

# OBJECT ORIENTED PROGRAMMING WITH JAVA (20CP204T)

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Outline

- Course Introduction- Teaching Scheme
- Syllabus
- Text/Reference Books
- Prerequisite
- OBE (Outcome Based Education), PEOs, POs, COs
- Traditional Education v/s OBE
- Pedagogy
- Office Hour
- Paradigm-1: Procedural Approach and example
- Paradigm-2: Object Oriented Approach and example
- Applications of Java

# Course Introduction- Teaching Scheme

20CP204T					Object Oriented Programming with Java					
Teaching Scheme					Examination Scheme					
L	T	P	C	Hrs/Week	Theory			Practical		Total Marks
					MS	ES	IA	LW	LE/Viva	
2	0	0	2	2	25	50	25			100

## Course Objectives:

- To build an understanding of basic concepts of object-oriented programming techniques
- To develop programming skills in Java programming language
- To implement object-oriented techniques using Java language features.
- To develop software using object-oriented programming paradigms

# Syllabus

<b>UNIT 1 BASICS OF JAVA</b> Features of Object Oriented Programming and Java, Basics of Java programming, Data types, Variables, Operators, Control structures including selection, Looping, Java methods, Overloading, Math class, Arrays in Java.	<b>7 Hrs.</b>
<b>UNIT 2 INHERITANCE</b> Basics of objects and classes in java, Constructors, Visibility modifiers, Inbuilt classes in Java, this reference; Inheritance in java, Overriding, Object class, Polymorphism, Dynamic binding, Abstract class, Interface in java, Package in java.	<b>7 Hrs.</b>
<b>UNIT 3 I/O PROGRAMMING, EXCEPTION AND MULTITHREADING</b> Introduction to Java IO streams, Character and Binary streams, reading data from and writing data to files, Difference between error and exception, Exception handling in Java, Multithreading in Java, Thread life cycle and methods, Runnable interface, Thread synchronization	<b>6 Hrs.</b>
<b>UNIT 4 EVENT HANDLING AND GUI PROGRAMMING</b> Event handling in Java, GUI Components and Layouts, Applet and its life cycle.	<b>6 Hrs.</b>
<b>Max. 26 Hrs.</b>	

# Text/Reference Books

- Brett D. McLaughlin, Head First Object-Oriented Analysis and Design, O' Reilly, 2006
- Matt Weisfeld, The Object-Oriented Thought Process, Addison-Wesley Professional, 2019
- Herbert Schildt, The Complete Reference, Java 2, McGraw Hill, 2020
- Balagurusamy, Programming with Java – A Primer, McGraw Hill, 2019

# Prerequisite

- No prerequisite subject
- Fundamental knowledge of programming

# OBE (Outcome Based Education)

- An education in which an emphasis is placed on
  - ▣ Clarity of Focus: a clearly articulated idea of what students are expected to know and be able to do
  - ▣ Designing down: the curriculum design must start with a clear definition of the intended outcomes about skills and knowledge students need to have, when they leave the school system
  - ▣ High expectations: high and challenging standards of performance in order to encourage students to engage deeply in what they are learning
  - ▣ Expanded opportunities: provide expanded opportunities for all students.
- Students are assisted when and where they have challenges.

# OBE (Outcome Based Education)...

OBE shifts from measuring input and process to include measuring the output (outcome).



Image Source: "Importance of Outcome Based Education (OBE) to Advance Educational Quality and enhance Global Mobility." (2018).

- 4 PEOs, 12 POs and 6 COs

# Program Education Objectives (PEOs)

1. To prepare graduates who will be successful professionals in industry, government, academia, research, entrepreneurial pursuit and consulting firms
2. To prepare graduates who will make technical contribution to the design, development and production of computing systems
3. To prepare graduates who will get engage in lifelong learning with leadership qualities, professional ethics and soft skills to fulfill their goals
4. To prepare graduates who will adapt state of the art development in the field of computer engineering

# Program Outcomes (POs)

1. Engineering knowledge
2. Problem analysis
3. Design / development of solutions
4. Conduct investigations of complex problems
5. Modern tool usage
6. The engineer and society
7. Environment and sustainability
8. Ethics
9. Individual and team work
10. Communication
11. Project management and finance
12. Life-long learning

# Program Specific Outcomes (PSOs)

- The graduates of CSE department will be able to:
  1. Develop computer engineering solutions for specific needs in different domains applying the knowledge in the areas of programming, algorithms, hardware-interface, system software, computer graphics, web design, networking and advanced computing.
  2. Analyze and test computer software designed for diverse needs.
  3. Pursue higher education, entrepreneurial ventures and research.

# Course Outcomes (COs)

- On completion of the course, student will be able to
  - CO1- Describe the basic features of Object-oriented programming and map them with the Java.
  - CO2- Distinguish Objects and Classes using Java.
  - CO3- Demonstrate Inheritance and Runtime Polymorphism
  - CO4- Apply I/O handling, exception handling for interactive problem.
  - CO5- Use the concepts of Event Handling in GUI Programming.
  - CO6- Construct object-oriented solutions for small systems involving multiple objects.

# **Traditional Education v/s OBE**

<b>Traditional Education</b>	<b>Outcome Based Education</b>
The approach is exam-driven.	Students are assessed on an ongoing basis.
Learning is textbook/worksheet-bound, subjective and teacher-centered	Learning is objective learner-centered, the teacher facilitates and constantly applies group work and team work for new tasks
Learners are passive	Learners are active
Emphasis on what lecturer hopes to achieve	Emphasis on specific outcomes
Rote learning	Critical thinking, reasoning and action
Textbook/worksheets focused and teacher centered	Learner centered and educator use group/team work

# Pedagogy

- Use of
  - Board
  - Powerpoint Presentations
  - Puzzles
  - Program Execution

# Office Hour

- Tuesday, 11:00 AM to 12:00 PM
- Take prior permission through mail and then come to meet in E-215 faculty cabin

# Paradigm-1: Procedural Approach

- 1. Procedure → Data
- 2. Procedure → Data
- 3. Procedure → Data
- 4. Procedure → Data

.

.

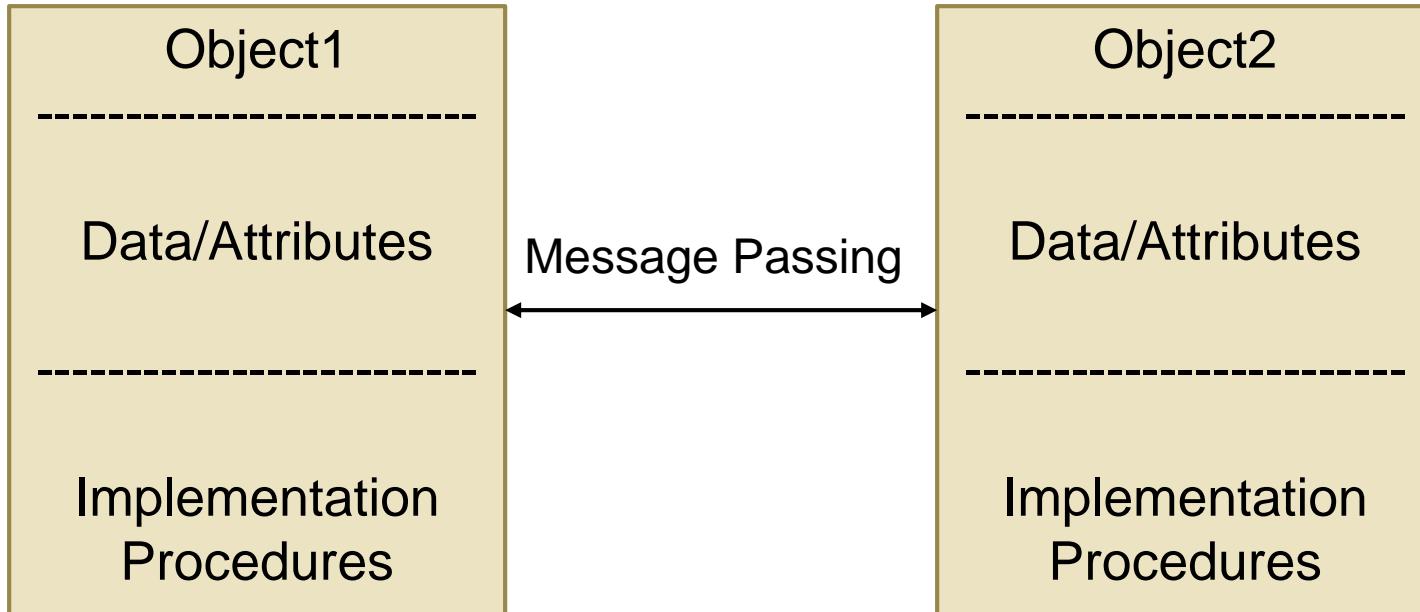
.

