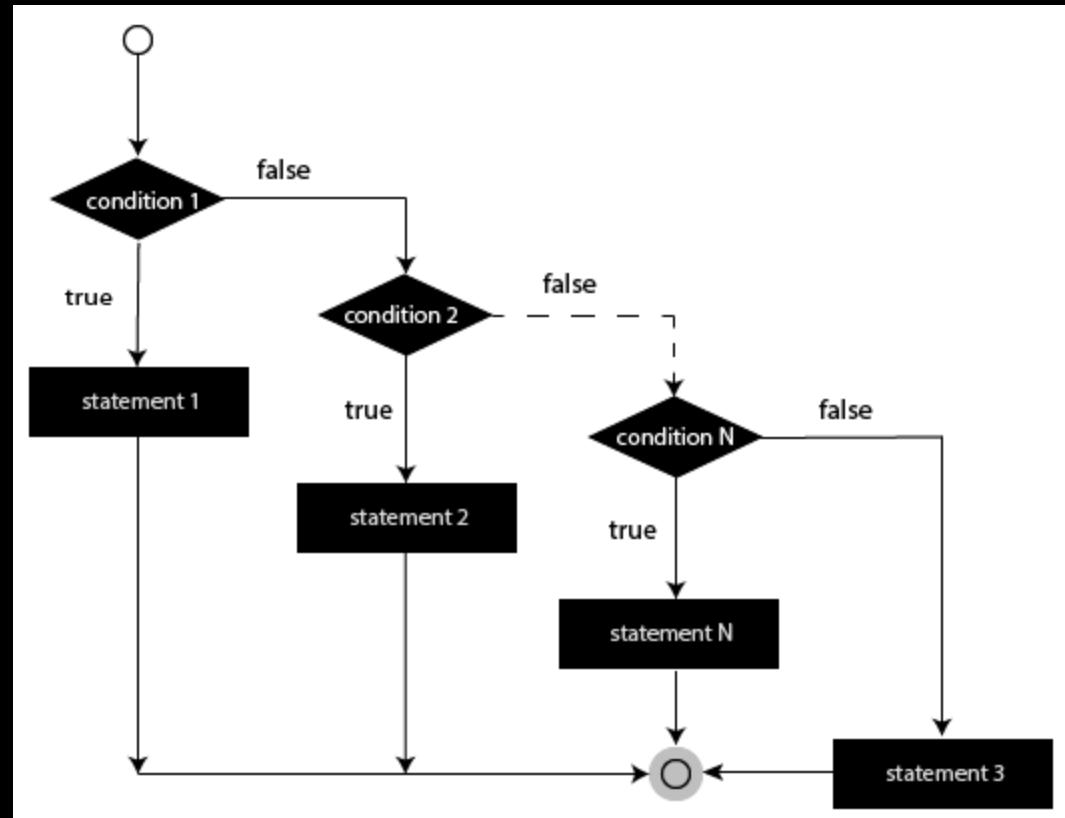
## SELECTION STATEMENTS (if)

### “else-if Ladder Statement”

Syntax:

```
if(condition)
{
    block1-stmt1
    block1-stmt2
}
else
{
    if(condition)
    {
        block2-stmt1
        block2-stmt2
    }
    else
    {
        block3-stmt1
        block3-stmt2
    }
}
```

Flow Chart



## SELECTION STATEMENTS (switch)

### “switch Statement”

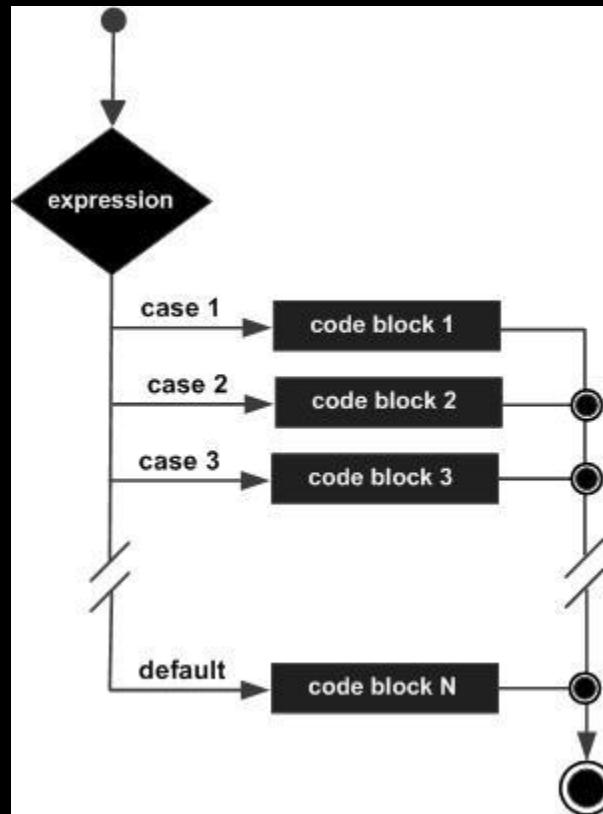
Syntax:

```
switch(expression)
{
    case value :
        // Statements
        break; // optional
    case value :
        // Statements
        break; // optional

    // You can have any
    number of case
    statements.

    default : // Optional
        // Statements
}
```

Flow Chart



Example

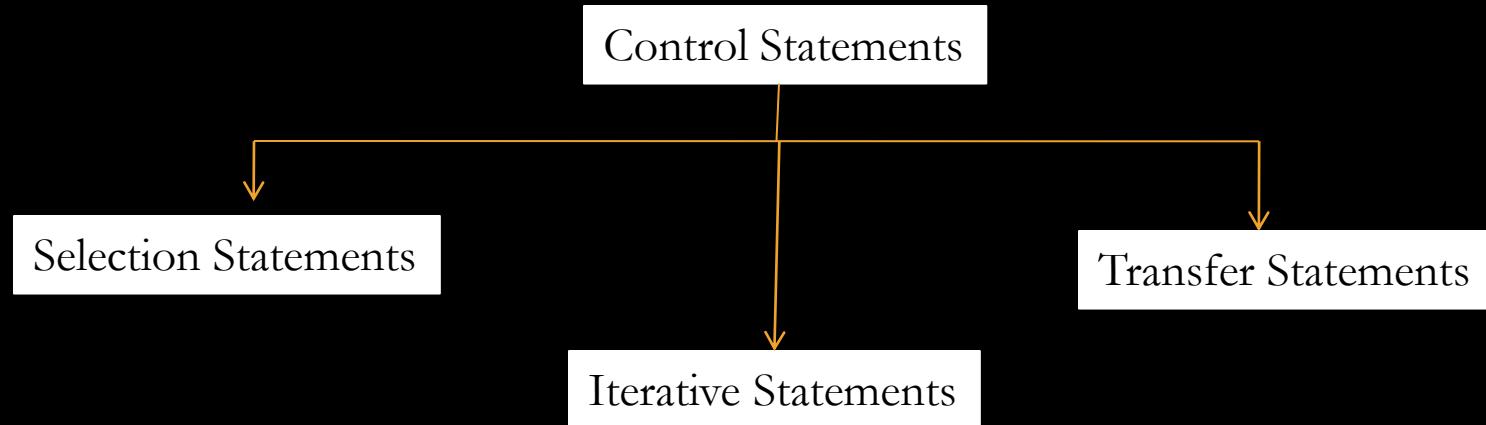
```
switch(grade)
{
    case 'A' :
        System.out.println("Excellent!");
        break;
    case 'B' :
        System.out.println("Well done");
        break;
    case 'C' :
        System.out.println("Passed");
        break;
    case 'D' :
        System.out.println("Fail");
        break;
    default :
        System.out.println("Invalid grade");
}
```

## OBJECT ORIENTED PROGRAMMING

# Lecture #9: Control Statements in Java (Part-2)

by

# CONTROL STATEMENTS



## ITERATIVE STATEMENTS (while)

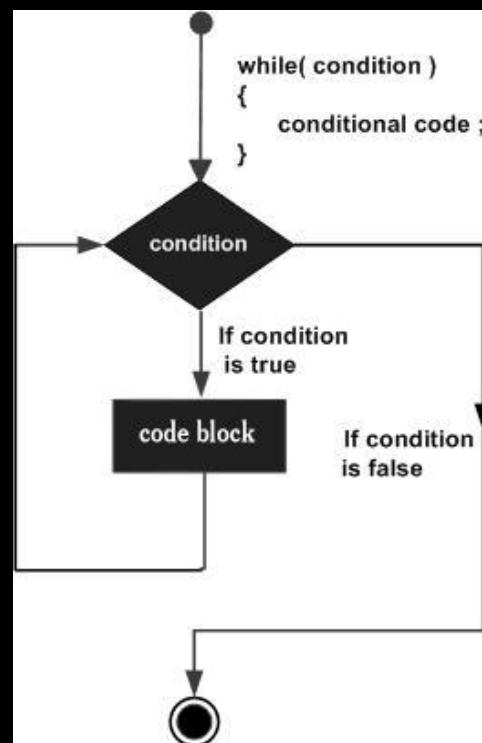
“while”

Repeatedly executes the block of statements as long as the condition is true  
Entry-controlled Loop

Syntax:

```
while(expression)
{
    block1- stmt1
    block1-stmt2
}
```

FlowChart



Example:

```
x=0
while( x<10)
{
    System.out.println(x);
    x=x+1;
}
print("End of while")
```

## ITERATIVE STATEMENTS (do-while)

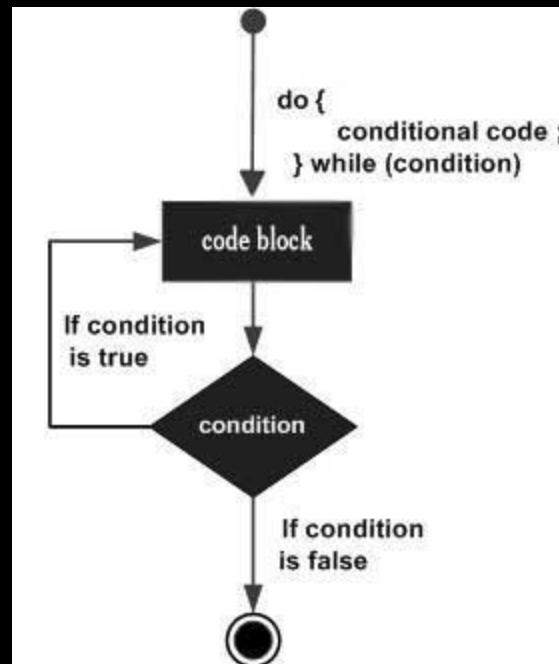
“do-while”

Repeatedly executes the block of statements until the condition becomes False  
Exit-controlled Loop

Syntax:

```
do
{
    block1- stmt1
    block1-stmt2
}
while(expression);
```

FlowChart



Example:

```
x=0
do
{
    System.out.println(x);
    x=x+1;
}
while( x<10);
print("End of do-while")
```

## ITERATIVE STATEMENTS (for)

**“for”**

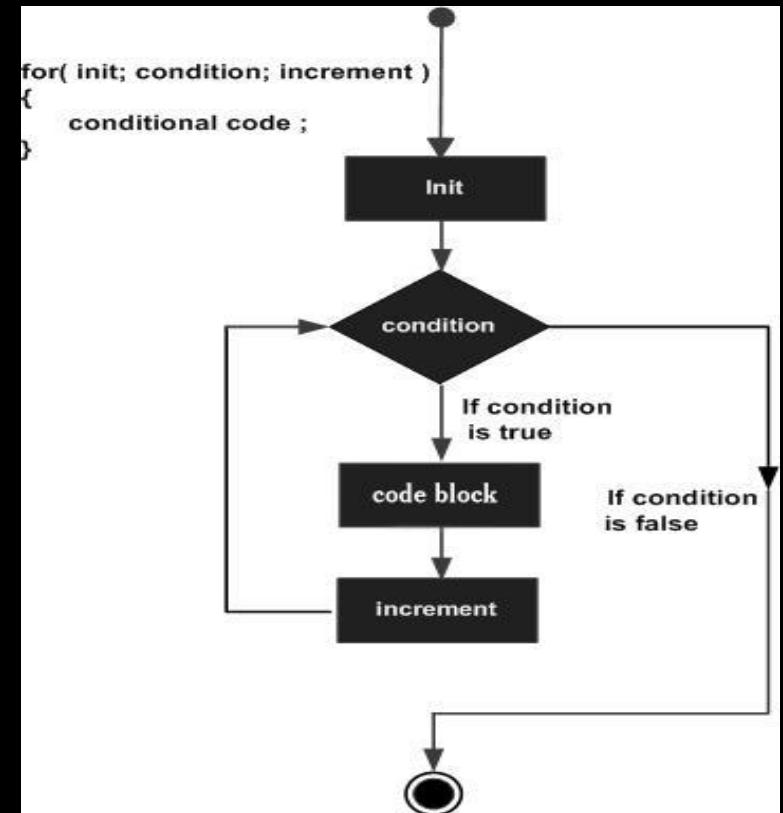
Syntax:

```
for(initialization; Boolean_expression; update)
{
// Statements
}
```

Example:

```
for(int x = 10; x < 20; x = x + 1)
{
    System.out.print("value of x :" + x );
}
```

Flowchart



## TRANSFER STATEMENTS (break)

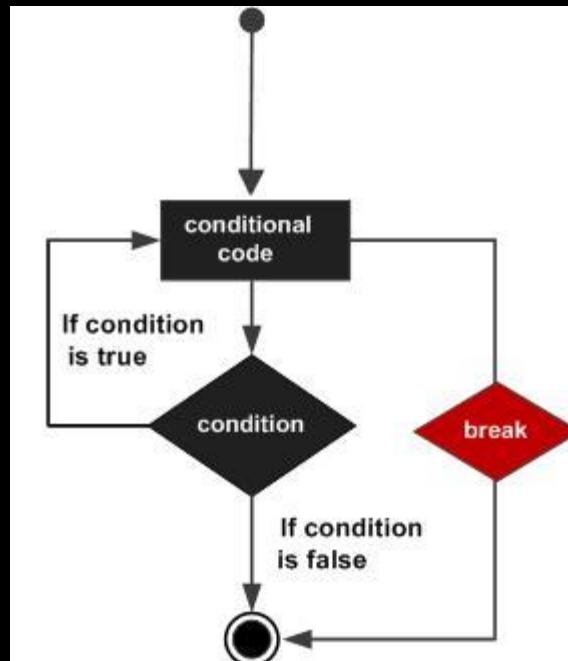
### **“break”**

When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

Flowchart

Syntax:

break;



Example:

```
for(int i = 1; i < 5; i ++)
{
    if( i == 3 )
        break;
    System.out.println( i );
}
```

Output:

```
1
2
```

## TRANSFER STATEMENTS

### (continue)

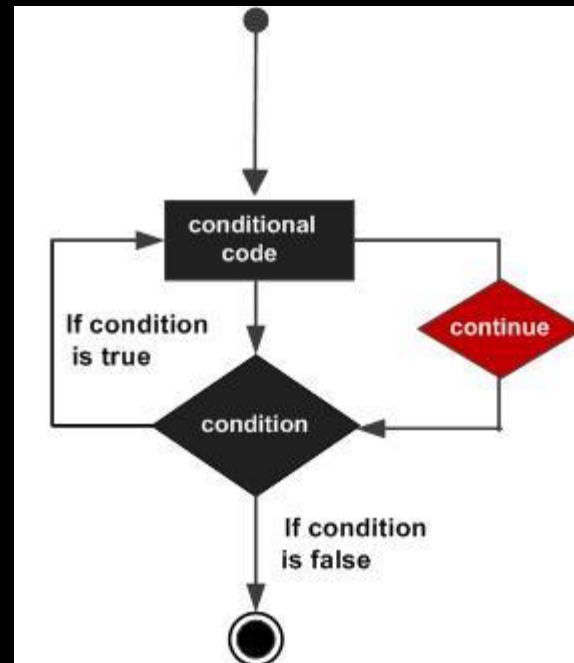
#### “continue”

In a loop, the continue keyword causes control to immediately jump to the next iteration.

Flowchart

Syntax:

continue;



Example:

```
for(int i = 1; i < 5; i ++)
{
    if( i == 3 )
        continue;
    System.out.println( i );
}
```

Output:

1  
2  
4

## PACKAGE DETAILS

### Java user input(Scanner)

The Scanner class is used to get user input and it is found in the java.util package.

To use the scanner class, create an object of the class and use any of the available methods found in the scanner class documentation.

There are some input types in Scanner package

nextBoolean() – reads a Boolean value from the user

nextByte() – reads a byte value from the user

nextDouble() - reads a double value from the user

nextFloat() – reads a float value from the user

nextInt() – reads a int value from the user

nextLine() - reads a string from the user

nextLong() – reads a long value from the user

nextShort() – reads a short value from the user

## JAVA LANG PACKAGE

For writing any java program, the most commonly required classes and interfaces are encapsulated in the separate packages which is nothing but `java.lang` package. It is not required to import `java` package in our program because it is available by default to every java program.

The following are the some of the important classes present in `java.lang` package.

1. Object
2. String
3. String buffer
4. All wrapper classes
5. Exception API
6. Thread API...etc.

## JAVA.LANG.OBJECT CLASS

For any java object whether it is predefined or customized the most commonly required methods are encapsulated into a separate class which is nothing but object class.

As object class act as a root (or) parent (or) super for all java classes , by default its methods are available to every java class.

The following are the list of all methods present in java.lang object class

1. public String toString()
2. public native int hashCode()
3. public boolean equals(Object o);
4. protected native Object clone() throws CloneNotSupportedException;
5. public final Class<?> getClass();
6. protected void finalize () throws Throwable;
7. public final void wait() throws InteruptedException;
8. public final void wait(long ms , int ns) throws InteruptedException;
9. public final native void notify();
10. public final native void notifyAll();

## ACCESS MODIFIERS

Access modifiers in java specifies the accessibility or scope of the field, method, constructor, or a class.

We can change the access levels fields by applying the modifiers in it.

There are 4 access modifiers in java.

Public- for every where.

Private – only within the class. It cannot be accessed from the outside of the class.

Protected- with in the package and outside the package through child class.

Default – only with in the package.

## ACCESS MODIFIERS

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

## TYPE CONVERSION

Converting one data type into another data type is known as type conversion

This is classified into 2 types

1. Automatic/ implicit type conversion
2. Explicit/ manual type conversion

Automatic Type Conversion:

It is possible to convert lower data type into higher data type .

Ex: byte a=5;

int b;

b=a;

This means a is of byte data type and b is int data type. If we write b=a means, a value is stored in b but the value is in int data type.

Explicit Type Conversion:

It is possible to convert higher data type into lower data type. But we may loss data while conversion.

## EXAMPLE

Ex: if float is converted into short data type

Float data type occupies 4 bytes of memory

Short data type occupies 2 bytes of memory

So while converting from higher data type to lower data type we lost some data.

Syntax: var= (target) variable;

Ex: float a;

int x;

x=(int) a;

Which means, a is converted into int data type and the result will be stored in x which is already an integer data type.

So we are compressing the size of our data so this is nothing but type casting.

## **LEXICAL ISSUES**

Java is a collection of

1. White spaces
2. Identifiers
3. Comments
4. Literals
5. Operators
6. Separators
7. Keywords

## WHITE SPACES

Java is a free form language.

We don't need to follow any special indentation rules

Ex: a program can be written in a single line or multi line or in any strange way , as long as there is at least one white space character between each token that is not already separated by an operator or separator.

In java , white space is a space or tab or new line

## SEPARATORS

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a <b>for</b> statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

# OBJECT ORIENTED PROGRAMMING

## Lecture #10: Class & Object

## CLASS DEFINITIONS

```
Class Definitions
Main method class
{
    main method
    definition
}
```

Optional

Essential

Any number of classes can be defined

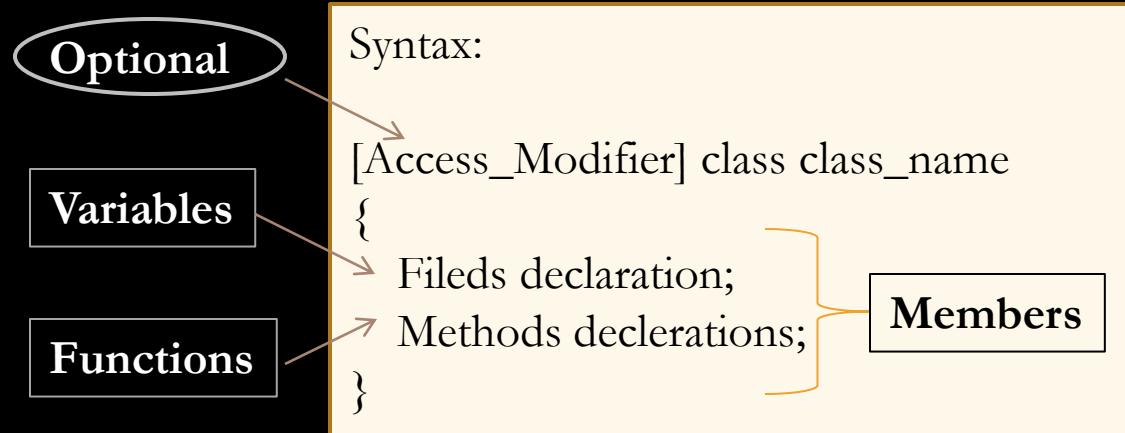
One class should have main method—Main method class

All the objects of class should be created in main method.

# CLASS

Defining a class:

A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.



# CLASS

class\_name:

Example:

```
class Student  
class Example
```

Field declaration:

Syntax:

Optional

[Access Modifier] type Variable\_name;  
[initial value]

Example:

```
int age;  
float marks;  
float studentHeight;
```

Instance  
Variable

Instance  
variables are  
created at the  
time of object  
creation

# CLASS

Method declaration:

Syntax:

[Access Modifier] **returntype identifier (Paramenter\_list)**  
    {  
        method\_body  
    }

Optional

Example:

```
int add( int x, float b)
{
    statements
}
```

Example:

```
int getStudentData( int x, float b)
{
    statements
}
```

## CLASS- EXAMPLE

Example:

```
class Student
{
    int a,b;
    int add( int x, float b)
    {
        statements
    }
}
```

# OBJECT

An object is an instance of a class.

To create an object “new” operator is used

Object creation:

2 parts-

- Declare the object
- Instantiate the object

Object creation:

```
Student obj;  
obj=new Student();
```

Object creation:

```
Student obj=new Student();
```

Example:

```
class Student  
{  
    int a,b;  
    int add( int x, float b)  
    {  
        statements  
    }  
}
```

# OBJECT

Accessing class members

using .(dot) Operator

Syntax:

```
objectname.variable_name=value;  
objectname.methodname(parameter_list);
```

Example:

```
obj.a=10;  
obj.add(25,33.5);
```

Class creation:

```
class Student  
{  
    int a,b;  
    int add( int x, float b)  
    {  
        statements  
    }  
}
```

Object creation:

```
Student obj=new Student();
```

# OBJECT

Multiple objects can be created from a single class

Class creation:

```
class Student  
{  
    int rollno;  
    int add( int x, float b)  
    {  
        statements  
    }  
}
```

Object creation:

```
Student s1=new Student();  
Student s2=new Student();  
Student s3=new Student();  
s1.rollno=12;  
s2.rollno=23;  
s3.rollno=45;
```

## PROGRAM WITH MULTIPLE CLASSES

```
class Example
{
    int a,b,c;
    void getData(int x, int y)
    {
        a=x;
        b=y;
    }
    void add()
    {
        c=a+b;
        System.out.print("Sum is :"+c);
    }
}
```

```
class Addition
{
    public static void main(String args[])
    {
        Example obj1=new Example();
        obj1.getData(25,25);
        obj1.add();
    }
}
```

Output:  
Sum is :50

## PROGRAM WITH MULTIPLE CLASSES

```
class Example
{
    int a,b,c;
    void getData(int x, int y)
    {
        a=x;
        b=y;
    }
    void add()
    {
        c=a+b;
        System.out.println("Sum is :"+c);
    }
}
```

```
Private class Addition
{
    public static void main(String args[])
    {
        Example obj1=new Example();
        obj1.getData(25,25);
        obj1.add();

        Example obj2=new Example();
        obj2.getData(5,2);
        obj2.add();
    }
}
```

Output:  
Sum is :50  
Sum is :7

## **OBJECT ORIENTED PROGRAMMING**

Lecture #11: Array in Java-Introduction

## WHY ARRAY?

Consider :

```
int a;  
a=10;  
a=20;  
System.out.print("a= "+a);
```

Output:  
a= 20

Consider :

```
int a;  
a=10;  
System.out.print("a= "+a);  
b=20;  
System.out.print("b= "+b);
```

Output:  
a= 10  
b=20

Array is used to hold more than one value with same variable name

A **Variable** will hold only **one value** at a time

## ARRAY

**Java array** is an object which contains elements of a similar data type.  
The elements of an array are stored in a contiguous memory location.

3 Steps:

- Declaration of array
- Instantiate the array
- Initialization of array

# ARRAY

## Declaration of Array:

Specifying Array name and datatype

2 forms

Syntax(Form-1):

```
type array_name[ ];
```

Syntax(Form-2):

```
type[ ] array_name;
```

Example(Form-1):

```
int roll_no[ ];
```

Example(Form-2):

```
int[ ] roll_no;
```

Don't Mention the  
size

# ARRAY

## Instantiate the Array:

Allocating the memory to the array  
uses “new” operator

Example:

```
int roll_no[ ];
```

Syntax:

```
array_name= new type[ size];
```

Example:

```
roll_no=new int[5 ];
```

## Combining Declaration & Instantiate the Array:

Syntax:

```
type array_name[ ]= new type[ size];
```

Example:

```
int roll_no[ ]=new int[5 ];
```

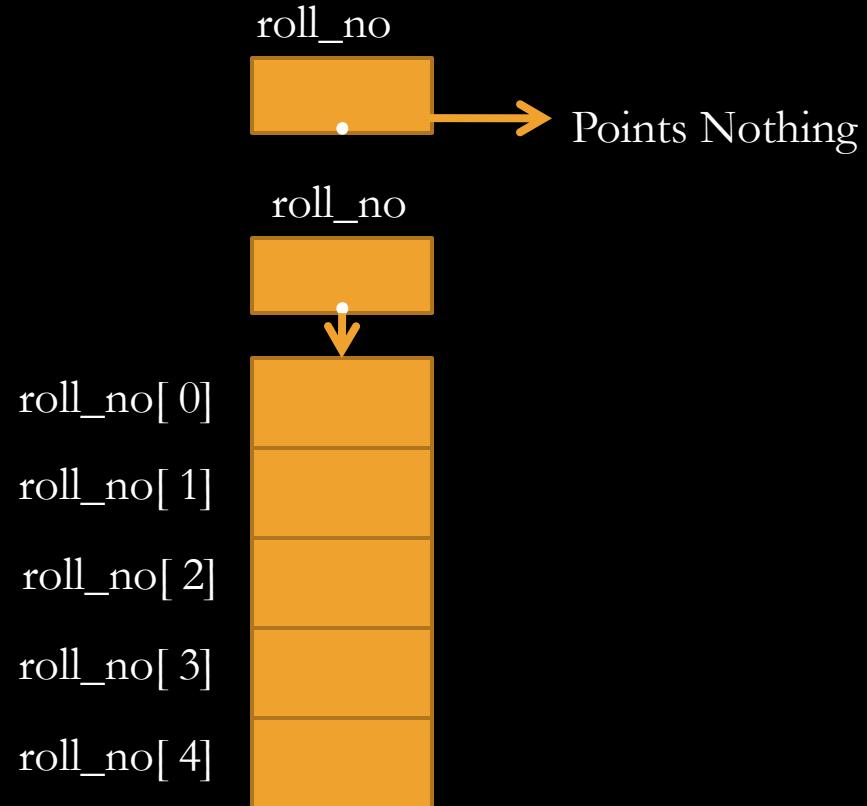
## ARRAY

Memory Allocation:

```
int roll_no[];
```

```
roll_no=new int[5];
```

Array in Java is index-based,  
the first element of the array is  
stored at the 0<sup>th</sup> index,  
2nd element is stored on 1st index  
and so on.



# ARRAY

## Initialization of Array:

Assigning values to the Array.