Assembly line –sequential tasks

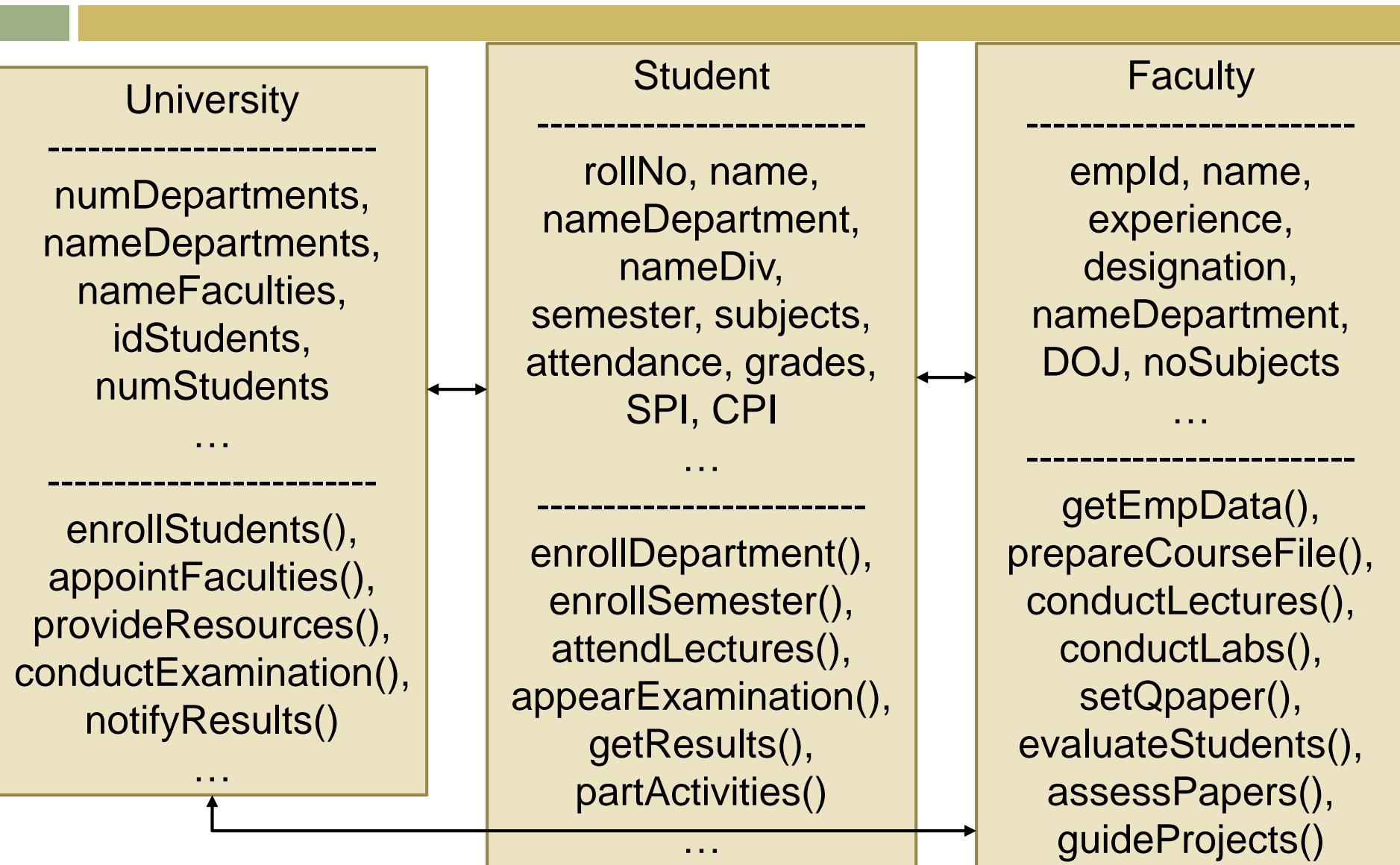
# Procedural Approach-Example

- Teaching process
  - Students take admission in department of university
  - Faculties take lectures and labs and evaluate students
  - Students attend lectures and labs
  - Students participate in different events
  - Students attend workshops
  - At the end of semester, university conducts exam
  - Faculties set questions papers
  - Students appear for the examination under Faculties' supervision
  - Faculties evaluate answersheets
  - Faculties conduct practical evaluation
  - University generate the results (SPI and CPI of students)
  - University considers performance appraisal of faculties

# Paradigm-2: Object Oriented Approach

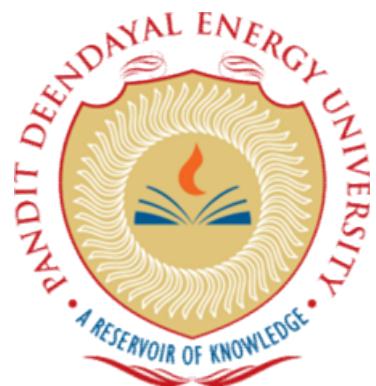


# Object Oriented Approach- Example



# Applications of Java

- Mobile Applications (Twitter, Minecraft)
- Desktop GUI Applications
- Web-based Applications
- Enterprise Applications
- Scientific Applications
- Gaming Applications
- Big Data Technologies
- Business Applications
- Distributed Applications
- Cloud-based Applications



# OBJECT ORIENTED PROGRAMMING WITH JAVA (20CP204T)

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,  
School of Technology,  
Pandit Deendayal Energy University

# Outline

- Applications of Java
- Major Impacts of Java on Internet
- OOP Principles
- What is Compiler?
- How Java works?

# Applications of Java

- Mobile Applications (Twitter, Minecraft)
- Desktop GUI Applications.
- Web-based Applications.
- Enterprise Applications.
- Scientific Applications.
- Gaming Applications.
- Big Data Technologies.
- Business Applications.
- Distributed Applications
- Cloud-based Applications

# Major Impacts of Java on Internet

- Java Applets
  - ▣ Java program to be transmitted over the Internet
  - ▣ Automatically executed inside a Java-compatible web browser
- Security
  - ▣ Applications are executed in Java execution environment
- Portability
  - ▣ Heterogeneous types of computers and operating systems

# OOP Principles

## 1. Encapsulation

Student

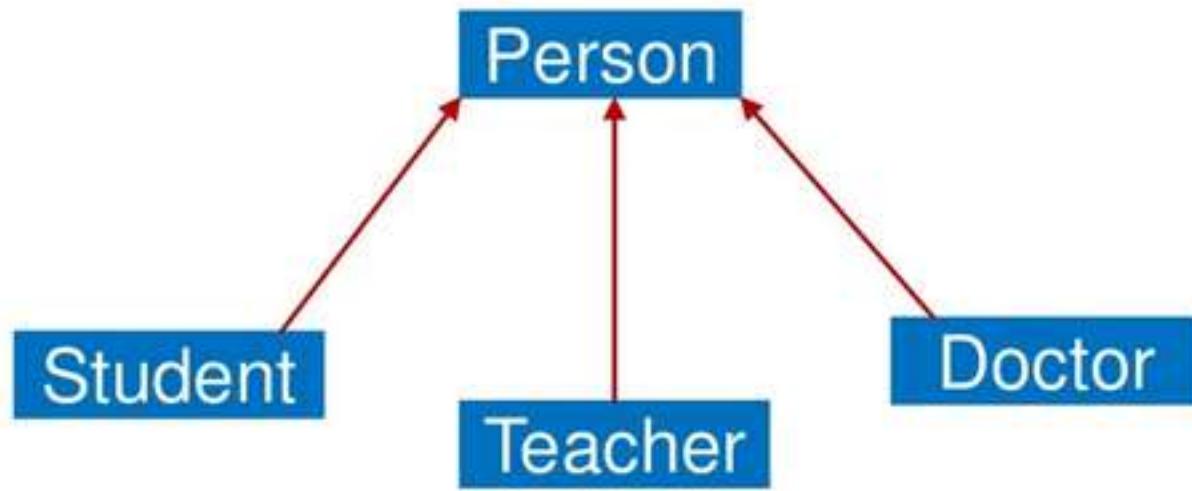
-----  
rollNo, name,  
nameDepartment,  
nameDiv,  
semester, subjects,  
attendance, grades,  
SPI, CPI  
...  
-----

enrollDepartment(),  
enrollSemester(),  
attendLectures(),  
appearExamination(),  
getResults()  
partActivities()  
...