Example:

```
int roll_no[ ]=new int[5];
```

### 1. Compile-time:

- Assigning individual array elements:

Syntax:

```
array_name[index]= value;
```

Example:

```
roll_no[0]= 10;  
roll_no[1]= 20;
```

- Combining declaration and initialization:

Syntax:

```
type array_name[ ]= {list of values};
```

Example:

```
int roll_no [ ]= {10,20,54,66,12};
```

In this case required memory will be automatically created, no need of “new” operator

# ARRAY

## Initialization of Array:

Reading values from user- **Run-time**

uses Scanner class (import java.util.\*)

- To read a variable value

```
int a;  
Scanner s;  
a=s.nextInt();
```

- To read array values

```
int a[]=new int[5];  
Scanner s;  
for(i=0;i<5;i++)  
{  
    a[i]=s.nextInt();  
}
```

- To print array values

Using for or foreach

```
for(i=0;i<5;i++)  
{  
    System.out.println(a[i]);  
}
```

```
for(int i : a)  
{  
    System.out.println( i);  
}
```

## OBJECT ORIENTED PROGRAMMING

Lecture #12: Two Dimensional Array

## **TYPES OF ARRAY**

**Java array** is an object which contains elements of a similar data type.  
The elements of an array are stored in a contiguous memory location.

3 Steps:

- Declaration of array
- Instantiate the array
- Initialization of array

3 types of Arrays:

1. One-Dimensional Array
2. Two-Dimensional Array
3. Multi-Dimensional Array

## ONE-DIMENSIONAL ARRAY

Combining Declaration & Instantiate the Array:

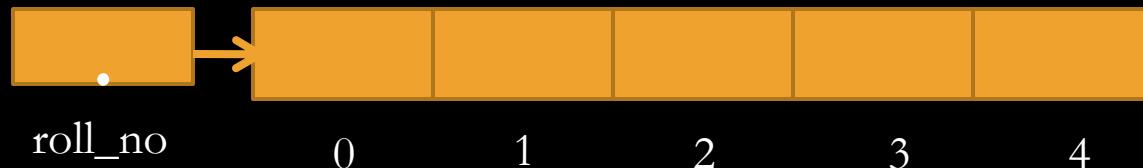
Syntax:

```
type array_name[ ]= new type[ size];
```

Example:

```
int roll_no[ ]=new int[5];
```

Memory Allocation:



Initialization:

Syntax:

```
type array_name[ ]= {list of values};
```

Example:

```
int roll_no [ ]= {10,20,54,66,12};
```

```
int a[] = new int[5];
```

```
Scanner s = new Scanner(System.in);
```

```
for(i=0;i<5;i++)
```

```
{
```

```
    a[i]=s.nextInt();
```

```
}
```

## TWO-DIMENSIONAL ARRAY

Used to store more than one value

Used to represent the data in the form of matrix or table

Declaration of Array:

Specifying Array name and data-type

Syntax:

```
type array_name[ ][ ];
```

Example:

```
int a[ ][];
```

Instantiate the Array:

Allocating the memory to the array

uses “new” operator

Syntax:

```
array_name= new type[ r-size][c-size];
```

Example:

```
a=new int[3][4];
```

Syntax:

```
type array_name[ ][ ]= new type [ r-size][c-size];
```

Example:

```
int a[ ][ ] = new int[3][4];;
```

## TWO-DIMENSIONAL ARRAY

Memory Allocation:

Example:

```
int a[ ][ ] = new int[3][4];;
```

Individual element is accessed:

a[r-index][c-index]

First element: a[0][0]

Second element: a[0][1]

Third element: a[0][2]

12<sup>th</sup> element: a[2][3]

	0	1	2	3
0				
1				
2				

## TWO-DIMENSIONAL ARRAY

### Initialization of Array:

Assigning values to the Array.

#### 1. Compile-time:

- Assigning individual array elements:

Syntax:

```
array_name[r-index][c-index]= value;
```

Example:

```
int a[ ][ ]=new int[3 ][4];
```

Example:

```
a[0][0]= 10;  
a[0][1]= 20;  
a[1][2]=65;  
a[2][3]=35;
```

## TWO-DIMENSIONAL ARRAY

Combining declaration and initialization:

Syntax:

```
type array_name[r-size][c-size] = {list of values};
```

Example:

```
int a[3][2] = {10,20,54,66,12,77};
```

Example:

```
int a[3][2] = {{10,20},{54,66},{12,77}};
```

In this case required memory will be automatically created, no need of “new” operator

a[0][0] = 10    a[1][0] = 54    a[2][0] = 12  
a[0][1] = 20    a[1][1] = 66    a[2][1] = 77

Values will be  
stored row by  
row

	0	1
0	10	20
1	54	66
2	12	77

## TWO-DIMENSIONAL ARRAY

### Initialization of Array:

Reading values from user- **Run-time**

uses Scanner class (import java.util.\*)

```
int a[][]=new int[3][2];
Scanner s= new Scanner(System.in);
for(i=0;i<3;i++)
{
    for (j=0; j<2;j++)
    {
        a[i][j]=s. nextInt();
    }
}
```

- To print array values

```
for(i=0;i<3;i++)
{
    for (j=0; j<2;j++)
    {
        System.out.println(a[i][j]);
    }
}
```

## OBJECT ORIENTED PROGRAMMING

# Lecture #13: Multi-Dimensional Array and Jagged Arrays

## MULTI-DIMENSIONAL ARRAY

The Multi Dimensional Array in Java programming language is nothing but an Array of Arrays (Having more than one dimension).

Two Dimensional Array, which is the simplest form of Java Multi Dimensional Array. we can declare n-dimensional array or Muti dimensional array by placing n number of brackets [ ], where n is dimension number.

### 3D-Array Declaration:

Syntax:

```
type array_name[ ][ ][ ];
```

Example:

```
int a[ ][ ][ ];
```

### Instantiate the 3D-Array:

Syntax:

```
array_name= new type[Tables][ r-size][c-size];
```

Example:

```
a= new int[2][ 3][4];
```

## MULTI-DIMENSIONAL ARRAY

### Initialization of Array:

Assigning values to the Array.

#### 1. Compile-time:

- Assigning individual array elements:

Example:

```
int a[ ][ ][ ]=new int[2][ 3][4];
```

Syntax:

```
array_name[table_no][r-index][c-index]= value;
```

Example:

```
a[0][0][0]= 10;  
a[0][0][1]= 20;  
a[1][1][2]=65;  
a[1][2][3]=35;
```

## MULTI-DIMENSIONAL ARRAY

Combining declaration and initialization:

Syntax:

```
type array_name[tables][r-size][c-size]= {list of values};
```

Example:

```
int a[2][3][2]= {{ {5,6},{8,12},{7,90}} , {{10,20},{54,66},{12,77}}};
```

In this case required memory will be automatically created, no need of “new” operator

## MULTI-DIMENSIONAL ARRAY

### Initialization of Array:

Reading values from user- **Run-time**  
uses Scanner class (import java.util.\*)

```
int a[][][] = new int[2][3][4];
Scanner s = new Scanner(System.in);
for(i=0;i<2;i++)
{
    for (j=0; j<3;j++)
    {
        for(k=0;k<4;k++)
        {
            a[i][j][k]=s.nextInt();
        }
    }
}
```

- To print array values

```
for(i=0;i<2;i++)
{
    for (j=0; j<3;j++)
    {
        for(k=0;k<4;k++)
        {
            System.out.print(a[i][j][k]);
        }
    }
}
```

## JAGGED ARRAY

Jagged array is a multidimensional array where member arrays are of different size.

For example, we can create a 2D array where first array is of 3 elements, and is of 4 elements.

Program:

```
int a[ ][ ];  
a=new int[3][ ];  
a[0]=new int[4];  
a[1]=new int[2];  
a[2]=new int[3];
```

	0	1	2	3
0				
1				
2				

## **ARRAY PACKAGE**

Array class is included in java.util package

It contains several static methods that you can use to compare, sort and search in arrays.

This class is a member of collection framework.

Different methods of this class are listed below.

Arrays.toString()

Arrays.asList()

Array.sort()

Array.binarySearch()

Array.copyOf()

Array.copyOfRange()

Arrays.fill() ... etc.

## METHODS IN ARRAYS

Arrays.ToString():

It returns the string representation of the array enclosed in the square braces[].

Adjacent elements are separated by the comma character.

Ex: int array[] = {1,2,3,4,5,6,7};

```
System.out.println(Arrays.ToString(array));
```

Arrays.asList():

It returns a list backed by a given array . In other words, both the list and array refer to the same location

Ex: List integerList= Array.asList(array); // returns a fixed size list backed by the specified array.

Arrays.sort():

This method sorts the specified array into ascending numerical order.

```
Arrays.sort(array)
```

## METHODS IN JAVA

Arrays.binarySearch():

It returns an integer value for the index of the specified key in an specified array. It returns a negative number if the key is not found and for this method to work properly , the array must be sorted.

Ex: int idx= Arrays.binarySearch(baseArray,21);

Arrays.copyOf():

This method copies the specified array, truncates or pads with zeros so the copy has the specified length.

Ex: int copyOfArray= Arrays.copyOf(baseArray,11);

# OBJECT ORIENTED PROGRAMMING

## Lecture #14: Strings

## STRINGS

Java string is a sequence of characters. They are objects of type String.

Once a String object is created it cannot be changed. Strings are Immutable.

Creating an empty string.

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

## CREATING STRINGS

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
```

```
String str = new String(data);
```

Construct a string object by passing another string object.

```
String str2 = new String(str);
```

## **STRING METHODS**

- length()**
- charAt()**
- equals()**
- equalsIgnoreCase()**
- startsWith()**
- endsWith()**
- compareTo()**
- compareToIgnoreCase()**
- indexOf()**
- lastIndexOf()**
- substring()**
- concat()**
- replace()**
- toLowerCase**

## STRING METHODS

length():

This method returns the length of the string

```
System.out.println("Hello".length()); //prints 5
```

```
String str1 = "CSE Department"
```

```
int a=str1.length(); //a=14
```

## STRING METHODS

### charAt( ):

Returns the character at the specified index. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

```
char ch;
```

```
ch = "abc".charAt(1); // ch = "b"
```

```
String str1 = "CSE Department"
```

```
char ch1;
```

```
ch1 = str1.charAt(4); // ch = "D"
```

## STRING METHODS

### equals():

Returns true if the invoking string contains the same character sequence.

```
String str1 = "CSE Department";
String str2 = "IT Department";
if(str1.equals(str2))
    System.out.println("str1 and str2 are equal");
else
    System.out.println("str1 and str2 are unequal");
```

## STRING METHODS

equalsIgnoreCase();

Compares this String to another String, ignoring case considerations.

```
String str1 = “CSE DEPARTMENT”;  
String str2 = “cse department”;  
if(str1.equalsIgnoreCase(str2)  
    System.out.println(“str1 and str2 are equal”);  
else  
    System.out.println(“str1 and str2 are unequal”);
```

## STRING METHODS

### StartsWith():

Tests if string starts with the specified prefix

```
“Figure”.startsWith(“Fig”); // true
```

```
String str1 = “CSE DEPARTMENT”;  
if(str1.startsWith(“IT”))  
    System.out.println(“yes it str1 starts with IT”);  
else  
    System.out.println(“No it str1 is not starting with IT”);
```

Similarly endsWith()

# OBJECT ORIENTED PROGRAMMING

Lecture #15: Strings (cont..)

## STRING METHODS

### compareTo():

Compares two strings lexicographically.

The result is a negative integer if this String object lexicographically precedes the argument string.

The result is a positive integer if this String object lexicographically follows the argument string.

The result is zero if the strings are equal.

```
String str1 = "CSE Department";
```

```
String str2 = "IT Department";
```

```
int result;
```

```
result= str1.compareTo(str2);
```

Similarly compareToIgnoreCase()

## STRING METHODS

### indexOf():

Searches for the first occurrence of a character or substring.  
Returns -1 if the character does not occur.

```
String str = "How was your day today?";  
str.indexOf('a'); // 5  
str.indexOf("was"); //4
```

## STRING METHODS

### lastIndexOf():

Searches for the last occurrence of a character or substring.

```
String str = "How was your day today?";
```

```
str.lastIndexOf('a'); // 20
```

## STRING METHODS

### substring():

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

"unhappy".substring(2) returns "happy"

```
String str = "How was your day today?";
```

```
String str2 = str.substring(8); //your day today
```

## STRING METHODS

### concat():

Concatenates the specified string to the end of this string.

"to".concat("get").concat("her") returns "together"

```
String s1="hello";
```

```
String s2="welcome";
```

```
String s3=s1.concat(s2); //hellowelcome
```

## STRING METHODS

### replace():

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

```
replace(char oldChar, char newChar)
```

```
"mesquite in your cellar" .replace('e', 'o')  
returns "mosquito in your collar"
```

## STRING METHODS

**toLowerCase()**: Converts all of the characters in a String to lower case.

**toUpperCase()**: Converts all of the characters in this String to upper case.

```
"HELLO THERE".toLowerCase();
```

```
"hello there".toUpperCase();
```

## EXPERIMENT

Write a Java program to demonstrate String handling methods.

## STRING BUFFER

Java string buffer class is used to create mutable(modifiable ) string.

The string buffer class in java is same as string class except it is mutable. i.e., it can be changed.

`StringBuffer()` – creates an empty string buffer with the initial capacity of 16.

`StringBuffer(String str)` – creates a string buffer with the specified string.

`StringBuffer(int capacity)` – creates an empty string buffer with the specified capacity as length.

`StringBuffer append()` – the `append()` concatenates the given argument with this string.

Ex: `StringBuffer sb=new StringBuffer("hello");`

`sb.append("java");`

`System.out.println(sb); // hellojava`

## STRING BUFFER METHODS

StringBuffer insert() – inserts the given string at the given position.

```
Ex: StringBuffer sb=new StringBuffer("hello");
    sb.insert(1,"java");
    System.out.println(sb); // hjavaello
```

StringBuffer replace() – it replaces the given string from the specified beginIndex and endIndex.

```
Ex: StringBuffer sb=new StringBuffer("hello");
    sb.replace(1,3,"java");
    System.out.println(sb); // hjavalo
```

StringBuffer delete() – the delete() od StringBuffer class deletes the string from the specified beginIndex and endsIndex.

```
Ex: StringBuffer sb=new StringBuffer("hello");
    sb.delete(1,3);
    System.out.println(sb); // hlo
```

## **STRING BUFFER METHODS**

StringBuffer reverse() – the reverse() of StringBuffer class reverse the current string.

```
Ex: StringBuffer sb=new StringBuffer("hello");
    sb.reverse();
    System.out.println(sb); //olleh
```

# OBJECT ORIENTED PROGRAMMING

## Lecture #16: Methods

## METHODS

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.

It provides the reusability of code.

Naming a Method:

Method name must start with a lowercase letter

In the multi-word method name, the first letter of each word must be in uppercase except the first word.

There are two types of methods in Java:

Example:  
check()  
sumOfNumbers()

1. Predefined Method
2. User-defined Method

## **PRE-DEFINED METHOD**

Method that is already defined in the Java class libraries is known as predefined methods.

Also called standard library method or built-in method.

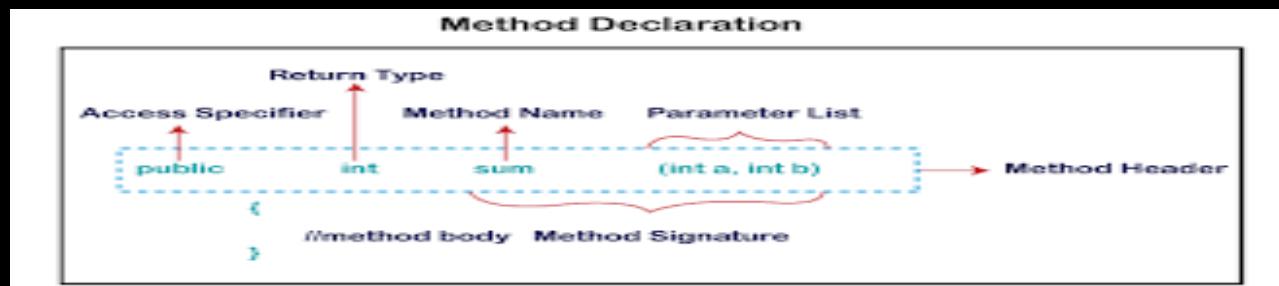
Examples: length(), equals(), compareTo(), sqrt(), etc.

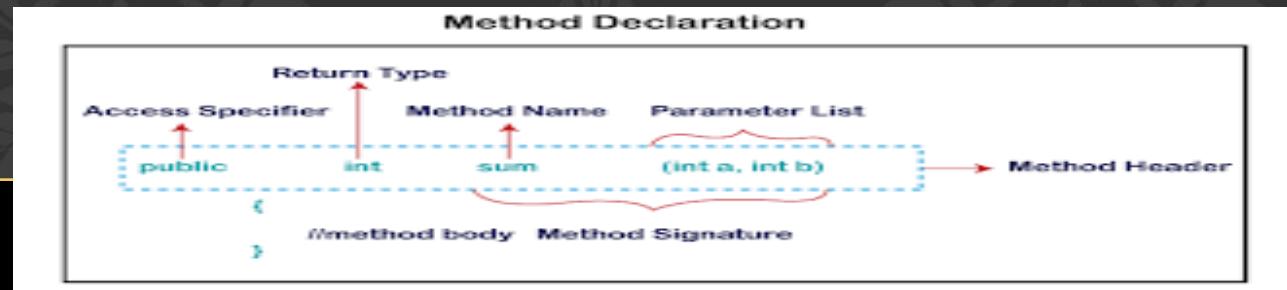
Import related package

## USER-DEFINED METHOD

The method written by the user or programmer is known as a user-defined method

We can modify the method according to our requirement.





## Access Specifier:

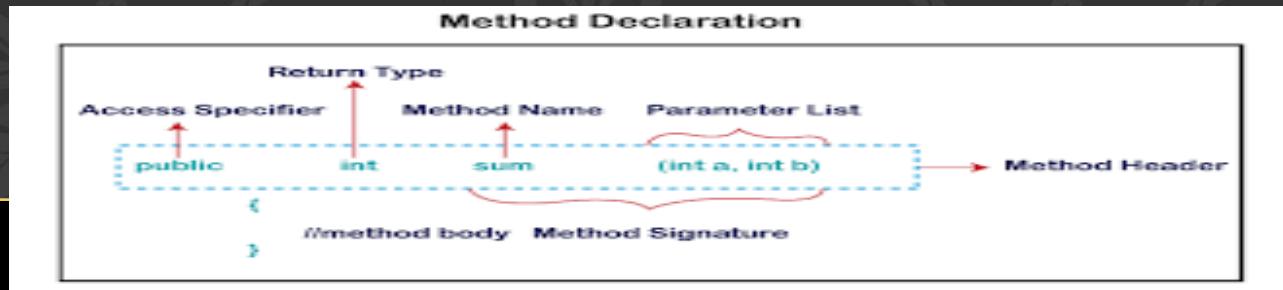
Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

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

**Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.

**Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

**Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.



## Return Type:

Return type is a data type that the method returns.

If the method does not return anything, we use void keyword.

## Method Name:

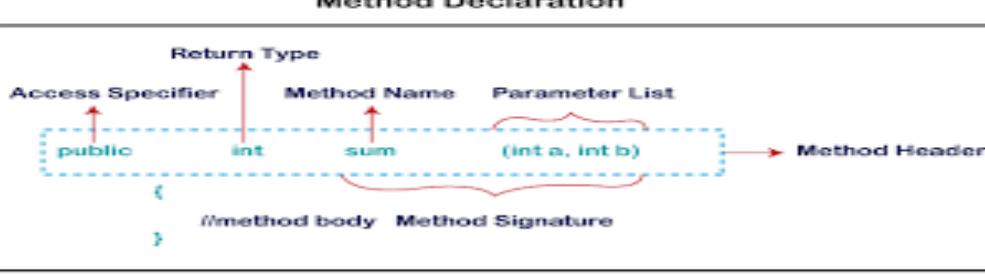
It is a unique name that is used to define the name of a method.

A method is invoked by its name.

## Parameter List:

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

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



```

class Example
{
    public static void main(String[] args)
    {
        int a = 15;
        int b = 5;
        Example e=new Example();
        int c = e.minNumber(a, b);
        System.out.println("Minimum Value = " + c);
    }

    public int minNumber(int n1, int n2)
    {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;
        return min;
    }
}
  
```

# OBJECT ORIENTED PROGRAMMING

## Lecture #17: Constructors

## CONSTRUCTORS

- A constructor initializes an object when it is created.
- we use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures
- It has the same name as the class in which it resides.
- It is syntactically similar to a method
- At the time of calling constructor, memory for the object is allocated in the memory.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default(initializes all member variables to zero)
- Once we define our own constructor, the default constructor is no longer used.

## CONSTRUCTORS

Rules for creating Java constructor

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

There are two types of constructors in Java:

1. No-arguments constructor
2. Parameterized constructor

```
Syntax:  
class ClassName  
{  
    ClassName()  
    {  
    }  
}
```

## NO-ARGUMENTS CONSTRUCTOR

No argument constructors of Java does not accept any parameters.

Instance variables of a method will be initialized with fixed values for all objects.

Syntax:

```
class ClassName {  
    ClassName()  
    {  
        var_name=value;  
    }  
}
```

Example:

```
Public class MyClass  
{  
    Int num;  
    MyClass()  
    {  
        num = 10;  
    }  
}
```

class Example

```
{  
    public static void main(String args[])  
    {  
        MyClass o1 = new MyClass();  
        MyClass o2 = new MyClass();  
        System.out.println(o1.num + " " + o2.num);  
    }  
}
```

Output:

```
10 10
```

## PARAMETERIZED CONSTRUCTOR

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Accepts one or more parameters

Parameters are added to a constructor in the same way that they are added to a method

Syntax:

```
class ClassName {  
    ClassName(parameters )  
    {  
        var_name=value;  
    }  
}
```

Example:

```
class MyClass  
{  
    int x;  
    MyClass(int i )  
    {  
        x = i;  
    }  
}
```

```
class Example  
{  
    public static void main(String args[])  
    {  
        MyClass o1 = new MyClass(10);  
        MyClass o2 = new MyClass(20);  
        System.out.println(o1.num + " " + o2.num);  
    }  
}
```

Output: 10 20

## CONSTRUCTORS VS METHODS

A constructor is used to Initialize the state of an object.

1

A method is used to expose the behavior of an object.

A constructor must not have a return type.

2

A method must have a return type.

The constructor is invoked Implicitly.

3

The method is Invoked explicitly.

The Java compiler provides a default constructor if you don't have any constructor in a class.

4

The method is not provided by the compiler in any case.

The constructor name must be same as the class name.

5

The method name may or may not be same as class name.

## OBJECT ORIENTED PROGRAMMING

### Lecture #18: Constructor Overloading

## **CONSTRUCTOR OVERLOADING**

It is a technique of having more than one constructor with different parameter lists.

Each constructor performs a different task.

They are differentiated by the compiler by the number of parameters in the list and their types.

```
class Student{  
    int id;  
    String name;  
    int age;  
    Student(int i, String n)  
    {  
        id = i;  
        name = n;  
    }  
    Student(int i, String n, int a)  
    {  
        id = i;  
        name = n;  
        age=a;  
    }  
    void display() {System.out.println(id+" "+name+" "+age);}  
    public static void main(String args[]){  
        Student s1 = new Student(11,"Jyothi");  
        Student s2 = new Student(22,"Rama",25);  
        s1.display();  
        s2.display();  
    }  
}
```

Output:  
11 jyothi 0  
22 Rama 25

## OBJECT ORIENTED PROGRAMMING

### Lecture #19: Method Overloading

## METHOD OVERLOADING

- Method overloading is one of the ways through which java achieves polymorphism.
- **Polymorphism** in Java is a concept by which we can perform a single action in different ways.
- There are two types of polymorphism in Java: **compile-time polymorphism and runtime polymorphism**. We can perform polymorphism in java by method overloading and method overriding.
- **Compile Time Polymorphism:** At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or **static or early binding**. Compile-time polymorphism is achieved through **method overloading**.

*Polymorphism  
means many  
forms.*

## METHOD OVERLOADING

- Method Overloading is a technique which allows declaring multiple methods with same name but different parameters in the same class.
- They are differentiated by the compiler by changing number of arguments or by changing the data type of arguments.
- If two or more method have same name and same parameter list but differs in return type can not be overloaded.
- There are two different ways of method overloading.
  1. Different datatype of arguments
  2. Different number of arguments

## METHOD OVERLOADING BY CHANGING DATA TYPE OF ARGUMENTS.

```
class Calculate
{
    void sum (int a, int b)
    {
        System.out.println("sum is"+(a+b));
    }
    void sum (float a, float b)
    {
        System.out.println("sum is"+(a+b));
    }
    Public static void main (String[] args)
    {
        Calculate cal = new Calculate();
        cal.sum (8,5);      //sum(int a, int b) is method is called.
        cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
    }
}
```

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

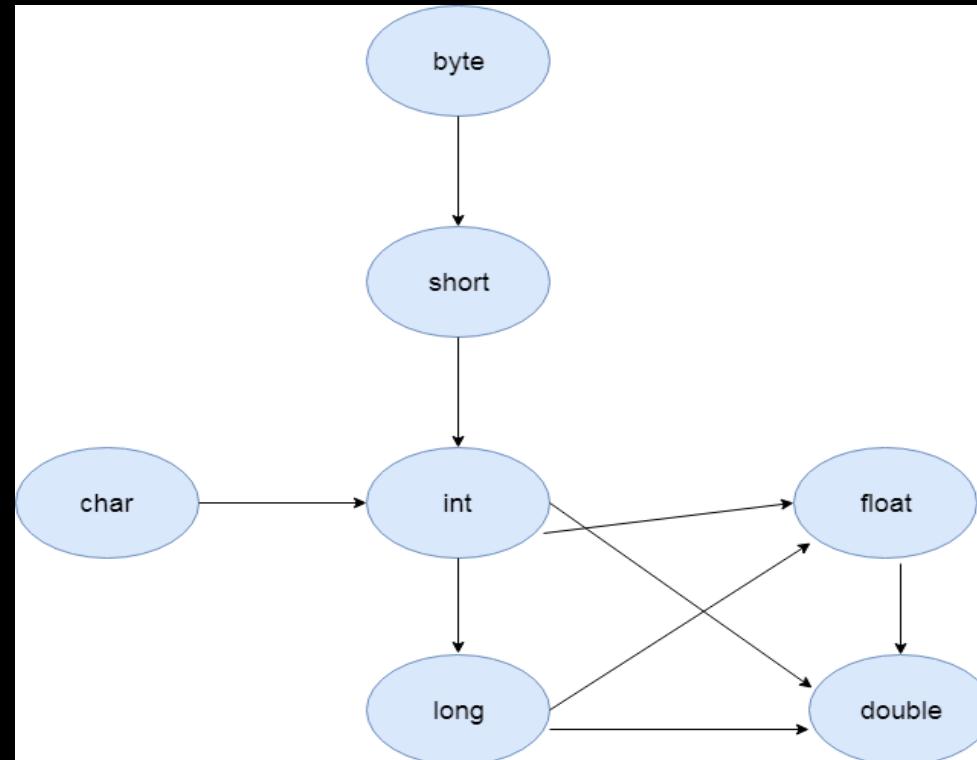
## METHOD OVERLOADING BY CHANGING NO. OF ARGUMENT

```
class Demo
{
    void multiply(int l, int b)
    {
        System.out.println("Result is"+(l*b));
    }
    void multiply(int l, int b,int h)
    {
        System.out.println("Result is"+(l*b*h));
    }
    public static void main(String[] args)
    {
        Demo a = new Demo();
        a.multiply(8,5); //multiply(int l, int b) is method is called
        a.multiply(4,6,2); //multiply(int l, int b,int h) is called
    }
}
```

*multiply() method is overloaded twice. The first method takes two arguments and the second method takes three arguments.*

## METHOD OVERLOADING AND TYPE PROMOTION

One type is promoted to another implicitly if no matching datatype is found.



## METHOD OVERLOADING AND TYPE PROMOTION

```
class Example
{
    void sum(int a,long b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
        System.out.println(a+b+c);
    }
    public static void main(String args[])
    {
        Example obj=new Example();
        obj.sum(20,20); //now second int literal will be promoted to long
        obj.sum(20,20,20);
    }
}
```

Output:  
40  
60

## METHOD OVERLOADING AND TYPE PROMOTION

```
class Example
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
    void sum(long a,long b)
    {
        System.out.println(a+b);
    }
    public static void main(String args[])
    {
        Example obj=new Example();
        obj.sum(20,20); //now int arg sum() method gets invoked
    }
}
```

Output:  
20  
20

## METHOD OVERLOADING AND TYPE PROMOTION

```
class Example
{
    void sum(int a,long b)
    {
        System.out.println("a method invoked");
    }
    void sum(long a,int b)
    {
        System.out.println("b method invoked");
    }
    public static void main(String args[])
    {
        Example obj=new Example();
        obj.sum(20,20); //now ambiguity
    }
}
```

Output: Compile Time Error

# OBJECT ORIENTED PROGRAMMING

Lecture #20: “this” Keyword

## THIS KEYWORD

refers to “this” object (object in which it is used)

usage:

with an instance variable or method of “this” class

as a function inside a constructor of “this” class

as “this” object, when passed as parameter

refers to “this” object’s data member