# OOP Principles...

## 2. Inheritance



# OOP Principles...

## 3. Polymorphism (same name, many forms)

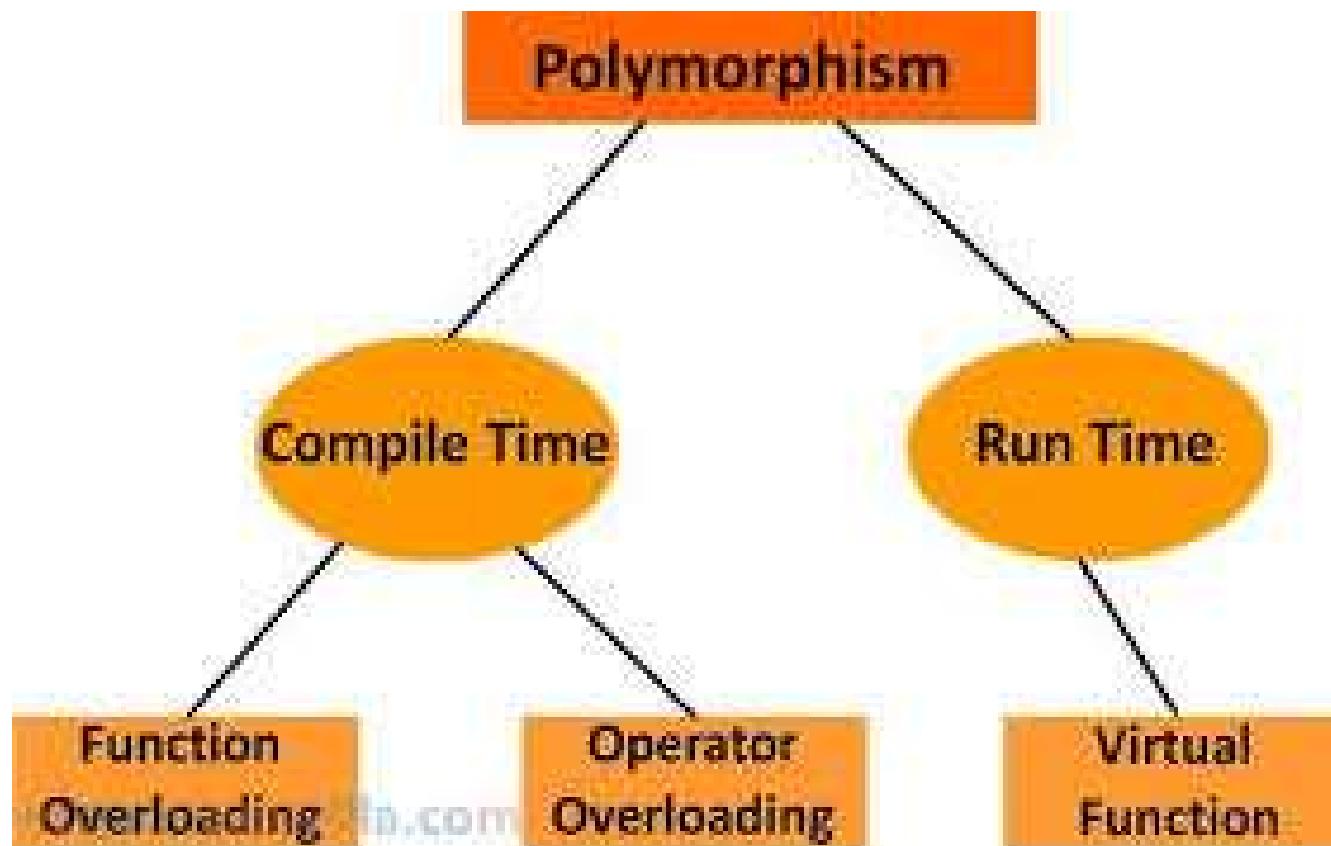


Image source: <http://spiroprojects.com/blog/cat-view-more.php?blogname=What-is-Polymorphism-in-Java?id=330>

# What is Compiler?

- Compiler
  - Converts the high-level language (human language) into lower level code
    - a sequence of executable machine instructions - directly executed by CPU
    - an assembly code that is processed by assembler
    - an intermediate representation that is interpreted

# How Java works?

- Java Compiler
  - Generates intermediate representation- **the Bytecode (.class file)**
    - platform-independent (executed on all operating systems)
    - adds to an important feature in the JAVA language termed as **portability**
    - needs an interpreter to execute on a machine - **JVM**
- JVM (Java Virtual Machine)
  - provides a runtime environment
  - **loads, verifies and executes** Java Bytecode
  - known as the interpreter or the core of Java programming language because it executes Java programming
  - does not exists physically, resides in memory as a software program
  - it actually calls the **main** method present in a java code

# How Java works?...

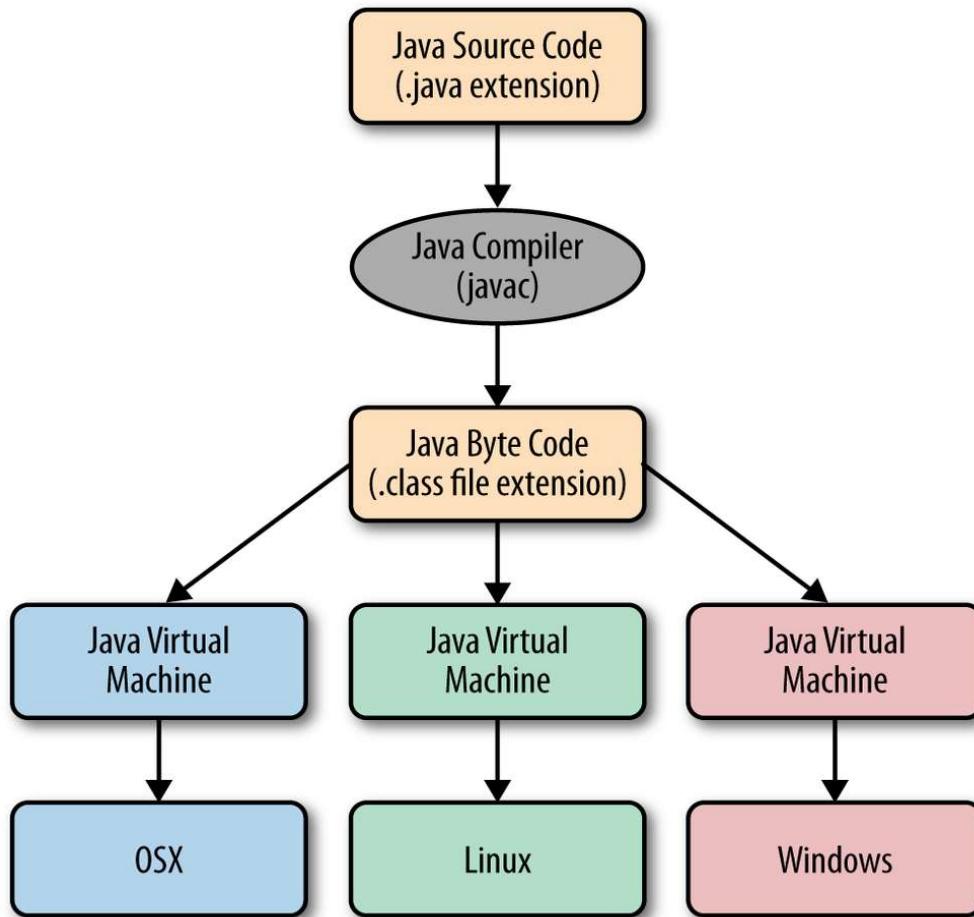


Image Source: "How does the Java compilation process work? What is JAS? : r/java"

- **Java is platform-independent but JVM is platform dependent**

# How Java works?...

- JRE (Java Runtime Environment)
  - ▣ a set of software tools which are used for **developing Java applications**
  - ▣ provides the **runtime environment (installation package)** that provides an environment to **only run (not develop)** the java program (or application)
  - ▣ the implementation of JVM, physically exists
  - ▣ contains a set of libraries + other files that JVM uses at runtime
- JDK (Java Development Kit)
  - ▣ a software development environment which is used to develop Java applications and **applets**
  - ▣ physically exists
  - ▣ a **kit (or package) that includes** JRE, an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed

# How Java works?...

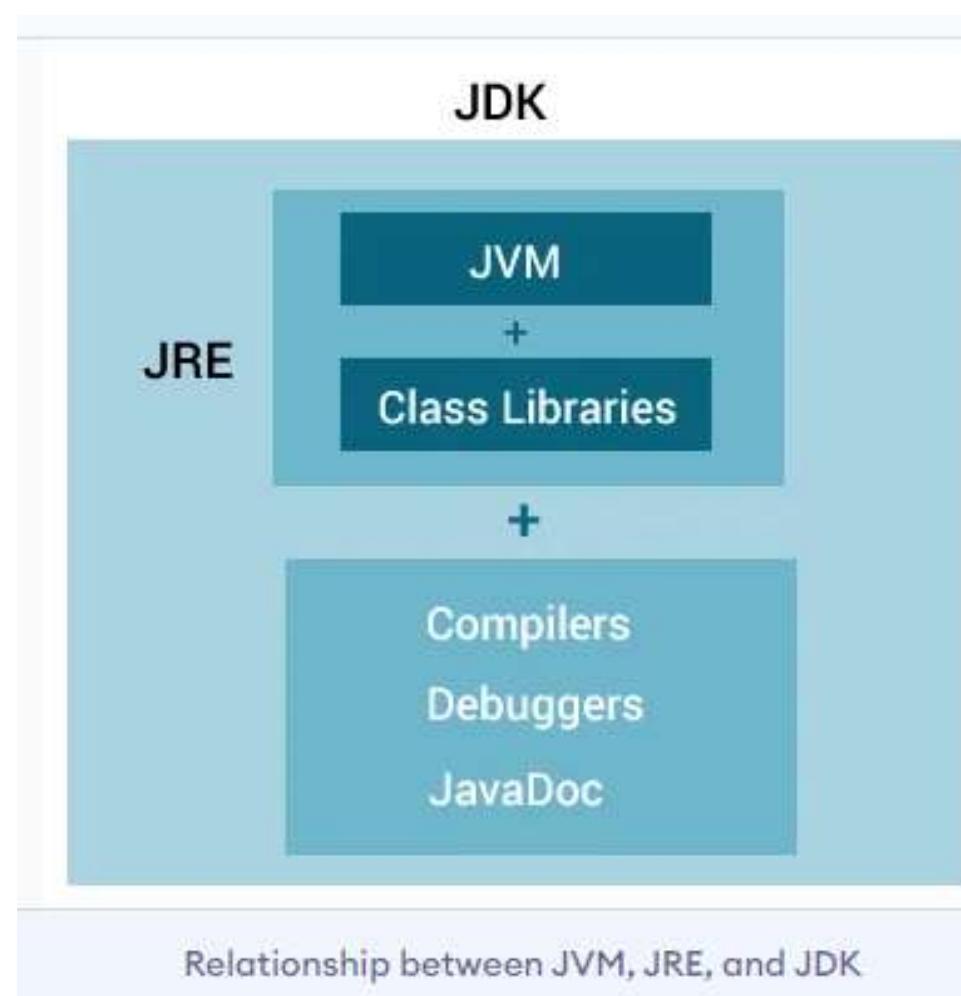
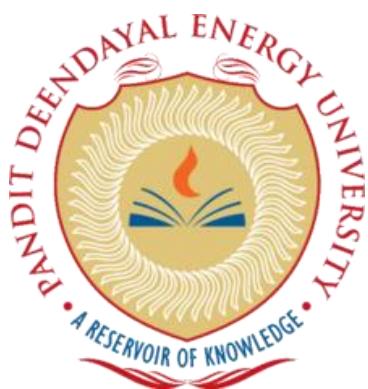


Image source: <https://www.programiz.com/java-programming/jvm-jre-jdk>



# OBJECT ORIENTED PROGRAMMING WITH JAVA (20CP204T)

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Outline

- Data Types
- Type Conversions
- Type Promotion in Expressions
- Scope of Variables
- Main Method
- Print Method
- Scan Variables
- Size and Type of Variables

# Data Types

- Java is strongly typed
  - ▣ Every variable and expression has a type
  - ▣ Every type is strictly defined
  - ▣ Type compatibility check in each assignment (direct or through parameter passing in method calls)
- Primitive data types
  - ▣ Eight primitive data types (**byte**, **short**, **int**, **long**, **char**, **float**, **double** and **boolean**)
- All data types have **a strictly defined range**, regardless of particular platform

# Data Types...

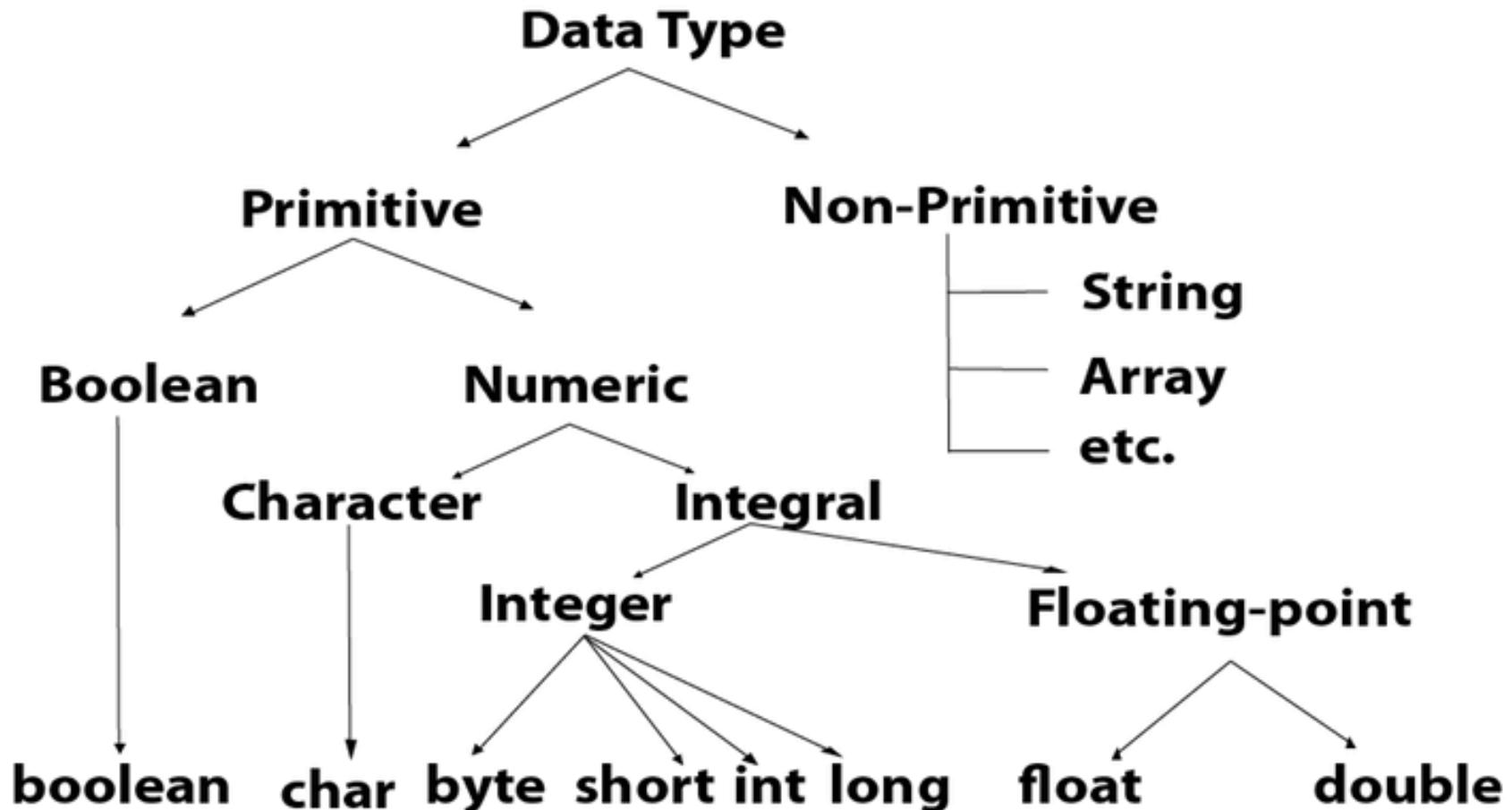


Image source: Java Data Types - Javatpoint

# Data Types- Integers

- byte, short, int, long
- all are **signed**, positive and negative
- Java does not support unsigned integers

Type	Contains	Data Types	Size	Range
byte	Signed integer	Byte	8 bit or 1 byte	-128 to 127
short	Signed integer	Short	16 bit or 2 bytes	-32,768 to 32767
int	Signed integer	Int	32 bit or 4 bytes	-2147,483,648 to 2147,483,647
long	Signed integer	Long	64 bit or 8 bytes	-263 to 263-1 or -9223,372,036,854,755,808 to 9223,372,036,854,755,807

# Data Types...

- Floating Point types
  - ▣ `float`, `double`
  - ▣ Stores numbers with fractional precision
- Characters – `char`, supports ASCII conversion, no negative char
- Booleans – Boolean – takes values `true` or `false`

Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2 <sup>31</sup> to 2 <sup>31</sup> -1
long	64	-2 <sup>63</sup> to 2 <sup>63</sup> -1
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

# Data Types and Variables- Examples

(1) short s, t;

s=55;

O/P: ?

System.out.println("10 > 5 is "+ (10 > 5));

(2) int lightspeed,

long seconds;

lightspeed = 20500;

seconds = 1000\*24\*60\*60;

(3) char ch1, ch2;

ch1=65;

ch2='B';

System.out.print("ch1 and ch2: " + ch1 + " " +ch2);

(4) boolean b=false;

if (b) System.out.println("in true block");

# Find Output

```
public class Main {  
    public static void main(String[] args) {  
        int num1 = 10;  
        int num2 = 20;  
        long num3 = 0;  
        num3 = num1 + num2 * 10 + Character.SIZE;  
        System.out.println(num3);  
    }  
}
```

# Type Conversions

- Automatic (implicit type conversion)
  - ▣ The two types are compatible
  - ▣ The destination type is larger than the source type
- Casting incompatible types (explicit type conversion)
  - ▣ (target-type) value
  - ▣ `int a = 257; byte b; double d = 323.142;`  
`b = (byte) a;`  
`a = (int) d;`  
`b = (byte) d;`

# Type Promotion in Expressions

- In expression, the range of intermediate values will sometimes exceed the range of either operand

Ex: byte a=40, b=50, c=100;

int d = a\*b + c;

- Thus, Java automatically promotes each byte, short or char operand to int during expression evaluation

  - So  $a * b = 40 * 50 = 2000$  is data type int, not byte

  - Also data type of result of  $a * b + c$  is int

- Compile-time error !

Ex: byte b=50;

byte a = b\*2; //Error, can't assign int to byte

- Solution?

byte a = (byte) b\*2;

# Scope of Variables

- Variables can be declared in any block – before its use

```
cnt =0;  
int cnt;
```

Error !

- Block begins with an opening curly bracket
- Scope of variable – within the block in which it is declared

# Find Output

```
public class Main {  
    public static void main(String[] args) {  
        int i;  
        for(i=1; i<= 3; i++)  
        {  
            int x = -1;  
            System.out.println("x is "+x);  
            x = 10;  
            System.out.println("now x is "+x);  
        }  
    }  
}
```

# Find Output...

```
public class Institutes {  
    public static void main(String[] args) {  
        String name = "PDEU";  
        {  
            System.out.println(name + " Institute");  
            name = "pdeu";  
        }  
        System.out.println(name + " Institute");  
    }  
}
```

# Find Output...

```
public class VarScope {  
    public static void main(String[] args) {  
        int x = 10;  
        {  
            int y = 20;  
            System.out.print(x + ", " + y);  
        }  
        {  
            y = 10;  
            x = 15;  
            System.out.print(" - " + x + ", " + y);  
        }  
        System.out.print(" - " + x + ", " + y);  
    }  
}
```

# Find Output...

```
public class ScopeOfVariables {  
    public static void main(String[] args) {  
        int x = 10;  
        int y = 20;  
        {  
            System.out.print(x + ", " + y);  
        }  
        {  
            x = 15;  
            System.out.print(" - " + x + ", " + y);  
        }  
        System.out.print(" - " + x + ", " + y);  
    }  
}
```

# Main Method

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

- **public:** This is the **access modifier** of the main method. It has to be public so that java runtime can execute this method.
- **static:** When java runtime starts, there is **no object of the class** present. That's why the main method has to be static so that JVM can load the class into memory and call the main method (there is only one **static variable** per class).
- **void:** Java programming mandates that every method provide the return type. Java main method **doesn't return anything**, that's why it's return type is void.
- **main:** This is the name of java main method. It's fixed and when we start a java program, it **looks for the main method**.