```
class Weight {  
    int lb; int oz;  
public Weight (int lb, int oz ) {  
    this.lb = lb; this.oz = oz;  
}  
}
```

# Usage of java this keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

1

**this** can be used to refer current class instance variable.

2

**this** can be used to invoke current class method (implicitly)

3

**this()** can be used to invoke current class constructor.

4

**this** can be passed as an argument in the method call.

5

**this** can be passed as argument in the constructor call.

6

**this** can be used to return the current class instance from the method.

## **USAGE OF THIS KEYWORD**

This can be used to refer current class instance variable

This can be used to refer current class method(implicitly)

This() can be used to invoke current class constructor.

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variable and parameters, this keyword resolves the problem of ambiguity.

## UNDERSTANDING THE PROBLEM WITHOUT THIS KEYWORD

```
class Student{  
    int rollno;  
    String name;  
    float fee;  
    Student(int rollno,String name,float fee){  
        rollno=rollno;  
        name=name;  
        fee=fee;  
    }  
    void display(){System.out.println(rollno+" "+name+" "+fee);}  
}  
  
class TestThis1{  
    public static void main(String args[]){  
        Student s1=new Student(111,"ankit",5000f);  
        Student s2=new Student(112,"sumit",6000f);  
        s1.display();  
        s2.display();  }}  
}
```

Here, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Output:  
0 null 0.0  
0 null 0.0

## SOLUTION OF THE ABOVE PROBLEM BY THIS KEYWORD

```
class Student{  
    int rollno;  
    String name;  
    float fee;  
  
    Student(int rollno, String name, float fee){  
        this.rollno=rollno;  
        this.name=name;  
        this.fee=fee;  
    }  
    void display(){System.out.println(rollno+" "+name+" "+fee);}  
}  
  
class TestThis2{  
    public static void main(String args[]){  
        Student s1=new Student(111,"ankit",5000f);  
        Student s2=new Student(112,"sumit",6000f);  
        s1.display();  
        s2.display();  }}  

```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword

**Output:**  
111 ankit 5000  
112 sumit 6000

## PROGRAM WHERE THIS KEYWORD IS NOT REQUIRED

```
class Student{  
    int rollno;  
    String name;  
    float fee;  
    Student(int r,String n,float f){  
        rollno=r;  
        name=n;  
        fee=f;  
    }  
    void display(){System.out.println(rollno+" "+name+" "+fee);}  
}  
class TestThis3{  
    public static void main(String args[]){  
        Student s1=new Student(111,"ankit",5000f);  
        Student s2=new Student(112,"sumit",6000f);  
        s1.display();  
        s2.display();  }}  
111 ankit 5000  
112 sumit 6000
```

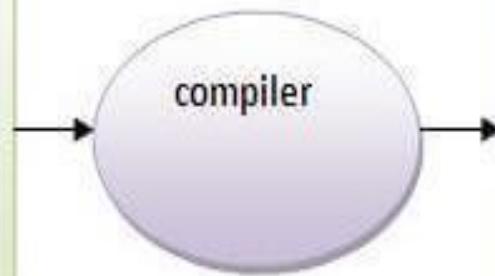
Output:  
111 ankit 5000  
112 sumit 6000

## **THIS: TO INVOKE CURRENT CLASS METHOD**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

## EXAMPLE

```
class A{  
  
void m(){}
  
  
void n(){
m();
}  
  
public static void main(String args[]){
new A().n();
}  
}
```



```
class A{  
  
void m(){}
  
  
void n(){
this.m();
}  
  
public static void main(String args[]){
new A().n();
}  
}
```

## EXAMPLE

```
class A
{
    void m()
    {
        System.out.println("hello m");
    }
    void n()
    {
        System.out.println("hello n");
        this.m();
    }
}
class TestThis4{
    public static void main(String args[])
    {
        A a=new A();
        a.n();
    }
}
```

Output:  
hello n  
hello m

## **THIS() : TO INVOKE CURRENT CLASS CONSTRUCTOR**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

## CALLING DEFAULT CONSTRUCTOR FROM PARAMETERIZED CONSTRUCTOR:

```
class A
{
A()
{
System.out.println("hello a");
}
A(int x)
{
this();
System.out.println(x);
}
}

class TestThis5
{
public static void main(String args[])
{
A a=new A(10);
}
}
```

Default constructor

**Output:**  
hello a  
10

## CALLING PARAMETERIZED CONSTRUCTOR FROM DEFAULT CONSTRUCTOR:

```
class A{  
    A()  
    {  
        this(5);  
        System.out.println("hello a");  
    }  
    A(int x)  
    {  
        System.out.println(x);  
    }  
}  
  
class TestThis6{  
    public static void main(String args[]){  
        A a=new A();  
    }  
}
```

Output:

5  
hello a

## EXAMPLE

```
class Student
{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course)
{
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee)
{
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}

class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}
}
```

Output:  
111 ankit java null  
112 sumit java 6000

# OBJECT ORIENTED PROGRAMMING

Lecture #21: “static” Keyword

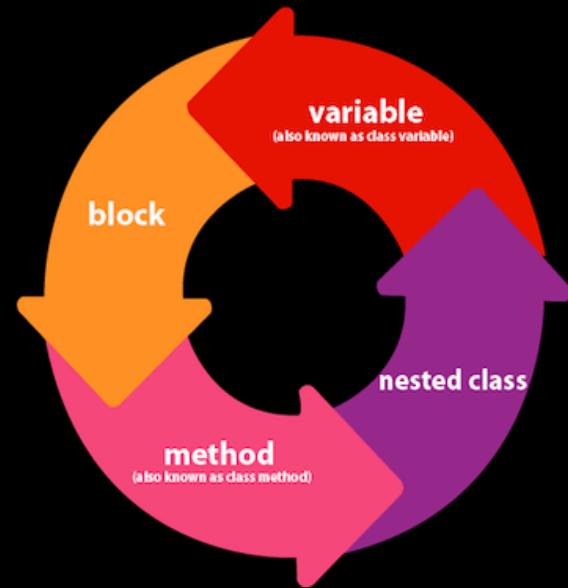
## “STATIC” KEYWORD

The **static** keyword in Java is used for memory management mainly.

The **static** keyword belongs to the class than an instance of the class.

The “**static**” Keyword can be applied to:

- Variable (also known as a instance variable)
- Block
- Method
- Nested class



## TO VARIBALE

Java static variable:

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

The static variable can be used to refer to the common property of all objects (which is not unique for each object)

for example: college name of students

The static variable gets memory only once in the class area at the time of class loading

It makes your program memory efficient (i.e., it saves memory).

Syntax:

```
static varibale_name=value;
```

## TO VARIBALE

```
class Student{  
    int rollno;  
    String name;  
    String college="GVPCOE";  
}
```

If we make college filed static, this field will get the memory only once.

```
class Student{  
    int rollno;  
    String name;  
    static String college="GVPCOE";  
}
```

Java static property is shared to all objects.

## TO VARIBALE

```
class Student{  
    int rollno;  
    String name;  
    static String college ="GVPCOE";//static variable  
    Student(int r, String n){  
        rollno = r;  
        name = n;  
    }  
    void display (){System.out.println(rollno+" "+name+" "+college);}  
}  
public class TestStaticVariable1 {  
    public static void main(String args[]){  
        Student s1 = new Student(111,"AKHILA");  
        Student s2 = new Student(222,"PRAGADA");  
        s1.display();  
        s2.display();  
    }  
}
```

Output:  
111 AKHILA GVPCOE  
222 PRAGADA GVPCOE

## TO VARIABLE

```
class Counter{  
int count=0;//will get memory each time when the instance is created
```

```
Counter(){  
count++; //incrementing value  
System.out.println(count);  
}
```

```
public static void main(String args[]){  
//Creating objects  
Counter c1=new Counter();  
Counter c2=new Counter();  
Counter c3=new Counter();  
}  
}
```

Output:  
1  
1  
1

will get memory  
only once and  
retain its value

## TO VARIABLE

```
class Counter2{  
    static int count=0;
```

```
Counter2(){  
    count++; //incrementing the value of static variable  
    System.out.println(count);  
}
```

```
public static void main(String args[]){  
    //creating objects  
    Counter2 c1=new Counter2();  
    Counter2 c2=new Counter2();  
    Counter2 c3=new Counter2();  
}  
}
```

Output:  
1  
2  
3

## **TO BLOCK**

A static block is used for initializing static variables.

The static block is executed once when the first object of the class is created.

Syntax:

```
static  
{  
    //code  
}
```

## TO BLOCK

```
class Example {  
    static int a;  
    static int b;  
  
    static {  
        a = 10;  
        b = 2 * a;  
    }  
  
    public static void main( String args[] )  
    {  
        Example e = new Example();  
        System.out.printf("a = %d\t b = %d ", e.a, e.b);  
    }  
}
```

Output:  
a = 10 b = 20

## OBJECT ORIENTED PROGRAMMING

Lecture #22: “static” Keyword(cont.)

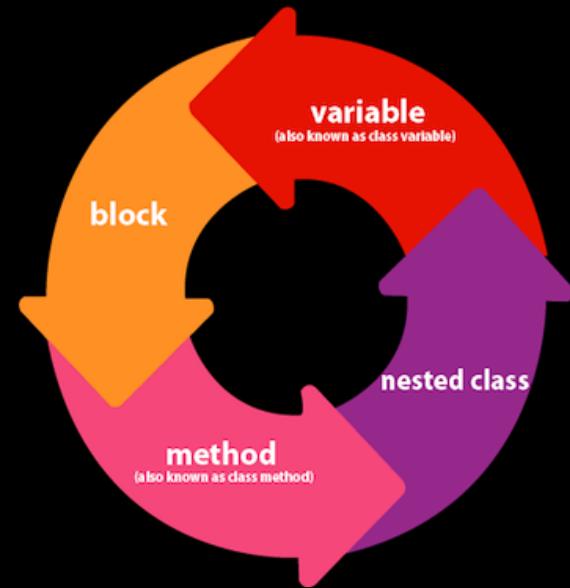
## “STATIC” KEYWORD

The **static** keyword in Java is used for memory management mainly.

The **static** keyword belongs to the class than an instance of the class.

The “**static**” Keyword can be applied to:

- Variable (also known as a instance variable)
- Block
- Method
- Nested class



## TO METHOD

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

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

```
class Student{  
    int rollno;  
    String name;  
    static String college = "GVP";  
    //static method to change the value of static variable  
    static void change(){  
        college = "GVPCOE";  
    }  
}
```

## TO METHOD

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
  
    public static void main(String args[])  
    {  
        int result=Calculate(cube(5);  
        System.out.println(result);  
    }  
}
```

Output:  
125

## TO METHOD

### Restrictions for the static method

- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

```
class A{  
    int a=40;//non static  
  
    public static void main(String args[]){  
        System.out.println(a);  
    }  
}
```

Output:  
Compile Time Error

Java main method is static? because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

## JAVA MAIN METHOD IS STATIC?

Java main() method is always static, so that compiler can call it without the creation of an object or before the creation of an object of the class.

In any Java program, the main() method is the starting point from where compiler starts program execution. So, the compiler needs to call the main() method.

If the main() is allowed to be non-static, then while calling the main() method JVM has to instantiate its class.

While instantiating it has to call the constructor of that class, There will be ambiguity if the constructor of that class takes an argument.

Static method of a class can be called by using the class name only without creating an object of a class.

The main() method in Java must be declared public, static and void. If any of these are missing, the Java program will compile but a runtime error will be thrown.

## **FINAL KEYWORD**

The final keyword is a non access modifier used for classes , attributes and methods, which makes them non changeable.  
(impossible to inherit or override)

The final keyword is useful when you want a variable to always store the same value like PI(3.14)

The final key word is a modifier.

### **Java Final Keyword**

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

## **FINAL KEYWORD**

The final keyword in java is used to restrict the user. The java final keyword can be used in many contexts.

1. Variable
2. Method
3. Class

It can be applied with the variables , a final variable that have no value it is called blank final variable or uninitialized 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.

## **FINAL VARIABLE**

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

Ex: there is a final variable speed limit, we are going to change the value of this variable.

But, it cannot be changed because final variable once assigned a value can never be changed.

## EXAMPLE

```
class Bike
{
final int speedlimit=90;
void run()
{
speedlimit=400;
}
public static void main(String args[])
{
Bike obj= new Bike();
obj.run();
}
}
Output:
Compile time error
```

## FINAL METHOD

```
class Bike
{
final void run()
{
System.out.println("running");
}
class Honda extends Bike
{
void run()
{
System.out.println("running safely with 100 kmph");
}
public static void main(String args[])
{
Honda honda = new Honda();
honda.run();
}
}
Output: Compile time error
```

## IS FINAL METHOD INHERITED?

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

Class Honda2 extends Bike
{
public static void main(String args[])
{
New Honda2().run();
}

Output: running...
```

## 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.

It can be initialized only in constructor.

Ex: class Student

```
{  
    int id;  
    String name;  
    final String PAN_CARD_NUMBER;  
}
```

## CAN WE INITIALIZE BLANK FINAL VARIABLE?

Yes, but only in constructors

```
class Bike10{  
    final int speedlimit;//blank final variable  
    Bike10(){  
        speedlimit=70;  
        System.out.println(speedlimit);  
    }  
    public static void main(String args[]){  
        new Bike10();  
    }  
}  
Output: 70
```

## GARBAGE COLLECTION

Java garbage collection is the process by which Java programs perform automatic memory management.

Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short.

When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program.

Eventually, some objects will no longer be needed.

The garbage collector finds these unused objects and deletes them to free up memory.

## **GARBAGE COLLECTION**

In java, garbage means unreferenced objects.

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

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

### **Advantage of Garbage Collection**

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

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

# How can an object be unreferenced?

01

By nulling the reference

02

By assigning a reference to  
another

03

By anonymous object etc.

By nulling a reference:

```
Employee e= new Employee();  
e=null;
```

By assigning a reference to another:

```
Employee e1= new Employee();  
Employee e2= new Employee();  
e1=e2;
```

//now the first object is referred by e1 is available for garbage collection

By anonymous object:

```
new Employee();
```

## HOW IT WORKS?

Java garbage collection is an automatic process.

The programmer does not need to explicitly mark objects to be deleted.

The garbage collection implementation lives in the JVM.

Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification.

Although there are many JVMs, Oracle's HotSpot is by far the most common.

It offers a robust and mature set of garbage collection options.

While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process.

## CONTD...

In the first step, unreferenced objects are identified and marked as ready for garbage collection.

In the second step, marked objects are deleted.

Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap.

The compaction process makes it easier to allocate memory to new objects sequentially after the block of memory allocated to existing objects.

All of HotSpot's garbage collectors implement a generational garbage collection strategy that categorizes objects by age.

The rationale behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.

## **FINALIZE() METHOD**

The `java.lang.Object.finalize()` is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

### **Declaration**

Following is the declaration for `java.lang.Object.finalize()` method

```
protected void finalize()
```

### **Parameters**

NA

### **Return Value**

This method does not return a value.

### **Exception**

`Throwable` – the Exception raised by this method

## EXAMPLE

The following example shows the usage of lang.Object.finalize() method.

```
import java.util.*;
public class ObjectDemo extends GregorianCalendar {
    public static void main(String[] args) {
        try {
ObjectDemo cal = new ObjectDemo();
        System.out.println("+" + cal.getTime());
        System.out.println("Finalizing...");
        cal.finalize();
        System.out.println("Finalized."); }
    catch (Throwable ex) {
        ex.printStackTrace(); }
    }
}
```

Output:

Sat Sep 22 00:27:21 EEST 2012  
Finalizing...  
Finalized.

# OBJECT ORIENTED PROGRAMMING

## Lecture #23: Inheritance-Introduction

## INHERITANCE

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

Reusability is achieved by INHERITANCE

Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.

The class whose properties are extended is known as **super** or **base** or **parent** class.

The class which extends the properties of super class is known as **sub** or **derived** or **child** class

A class can either extends another class or can implement an interface

## DEFINING A SUBCLASS

Syntax :

```
class <subclass name> extends <superclass name>
{
    variable declarations;
    method declarations;
}
```

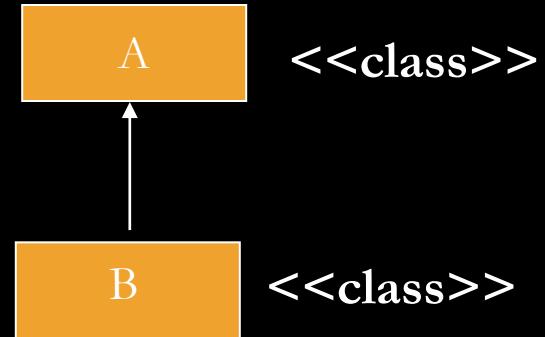
- extends keyword signifies that properties of the super class are extended to sub class
- Sub class will not inherit private members of super class

## INHERITANCE

**class B extends A { ..... }**

**A super class**

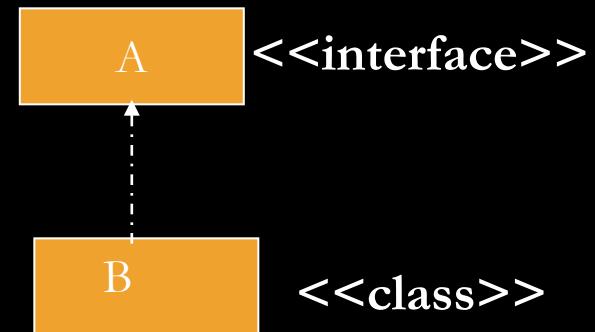
**B sub class**



**class B implements A { ..... }**

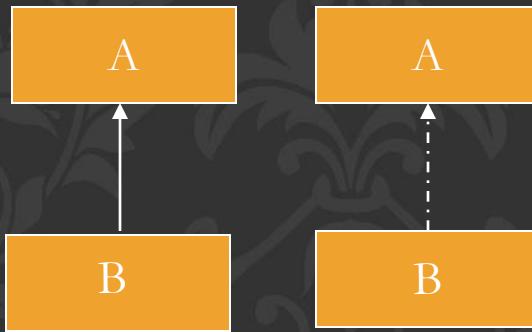
**A interface**

**B sub class**

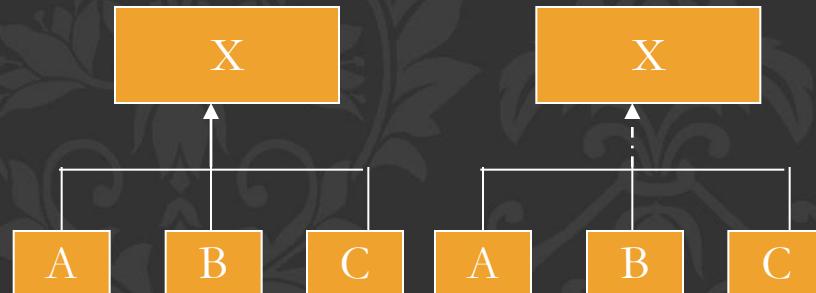


## TYPES OF INHERITANCE

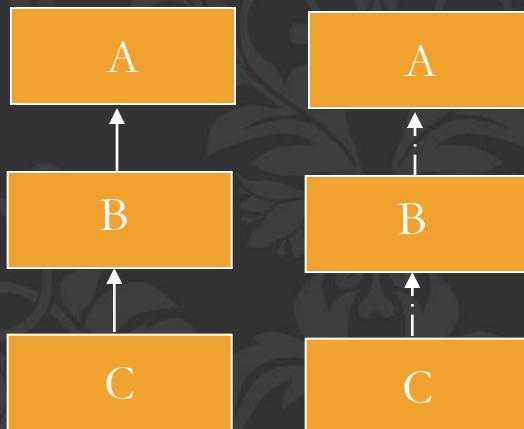
Single Inheritance



Hierarchical Inheritance

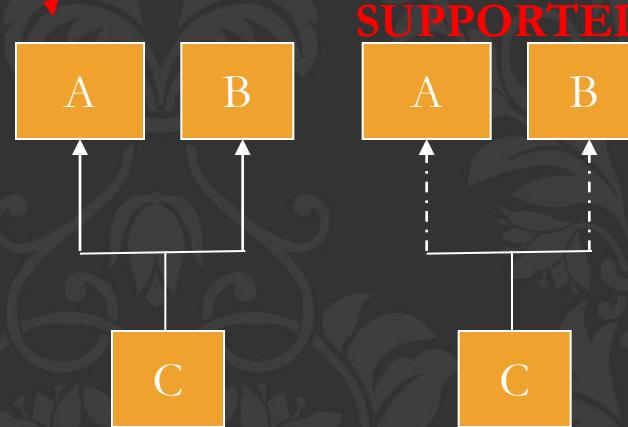


MultiLevel Inheritance



**NOT SUPPORTED BY JAVA**

Multiple Inheritance



**SUPPORTED BY JAVA**

## TYPES OF INHERITANCE

- Multiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes

Example :

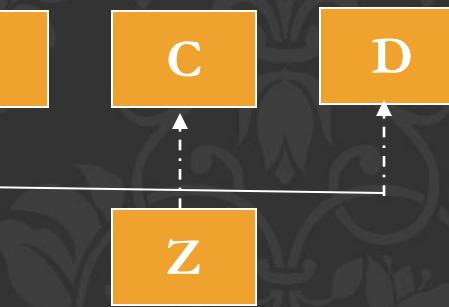
class Z extends A implements C , D  
{ ..... }

OK

class Z extends A ,B  
{ }

**WRONG**

OR



class Z extends A extends B  
{ }

**WRONG**

# OBJECT ORIENTED PROGRAMMING

## Lecture #24: Single Inheritance

## INHERITANCE

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

Reusability is achieved by INHERITANCE

Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.

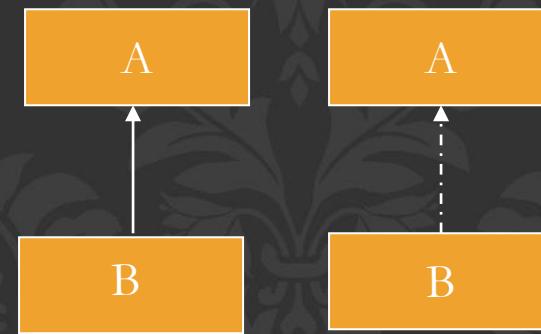
The class whose properties are extended is known as super or base or parent class.

The class which extends the properties of super class is known as sub or derived or child class

A class can either extends another class or can implement an interface

## TYPES OF INHERITANCE

### Single Inheritance



## SINGLE INHERITANCE

```
class A
{
    int i,j;
    void showij()
    {
        System.out.println("i and j : "+i+j);
    }
}
class B extends A
{
    int k;
    void showk()
    {
        System.out.println("k : "+k);
    }
    void sum()
    {
        System.out.println("i+j+k: "+(i+j+k));
    }
}
```

```
class Example
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        a.i=10;
        a.j=20;
        System.out.println("contents of a:");
        a.showij();
        b.i=1;
        b.j=2;
        b.k=3;
        System.out.println("contents of b:");
        b.showij();
        b.showk();
        System.out.println("sum of i,j,k of b:");
        b.sum();
    }
}
```

# OBJECT ORIENTED PROGRAMMING

## Lecture #25: Multilevel Inheritance

## INHERITANCE

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

Reusability is achieved by INHERITANCE

Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.

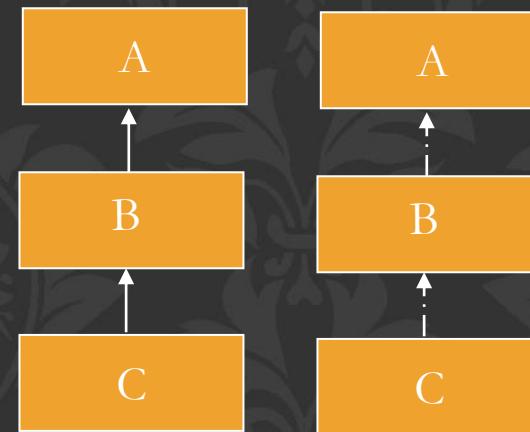
The class whose properties are extended is known as super or base or parent class.

The class which extends the properties of super class is known as sub or derived or child class

A class can either extends another class or can implement an interface

## TYPES OF INHERITANCE

### MultiLevel Inheritance



## MULTI LVEL INHERITANCE

```
class A
{
    int i;
    void showij()
    {
        System.out.println("i :" +i);
    }
}

class B extends A
{
    int j;
    void showj()
    {
        System.out.println("j : "+j);
    }
}
```

```
class C extends B
{
    int k;
    void showk()
    {
        System.out.println("k : "+k);
    }
    void sum()
    {
        System.out.println("i+j+k: "+(i+j+k));
    }
}

class Example
{
    public static void main(String args[])
    {
        C c=new C();
        c.i=10;
        c.j=20;
        c.k=30;
        c.sum()
    }
}
```

# OBJECT ORIENTED PROGRAMMING

Lecture #26: Hierarchical Inheritance

## INHERITANCE

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

Reusability is achieved by INHERITANCE

Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.

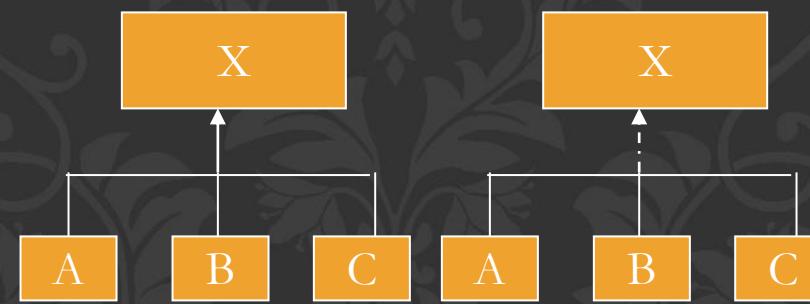
The class whose properties are extended is known as super or base or parent class.

The class which extends the properties of super class is known as sub or derived or child class

A class can either extends another class or can implement an interface

## TYPES OF INHERITANCE

### Hierarchical Inheritance



## HIERARCHICAL INHERITANCE

```
class A
{
    int i;
    void showij()
    {
        System.out.println("i :" + i);
    }
}
class B extends A
{
    int j;
    void showj()
    {
        System.out.println("j : " + j);
    }
    void sum()
    {
        System.out.println("i+j: " + (i+j));
    }
}
```

```
class C extends A{
    int k;
    void showk()
    {
        System.out.println("k : " + k);
    }
    void sum()
    {
        System.out.println("i+k: " + (i+k));
    }
}
class Example
{
    public static void main(String args[])
    {
        C c=new C();
        c.i=10;
        c.k=30;
        c.sum()
    }
}
```

# OBJECT ORIENTED PROGRAMMING

Lecture #27: “super” keyword

## SUPER KEYWORD

The **super keyword** in Java is used in subclasses to access superclass members (attributes, constructors and methods).

Can be used to call super class constructor

1. **super()**
2. **super(<parameter-list>)**

Can refer to super class instance variables/Methods

3. **super . <super class instance variable/Method>**

## USE OF SUPER KEYWORD FOR CONSTRUCTORS

Whenever a sub class object is created ,super class constructor is called first.

If super class constructor does not have any constructor of its own OR has an unparametrized constructor then it is automatically called by Java Run Time by using call **super()**

If a super class has a parameterized constructor then it is the responsibility of the sub class constructor to call the super class constructor by call

**super(<parameters required by super class>)**

Call to super class constructor must be the first statement in sub class constructor

## 1. WHEN SUPER CLASS HAS A UNPARAMETRIZED CONSTRUCTOR

```
class A
{
A()
{
System.out.println("This is constructor
of class A");
}
} // End of class A
class B extends A
{
B()
{
super();
System.out.println("This is constructor of
class B");
}
} // End of class B
```

class inhtest

{

public static void main(String args[])

{

B b1 = new B();

}

}

Optional

## OUTPUT

This is constructor of class A  
This is constructor of class B

```
class A
{
A()
{
System.out.println("This is class A");
}
}

class B extends A
{
B()
{
System.out.println("This is class B");
}
}

class inherit1
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

File Name is inherit1.java

Output:

This is class A  
This is class B

1. When super class has a Unparametrized constructor

1. When super class has a Unparametrized constructor

```
class A
{
private A()
{
System.out.println("This is class A");
}
}

class B extends A
{
B()
{
System.out.println("This is class B");
}
}

class inherit2
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

**Output:**

**Error: A() has private access in A**

```
class A
{
private A()
{
System.out.println("This is class A");
}

A()
{
System.out.println("This is class A");
}

class B extends A
{
B()
{
System.out.println("This is class B");
}

}

class inherit2
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

Output:

A() is already defined in A  
A() has private access in A

2 errors

1. When super class has a Unparametrized constructor

## 2. When super class has a parametrized constructor

```
class A
{
    int a;
    A( int a1)
    {
        a =a1;
        System.out.println("This is constructor of class A");
    }
}

class B extends A
{
    int b;
    double c;
    B(int b1,double c1)
    {
        b=b1;
        c=c1;
        System.out.println("This is constructor of class B");
    }
}
```

B b1 = new B(10,8.6);

D:\java\bin>javac inhtest.java  
inhtest.java:15: cannot find  
symbol  
symbol : constructor A()  
location: class A  
{  
^  
1 errors

## 2. When super class has a parametrized constructor

```
class A
{
    int a;
    A( int a1)
    {
        a =a1;
        System.out.println("This is constructor
of class A");
    }
}

class B extends A
{
    int b;
    double c;
    B(int a2,int b1,double c1)
    {
        super(a2);
        b=b1;
        c=c1;
        System.out.println("This is constructor
of class B");
    }
}
```

B b1 = new B(8,10,8.6);

### OUTPUT

This is constructor of class A  
This is constructor of class B

# OBJECT ORIENTED PROGRAMMING

Lecture #28: “super” keyword

## SUPER KEYWORD

The **super keyword** in Java is used in subclasses to access superclass members (attributes, constructors and methods).