# Print Method

- used to display a text on the console
1. `public void print(String s)`
    - an overloaded method of the **PrintStream** class
    - Call: `System.out.print(parameter);`

Can't create object of PrintSteam class directly, so use instance of the PrintStream class that is **System.out**
  2. `public void println(String s)`
    - an overloaded method of the **PrintStream** class
  3. `public PrintStream printf(String format, Object... args)`
    - an overloaded method of the **PrintStream** class
    - print the formatted string to the console using the specified format string and arguments

# Scan Variables

- Scanner method:

```
import java.util.Scanner;  
class Program{  
    public static void main(String []args){  
        Scanner myObj = new Scanner(System.in);  
        String userName = myObj.nextLine();  
        int i = myObj.nextInt();  
    }  
}
```

# Size and Type of Variables

- Get size of variable- size operator
  - ▣ EX: size of integer : Integer.SIZE/8  
size of character: Character.SIZE/8
  
- Get type of variable
  - ▣ Call getClass().getSimpleName() via the object
  - ▣ EX: String str;  

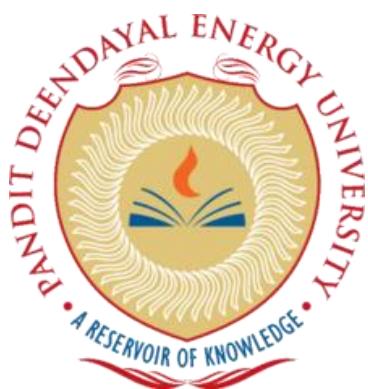
```
System.out.println(str.getClass().getSimpleName());
```

```
int i;
```

```
System.out.println(((Object)i).getClass().getSimpleName());
```



# LOOPS AND ARRAYS IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

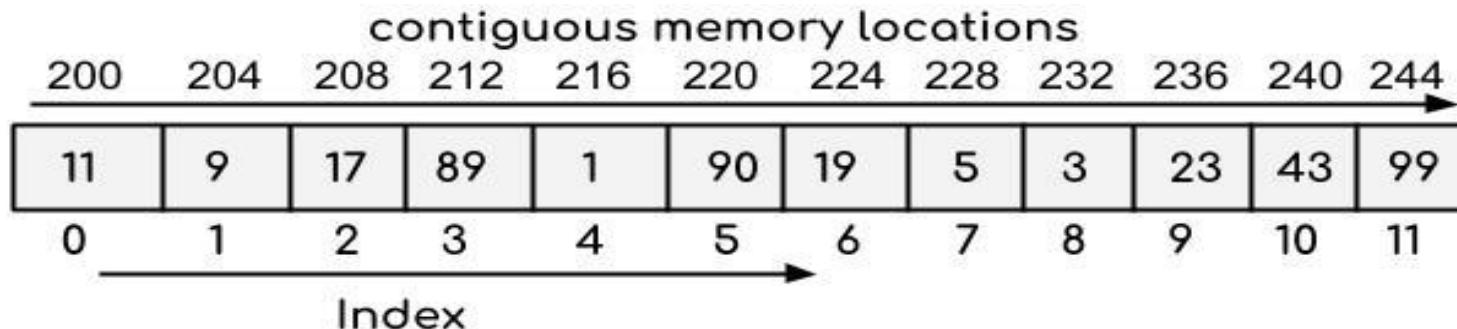
Pandit Deendayal Energy University

# Outline

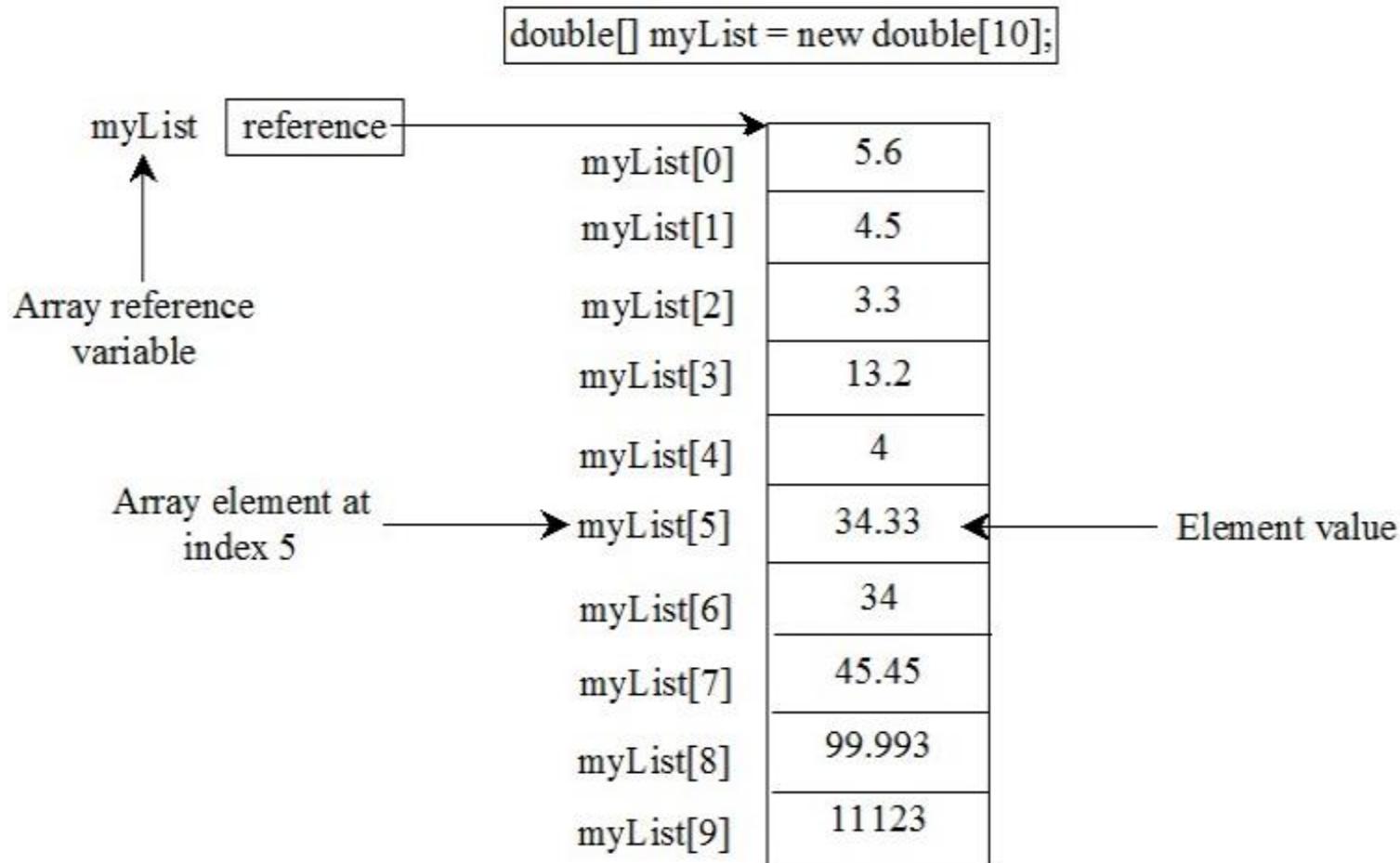
- What is an array?
- Example of Array
- Array Declaration, Instantiation, Initialization
- Length of an Array
- Default Values
- Loops
- Few words about Strings
- Two-dimensional Arrays
- Ragged Arrays

# What is an array?

- A group or collection of **like-typed variables** that are referred to by a **common name**
- Can have one or more dimensions
- Element is accessed by **index**
- **Contiguous memory allocation**



# Example of Array



# Array Declaration

- Declaration:
  - ▣ Datatype[] arrayRefVar; //Preferable
  - ▣ Datatype []arrayRefVar;
  - ▣ Datatype arrayRefVar[]; //Allowed, but not preferred
  - ▣ EX:  
int[] totalMarks;
- Instantiation:  
`arrayRefVar = new datatype [arraySize];`  
EX:  
`totalMarks=new int[5]; //declaration and instantiation`

# Array Instantiation and Initialization

- Only the declaration of the array is not sufficient
- To store values in the array, it is required to initialize it after declaration

## 1. Without assigning values:

```
int[] iArray = new int[5];
System.out.println(iArray[2]);    //0
for (i=0; i<5;i++)
    System.out.println(iArray[i]);
```

## 2. After the declaration of the array:

```
int[] numbers;
numbers = new int[]{22,33,44,55,66};
```

# Array Initialization

### 3. Initialize and assign values together:

```
int[] numbers = {22,33,44,55,66}; //Shorthand notation
```

### 4. Assign values using index:

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

```
iArray[0]=10;
```

```
iArray[1]=20;
```

...