Can be used to call super class constructor

1. **super()**
2. **super(<parameter-list>)**

Can refer to super class instance variables/Methods

3. **super . <super class instance variable/Method>**

### 3. Refer to super class instance variables/Methods

```
class A
{
    int a;
    A( int a)
    {
        this.a =a;
        System.out.println("This is constructor of
                           class A");
    }
    void print()
    {
        System.out.println("a="+a);
    }
    void display()
    {
        System.out.println("hello This is Display in
                           A");
    }
} // End of class A
```

```
class B extends A
{
    int b;
    double c;
    B(int a,int b,double c)
    {
        super(a);
        this.b=b;
        this.c=c;
        System.out.println("This is constructor of
                           class B");
    }
    void show()
    {
        print();
        System.out.println("b="+b);
        System.out.println("c="+c);
    }
} // End of class B
```

### 3. Refer to super class instance variables/Methods

```
class inhtest1
{
    public static void main(String args[])
    {
        B b1 = new B(10,8,4.5);
        b1.show();
    }
}
```

#### OutPut

D:\java\bin>java inhtest1

This is constructor of class A

This is constructor of class B

a=10

b=8

c=4.5

### 3. Refer to super class instance variables/Methods

```
class A
{
    int a;
    A( int a)
    {
        this.a =a;
        System.out.println("This is constructor of
class A");
    }
    void show()
    {
        System.out.println("a="+a);
    }
    void display()
    {
        System.out.println("hello This is Display in
A");
    }
}
```

```
class B extends A
{
    int b;
    private double c;
    B(int a,int b,double c)
    {
        super(a);
        this.b=b;
        this.c=c;
        System.out.println("This is constructor of
class B");
    }
    void show()
    {
        super.show();
        System.out.println("b="+b);
        System.out.println("c="+c);
        display();
    }
}
```

### 3. Refer to super class instance variables/Methods

```
class inhtest1
{
    public static void main(String args[])
    {
        B b1 = new B(10,8,4.5);
        b1.show();
    }
}
/* OutPut
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
hello This is Display in A
*/
```

```
class A
{
int a;
A( int a)
{
this.a =a;
}
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is
Display in A");
}
```

```
class B extends A
{
int b;
double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
}
void show()
{
//super.show();
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}
```

```
class inhtest2
{
public static void main(String
args[])
{
B b1 = new B(10,20,8.4);
b1.show();
}}
```

```
D:\java\bin>java inhtest2
a=10
b=20
c=8.4
```

3. Refer to super class instance variables/Methods

```
class A
{
    int a;
    A( int a)
    {
        this.a =a;
    }
}

class B extends A
{
    // super class variable a hides here
    int a;
    int b;
    double c;
    B(int a,int b,double c)
    {
        super(100);
        this.a = a;
        this.b=b;
        this.c=c;
    }
    void show()
    {
        System.out.println("Super class a="+super.a);
        System.out.println("a="+a);
        System.out.println("b="+b);
        System.out.println("c="+c);
    }
}
```

```
class inhtest2
{
    public static void main(String args[])
    {
        B b1 = new B(10,20,8.4);
        b1.show();
    }
}
```

3. Refer to super class instance variables/Methods

### Out Put

D:\java\bin>java inhtest2  
Super class a=100  
a=10  
b=20  
c=8.4

# OBJECT ORIENTED PROGRAMMING

## Lecture #29: Method Overriding

## METHOD OVERRIDING

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

### Uses:

- 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.

## METHOD OVERRIDING

```
class Vehicle  
{  
    void run()  
    {  
        System.out.println("Vehicle is running");  
    }  
}
```

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

Output:  
Vehicle is running

```
class Vehicle  
{  
    void run()  
    {  
        System.out.println("Vehicle is running");  
    }  
}
```

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

```
void run()  
{  
    System.out.println("Bike is running safely");  
}
```

Output:  
Bike is running safely

## METHOD OVERRIDING

```
class Vehicle
{
    void run()
    {
        System.out.println("Vehicle is running");
    }
}
```

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

Output:  
Vehicle is running

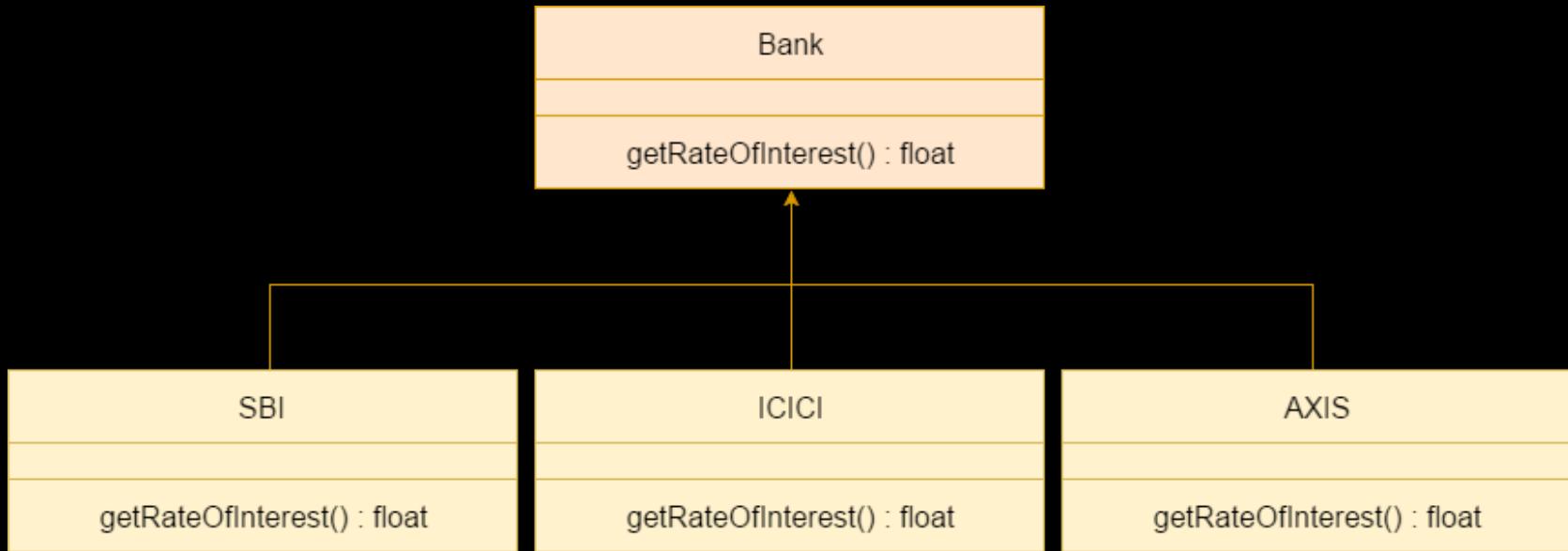
Output:  
Vehicle is running  
Bike is running safely

```
class Vehicle
{
    void run()
    {
        System.out.println("Vehicle is running");
    }
}
```

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

```
void run()
{
    super.run();
    System.out.println("Bike is running safely");
}
```

## METHOD OVERRIDING REALTIME EXAMPLE



Example

## METHOD OVERRIDING

Note:

A method declared as static method cannot be overridden

Main method cannot be overridden

A method declared final cannot be overridden.

Constructors cannot be overridden.

## METHOD OVERLOADING VS METHOD OVERRIDING

Method overloading	Method overriding
<b>1. More than one method with same name, different prototype in same scope is called method overloading.</b>	<b>1. More than one method with same name, same prototype in different scope is called method overriding.</b>
<b>2. In case of method overloading, parameter must be different.</b>	<b>2. In case of method overriding, parameter must be same.</b>
<b>3. Method overloading is the example of <u>compile time polymorphism</u>.</b>	<b>3. Method overriding is the example of <u>run time polymorphism</u>.</b>
<b>4. Method overloading is performed within class.</b>	<b>4. Method overriding occurs in two classes.</b>
<b>5. In case of method overloading, Return type can be same or different.</b>	<b>5. In case of method overriding Return type must be same.</b>
<b>6. Static methods can be overloaded which means a class can have more than one static method of same name.</b>	<b>6. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.</b>
<b>7. <u>Static binding</u> is being used for overloaded methods</b>	<b>7. <u>dynamic binding</u> is being used for overridden/overriding methods.</b>

# OBJECT ORIENTED PROGRAMMING

## Lecture #30: Dynamic Method Dispatch

## DYNAMIC METHOD DISPATCH

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime.

This is how java implements runtime polymorphism.

When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to.

In simple words the type of object which it referred determines which version of overridden method will be called.

Parent



extends

Child

Parent p = new Parent();

Child c = new Child();

Parent p = new Child();



Upcasting

~~Child c = new Parent();~~

incompatible type

## UPCASTING IN JAVA

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**.

In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class.

In those cases we can create a parent class reference and assign child class objects to it.

Example:

Notice the last output. This is because of the statement, gm = ck;. Now gm.type() will call the Cricket class version of type() method. Because here gm refers to the cricket object.

# EXAMPLE

## EXAMPLE

```
class Game{  
    public void type(){  
        System.out.println("Indoor & outdoor"); }  
}  
  
Class Cricket extends Game  
{  
    public void type(){  
        System.out.println("outdoor game");  
    }  
    public static void main(String[] args){  
        Game gm = new Game();  
        Cricket ck = new Cricket();  
        gm.type();  
        ck.type();  
        gm = ck; //gm refers to Cricket object  
        gm.type(); //calls Cricket's version of type}  
}
```

Output:  
Indoor & outdoor  
Outdoor game  
Outdoor game

## DIFFERENCE BETWEEN STATIC BINDING AND DYNAMIC BINDING IN JAVA?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

In simpler terms, Static binding means when the type of object which is invoking the method is determined at compile time by the compiler. While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler.

# OBJECT ORIENTED PROGRAMMING

Lecture #31: “final” keyword

## “FINAL” KEYWORD

The final keyword in java is used to restrict the user

final can be:

variable

method

class

### Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

## JAVA FINAL VARIABLE

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

```
class FinalVariable
{
    final int var = 50;
    var = 60 // error
}
```



compile-time  
errors

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

## JAVA FINAL METHOD

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



compile-time errors

```
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();
    }
}
```

## 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();  
    }  
}
```



compile-time errors

## “FINAL” KEYWORD

### uninitialized final variable:

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

It can be initialized only in constructor.

Ex: PAN CARD NUMBER

```
class Bike{  
    final int speedlimit;//blank final variable  
  
    Bike()  
    {  
        speedlimit=70;  
        System.out.println(speedlimit);  
    }  
  
    public static void main(String args[])  
    {  
        Bike b1=new Bike();  
    }  
}
```

## “FINAL” KEYWORD

### static blank final variable

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

It can be initialized only in static block.

```
class A
{
    static final int data; // static blank final variable
    static
    {
        data=50;
    }
    public static void main(String args[])
    {
        System.out.println(A.data);
    }
}
```

## “FINAL” KEYWORD

### final parameter

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



compile-time errors

```
class Bike
{
    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[])
    {
        Bike b=new Bike();
        b.cube(5);
    }
}
```

## “FINAL” KEYWORD

**Final Variable**  **To create constant variables**

**Final Methods**  **Prevent Method Overriding**

**Final Classes**  **Prevent Inheritance**

# OBJECT ORIENTED PROGRAMMING

## Lecture #32: Abstract class

## ABSTRACTION

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

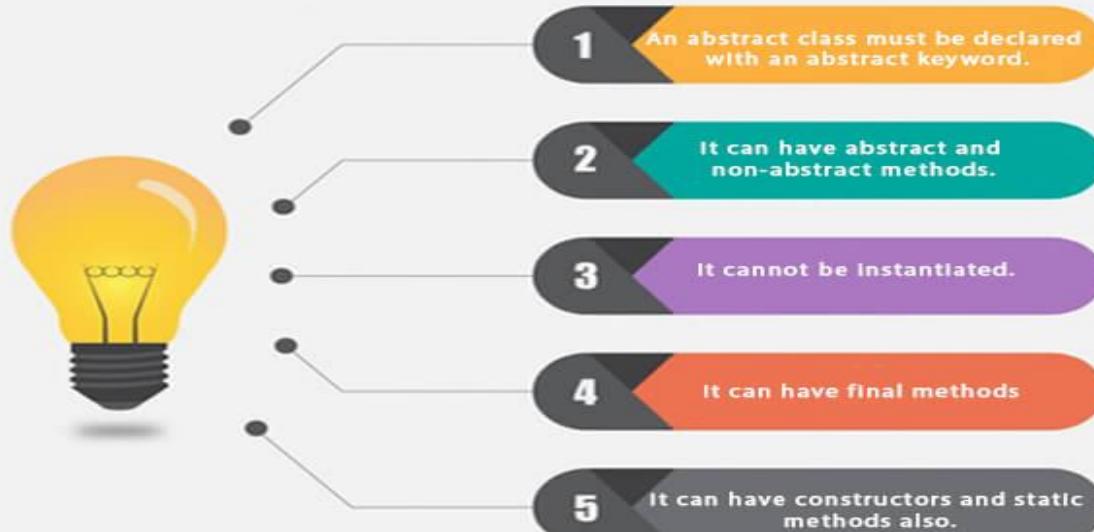
There are two ways to achieve abstraction in java

- Abstract class
- Interface

## ABSTRACT CLASS

A class which contains the abstract keyword in its declaration is known as abstract class.

### Rules for Java Abstract class

- 
- 1 An abstract class must be declared with an abstract keyword.
  - 2 It can have abstract and non-abstract methods.
  - 3 It cannot be instantiated.
  - 4 It can have final methods
  - 5 It can have constructors and static methods also.

## ABSTRACT CLASS

An abstract class can have both abstract and regular methods

The abstract keyword is a non-access modifier, used for classes and methods:

**Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Syntax:

```
abstract class_name{ }
```

**Abstract method:** can only be used in an abstract class. It does not have a body. The body is provided by the subclass (inherited).

no method  
body

Syntax:

```
abstract type method_name(paramenter_list);
```

```
abstract class Figure
{
    int dim1,dim2;
    Figure(int d1, int d2)
    {
        dim1=d1;
        dim2=d2;
    }
    void display()
    {
        System.out.println("Area is: ");
    }

    abstract void area(); //abstract method
}
```

Any subclass of an abstract class must either implement all of the abstract methods of super class or be declared itself abstract

```
class Rectangle extends Figure
{
    Rectangle(int a, int b)
    {
        super(a,b);
    }

    void area() //overriding abstract method
    {
        System.out.println(dim1*dim2);
    }
}
```

```
class Example32{
    public static void main(String args[])
    {
        Rectangle r=new rectangle(10,20);
        Figure f;
        f=r;
        f.display();
        f.area();
    }
}
```

# OBJECT ORIENTED PROGRAMMING

## Lecture #33: Interface

## ABSTRACTION

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

- Abstract class
- Interface

## INTERFACE

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the “**interface**” keyword to create an interface in Java

### Uses:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Syntax:

```
interface <interface_name> {
```

```
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

## INTERFACE

### Implementing an Interface

- Like abstract classes, we cannot create objects of interfaces.
- To use an interface, other classes must implement it. We use the “**implements**” keyword to implement an interface.

### Note:

- It is similar to class. It is a collection of abstract methods.
- Method bodies exist only for default methods and static methods.
- An interface may also contain constants, default methods, static methods, and nested types.
- all the methods of the interface need to be defined in the class.
- cannot instantiate an interface.
- does not contain any constructors.

```
interface Polygon  
{  
    void getArea(int length, int breadth);  
}
```

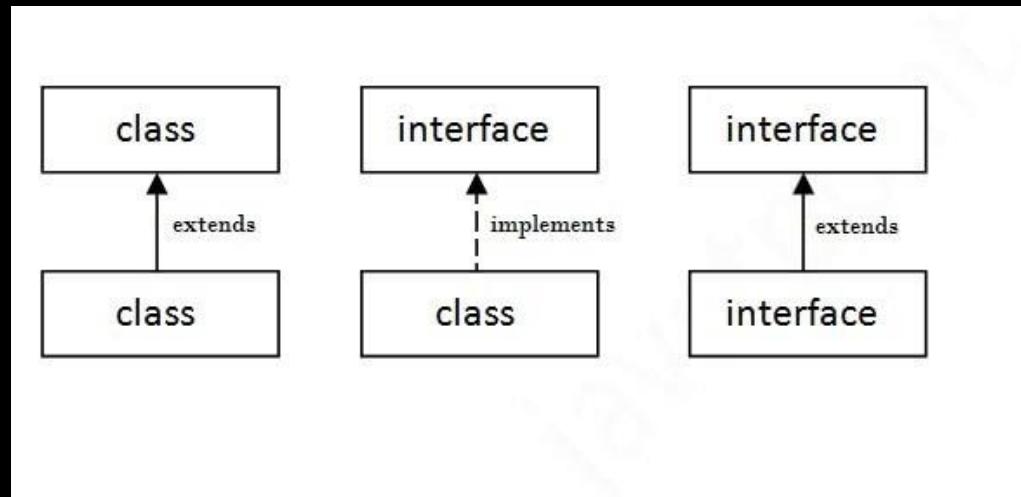
```
class Rectangle implements Polygon  
{  
    public void getArea(int length, int breadth)  
    {  
        System.out.println("The area of the rectangle is " + (length * breadth));  
    }  
}
```

A class that implements an interface must implement all the methods declared in the interface.

```
class Example{  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        r1.getArea(5, 6);  
    }  
}
```

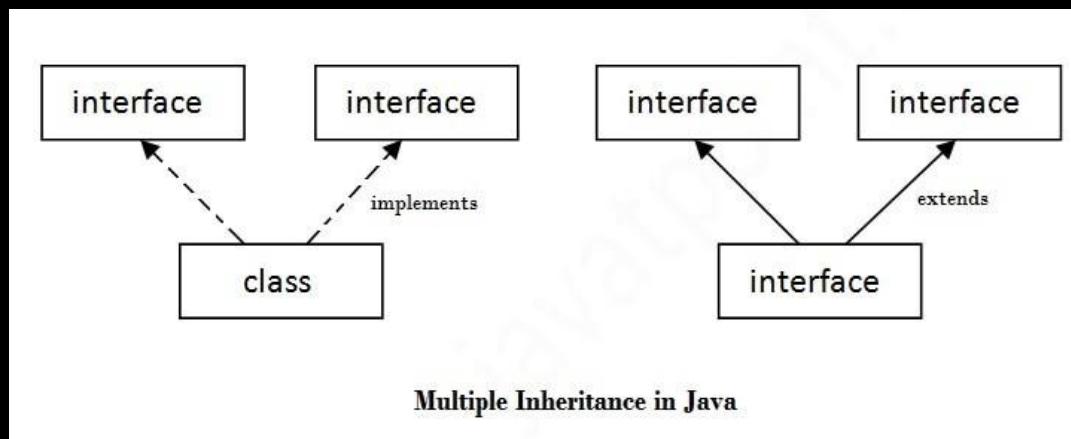
# INTERFACE

The relationship between classes and interfaces



# INTERFACE

Multiple inheritance in Java by interface



# INTERFACE

## Implementing Multiple Interfaces

```
interface A {  
    // members of A  
}  
  
interface B {  
    // members of B  
}  
  
class C implements A, B {  
    // abstract members of A  
    // abstract members of B  
}
```

## Extending an Interface

```
interface Line {  
    // members of Line interface  
}  
  
// extending interface  
interface Polygon extends Line {  
    // members of Polygon interface  
    // members of Line interface  
}
```

*if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.*

## Abstract class

## Interface

1) Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance</b> .	Interface <b>supports multiple inheritance</b> .
3) Abstract class <b>can have final, non-final, static and non-static variables</b> .	Interface has <b>only static and final variables</b> .
4) Abstract class <b>can provide the implementation of interface</b> .	Interface <b>can't provide the implementation of abstract class</b> .
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.
6) An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface only.
7) An <b>abstract class</b> can be extended using keyword "extends".	An <b>interface</b> can be implemented using keyword "implements".
8) A Java <b>abstract class</b> can have class members like private, protected, etc.	Members of a Java interface are public by default.
<b>9) Example:</b> <pre>public abstract class Shape{     public abstract void draw(); }</pre>	<b>Example:</b> <pre>public interface Drawable{     void draw(); }</pre>

# OBJECT ORIENTED PROGRAMMING

## Lecture #35: Packages

## PACKAGES

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations)

Packages are used for:

- Preventing naming conflicts.
- Making searching/locating and usage of classes, interfaces, enumerations easier
- Providing controlled access
- Packages can be considered as data encapsulation (or data-hiding).

Package names and  
directory structure  
are closely related.

## PACKAGES

Packages

User Defined  
Packages

In Built  
Packages



## BUILT-IN PACKAGES

These packages consist of a large number of classes which are a part of Java API.

- **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
- **java.io:** Contains classed for supporting input / output operations.
- **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- **java.applet:** Contains classes for creating Applets.
- **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
- **java.net:** Contain classes for supporting networking operations.

## USER-DEFINED PACKAGES

These are the packages that are defined by the user.

### Creating a Package:

- choose a name for the package
- include a “**package**” statement along with that name
- The package statement should be the first line in the source file.
- There can be only one package statement in each source file

Syntax:

```
package package_name;
```

It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Example:

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

## USER-DEFINED PACKAGES

### compile java package

Syntax:

```
javac -d directory javfilename
```

Example:

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

### run java package program

You need to use fully qualified name

Example:

```
java mypack.Simple
```

The -d switch specifies the destination where to put the generated class file.

## **ACCESS PACKAGE FROM ANOTHER PACKAGE**

three ways to access

1. import package.\*;
2. import package.classname;
3. fully qualified name.

## ACCESS PACKAGE FROM ANOTHER PACKAGE

import package.\*;

- “**import**” keyword is used to make the classes and interface of another package accessible to the current package.
- All the classes and interfaces of this package will be accessible but not subpackages.

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

save as:  
A.java

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

save as: B.java

### Output:

```
javac -d . A.java  
javac -d . B.java  
java mypack.B  
Hello
```

## ACCESS PACKAGE FROM ANOTHER PACKAGE

import package.classname;

- “**import**” keyword is used to make the classes and interface of another package accessible to the current package.
- only declared class of this package will be accessible.

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

save as:  
A.java

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

save as: B.java

### Output:

```
javac -d . A.java  
javac -d . B.java  
java mypack.B  
Hello
```

## ACCESS PACKAGE FROM ANOTHER PACKAGE

### fully qualified name.

- 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.
- only declared class of this package will be accessible.

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

save as:  
A.java

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

save as: B.java

### Output:

```
javac -d . A.java  
javac -d . B.java  
java mypack.B  
Hello
```

## ACCESS SPECIFIERS

- 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.
- By controlling access, you can prevent misuse.

There are four types of Java access modifiers:

1. Default
2. public
3. private
4. protected

## ACCESS SPECIFIERS

Access Modifiers	Default	private	protected	public
Accessible inside the class	yes	yes	yes	yes
Accessible within the subclass inside the same package	yes	no	yes	yes
Accessible outside the package	no	no	no	yes
Accessible within the subclass outside the package	no	no	yes	yes

## 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.

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

## USES OF WRAPPER CLASSES

- **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.

## Primitive Type

## Wrapper class

boolean

Boolean

char

Character

byte

Byte

short

Short

int

Integer

long

Long

float

Float

double

Double

## 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.

```
class WrapperExample{  
    public static void main(String args[]){  
        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);  
}} Output: 20 20 20
```

## 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.

```
class WrapperExample{  
    public static void main(String args[]){  
        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);  
    } } output: 3 3 3
```

## CUSTOM WRAPPER CLASSES

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

## MATH CLASS

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Method	Description
Math.abs()	It will return the Absolute value of the given value.
Math.max()	It returns the Largest of two values.
Math.min()	It is used to return the Smallest of two values.
Math.round()	It is used to round off the decimal numbers to the nearest value.
Math.sqrt()	It is used to return the square root of a number.
Math.cbrt()	It is used to return the cube root of a number.
Math.pow()	It returns the value of first argument raised to the power to second argument.
Math.signum()	It is used to find the sign of a given value.
Math.ceil()	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.

Math.copySign()	It is used to find the Absolute value of first argument along with sign specified in second argument.
Math.nextAfter()	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
Math.nextUp()	It returns the floating-point value adjacent to d in the direction of positive infinity.
Math.nextDown()	It returns the floating-point value adjacent to d in the direction of negative infinity.
Math.floor()	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
Math.floorDiv()	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
Math.random()	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
Math.rint()	It returns the double value that is closest to the given argument and equal to mathematical integer.

Math.hypot()	It returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
Math.ulp()	It returns the size of an ulp of the argument.
Math.getExponent()	It is used to return the unbiased exponent used in the representation of a value.
Math.IEEEremainder()	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
Math.addExact()	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.
Math.subtractExact()	It returns the difference of the arguments, throwing an exception if the result overflows an int.
Math.multiplyExact()	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.
Math.incrementExact()	It returns the argument incremented by one, throwing an exception if the result overflows an int.

<code>Math.decrementExact()</code>	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
<code>Math.negateExact()</code>	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
<code>Math.toIntExact()</code>	It returns the value of the long argument, throwing an exception if the value overflows an int.

## Logarithmic Math Methods

<b>Method</b>	<b>Description</b>
<code>Math.log()</code>	It returns the natural logarithm of a double value.
<code>Math.log10()</code>	It is used to return the base 10 logarithm of a double value.
<code>Math.log1p()</code>	It returns the natural logarithm of the sum of the argument and 1.
<code>Math.exp()</code>	It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828.
<code>Math.expm1()</code>	It is used to calculate the power of E and subtract one from it.

# Trigonometric Math Methods

Method	Description
Math.sin()	It is used to return the trigonometric Sine value of a Given double value.
Math.cos()	It is used to return the trigonometric Cosine value of a Given double value.
Math.tan()	It is used to return the trigonometric Tangent value of a Given double value.
Math.asin()	It is used to return the trigonometric Arc Sine value of a Given double value
Math.acos()	It is used to return the trigonometric Arc Cosine value of a Given double value.
Math.atan()	It is used to return the trigonometric Arc Tangent value of a Given double value.

# Hyperbolic Math Methods

Method	Description
Math.sinh()	It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.
Math.cosh()	It is used to return the trigonometric Hyperbolic Sine value of a Given double value.
Math.tanh()	It is used to return the trigonometric Hyperbolic Tangent value of a Given double value.

# Angular Math Methods

Method	Description
Math.toDegrees	It is used to convert the specified Radians angle to equivalent angle measured in Degrees.
Math.toRadians	It is used to convert the specified Degrees angle to equivalent angle measured in Radians.

# OBJECT ORIENTED PROGRAMMING

Lecture #34: Object class

## OBJECT CLASS

The Object class is the parent class of all the classes in java by default  
Object class is present in `java.lang` package

If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

`String toString()`  
`boolean equals(Object obj)`  
`int hashCode()`

`Object clone()`  
`void finalize()`  
`Class getClass()`

`void wait()`  
`void notify()`  
`void notifyAll()`

## EQUALS

It is used to compare the two objects dynamically.

Syntax:

```
public boolean equals(Object obj)
```

This method returns true if this object is the same as the obj argument; false otherwise.

## OBJECT CLONING

The object cloning is a way to create exact copy of an object.

The `clone()` method of `Object` class is used to clone an object.

The `java.lang.Cloneable` interface must be implemented by the class whose object clone we want to create

Syntax:

```
protected Object clone() throws CloneNotSupportedException
```

Advantages:

- easy way of copying objects
- don't need to write lengthy and repetitive codes.

```
class Student implements Cloneable{  
int rollno;  
String name;  
  
Student(int rollno,String name){  
this.rollno=rollno;  
this.name=name;  
}  
public Object clone()throws CloneNotSupportedException{  
return super.clone();  
}  
public static void main(String args[]){  
try{  
Student s1=new Student(101,"amit");  
  
Student s2=(Student)s1.clone();  
  
System.out.println(s1.rollno+" "+s1.name);  
System.out.println(s2.rollno+" "+s2.name);  
}  
catch(CloneNotSupportedException c){}  
}
```

Output:  
101 amit  
101 amit

## REFLECTION

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

There is a class in Java named “Class” that keeps all the information about objects and classes at runtime.