**Caution:** In shorthand notation, declare, create and initialize array all in one statement, otherwise error !!

# Length of an Array

- Once an array is created, its size is fixed. It cannot be changed.
- To find size:  
`arrayRefVar.length`

EX:

`totalMarks.length` returns 5

# Default Values

- Default values of array elements are:
  - Numeric primitive data types: 0
  - Character data types: '\u0000'
  - Boolean data types: **false**

# Loops

- For-each Loop for Java Array
  - ▣ the Java array can be printed using **for-each loop** also
  - ▣ it prints the array elements one by one
  - ▣ it holds an array element in a variable

Syntax:

```
for(elementType ele:arrayRefVar){  
    //body of the loop  
}
```

# Loops...

Example of for each loop:

```
class TestArray1{  
    public static void main(String args[]){  
        int iArray[]={10, 20, 30, 40, 50};  
            //printing array using for-each loop  
        for(int ele : iArray)  
            System.out.println( ele );  
        for(int j=0;j<5;j++)  
            System.out.println(iArray[ j ]);  
    }  
}
```

# Few words about Strings

- String
  - Not primitive data type
  - Not an array of characters
  - It is an object
  - Characters of the string can't be accessed using index

```
String str = "Welcome";  
System.out.println(str[1]); //Error
```

# Two-dimensional Arrays

- // Declare array ref var
  - ▣ dataType[][] refVar;
- // Create array and assign its reference to variable
  - ▣ refVar = new dataType[10][10];
- // Combine declaration and creation in one statement
  - ▣ dataType[][] refVar = new dataType[10][10];
- // Alternative syntax
  - ▣ dataType refVar[][] = new dataType[10][10];

# Two-dimensional Arrays...

```
int[][] matrix = new int[10][10];
for(int i=0 ; i < matrix.length ; i++)
    for(int j=0 ; j < matrix[ i ].length ; i++)
        matrix[ i ][ j ] = matrix[ i ][ j ] + 100;
```

- Questions:

int[][] matrix = new int[3][5];

matrix.length ? 3

matrix[0].length ? 5

matrix[2].length ? 5

# Ragged Arrays

- Each row in a two-dimensional array is itself an array. So, the **rows can have different lengths**. Such an array is known as a ragged array.
- EX:

```
int[][] matrix = { {1, 2, 3, 4, 5},  
                  {2, 3, 4, 5},  
                  {3, 4, 5},  
                  {4, 5},  
                  {5}  
};
```

matrix.length is 5  
matrix[0].length is 5  
matrix[1].length is 4  
matrix[2].length is 3  
matrix[3].length is 2  
matrix[4].length is 1

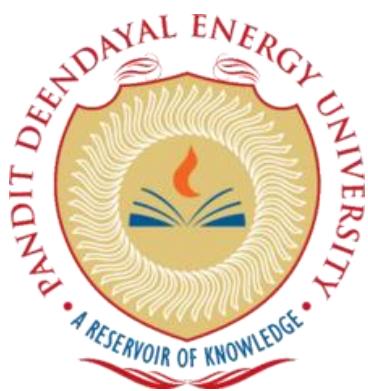
# Ragged Arrays...

- Another declaration

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

EX:

```
for(i=0; i<3; i++)  
    for(j=0; j<matrix.length; j++)  
        matrix[ i ][ j ] = i + j;
```



# METHOD OVERLOADING IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

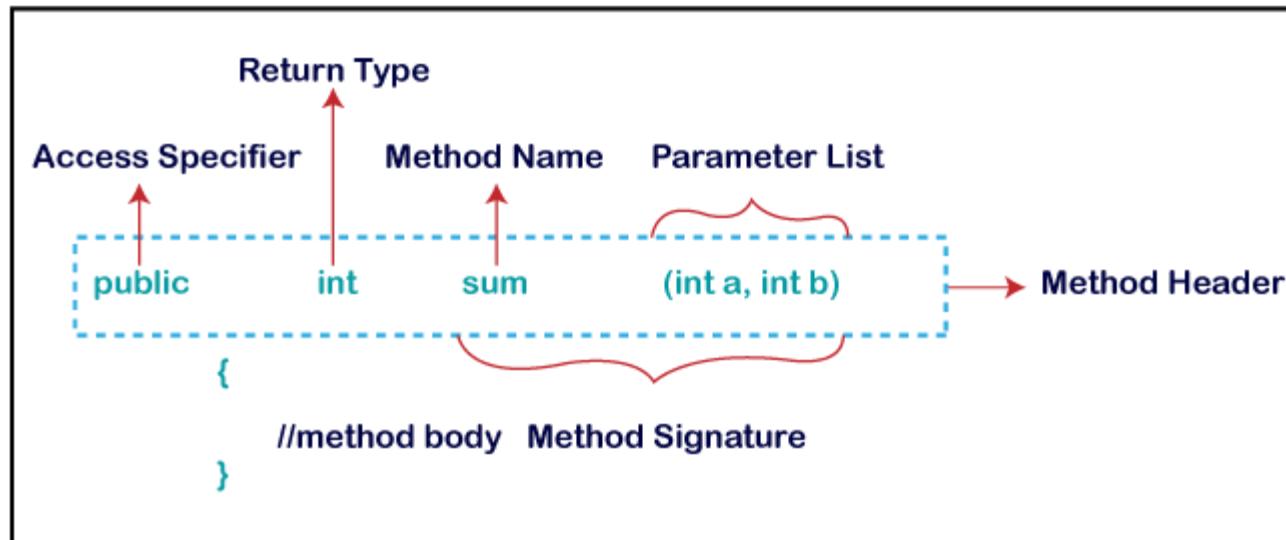
Pandit Deendayal Energy University

# Outline

- Java Methods
- Method Overloading
- Call by Reference
- Math Class

# Java Methods

- A method
  - a block of code which only runs when it is called
  - parameters- data passed into a method
  - return data
  - used to achieve the **reusability** of code
  - **must be declared within a class**



# Java Methods

- Types of methods
  - ▣ System defined methods (predefined, built-in)
    - Math.sqrt()
    - System.out.println()
  - ▣ User defined methods

# Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- [Method Overloading in Java - Javatpoint](#)

# Overload Main Method !!

```
Class overloadMain{  
    public static void main(String[] args){  
        System.out.println("in main with string arguments");  
        overloadMain objMain = new overloadMain();  
        objMain.main();  
    }  
    public static void main(){  
        System.out.println("in main without string arguments");  
    }  
}
```

# Call by Reference

- when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- [Passing and Returning Objects in Java - GeeksforGeeks](#)

# Example in C++

```
void swap (int &x, int &y){  
    int t;  
    t = x;  
    x = y;  
    y = t;  
}  
void main(){  
    int a=5, b=10;  
    printf("%d %d", a, b);  
    swap(a, b);  
    printf("%d %d", a, b);  
}
```

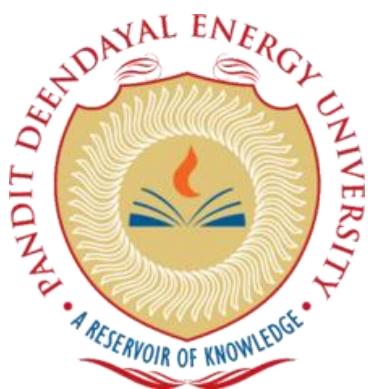
# Math Class

- provides several methods to work on math calculations
- `java.lang.Math` class contains various methods
- `min()`, `max()`, `avg()`, `sin()`, `cos()`, `tan()`, `round()`, `ceil()`, `floor()`, `abs()` etc.
- All methods
  - ▣ [Java Math class with Methods - Javatpoint](#)

# Math Class

EX:

```
public class JavaMathExample
{
    public static void main(String[] args)
    {
        double x = 28;
        double y = 4;
        // return the maximum of two numbers
        System.out.println("Maximum number is: " +Math.max(x, y));
    }
}
```



# OBJECT ORIENTED PROGRAMMING

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Programming Paradigms- Procedural

- Based on the concept of using **procedures (functions)**
  - Procedure is a sequence of commands to be executed
  - Any procedure can be called from any point within the general program, including other procedures or even itself
  - Data Variable Scope:
    - Global
    - Local
- Procedure & Program is divided into modules
- Every module has its **own data and function** which can be called by other modules.

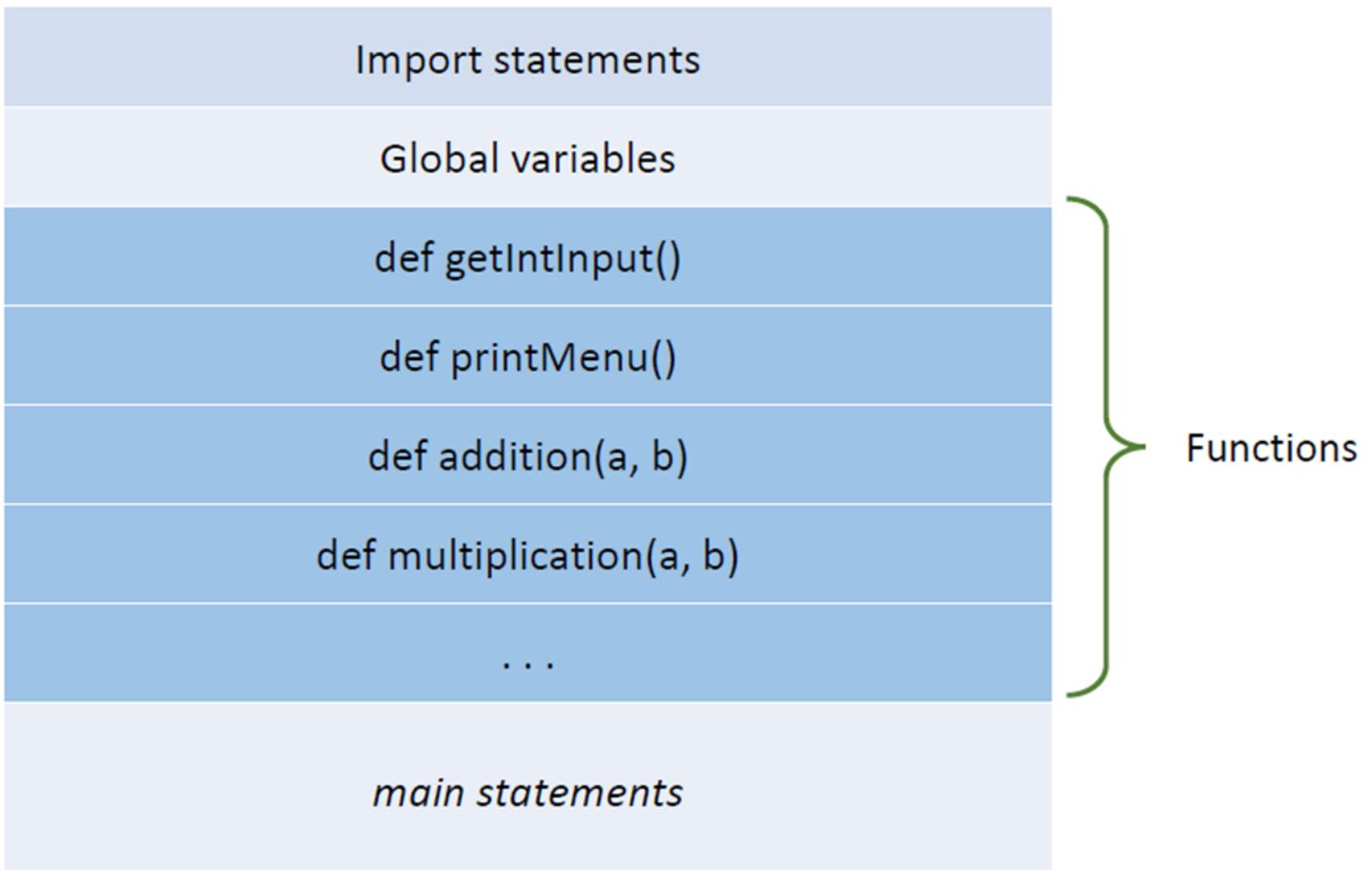
# Programming Paradigms- Procedural...

## □ Teaching-learning process

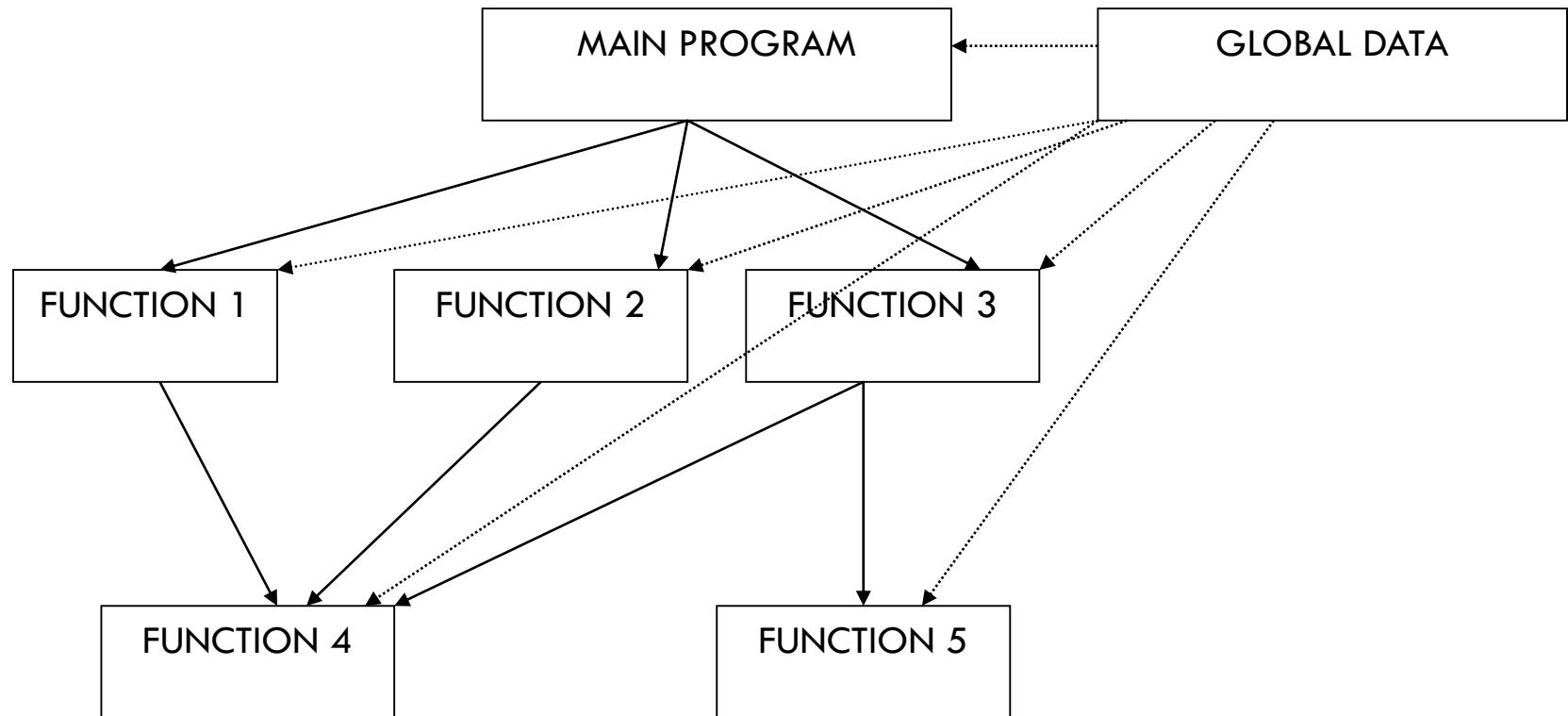
- Students take admission in department of university
- Faculties take lectures and labs and evaluate students
- Students attend lectures and labs
- Students participate in different events
- Students attend workshops
- At the end of semester, university conducts exam
- Faculties set questions papers
- Students appear for the examination under Faculties' supervision
- Faculties evaluate answersheets
- Faculties conduct practical evaluation
- University generate the results (SPI and CPI of students)
- University considers performance appraisal of faculties

# Programming Paradigms- Procedural...

## □ Arithmetic Calculator



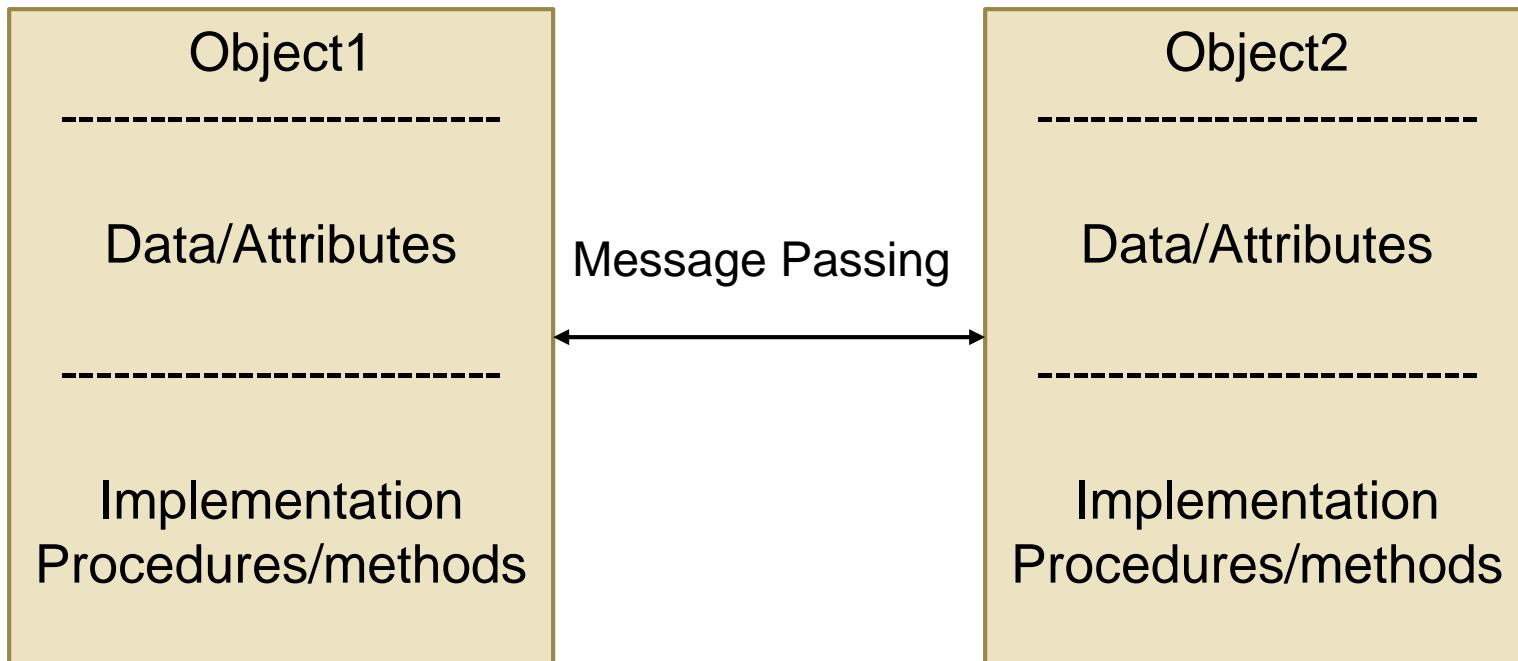
# Programming Paradigms- Procedural...