The object of “Class” can be used to perform reflection.

Steps:

- create an object of Class.
- using the object we can call various methods to get information about methods, fields, and constructors present in a class.

## REFLECTION

Reflection can be used to get information about –

Class The **getClass()** method is used to get the name of the class to which an object belongs.

Constructors The **getConstructors()** method is used to get the public constructors of the class to which an object belongs.

Methods The **getMethods()** method is used to get the public methods of the class to which an objects belongs.

```
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;
class Test
{
    private String s;
    public Test() { s = "Java Programming"; }
    public void method1() {
        System.out.println("The string is " + s);
    }
}
```

```
System.out.println("The name of constructor
is " + constructor.getName());
System.out.println("The public methods
of class are : ");
Method[] methods = cls.getMethods();
System.out.println(method.getName());
}
}
```

```
class Demo
{
    public static void main(String args[]) throws
Exception
{
    Test obj = new Test();
    Class cls = obj.getClass();
    System.out.println("The name of class is " +
cls.getName());
    Constructor constructor = cls.getConstructor();
```

Outpt:  
The name of class is Test  
The name of constructor is Test  
The public methods of class are  
:  
method1

## RUNTIME JAVA CLASSES

**Java Runtime** class is used to *interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of `java.lang.Runtime` class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

# Important methods of Java Runtime class

No.	Method	Description
1)	public static Runtime getRuntime()	returns the instance of Runtime class.
2)	public void exit(int status)	terminates the current virtual machine.
3)	public void addShutdownHook(Thread hook)	registers new hook thread.
4)	public Process exec(String command) throws IOException	executes given command in a separate process.
5)	public int availableProcessors()	returns no. of available processors.
6)	public long freeMemory()	returns amount of free memory in JVM.
7)	public long totalMemory()	returns amount of total memory in JVM.

## Java Runtime exec() method

```
public class Runtime1{  
    public static void main(String args[]) throws Exception{  
        Runtime.getRuntime().exec("notepad");//will open a new notepad  
    }  
}
```

## How to shutdown system in Java

You can use *shutdown -s* command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

```
public class Runtime2{  
    public static void main(String args[]) throws Exception{  
        Runtime.getRuntime().exec("shutdown -s -t 0");  
    }  
}
```

## How to restart system in Java

```
public class Runtime3{  
    public static void main(String args[])throws Exception{  
        Runtime.getRuntime().exec("shutdown -r -t 0");  
    }  
}
```

## Java Runtime availableProcessors()

```
public class Runtime4{  
    public static void main(String args[])throws Exception{  
        System.out.println(Runtime.getRuntime().availableProcessors());  
    }  
}
```

## Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

```
public class MemoryTest{  
    public static void main(String args[])throws Exception{  
        Runtime r=Runtime.getRuntime();  
        System.out.println("Total Memory: "+r.totalMemory());  
        System.out.println("Free Memory: "+r.freeMemory());  
  
        for(int i=0;i<10000;i++){  
            new MemoryTest();  
        }  
        System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());  
        System.gc();  
        System.out.println("After gc(), Free Memory: "+r.freeMemory());  
    }  
}
```

```
Total Memory: 100139008  
Free Memory: 99474824  
After creating 10000 instance, Free Memory: 99310552  
After gc(), Free Memory: 100182832
```

## COLLECTIONS IN JAVA

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](#), [Vector](#), [LinkedList](#), [PriorityQueue](#), [HashSet](#), [LinkedHashSet](#), [TreeSet](#)).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

## FRAMEWORK IN JAVA

What is a framework in Java

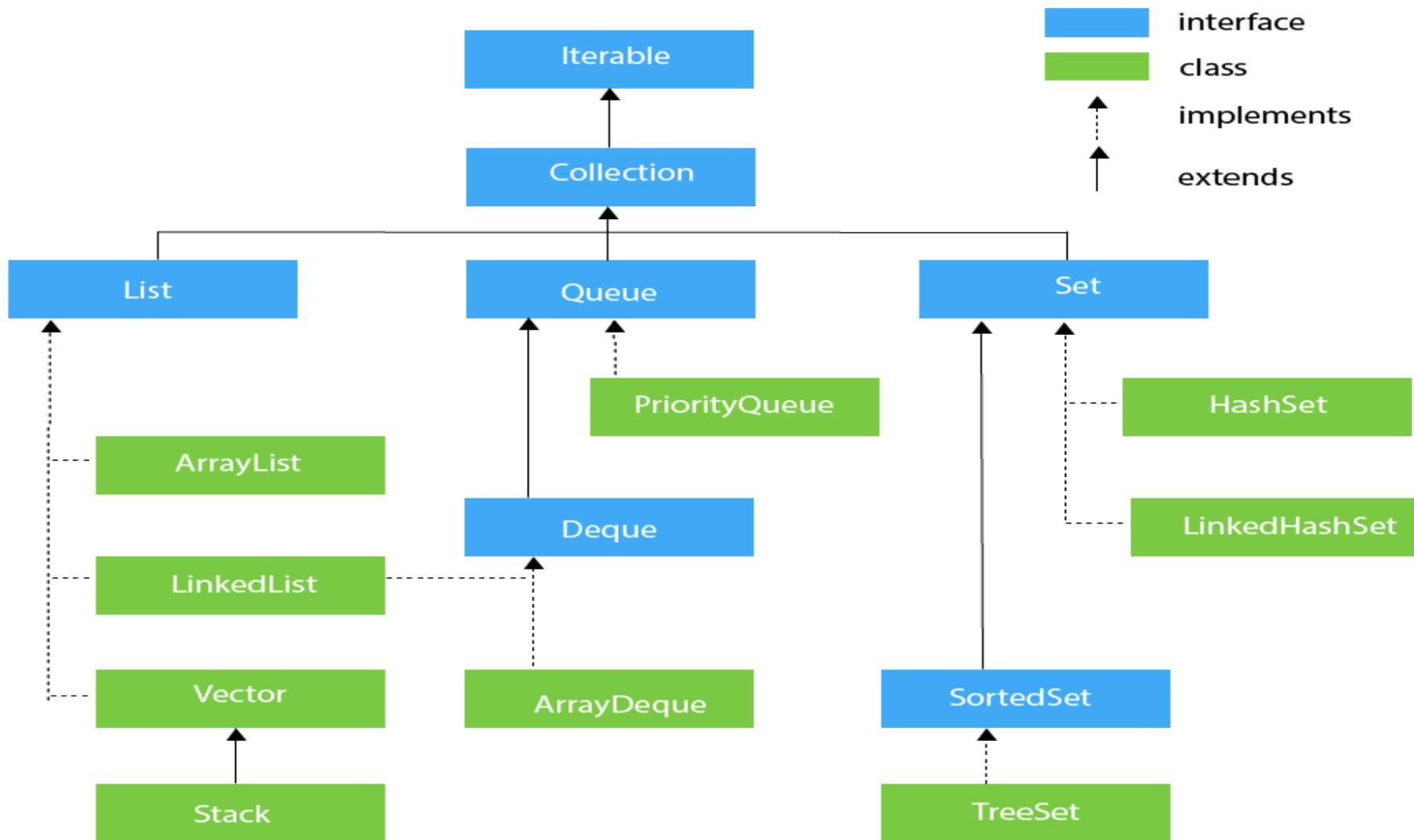
- It provides readymade architecture.
- It represents a set of classes and interfaces.
  - It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

# HIERARCHY OF COLLECTION FRAMEWORK



# Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.

10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

# Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

## Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

# Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

`Iterator<T> iterator()`

It returns the iterator over the elements of type T.

## COLLECTION INTERFACE

The Collection interface is the interface which is implemented by all the classes in the collection framework.

It declares the methods that every collection will have.

In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## LIST INTERFACE

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

- 1.List <data-type> list1= **new** ArrayList();
- 2.List <data-type> list2 = **new** LinkedList();
- 3.List <data-type> list3 = **new** Vector();
- 4.List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

## ARRAYLIST

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("akhila");//Adding object in arraylist
list.add("rajesh");
list.add("asha");
list.add("mouli");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } } }
Output: akhila rajesh asha mouli
```

## ARRAY LIST

Java **ArrayList** class uses a *dynamic array* for storing the elements.

It is like an array, but there is *no size limit*.

We can add or remove elements anytime.

So, it is much more flexible than the traditional array.

It is found in the *java.util* package.

It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also.

It implements the List interface so we can use all the methods of List interface here.

The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

## ARRAY LIST

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

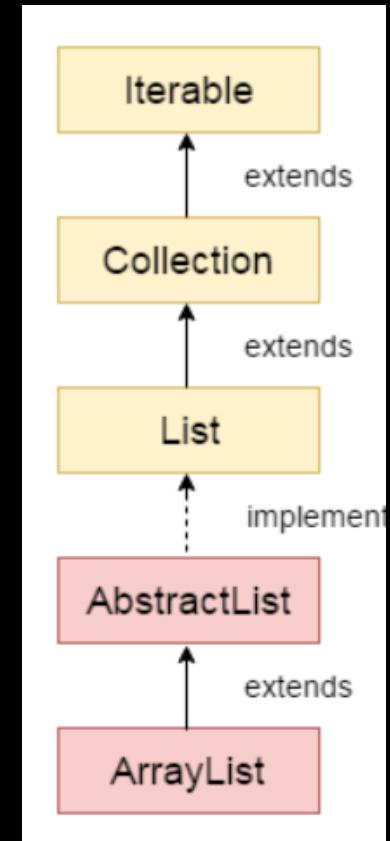
## HIERARCHY OF ARRAYLIST CLASS

As shown in the above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends the Collection and Iterable interfaces in hierarchical order.

### ArrayList class declaration

Let's see the declaration for java.util.ArrayList class.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```



# Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

## Methods of ArrayList

Method	Description
void <code>add(int index, E element)</code>	It is used to insert the specified element at the specified position in a list.
boolean <code>add(E e)</code>	It is used to append the specified element at the end of a list.
boolean <code>addAll(Collection&lt;? extends E&gt; c)</code>	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean <code>addAll(int index, Collection&lt;? extends E&gt; c)</code>	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void <code>clear()</code>	It is used to remove all of the elements from this list.
void <code>ensureCapacity(int requiredCapacity)</code>	It is used to enhance the capacity of an ArrayList instance.
E <code>get(int index)</code>	It is used to fetch the element from the particular position of the list.
boolean <code>isEmpty()</code>	It returns true if the list is empty, otherwise false.

<code>int lastIndexOf(Object o)</code>	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
<code>Object[] toArray()</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>&lt;T&gt; T[] toArray(T[] a)</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>Object clone()</code>	It is used to return a shallow copy of an ArrayList.
<code>boolean contains(Object o)</code>	It returns true if the list contains the specified element
<code>int indexOf(Object o)</code>	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
<code>E remove(int index)</code>	It is used to remove the element present at the specified position in the list.
<code>boolean remove(Object o)</code>	It is used to remove the first occurrence of the specified element.
<code>boolean removeAll(Collection&lt;?&gt; c)</code>	It is used to remove all the elements from the list.
<code>boolean removeIf(Predicate&lt;? super E&gt; filter)</code>	It is used to remove all the elements from the list that satisfies the given predicate.
<code>protected void removeRange(int fromIndex, int toIndex)</code>	It is used to remove all the elements lies within the given range.
<code>void replaceAll(UnaryOperator&lt;E&gt; operator)</code>	It is used to replace all the elements from the list with the specified element.

## LINKEDLIST

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Akhila");
al.add("Rajesh");
al.add("Asha");
al.add("Mouli");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } } }
```

Output: Akhila Rajesh Asha Mouli

## VECTOR

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } } }
```

## STACK

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;  
public class TestJavaCollection4{  
    public static void main(String args[]){  
        Stack<String> stack = new Stack<String>();  
        stack.push("Ayush");  
        stack.push("Garvit");  
        stack.push("Amit");  
        stack.push("Ashish");  
        stack.push("Garima");  
        stack.pop();  
        Iterator<String> itr=stack.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next()); } } }
```

## SET INTERFACE

Set Interface in Java is present in `java.util` package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by `HashSet`, `LinkedHashSet`, and `TreeSet`.

Set can be instantiated as:

1. `Set<data-type> s1 = new HashSet<data-type>();`
2. `Set<data-type> s2 = new LinkedHashSet<data-type>();`
3. `Set<data-type> s3 = new TreeSet<data-type>();`

## HASH SET

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;  
public class TestJavaCollection7{  
    public static void main(String args[]){  
        //Creating HashSet and adding elements  
        HashSet<String> set=new HashSet<String>();  
        set.add("Ravi");  
        set.add("Vijay");  
        set.add("Ravi");  
        set.add("Ajay");  
        //Traversing elements  
        Iterator<String> itr=set.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next()); } } }
```

## HASH SET

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashCode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

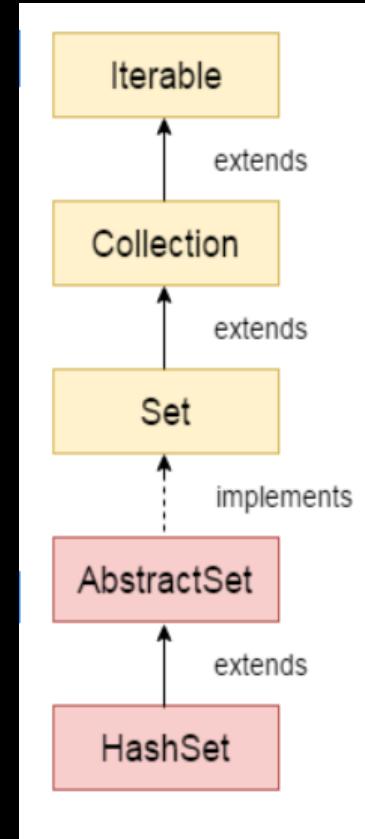
## HIERARCHY OF HASHSET CLASS

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

HashSet class declaration

Let's see the declaration for java.util.HashSet class.

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
```



## Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

## METHODS IN JAVA

SN	Modifier & Type	Method	Description
1)	boolean	<code>add(E e)</code>	It is used to add the specified element to this set if it is not already present.
2)	void	<code>clear()</code>	It is used to remove all of the elements from the set.
3)	object	<code>clone()</code>	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	<code>contains(Object o)</code>	It is used to return true if this set contains the specified element.
5)	boolean	<code>isEmpty()</code>	It is used to return true if this set contains no elements.
6)	Iterator<E>	<code>iterator()</code>	It is used to return an iterator over the elements in this set.
7)	boolean	<code>remove(Object o)</code>	It is used to remove the specified element from this set if it is present.
8)	int	<code>size()</code>	It is used to return the number of elements in the set.
9)	Spliterator<E>	<code>spliterator()</code>	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

## LINKED HASH SET

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;  
public class TestJavaCollection8{  
    public static void main(String args[]){  
        LinkedHashSet<String> set=new LinkedHashSet<String>();  
        set.add("Ravi");  
        set.add("Vijay");  
        set.add("Ravi");  
        set.add("Ajay");  
        Iterator<String> itr=set.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next()); } } }
```

## TREE SET

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } } }
```

## HASH MAP

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique.

If you try to insert the duplicate key, it will replace the element of the corresponding key.

It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized.

It allows us to store the null elements as well, but there should be only one null key.

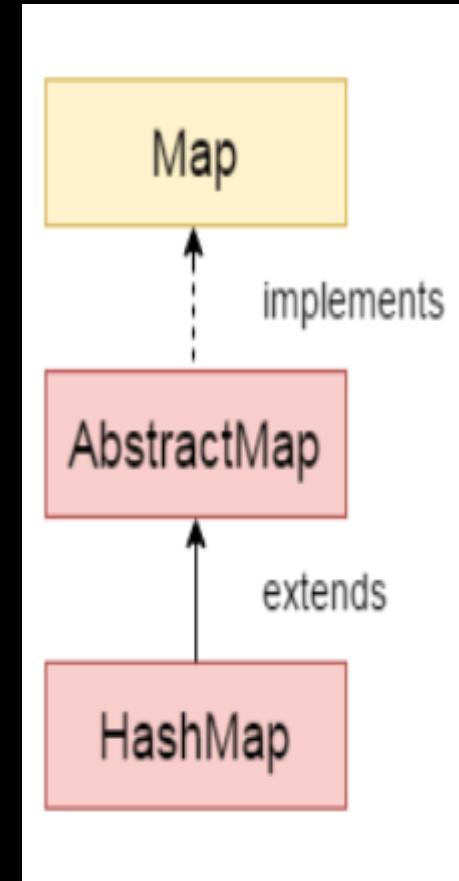
Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

## HASH MAP

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.



## HASH MAP DECLARATION

Let's see the declaration for `java.util.HashMap` class.

```
public class HashMap<K,V> extends AbstractMap<K,V> implements  
    Map<K,V>, Cloneable, Serializable
```

HashMap class Parameters

Let's see the Parameters for `java.util.HashMap` class.

- **K:** It is the type of keys maintained by this map.
- **V:** It is the type of mapped values.

## Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

## Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.

V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.

## EXAMPLE

```
import java.util.*;
public class HashMapExample1{
    public static void main(String args[]){
HashMap<Integer,String> map=new HashMap<Integer,String>();
//Creating HashMap
map.put(1,"Mango"); //Put elements in Map
map.put(2,"Apple");
map.put(3,"Banana");
map.put(4,"Grapes");
System.out.println("Iterating Hashmap...");
for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());  }  } }
Output: Iterating Hashmap...
1. Mango 2. Apple 3. Banana 4. Grapes
```

In this example, we are storing Integer as the key and String as the value, so we are using `HashMap<Integer,String>` as the type. The `put()` method inserts the elements in the map.

To get the key and value elements, we should call the `getKey()` and `getValue()` methods. The `Map.Entry` interface contains the `getKey()` and `getValue()` methods. But, we should call the `entrySet()` method of `Map` interface to get the instance of `Map.Entry`.

## STRING TOKENIZER

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

## Constructor

## Description

StringTokenizer(String str)

creates StringTokenizer with specified string.

StringTokenizer(String str, String  
delim)

creates StringTokenizer with specified string and delimiter.

StringTokenizer(String str, String  
delim, boolean returnValue)

creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

## Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimiter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

## EXAMPLE

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
import java.util.StringTokenizer;
public class Simple{
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("my name is khan","");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken()); } } }
```

Output: my  
name  
is  
khan

## NEXTTOKEN(STRING DELIM) METHOD OF STRINGTOKENIZER CLASS

StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

```
import java.util.*;  
public class Test {  
    public static void main(String[] args) {  
        StringTokenizer st = new StringTokenizer("my,name,is,khan");  
        // printing next token  
        System.out.println("Next token is : " + st.nextToken(",,"));    }    }
```

Output: Next token is : my

## CALENDAR CLASS

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

Java Calendar class declaration

Let's see the declaration of java.util.Calendar class.

```
public abstract class Calendar extends Object  
implements Serializable, Cloneable, Comparable<Calendar>
```

## EXAMPLE

```
import java.util.Calendar;
public class CalendarExample1 {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        System.out.println("The current date is : " + calendar.getTime());
        calendar.add(Calendar.DATE, -15);
        System.out.println("15 days ago: " + calendar.getTime());
        calendar.add(Calendar.MONTH, 4);
        System.out.println("4 months later: " + calendar.getTime());
        calendar.add(Calendar.YEAR, 2);
        System.out.println("2 years later: " + calendar.getTime());  }  }
```

Output:

The current date is : Thu Jan 19 18:47:02 IST 2017

15 days ago: Wed Jan 04 18:47:02 IST 2017

4 months later: Thu May 04 18:47:02 IST 2017

2 years later: Sat May 04 18:47:02 IST 2019

## RANDOM CLASS

Java Random class is used to generate a stream of pseudorandom numbers. The algorithms implemented by Random class use a protected utility method than can supply up to 32 pseudorandomly generated bits on each invocation.

Random class is used to generate pseudo-random numbers in java. An instance of this class is thread-safe. The instance of this class is however cryptographically insecure. This class provides various method calls to generate different random data types such as float, double, int.

### Constructors:

- **Random():** Creates a new random number generator
- **Random(long seed):** Creates a new random number generator using a single long seed

## Methods

Methods	Description
doubles()	Returns an unlimited stream of pseudorandom double values.
ints()	Returns an unlimited stream of pseudorandom int values.
longs()	Returns an unlimited stream of pseudorandom long values.
next()	Generates the next pseudorandom number.
nextBoolean()	Returns the next uniformly distributed pseudorandom boolean value from the random number generator's sequence
nextByte()	Generates random bytes and puts them into a specified byte array.
nextDouble()	Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence
nextFloat()	Returns the next uniformly distributed pseudorandom Float value between 0.0 and 1.0 from this random number generator's sequence
nextGaussian()	Returns the next pseudorandom Gaussian double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.

## JAVA IO PACKAGE

**Java I/O** (Input and Output) is used *to process the input and produce the output.*

Java uses the concept of a stream to make I/O operation fast. The `java.io` package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

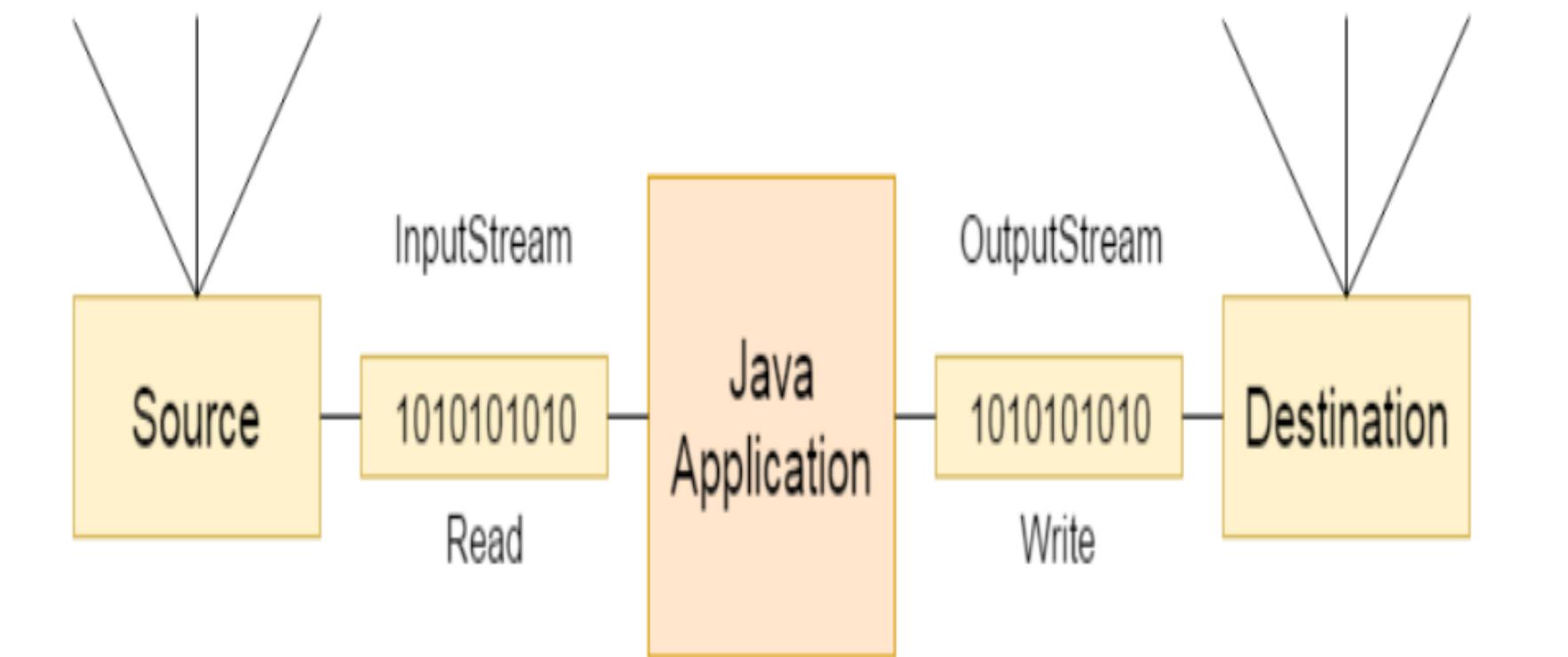
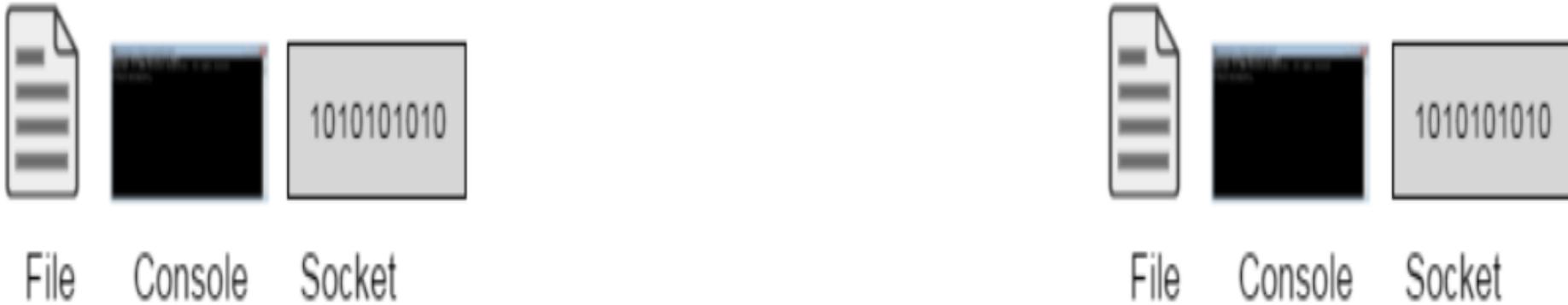
### Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:** standard output stream

**2) System.in:** standard input stream



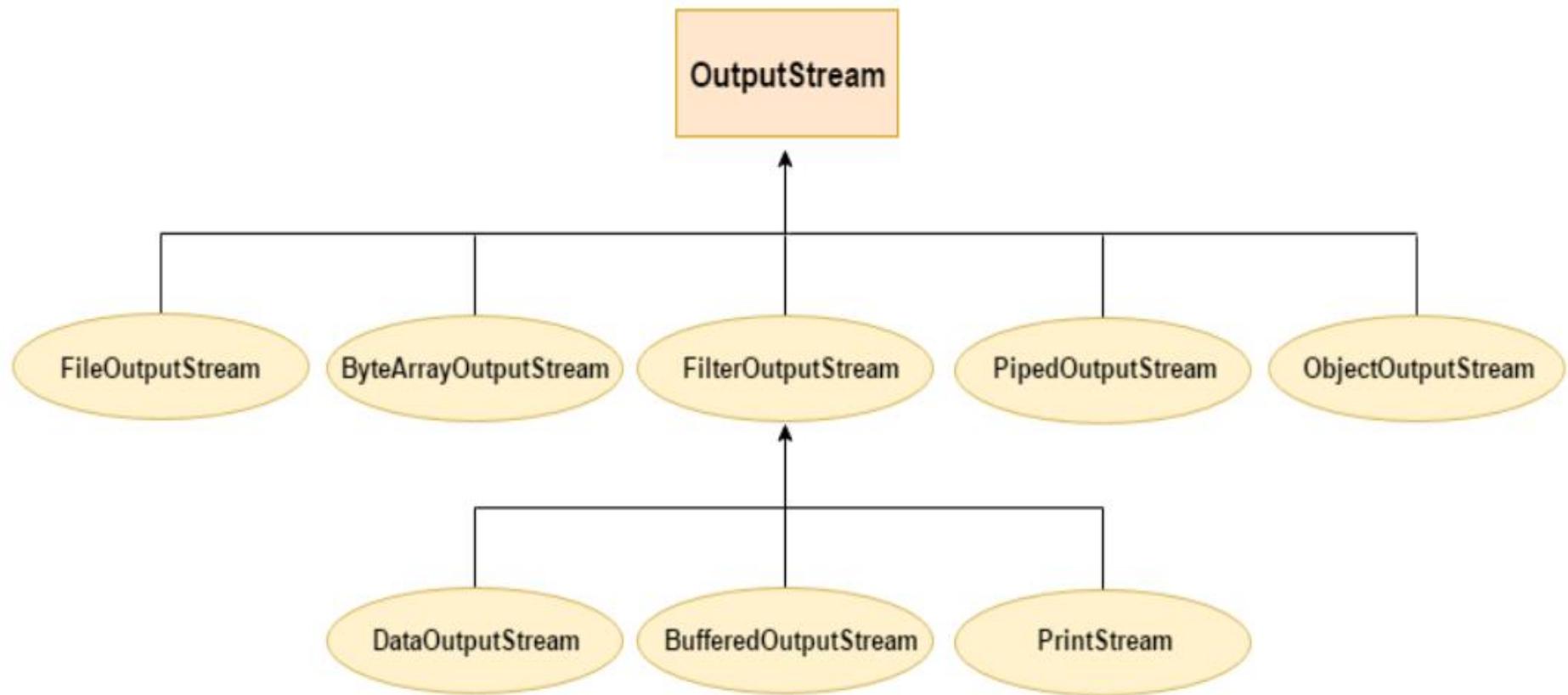
# OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

## Useful methods of OutputStream

Method	Description
1) public void write(int) throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException	flushes the current output stream.
4) public void close() throws IOException	is used to close the current output stream.

## OutputStream Hierarchy



## INPUTSTREAM

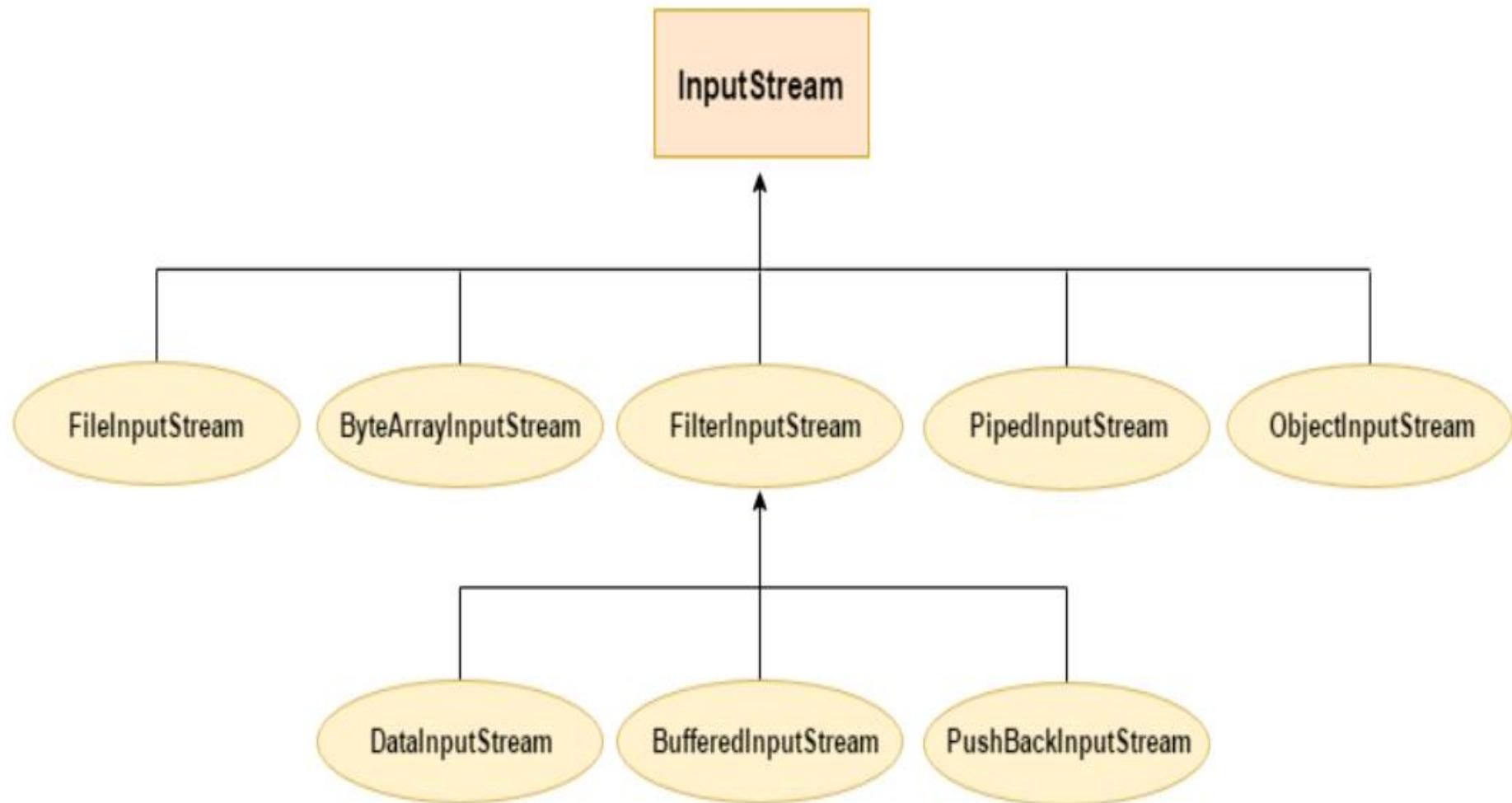
### InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

## InputStream Hierarchy



## FILE INPUT STREAM CLASS

Java FileInputStream class obtains input bytes from a file.

It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Let's see the declaration for java.io.FileInputStream class:  
**public class** FileInputStream **extends** InputStream ass.

## Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to <b>b.length</b> bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to <b>len</b> bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the <b>FileDescriptor</b> object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the <b>stream</b> .

## FILE OUTPUT STREAM CLASS

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

Let's see the declaration for Java.io.FileOutputStream class:

```
public class FileOutputStream extends OutputStream
```

## FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write <b>ary.length</b> bytes from the byte array <b>array</b> to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write <b>len</b> bytes from the byte array starting at offset <b>off</b> to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

## BUFFEREDOUTPUTSTREAM

Java BufferedOutputStream [class](#) is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D  
:\\"IO Package\\testout.txt"));
```

Let's see the declaration for Java.io.BufferedOutputStream class:

```
public class BufferedOutputStream extends FilterOutputStream
```

## Java BufferedOutputStream class constructors

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

## Java BufferedOutputStream class methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

## BUFFEREDINPUTSTREAM

Java `BufferedInputStream` class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about `BufferedInputStream` are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a `BufferedInputStream` is created, an internal buffer array is created.

Let's see the declaration for `Java.io.BufferedInputStream` class:

**public class** `BufferedInputStream` **extends** `FilterInputStream`

## Java BufferedInputStream class constructors

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves its argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves its argument, the input stream IS, for later use.

## Java BufferedInputStream class methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It reads the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It reads the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

## BYTE ARRAY OUTPUT STREAM

Java `ByteArrayOutputStream` class is used to **write common data** into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.

The `ByteArrayOutputStream` holds a copy of data and forwards it to multiple streams.

The buffer of `ByteArrayOutputStream` automatically grows according to data. Let's see the declaration for `Java.io.ByteArrayOutputStream` class:

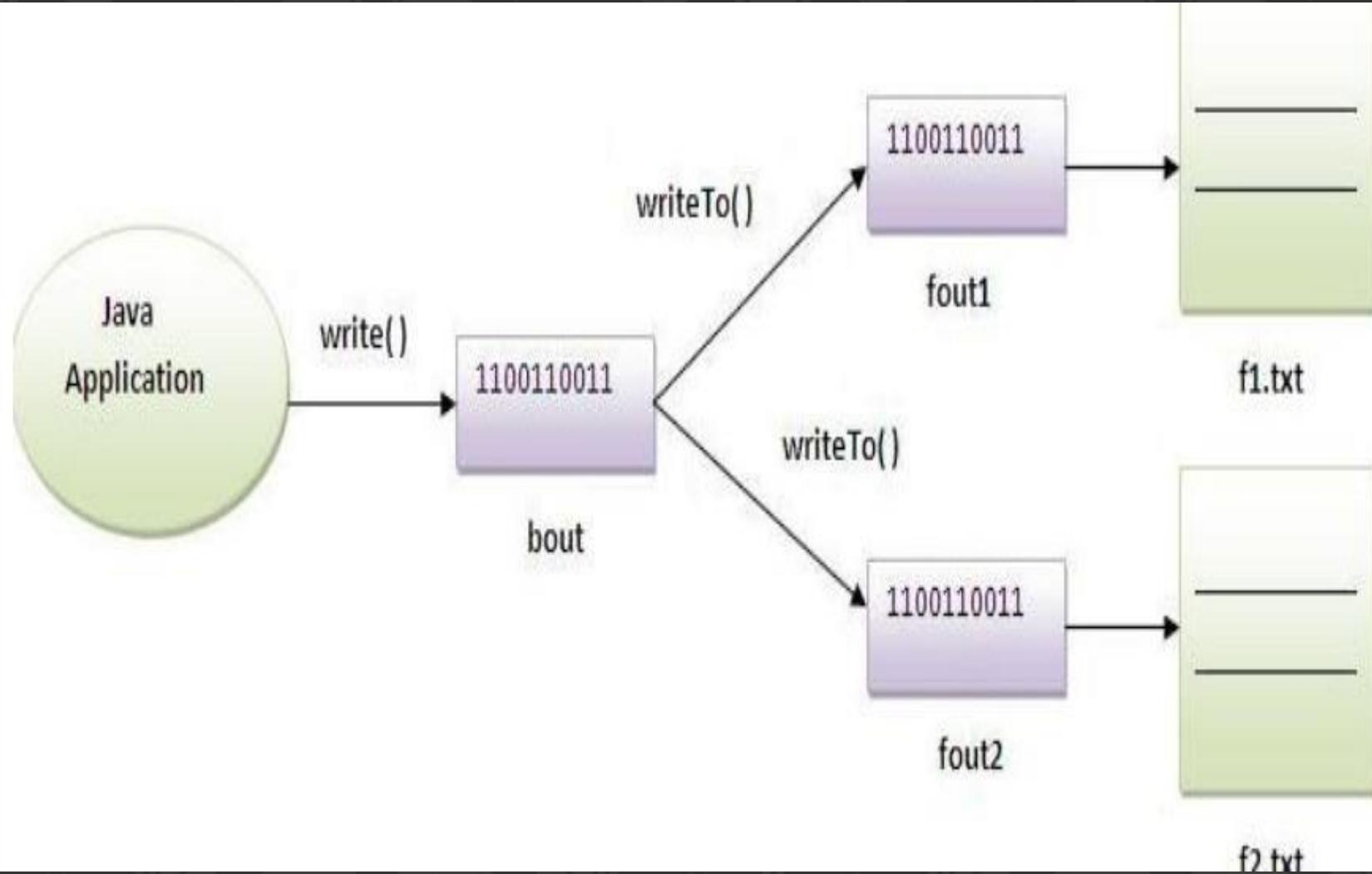
```
public class ByteArrayOutputStream extends OutputStream
```

# Java ByteArrayOutputStream class constructors

Constructor	Description
ByteArrayOutputStream()	Creates a new byte array output <b>stream</b> with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

## Java ByteArrayOutputStream class methods

Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a <b>string</b> decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len)	It is used for writing <b>len</b> bytes from specified byte array starting from the offset <b>off</b> to the byte array output stream.
void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.



## BYTE ARRAY INPUT STREAM

The `ByteArrayInputStream` is composed of two words: `ByteArray` and `InputStream`. As the name suggests, it can be used to read byte array as input stream.

Java `ByteArrayInputStream` class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

The buffer of `ByteArrayInputStream` automatically grows according to data.

Let's see the declaration for `Java.io.ByteArrayInputStream` class:

```
public class ByteArrayInputStream extends InputStream
```

## Java ByteArrayInputStream class constructors

Constructor	Description
ByteArrayInputStream(byte[] ary)	Creates a new byte array input stream which uses <b>ary</b> as its buffer array.
ByteArrayInputStream(byte[] ary, int offset, int len)	Creates a new byte array input stream which uses <b>ary</b> as its buffer array that can read up to specified <b>len</b> bytes of data from an array.

## Java ByteArrayInputStream class methods

Methods	Description
int available()	It is used to return the number of remaining bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read up to len bytes of data from an array of bytes in the input stream.
boolean markSupported()	It is used to test the input stream for mark and reset method.
long skip(long x)	It is used to skip the x bytes of input from the input stream.
void mark(int readAheadLimit)	It is used to set the current marked position in the stream.
void reset()	It is used to reset the buffer of a byte array.
void close()	It is used for closing a ByteArrayInputStream.

## JAVA FILE WRITER CLASS

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

### Java FileWriter class declaration

Let's see the declaration for Java.io.FileWriter class:

```
public class FileWriter extends OutputStreamWriter
```

### Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

## Methods of FileWriter class

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

## FILE READER CLASS IN JAVA

Java FileReader class is used to read data from the file. It returns data in byte format like `FileInputStream` class.

It is character-oriented class which is used for `file` handling in `java`.

### Java FileReader class declaration

Let's see the declaration for `Java.io.FileReader` class:

```
public class FileReader extends InputStreamReader
```

### Constructors of FileReader class

Constructor	Description
<code>FileReader(String file)</code>	It gets filename in <code>string</code> . It opens the given file in read mode. If file doesn't exist, it throws <code>FileNotFoundException</code> .
<code>FileReader(File file)</code>	It gets filename in <code>file</code> instance. It opens the given file in read mode. If file doesn't exist, it throws <code>FileNotFoundException</code> .

## Methods of FileReader class

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.
void close()	It is used to close the FileReader class.



# JAVA BUFFERED WRITER

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits [Writer](#) class. The buffering characters are used for providing the efficient writing of single [arrays](#), characters, and [strings](#).

## Class declaration

Let's see the declaration for Java.io.BufferedWriter class:

```
public class BufferedWriter extends Writer
```

## Class constructors

Constructor	Description
BufferedWriter(Writer wrt)	It is used to create a buffered character output stream that uses the default size for an output buffer.
BufferedWriter(Writer wrt, int size)	It is used to create a buffered character output stream that uses the specified size for an output buffer.

## Class methods

Method	Description
void newLine()	It is used to add a new line by writing a line separator.
void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

## BUFFERED READER CLASS

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by `readLine()` method. It makes the performance fast. It inherits [Reader class](#).

### Java BufferedReader class declaration

Let's see the declaration for Java.io.BufferedReader class:

```
public class BufferedReader extends Reader
```

### Java BufferedReader class constructors

Constructor	Description
<code>BufferedReader(Reader rd)</code>	It is used to create a buffered character input stream that uses the default size for an input buffer.
<code>BufferedReader(Reader rd, int size)</code>	It is used to create a buffered character input stream that uses the specified size for an input buffer.

## Java BufferedReader class methods

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an <a href="#">array</a> .
boolean markSupported()	It is used to test the input stream support for the mark and reset method.
String readLine()	It is used for reading a line of text.
boolean ready()	It is used to test whether the input stream is ready to be read.
long skip(long n)	It is used for skipping the characters.
void reset()	It repositions the <a href="#">stream</a> at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	It is used for marking the present position in a stream.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

# CHAR ARRAY READER CLASS

The CharArrayReader is composed of two words: CharArray and Reader. The CharArrayReader class is used to read character **array** as a reader (stream). It inherits **Reader** class.

## Java CharArrayReader class declaration

Let's see the declaration for Java.io.CharArrayReader class:

```
public class CharArrayReader extends Reader
```

## Java CharArrayReader class methods

Method	Description
int read()	It is used to read a single character
int read(char[] b, int off, int len)	It is used to read characters into the portion of an array.
boolean ready()	It is used to tell whether the stream is ready to read.
boolean markSupported()	It is used to tell whether the stream supports mark() operation.
long skip(long n)	It is used to skip the character in the input stream.
void mark(int readAheadLimit)	It is used to mark the present position in the stream.
void reset()	It is used to reset the stream to a most recent mark.
void close()	It is used to closes the stream.

## CHAR ARRAY WRITER CLASS

The CharArrayWriter class can be used to write common data to multiple files. This class inherits Writer class. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

### Java CharArrayWriter class declaration

Let's see the declaration for Java.io.CharArrayWriter class:

```
public class CharArrayWriter extends Writer
```

## Java CharArrayWriter class Methods

Method	Description
int size()	It is used to return the current size of the buffer.
char[] toCharArray()	It is used to return the copy of an input data.
String toString()	It is used for converting an input data to a <a href="#">string</a> .
CharArrayWriter append(char c)	It is used to append the specified character to the writer.
CharArrayWriter append(CharSequence csq)	It is used to append the specified character sequence to the writer.
CharArrayWriter append(CharSequence csq, int start, int end)	It is used to append the subsequence of a specified character to the writer.
void write(int c)	It is used to write a character to the buffer.
void write(char[] c, int off, int len)	It is used to write a character to the buffer.
void write(String str, int off, int len)	It is used to write a portion of string to the buffer.
void writeTo(Writer out)	It is used to write the content of buffer to different character stream.
void flush()	It is used to flush the stream.
void reset()	It is used to reset the buffer.
void close()	It is used to close the stream.

**UNIT-4**  
**EXCEPTION HANDLING**

## EXCEPTIONS

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception?

**Dictionary Meaning:** 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.

Advantage of Exception Handling

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 use exception handling.

# EXCEPTIONS

- *Exceptions* are events that occur during the execution of programs that disrupt the normal flow of instructions (e.g. divide by zero, array access out of bound, etc.).
- In Java, an exception is an **object** that wraps an error event that occurred within a method and contains:
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information
- Exception objects can be *thrown* and *caught*.

Exceptions are used to indicate many different types of error conditions.

## • **JVM Errors:**

- OutOfMemoryError
- StackOverflowError
- LinkageError

## System errors:

- FileNotFoundException
- IOException
- SocketTimeoutException

## • **Programming errors:**

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

## EXAMPLE

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in [Java](#).

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; //exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

# EXCEPTIONS

All exceptions and errors extend from a common `java.lang.Throwable` parent class. Only `Throwable` objects can be thrown and caught.

Other classes that have special meaning in Java are

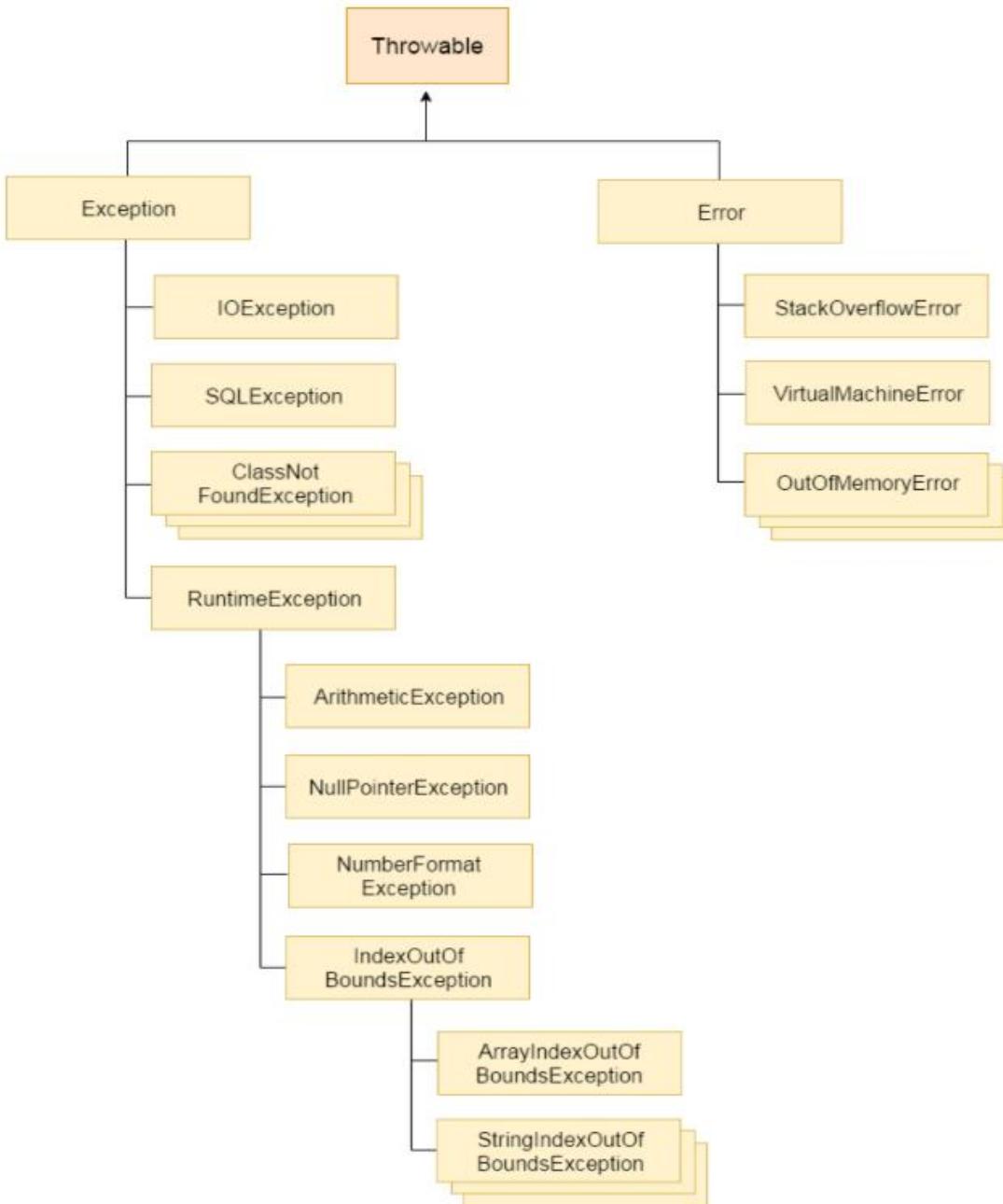
`java.lang.Error` (JVM Error), `java.lang.Exception` (System Error), and `java.lang.RuntimeException` (Programming Error).

```
public class java.lang.Throwable extends Object
    implements java.io.Serializable {
    public Throwable();
    public Throwable(String msg);
    public Throwable(String msg, Throwable cause);
    public Throwable(Throwable cause);
    public String getMessage();
    public String getLocalizedMessage();
    public Throwable getCause();
    public Throwable initCause(Throwable cause);
    public String toString();
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream);
    public void printStackTrace(java.io.PrintWriter);
    public Throwable fillInStackTrace();
    public StackTraceElement[] getStackTrace();
    public void setStackTrace(StackTraceElement[] stackTrace);}
```

	Error	Exception
Similarities	Both of them are generated in <code>java.lang.Throwable</code> package	
Differences	<b>Cannot be</b> handled	<b>Can be</b> handled
Examples	<code>java.lang.OutOfMemoryError</code>	<code>Java.lang.ArrayIndexOutOfBoundsException</code>
	<code>java.lang.NoClassDefFoundError</code>	<code>Java.lang.ArithmetricException</code>
	<code>java.lang.UnSupportedClassVersionError</code>	<code>Java.lang.NullPointerException</code>
	<code>Java.lang.NoSuchMethodError</code>	<code>Java.lang.NumberFormatException</code>
		<code>Java.lang.ClassNotFoundException</code>
		<code>Java.lang.NegativeArraySizeException</code>

## Hierarchy of Java Exception classes

The `java.lang.Throwable` class is the root class of Java Exception hierarchy which is inherited by two subclasses: `Exception` and `Error`. A hierarchy of Java Exception classes are given below:



# Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error



## TYPES

### Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionException etc.

# Checked vs. Unchecked Exceptions

Checked Exceptions	Unchecked Exceptions
Not subclass of RuntimeException	Subclass of RuntimeException
if not caught, method <i>must</i> specify it to be thrown	if not caught, method <i>may</i> specify it to be thrown
for errors that the programmer <i>cannot</i> directly prevent from occurring	For errors that the programmer <i>can</i> directly prevent from occurring
<code>IOException</code> , <code>FileNotFoundException</code> , <code>SocketException</code> , etc.	<code>NullPointerException</code> , <code>IllegalArgumentException</code> , <code>IllegalStateException</code> , etc.

## Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

# Java try 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 of 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.

## PROBLEM WITHOUT EXCEPTION HANDLING

Let's try to understand the problem if we don't use a try-catch block.

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

output: Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

## SOLUTION BY EXCEPTION HANDLING

Let's see the solution of the above problem by a java try-catch block.

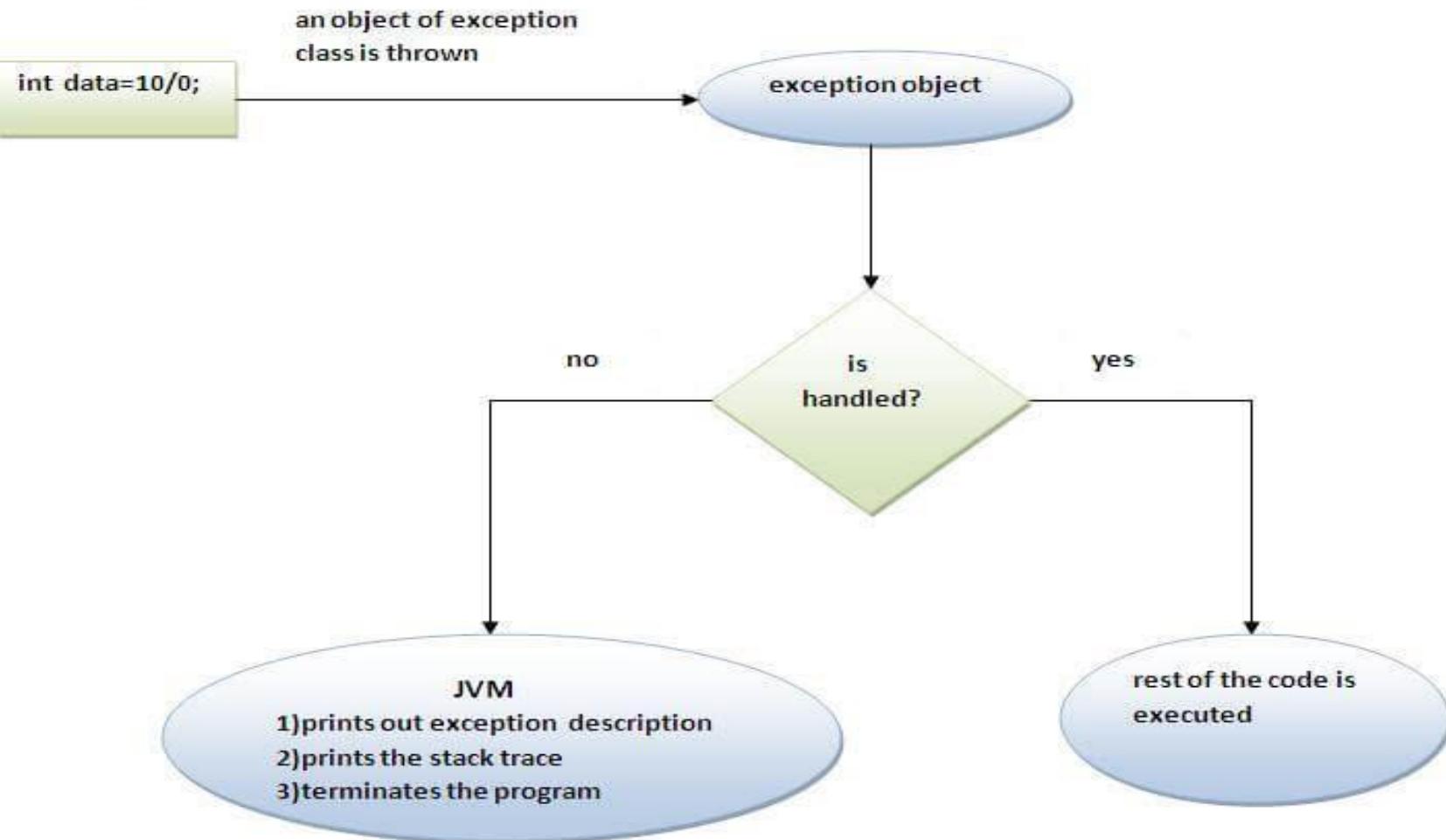
Example 2

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

Output: java.lang.ArithmetcException: / by zero rest of the code

Now, as displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

## INTERNAL WORKING OF TRY AND CATCH BLOCK



## INTERNAL WORKING OF JAVA TRY-CATCH BLOCK

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:

Prints out exception description.

Prints the stack trace (Hierarchy of methods where the exception occurred).

Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

## JAVA MULTI-CATCH BLOCK

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 `ArithmaticException` must come before catch for `Exception`.

## JAVA NESTED TRY BLOCK

The try block within a try block is known as nested try block in java. 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 1;  
        statement 2;      }  
    catch(Exception e)  
    { } } }  
catch(Exception e)  
{ } ....
```

## JAVA FINALLY BLOCK

**Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

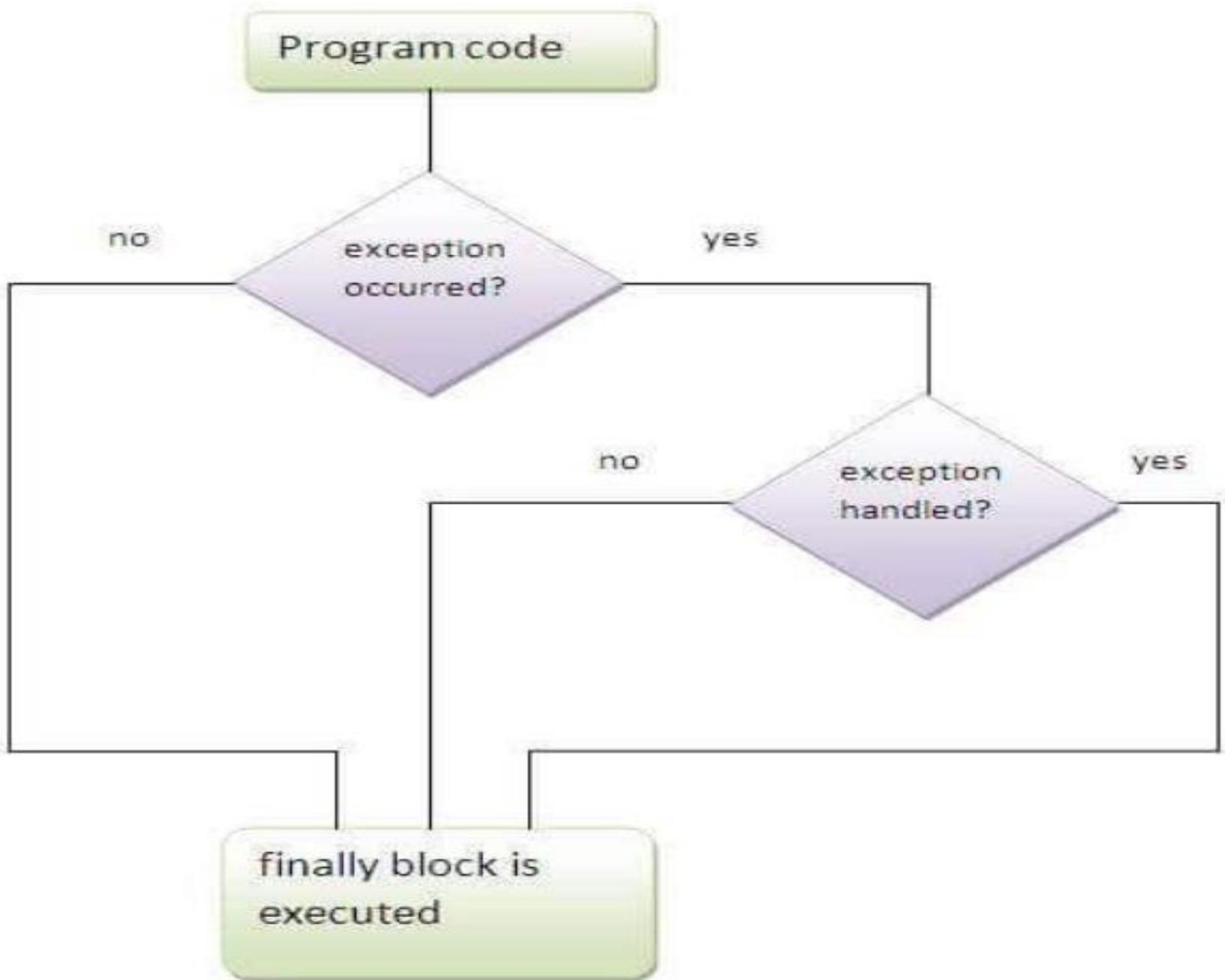
Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Let's see the different cases where java finally block can be used.

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).



## JAVA THROW KEYWORD

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

**throw** exception;

Let's see the example of throw IOException.

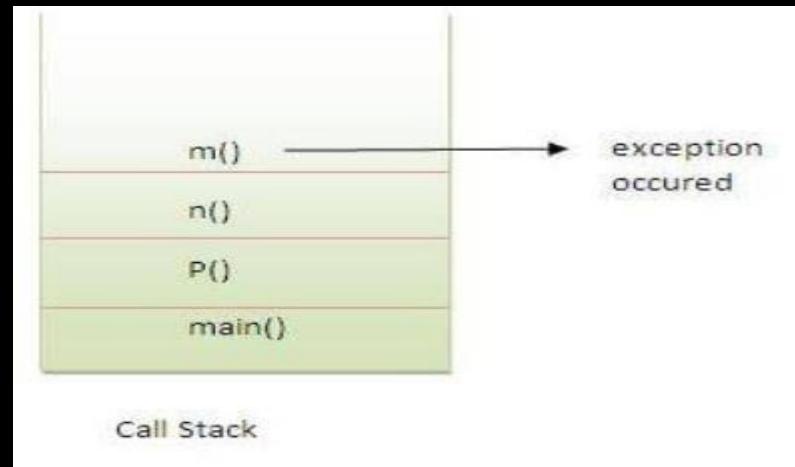
**throw new IOException("sorry device error");**

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

## JAVA EXCEPTION PROPAGATION

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).



## 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 normal flow 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 performing check up before the code being used.

```
return_type method_name() throws exception_class_name{  
//method code }
```

Which exception should be declared

**Ans)** checked exception only, because:

- unchecked Exception:** under your control so correct your code.
- error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

## JAVA THROWS EXAMPLE

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

**Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.**

1. There are two cases:**Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

Case2: You declare the exception

- A) In case you declare the exception, if exception does not occur, the code will be executed fine.
- B) In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

<b>throw</b>	<b>throws</b>
1. Java throw keyword is used to explicitly throw an exception	1. Java throws keyword is used to declare an exception.
2. void m(){ throw new ArithmeticException("sorry"); }	2. void m()throws ArithmeticException{ //method code }
3. Checked exception cannot be propagated using throw only.	3. Checked exception can be propagated with throws.
4. Throw is followed by an instance.	4. Throw is followed by a class.
5. Throw is used within the method.	5. Throws is used with the method signature.
6. You cannot throw multiple exceptions.	6. You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

# Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

## Types of Exceptions

User-Defined Exception

Built-in Exception

Checked Exceptions

Unchecked Exceptions

- ClassNotFoundException
- InterruptedException
- IOException
- InstantiationException
- SQLException
- FileNotFoundException

- ArithmeticException
- ClassCastException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ArrayStoreException
- IllegalThreadStateException

## BUILT IN EXCEPTIONS

Many exceptions and errors are automatically generated by the java virtual machine.

Errors generally abnormal situations in the JVM such as:

Running out of memory

Infinite recursion

Inability to link to another class

Run time exceptions, are generally a result of programming errors such as:

Dereferencing a null reference

Trying to read outside the bounds of an array

Dividing an integer value by zero.

## CHECKED EXCEPTION

**Checked** exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensure whether the programmer handles the exception or not.

```
import java.io.*;
class CheckedExceptionExample {
    public static void main(String args[]) {
        FileInputStream file_data = null;
        file_data = new FileInputStream("Akhila.txt");
        int m;
        while(( m = file_data.read() ) != -1) {
            System.out.print((char)m);
        }
        file_data.close();
    }
}
```

In the above code, we are trying to read the **Akhila.txt** file and display its data or content on the screen. The program throws following exceptions

The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.

The **read()** method of the **FileInputStream** class throws the **IOException**.

The **close()** method also throws the **IOException**.

## HOW TO RESOLVE THE ERROR?

There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws**. We only declare the IOException, not FileNotFoundException, because of the child-parent relationship. The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

```
class Exception{  
    public static void main(String args[]) throws IOException {  
        ...  
        ... }  
    }
```

If we compile and run the code, the errors will disappear, and we will see the data of the file.

CONT...

We can also handle these exception using **try-catch** However, the way which we have used above is not correct.

We have to give a meaningful message for each exception type.

By doing that it would be easy to understand the error.

We will use the try-catch block in the following way:

## UNCHECKED EXCEPTION

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time.

In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Usually, it occurs when the user provides bad data during the interaction with the program.

```
class UncheckedExceptionExample1 {  
    public static void main(String args[]) {  
        int positive = 35;  
        int zero = 0;  
        int result = positive/zero;  
        //Give Unchecked Exception here.  
        System.out.println(result); } }
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an `ArithmaticException` error at runtime. On dividing a number by 0 throws the divide by zero exception.

CONT...

Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

```
class UncheckedException1 {  
    public static void main(String args[]) {  
        int num[] = {10,20,30,40,50,60};  
        System.out.println(num[7]);    } }
```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

## USER DEFINED EXCEPTION

In Java, we already have some built-in exception classes like **ArrayIndexOutOfBoundsException**, **NullPointerException**, and **ArithmaticException**.

These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the **Exception** class.

We can throw our own exception on a particular condition using the **throw** keyword.

For creating a user-defined exception, we should have basic knowledge of the **try-catch** block and **throw keyword**.

Bugs or errors that we don't want and restrict the normal execution of the programs are referred to as **exceptions**.

**ArithmaticException**, **ArrayIndexOutOfBoundsExceptions**, **ClassNotFoundExc**  
**ptions** etc. are come in the category of **Built-in Exception**.

Sometimes, the built-in exceptions are not sufficient to explain or describe certain situations.

For describing these situations, we have to create our own exceptions by creating an exception class as a subclass of the **Exception** class.

These types of exceptions come in the category of **User-Defined Exception**.

S.No	<b>Checked Exception</b>	<b>Unchecked Exception</b>
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidCastException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

## CHAINED EXCEPTIONS

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception.

For example, consider a situation in which a method throws an `ArithmaticException` because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only `ArithmaticException` to the caller. So the caller would not come to know about the actual cause of exception.

Chained Exception is used in such type of situations.

**Constructors** Of `Throwable` class Which support chained exceptions in java:

1. `Throwable(Throwable cause)` :- Where `cause` is the exception that causes the current exception.

2. `Throwable(String msg, Throwable cause)` :- Where `msg` is the exception message and `cause` is the exception that causes the current exception.

**Methods** Of `Throwable` class Which support chained exceptions in java :

1. `getCause()` method :- This method returns actual cause of an exception.

2. `initCause(Throwable cause)` method :- This method sets the cause for

## MULTI THREADING IN JAVA

## MULTI THREADING IN JAVA

**Multithreading in Java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area.

They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- 2) You **can perform many operations together, so it saves time.**

Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## MULTI TASKING

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
  - Thread-based Multitasking (Multithreading)
- 1) Process-based Multitasking (Multiprocessing)
    - Each process has an address in memory. In other words, each process allocates a separate memory area.
    - A process is heavyweight.
    - Cost of communication between the process is high.
    - Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.
  - 2) Thread-based Multitasking (Multithreading)
    - Threads share the same address space.
    - A thread is lightweight.
    - Cost of communication between the thread is low.
    - Note: At least one process is required for each thread.

## THREAD

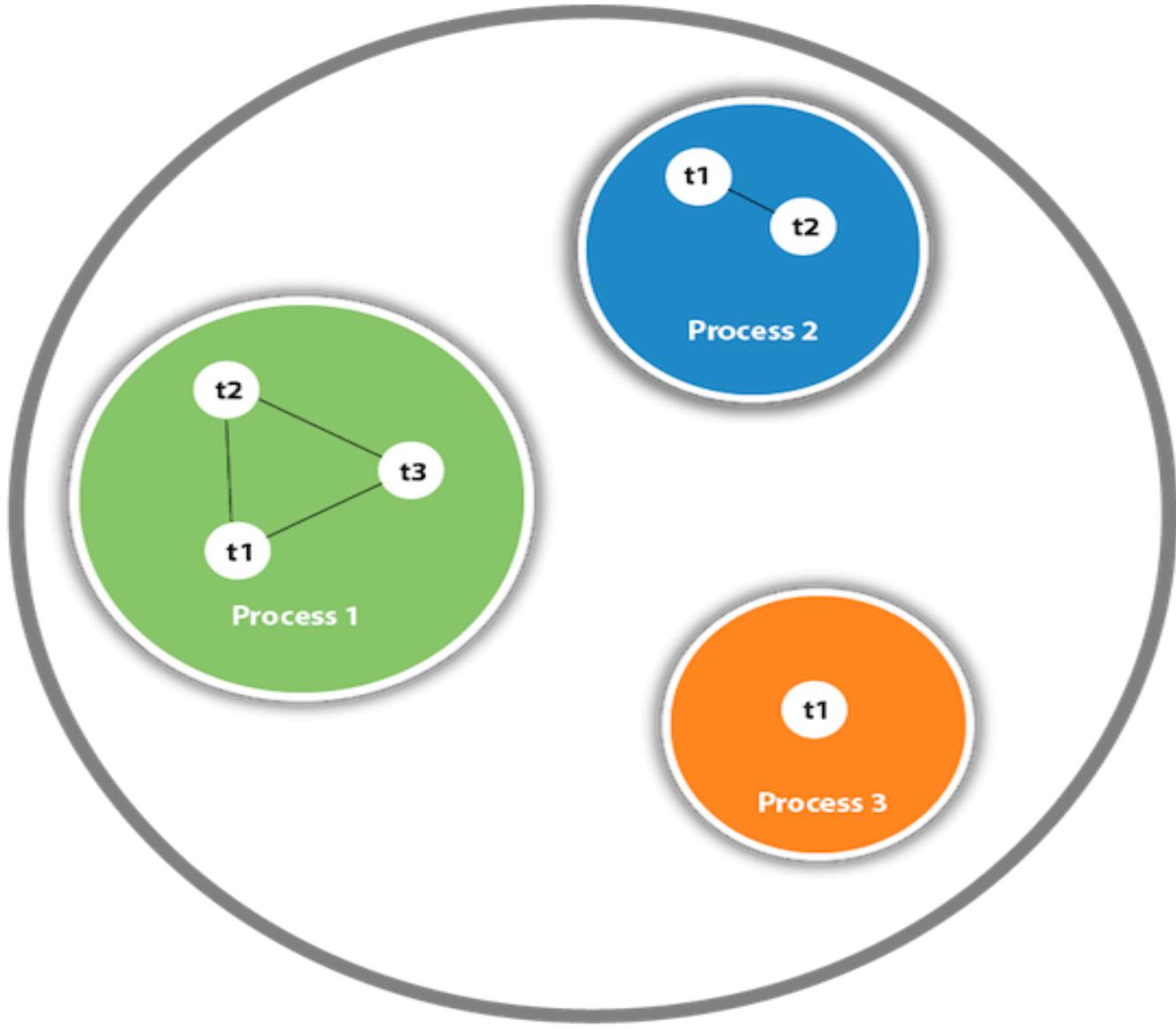
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.



**OS**

## Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	<code>start()</code>	It is used to start the execution of the thread.
2)	void	<code>run()</code>	It is used to do an action for a thread.
3)	static void	<code>sleep()</code>	It sleeps a thread for the specified amount of time.
4)	static Thread	<code>currentThread()</code>	It returns a reference to the currently executing thread object.
5)	void	<code>join()</code>	It waits for a thread to die.
6)	int	<code>getPriority()</code>	It returns the priority of the thread.
7)	void	<code>setPriority()</code>	It changes the priority of the thread.
8)	String	<code>getName()</code>	It returns the name of the thread.
9)	void	<code>setName()</code>	It changes the name of the thread.
10)	long	<code>getId()</code>	It returns the id of the thread.
11)	boolean	<code>isAlive()</code>	It tests if the thread is alive.
12)	static void	<code>yield()</code>	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	<code>suspend()</code>	It is used to suspend the thread.
14)	void	<code>resume()</code>	It is used to resume the suspended thread.
15)	void	<code>stop()</code>	It is used to stop the thread.
16)	void	<code>destroy()</code>	It is used to destroy the thread group and all of its subgroups.

17)	boolean	<code>isDaemon()</code>	It tests if the thread is a daemon thread.
18)	void	<code>setDaemon()</code>	It marks the thread as daemon or user thread.
19)	void	<code>interrupt()</code>	It interrupts the thread.
20)	boolean	<code>isInterrupted()</code>	It tests whether the thread has been interrupted.
21)	static boolean	<code>interrupted()</code>	It tests whether the current thread has been interrupted.
22)	static int	<code>activeCount()</code>	It returns the number of active threads in the current thread's thread group.
23)	void	<code>checkAccess()</code>	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	<code>holdLock()</code>	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	<code>dumpStack()</code>	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	<code>getStackTrace()</code>	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	<code>enumerate()</code>	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	<code>getState()</code>	It is used to return the state of the thread.
29)	ThreadGroup	<code>getThreadGroup()</code>	It is used to return the thread group to which this thread belongs
30)	String	<code>toString()</code>	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.

31)	void	<code>notify()</code>	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	<code>notifyAll()</code>	It is used to give the notification to all waiting threads of a particular object.
33)	void	<code>setContextClassLoader()</code>	It sets the context ClassLoader for the Thread.
34)	ClassLoader	<code>getContextClassLoader()</code>	It returns the context ClassLoader for the thread.
35)	static  Thread.UncaughtExceptionHandler	<code>getDefaultUncaughtExceptionHandler()</code>	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	<code>setDefaultUncaughtExceptionHandler()</code>	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

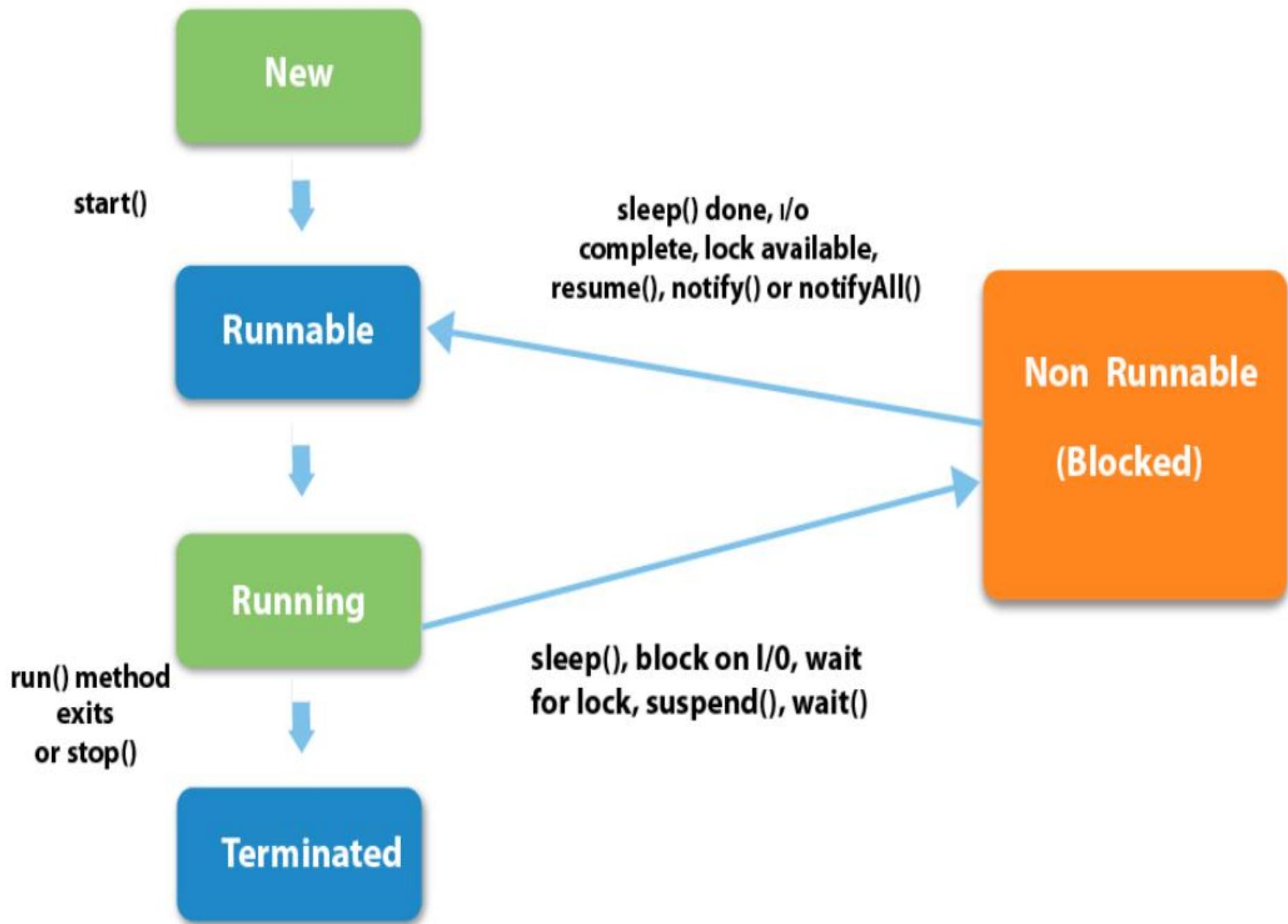
## LIFE CYCLE OF A THREAD

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

- 1.New
- 2.Runnable
- 3.Running
- 4.Non-Runnable (Blocked)
- 5.Terminated



## 1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

## 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

## 3) Running

The thread is in running state if the thread scheduler has selected it.

## 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

## 5) Terminated

A thread is in terminated or dead state when its run() method exits.

## OBJECT ORIENTED PROGRAMMING

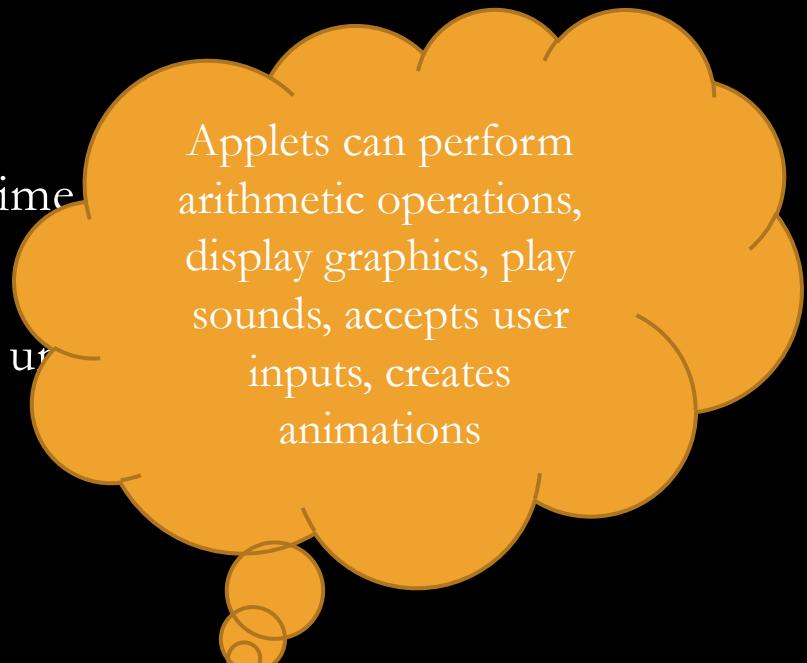
Lecture #36: Graphics Programming

## JAVA APPLET

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

### Advantage of Applet

- It works at client side so less response time
- Secured
- It can be executed by browsers running under Linux, Windows, Mac Os etc.



Applets can perform arithmetic operations, display graphics, play sounds, accepts user inputs, creates animations

## APPLETS

How applets differ from applications:

- Applets do not use main() method
- Applets cannot run independently. They run inside a webpage using HTML code
- Applets cannot read from or write to the files in the local computer
- Applet are restricted from using libraries from other languages such as C or C++.

These restrictions ensure security and cannot do any damage to the local system

## APPLETS

Steps involved in developing and testing an applet are:

1. Building an applet code (.java code)
2. Creating an executable applet (.class file)
3. Designing a webpage using HTML tags
4. Preparing <APPLET> tag
5. Incorporating <APPLET> tag into web page (creating HTML file)
6. Testing the applet code.

## 1. BUILDING AN APPLET

### Applet Life cycle:

`java.applet.Applet class`

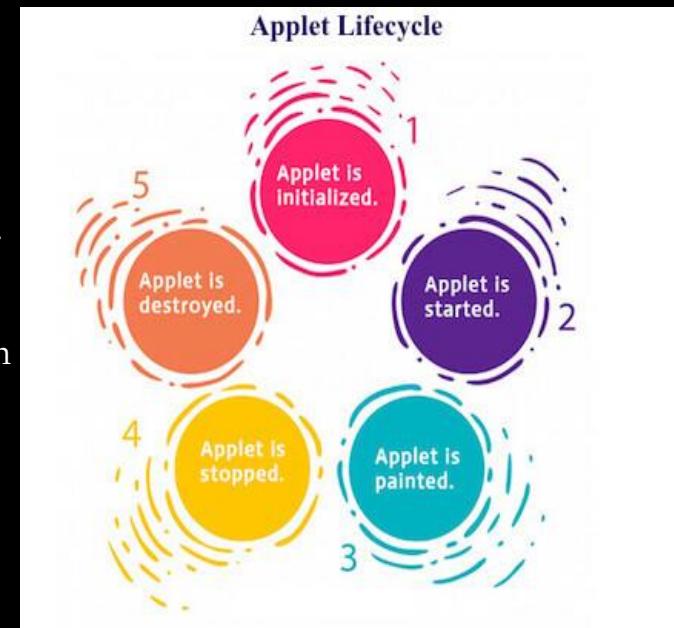
For creating any applet `java.applet.Applet` class must be inherited. It provides 4 life cycle methods of applet.

1. `public void init()`: is used to initialized the Applet. It is invoked only once.
2. `public void start()`: is invoked after the `init()` method or browser is maximized. It is used to start the Applet.
3. `public void stop()`: is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. `public void destroy()`: is used to destroy the Applet. It is invoked only once.

`java.awt.Component class`

The Component class provides 1 life cycle method of applet.

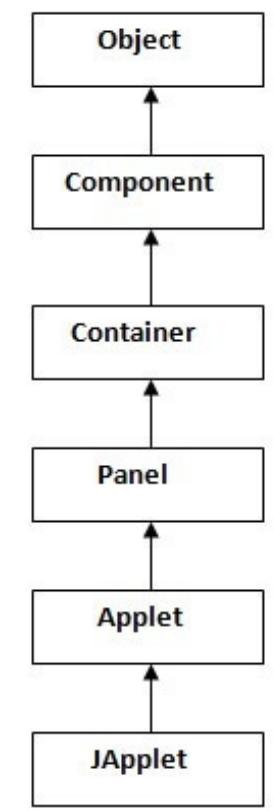
1. `public void paint(Graphics g)`: is used to paint the Applet. It provides `Graphics` class object that can be used for drawing oval, rectangle, arc etc.



```
import java.applet.*;
import java.awt.*;
public class appletclassname extends Applet
{
    .....
    .....
    public void paint( Graphics g)
    {
        ..... // Applet operations code
        .....
    }
    .....
}
```

```
import java.applet.*;
import java.awt.*;
public class Helloapplet extends Applet
{
    public void paint( Graphics g)
    {
        g.drawString("Hello Java",10,100);
    }
}
```

Hierarchy of Applet---->



Helloapplet.java

## **2. CREATING AN EXECUTABLE APPLET (.CLASS FILE)**

.class file is obtained by compiling the source code of the applet.

```
javac Helloapplet.java
```

this generates Helloapplet.class file.

### **3. DESIGNING A WEBPAGE USING HTML TAGS**

In order to run a java applet, it is first necessary to have a webpage that references the applet.

Web page is made with HTML tags.

```
<HTML>
<HEAD>
...
</HEAD>
<BODY>
....
</BODY>
</HTML>
```

<APPLET> tag is used.

## 4.PREPARING <APPLET> TAG

<APPLET> tag supplies the name of the applet to be loaded and tells browser how much space required by the applet.

This specifies three things

1. name of the applet
2. Width of the applet(in pixels)
3. Height of the applet(in pixels)

Example:

```
<APPLET  
code=Helloapplet.class  
width=400  
height=200 >  
</APPLET>
```

## 5. CREATING HTML FILE

```
<HTML>
<HEAD>
    <TITLE> Welcome to Java Applets</TITLE>
</HEAD>
<BODY>

<APPLET
    code=Helloapplet.class
    width=400
    height=200 >
</APPLET>

</BODY>

</HTML>
```

Helloapplet.html

## 6.TESTING THE APPLET CODE.

Our current directory will have following things:

Helloapplet.java

Helloapplet.class

Helloapplet.html

To run an applet, we require one of the folowing:

1. java enabled web browser(entire webpage can be seen)
2. java applet viewer(only applet output)

Syntax:

appletviewer Helloapplet.html

## OBJECT ORIENTED PROGRAMMING

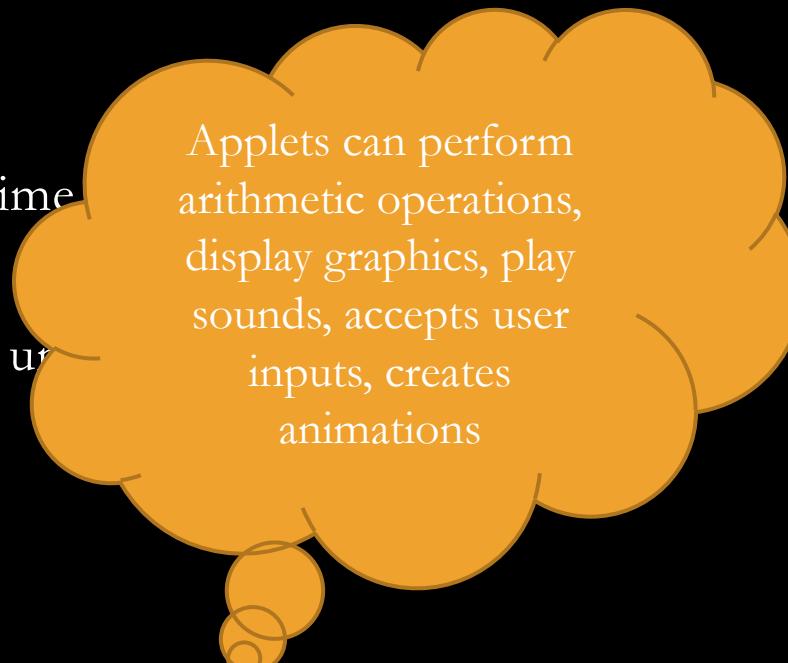
Lecture #37: Graphics Programming  
(Developing and Testing an Applet)

## JAVA APPLET

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

### Advantage of Applet

- It works at client side so less response time
- Secured
- It can be executed by browsers running under Linux, Windows, Mac Os etc.



Applets can perform arithmetic operations, display graphics, play sounds, accepts user inputs, creates animations

## APPLETS

Steps involved in developing and testing an applet are:

1. Building an applet code (.java code)
2. Creating an executable applet (.class file)
3. Designing a webpage using HTML tags
4. Preparing <APPLET> tag
5. Incorporating <APPLET> tag into web page (creating HTML file)
6. Testing the applet code.

```
import java.applet.*;
import java.awt.*;
public class Helloapplet extends Applet
{
    public void paint( Graphics g)
    {
        g.drawString("Hello Java",10,100);
    }
}
```

Helloapplet.java

Helloapplet.class

javac Helloapplet.java

Helloapplet.html

```
<HTML>
<HEAD>
    <TITLE> Welcome to Java Applets</TITLE>
</HEAD>
<BODY>

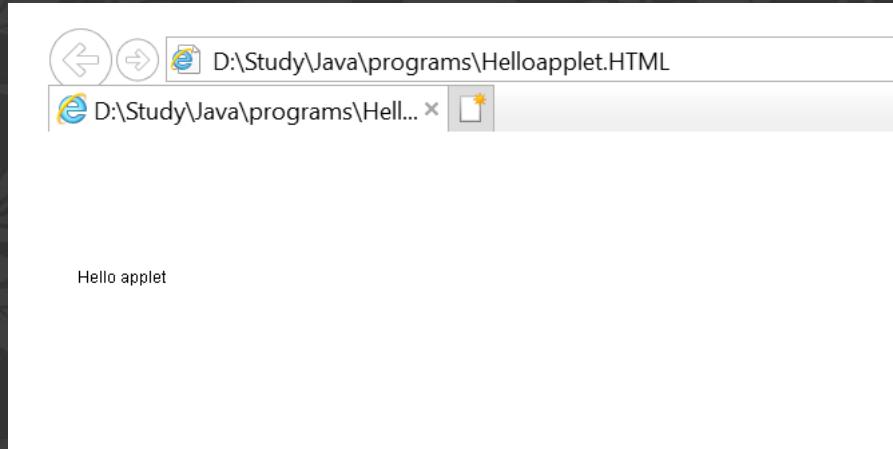
<APPLET
    code=Helloapplet.class
    width=400
    height=200 >
</APPLET>

</BODY>

</HTML>
```

To run an applet, we require one of the following:

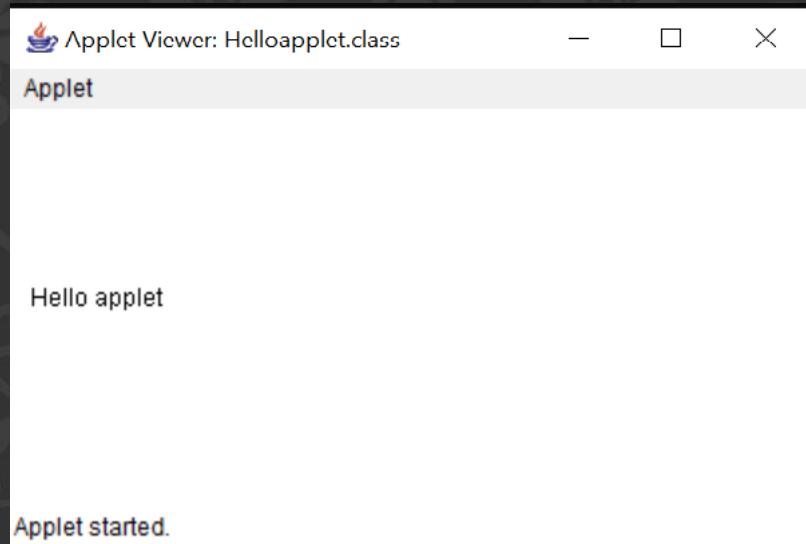
1. java enabled web browser(entire webpage can be seen)
1. java applet viewer(only applet output)



Applet Window----->

Syntax:

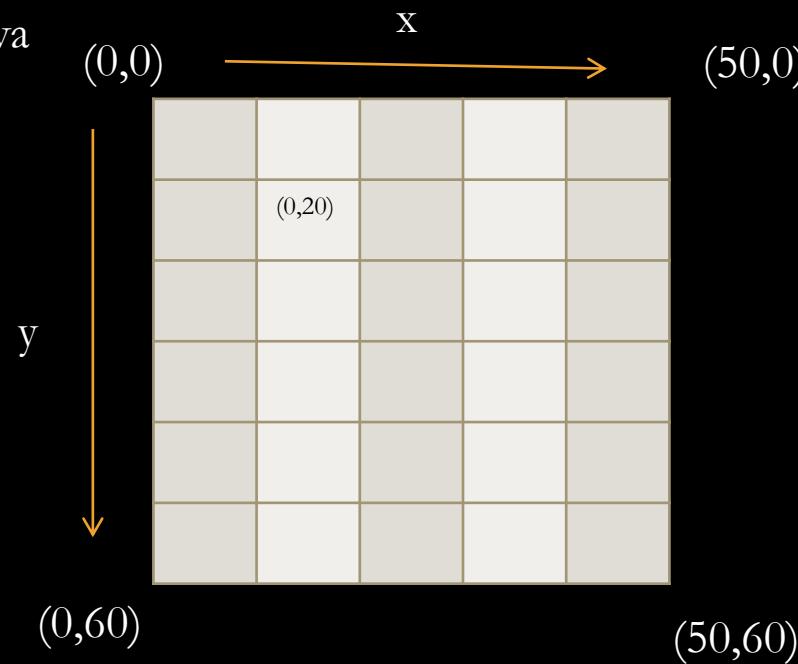
appletviewer Helloapplet.html



## WORKING WITH 2D SHAPES

Graphics class includes methods for drawing many different types of shapes.

Coordinate System of java



## DRAWING METHODS

### Graphics class

Method	Description
<code>drawString()</code>	Displays a text String
<code>drawLine()</code>	Draws a straight Line
<code>drawArc()</code>	Draws a hollow Arc
<code>drawRect()</code>	Draws a hollow Rectangle
<code>drawOval()</code>	Draws a hollow Oval
<code>drawRoundRect()</code>	Draws a hollow rectangle with rounded corners
<code>drawPolygon()</code>	Draws a hollow polygon
<code>fillArc()</code>	Draws a filled Arc
<code>fillPolygon()</code>	Draws a filled Polygon
<code>fillRect()</code>	Draws a filled Rectangle
<code>fillRoundRect()</code>	Draws a filled rectangle with rounded corners
<code>getColor()</code>	Retrieves the current drawing color
<code>getFont()</code>	Retrieves the currently used font
<code>setColor()</code>	Sets the drawing color
<code>setFont()</code>	Sets the font

# OBJECT ORIENTED PROGRAMMING

Lecture #38: Nested class/Inner class

## INNER CLASS OR NESTED CLASS

In java, just like methods and variables, it can have a class too.

Java inner class or nested class is a class which is declared inside the class or interface.

Syntax:

```
class Java_Outer_class{  
    //code  
    class Java_Inner_class{  
        //code  
    }  
}
```

## INNER CLASS OR NESTED CLASS

### Types of Nested classes

Two types

1. Static nested class
2. Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class

## STATIC NESTED CLASS

A static class i.e. created inside a class is called static nested class in java

Syntax:

```
class Java_Outer_class{  
    //code  
    static class Java_Inner_class{  
        //code  
    }  
}
```

static inner class instance can be created outside the class

Syntax

Syntax:

```
Outer.Inner o1=new Outer.Inner();
```

No need to  
create outer  
class object

## **STATIC NESTED CLASS**

Rules:

- It can be accessed by outer class name.
- Static nested class cannot access non-static (instance) data member or method.
- It can access static data members of outer class including private.
- static inner class can be public, private, protected and default
- Outer class should be public or default.

## **INNER CLASS OR NESTED CLASS**

### Types of Nested classes

Two types

1. Static nested class
2. Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class

## NON-STATIC NESTED CLASS (INNER CLASS)

- Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Java_Outer_class{  
    //code  
    class Java_Inner_class{  
        //code  
    }  
}
```

Example

Outer class Object is required.

## **NON-STATIC NESTED CLASS (INNER CLASS)**

- Anonymous inner class

A class that have no name is known as anonymous inner class in java.

It should be used if you have to override method of class or interface.

Example

## **NON-STATIC NESTED CLASS (INNER CLASS)**

- Local inner class

A class i.e. created inside a method is called local inner class in java.

If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Example

# OBJECT ORIENTED PROGRAMMING

## Lecture #39: Event Handling (Introduction)

## EVENT HANDLING

### Event?

Change in the state of an object is known as event i.e. event describes the change in state of source.

Events are generated as result of user interaction with the graphical user interface components.

For example: press a button, enter a character in textbox, click or drag a mouse, etc.

### Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.

This mechanism have the code which is known as event handler that is executed when an event occurs. (Delegation Event Model)

## DELEGATION EVENT MODEL)

### key participants

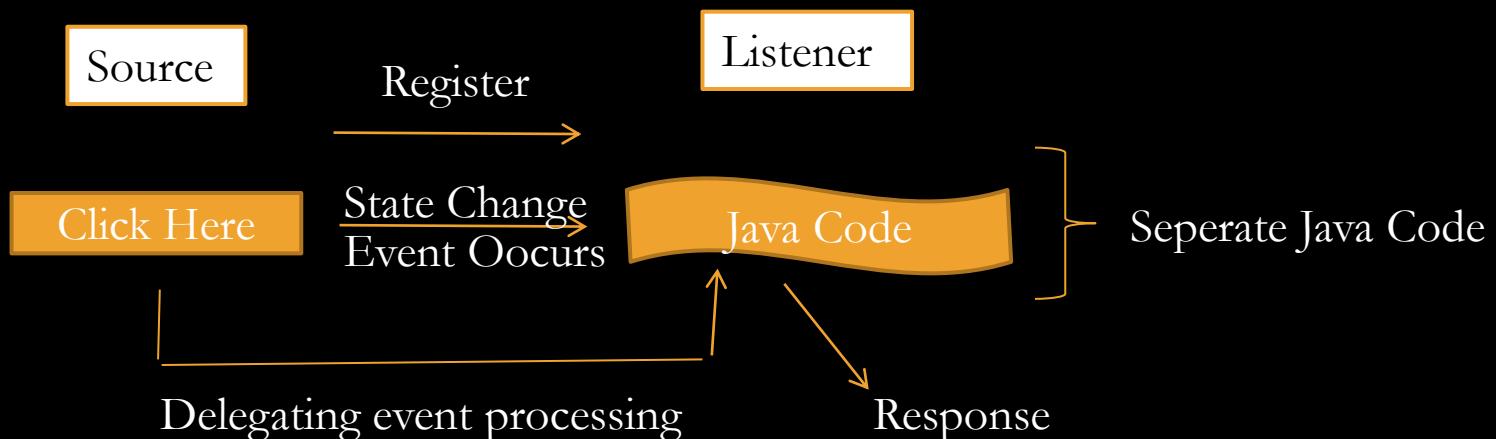
- Source

The source is an object on which event occurs.

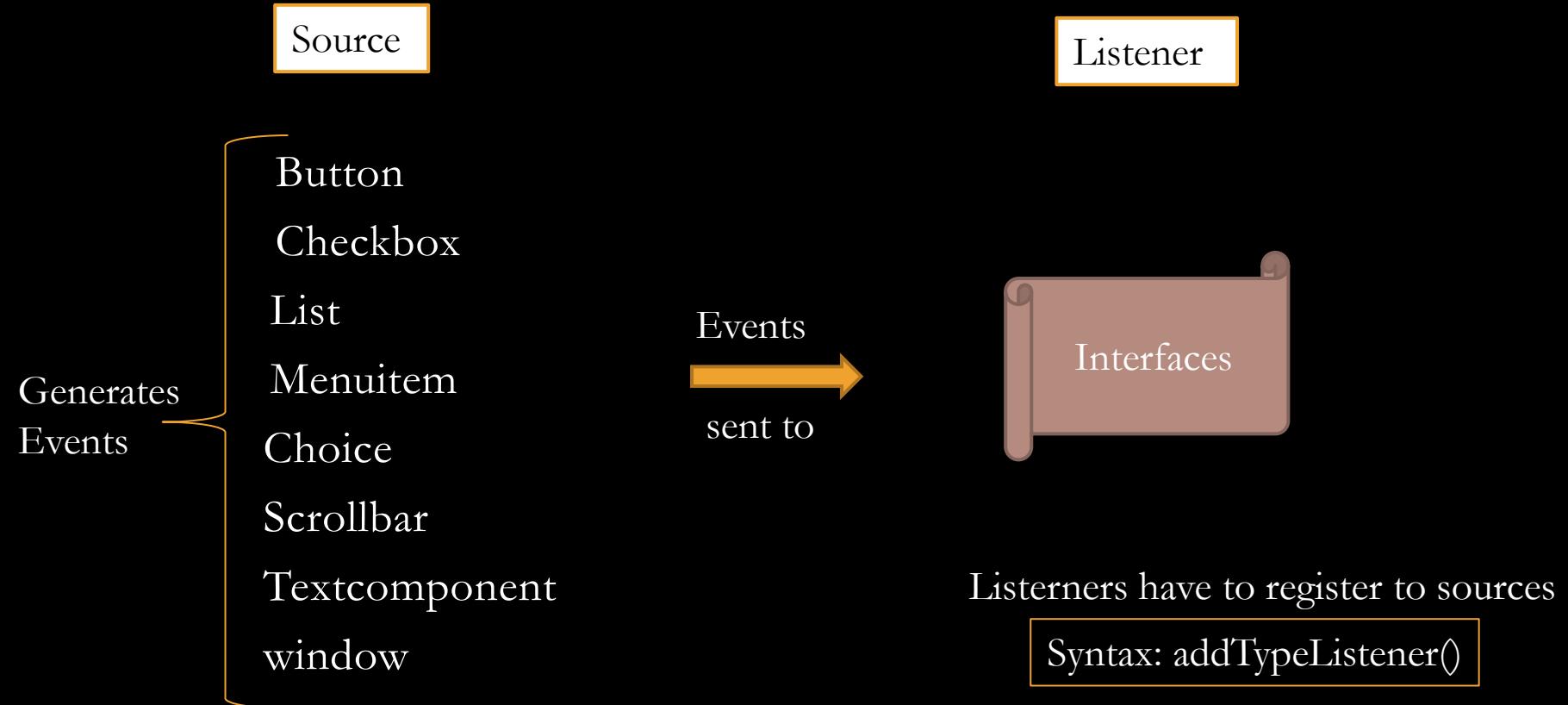
- Listener

It is also known as event handler. Listener is responsible for generating response to an event.

Benefit:  
User interface  
design is separated  
from event handler



## EVENT HANDLING



## EVENT HANDLING

import  
java.awt.event.\*  
package

### Event Classes

ActionEvent  
AdjustmentEvent  
ComponentEvent  
ContainerEvent  
FocusEvent  
ItemEvent  
KeyEvent  
MouseEvent  
MouseWheelEvent  
TextEvent  
WindowEvent

### Source

Button, MenuItem, List  
Component  
Component  
Component  
Component  
Checkbox, choice  
Textcomponent  
Mouse movements  
MouseWheelmovement  
Textcomponent  
Window

### Interfaces

ActionListener  
AdjustmentListner  
ComponentListener  
ContainerListner  
FocusListener  
ItemListener  
KeyListener  
MouseListener, MouseMotionListener  
MouseWheelListener  
TextListener  
WindowListener

## EVENT HANDLING

Template:

1. import packages

```
import java.awt.*;  
java.awt.event.*;  
java.applet.*;
```

2. Class should extend the Applet class and implement the interfaces

3. Write applet code

4. init() method: Register the Listener interface

Syntax: addTypeListener()

5. Write Event methods defintion

# OBJECT ORIENTED PROGRAMMING

## Lecture #40: Mouse Events

## EVENT HANDLING

### Event?

Change in the state of an object is known as event i.e. event describes the change in state of source.

Events are generated as result of user interaction with the graphical user interface components.

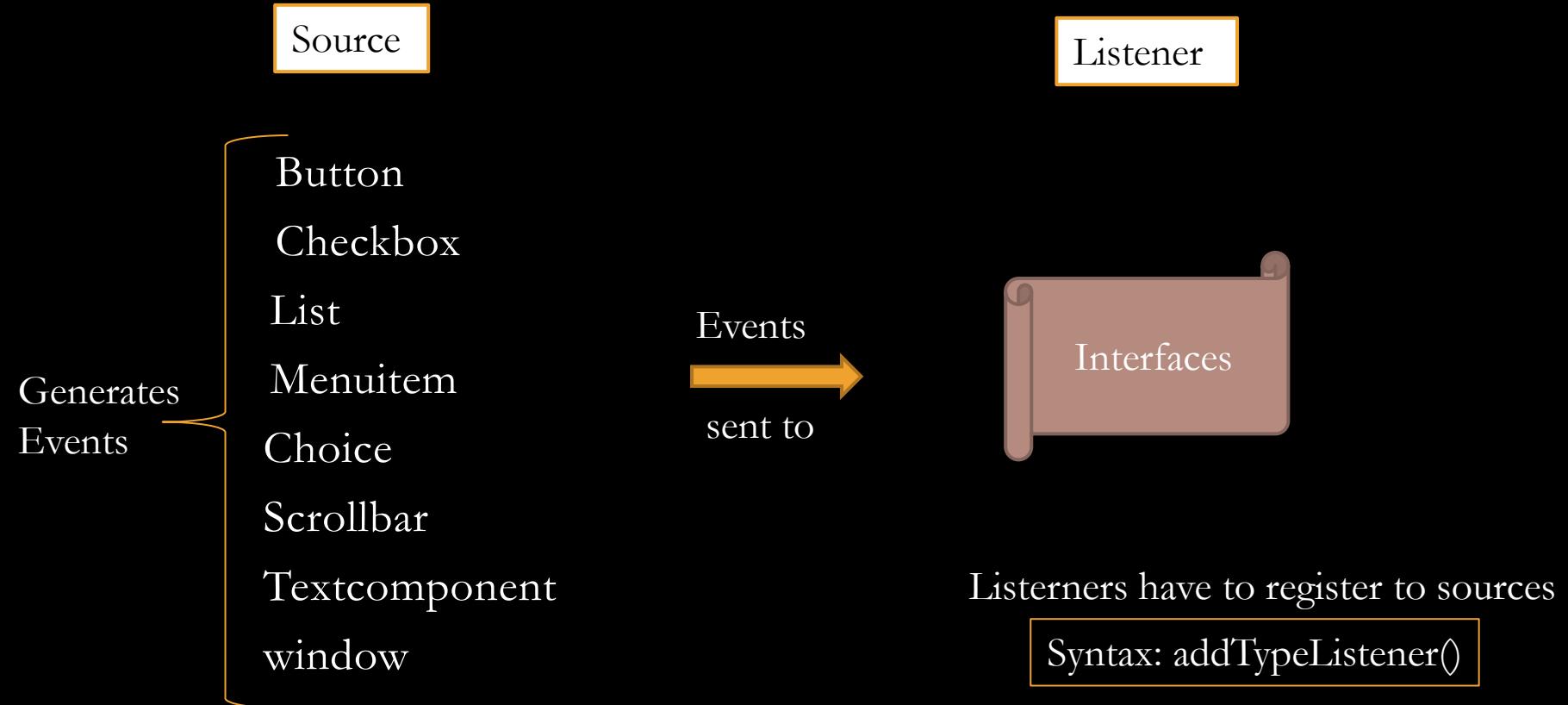
For example: press a button, enter a character in textbox, click or drag a mouse, etc.

### Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.

This mechanism have the code which is known as event handler that is executed when an event occurs. (Delegation Event Model)

## EVENT HANDLING



## EVENT HANDLING

Event Classes	Source	Interfaces
ActionEvent	Button, MenuItem, List	ActionListener
AdjustmentEvent	Component	AdjustmentListner
ComponentEvent	Component	ComponentListener
ContainerEvent	Component	ContainerListner
FocusEvent	Component	FocusListener
ItemEvent	Checkbox, choice	ItemListener
<u>KeyEvent</u>	Textcomponent	KeyListener
<u>MouseEvent</u>	Mouse movements	MouseListener, MouseMotionListener
MouseWheelEvent	MouseWheelmovement	MouseWheelListener
TextEvent	Textcomponent	TextListener
WindowEvent	Window	WindowListener

## MOUSE EVENTS

Class

Interfaces

Methods

MouseListener

MouseEvent

MouseMotionListener

mousePressed(),  
mouseClicked(),  
mouseEntered(),  
mouseExited(),  
mouseReleased()

mouseMoved(),  
mouseDragged

getX(), getY()

## MOUSE EVENTS

Template:

1. import packages

```
import java.awt.*;  
java.awt.event.*;  
java.applet.*;
```

2. Class should extend the Applet and implement the interfaces  
(MouseListener, MouseMotionListener)

3. Write applet code

4. init() method: Register the Listener interface

5. Write Event methods defintion

Example

Syntax:  
addMouseListener()  
addMouseMotionListener()

# OBJECT ORIENTED PROGRAMMING

## Lecture #41: AWT Hierarchy

# OBJECT ORIENTED PROGRAMMING

Lecture #42: Layout managers

## AWT VERSUS SWING

AWT	Swing
AWT components are heavyweight components	Swing components are lightweight components
AWT doesn't support pluggable look and feel	Swing supports pluggable look and feel
AWT programs are not portable	Swing programs are portable
AWT is old framework for creating GUIs	Swing is new framework for creating GUIs
AWT components require java.awt package	Swing components require javax.swing package
AWT supports limited number of GUI controls	Swing provides advanced GUI controls like Jtable, JTabbedPane etc
More code is needed to implement AWT controls functionality	Less code is needed to implement swing controls functionality
AWT doesn't follow MVC	Swing follows MVC

<b>EVENTS</b>	<b>SOURCE</b>	<b>LISTENERS</b>
Action Event	Button, List,MenuItem,Text field	ActionListener
Component Event	Component	Component Listener
Focus Event	Component	FocusListener
Item Event	Checkbox,CheckboxMen uItem, Choice, List	ItemListener
Key Event	when input is received from keyboard	KeyListener
Text Event	Text Component	TextListener
Window Event	Window	WindowListener
Mouse Event	Mouse related event	MouseListener

Java Generic Methods or generic classes enable Programmers to specify, with a single method declarations, a set of related methods or, with a single class declaration, a set of related types.

## GENERIC METHODS

We can write a single method declaration that can be called with arguments of different types.

Rules:

- All generic method declarations have a type parameter section delimited by angle brackets(< and >) that precedes the methods return type
- Each type parameter section contains one or more type parameters separated by commas
- The type parameter can be used to declare the return type
- Type parameters can represent only reference types not primitive types(like int, double and char)

## **GENERIC CLASSES**

Java Generic Methods or generic classes enable Programmers to specify, with a single method declarations, a set of related methods or, with a single class declaration, a set of related types.

Rules:

- Generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with Generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas

# OBJECT ORIENTED PROGRAMMING

Lecture #54: Generic and Inheritance

## **GENERIC AND INHERITANCE**

A class acquiring the properties of another class-Inheritance.

Java Generic Methods or generic classes enable Programmers to specify, with a single method declarations, a set of related methods or, with a single class declaration, a set of related types.

3 different ways of Generic and Inheritance

1. Only Super class is Generic
2. Only Subclass is Generic
3. Super and Sub class both are Generic

## ONLY SUPER CLASS IS GENREIC

```
class SuperA<T>
{
    T a;
    SuperA(T n)
    {
        a=n;
    }
    T getA()
    {
        return a;
    }
}
```

```
class SubA extends SuperA<T>
{
    SubA(Integer n1)
    {
        super(n1);
    }
}
```

```
class Ex
{
    public static void main(String args[])
    {
        SubA<Integer> s=new SubA<Integer>(10);
        System.out.println(s.getA());
    }
}
```

Output:  
10

## ONLY SUBCLASS IS GENERIC

```
class SuperA  
{  
    int a;  
    SuperA(int n)  
    {  
        a=n;  
    }  
    int getA()  
    {  
        return a;  
    }  
}
```

```
class SubA<T> extends SuperA{  
    T t;  
    SubA(Integer n1, T x)  
    {  
        super(n1);  
        t=x;  
    }  
    T gett()  
    {  
        return t;  
    }}  
}
```

```
class Ex  
{  
    public static void main(String args[])  
    {  
        SubA<String> s=new SubA<String>(10,"Jyothi");  
        System.out.println(s.getA());  
        System.out.println(s.gett());  
    }  
}
```

Output:

```
10  
Jyothi
```

## SUPER AND SUB CLASS BOTH ARE GENERIC

```
class SuperA<T>
{
    T a;
    SuperA(T n)
    {
        a=n;
    }
    T getA()
    {
        return a;
    }
}
```

```
class SubA<T> extends SuperA<T>
{
    SubA(T n1)
    {
        super(n1);
    }
}
```

```
class Ex
{
    public static void main(String args[])
    {
        SubA<Integer> s=new SubA<Integer>(10);
        System.out.println(s.getA());
        SubA<String> s1=new SubA<String>("Jyothi");
        System.out.println(s1.getA());
    }
}
```

Output:  
10  
Jyothi

## GENERIC AND INHERITANCE

3 different ways of Generic and Inheritance

1. Only Super class is Generic
2. Only Subclass is Generic
3. Super and Sub class both are Generic

subclass is free to  
use its own type  
parameters if  
needed

## SUBCLASS IS FREE TO USE ITS OWN TYPE PARAMETERS IF NEEDED

```
class SuperA<T>
{
    T a;
    SuperA(T n)
    {
        a=n;
    }
    T getA()
    {
        return a;
    }
}
```

```
class SubA<T> extends SuperA<T>
{
    SubA(T n1)
    {
        super(n1);
    }
}
```

```
class Ex
{
    public static void main(String args[])
    {
        SubA<Integer> s=new SubA<Integer>(10);
        System.out.println(s.getA());
        SubA<String> s1=new SubA<String>("Jyothi");
        System.out.println(s1.getA());
    }
}
```

Output:  
10  
Jyothi

## SUBCLASS IS FREE TO USE IYS OWN TYPE PARAMETERS IF NEEDED

```
class SuperA<T>
{
    T a;
    SuperA(T n)
    {
        a=n;
    }
    T getA()
    {
        return a;
    }
}
```

```
class SubA<T,V> extends SuperA<T>
{
    V t;
    SubA(T n1, T x)
    {
        super(n1);
        t=x;
    }
    V gett()
    {
        return t;
    }
}
```

```
class Ex
{
    public static void main(String args[])
    {
        SubA<Integer, String> s=new SubA<Integer, String>(10, "Jyothi");
        System.out.println(s.getA());
        System.out.println(s.gett());
    }
}
```

Output:  
10  
Jyothi

## OVERRIDING METHODS OF GENRIC CLASS

```
class SuperA<T>
{
    T a;
    SuperA(T n)
    {
        a=n;
    }
    T getA()
    {
        return a;
    }
}
```

```
class SubA<T> extends SuperA<T>
{
    SubA(T n1)
    {
        super(n1);
    }
    T getA()
    {
        return a;
    }
}
```

```
class Ex
{
    public static void main(String args[])
    {
        SuperA<Integer> s1=new SuperA<Integer> (10);
        SubB<Integer> s2=new SubB<Integer> (20);
        SubB<String> s3=new SubB<String> ("Jyothi");
        System.out.println(s1.getA());
        System.out.println(s2.getA());
        System.out.println(s3.getA());
    }
}
```

Output:  
10  
20  
Jyothi