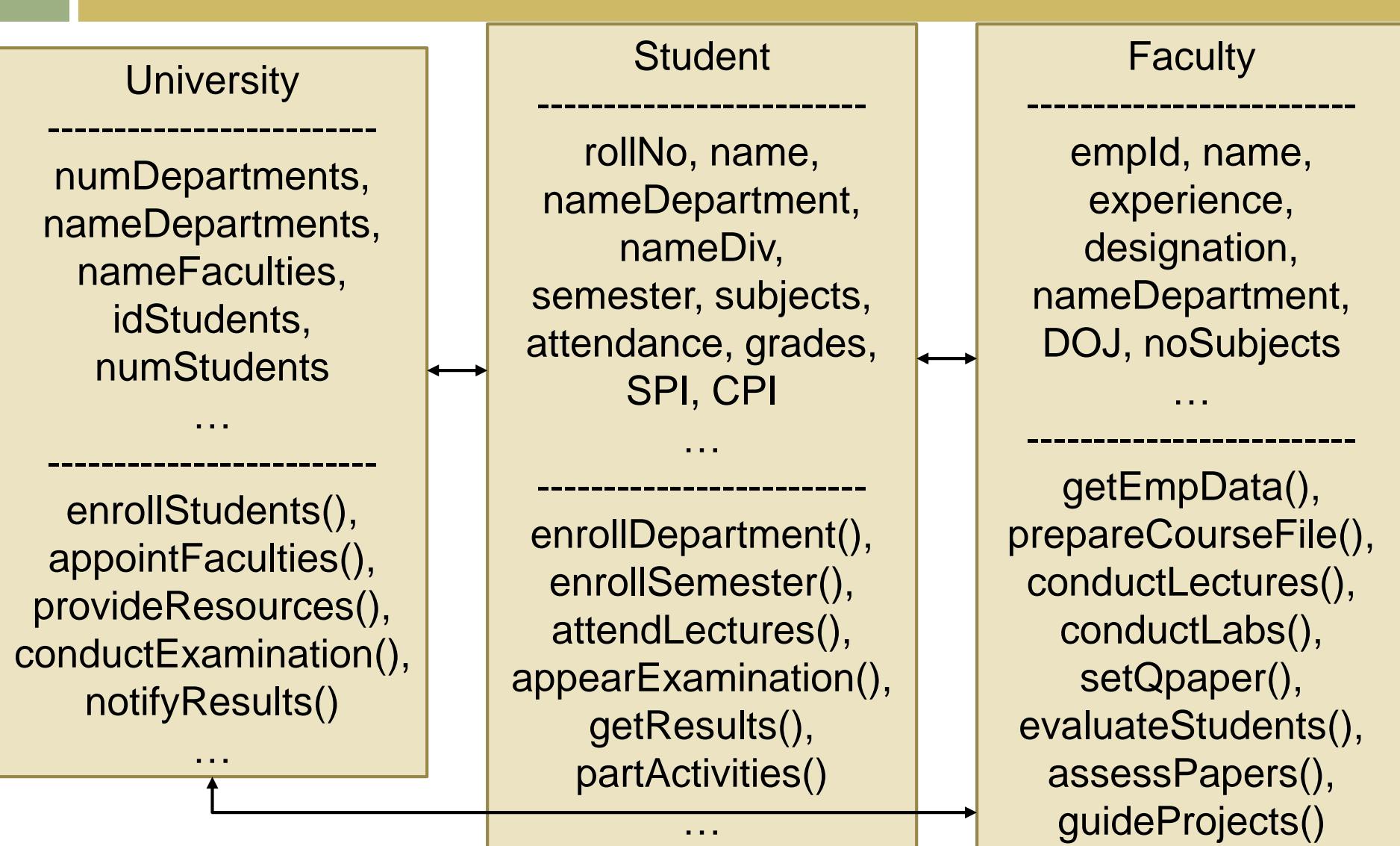
# Programming Paradigms – OOP

- Object Oriented Programming
  - ▣ It is based on the concept of using classes and its objects.
  - ▣ Inspired from the real-world.
- **Class:** It is a blueprint of the properties & behavior.
  - ▣ It is a data-type.
- **Object:** It is an instance of a particular class.
- Variable and function scope:
  - ▣ Public
  - ▣ Private
  - ▣ etc.

# Programming Paradigms – OOP...



# Object Oriented Programming- Example



# What is Object Oriented Programming?

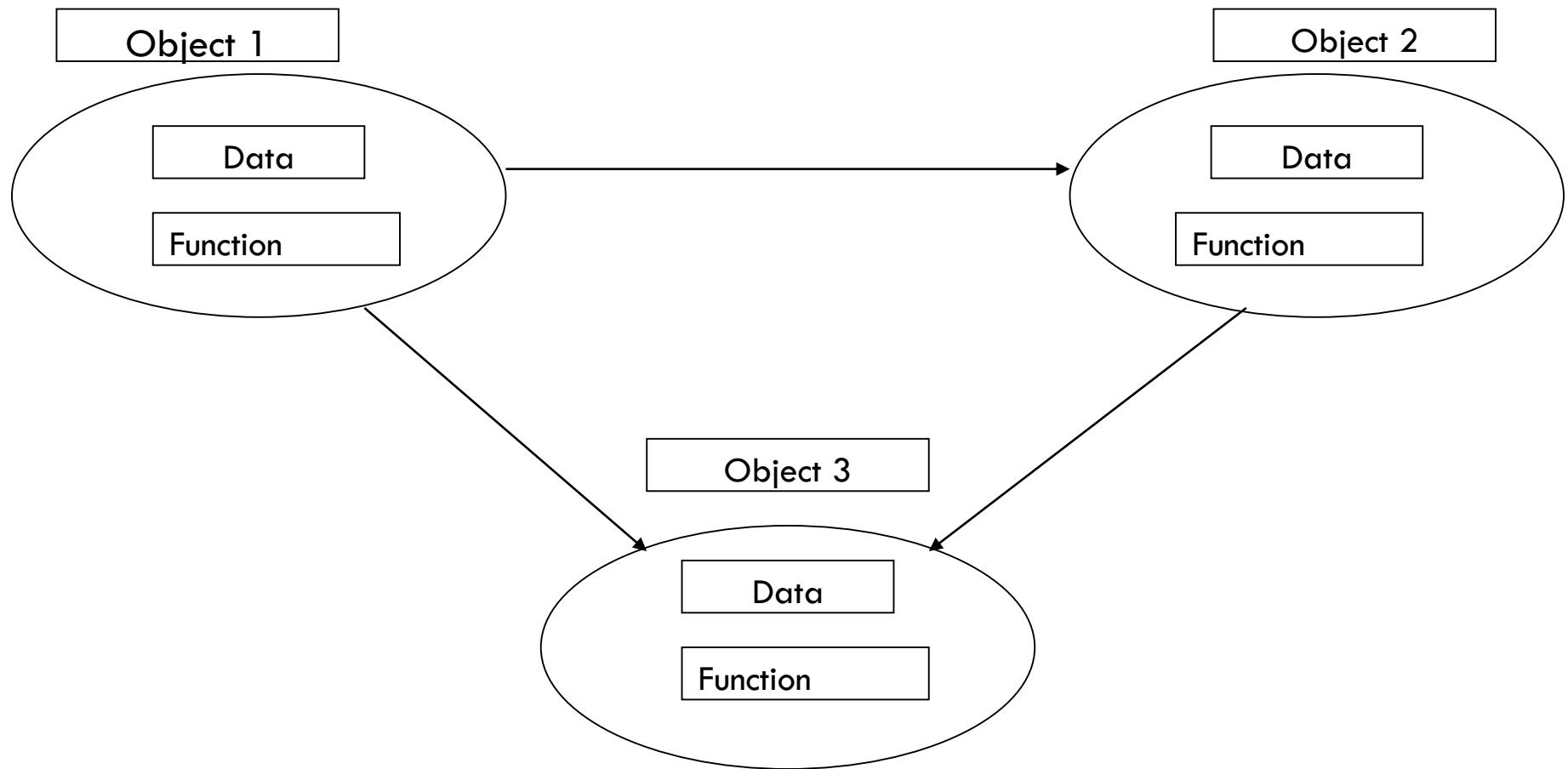
- Identifying **objects** and assigning responsibilities to these objects.
- Objects communicate to other objects by sending **messages**.
- Messages are received by the **methods** of an object
- An object is like a black box.
  - ▣ The internal details are hidden.



# Programming Paradigms – OOP

- Objects have both **data and methods**
- Objects of the same class have the **same data elements and methods**
- Objects send and receive messages to invoke actions
  
- Key idea in object-oriented:
- The real world can be accurately described as a collection of objects that interact.

# Programming Paradigms – OOP...

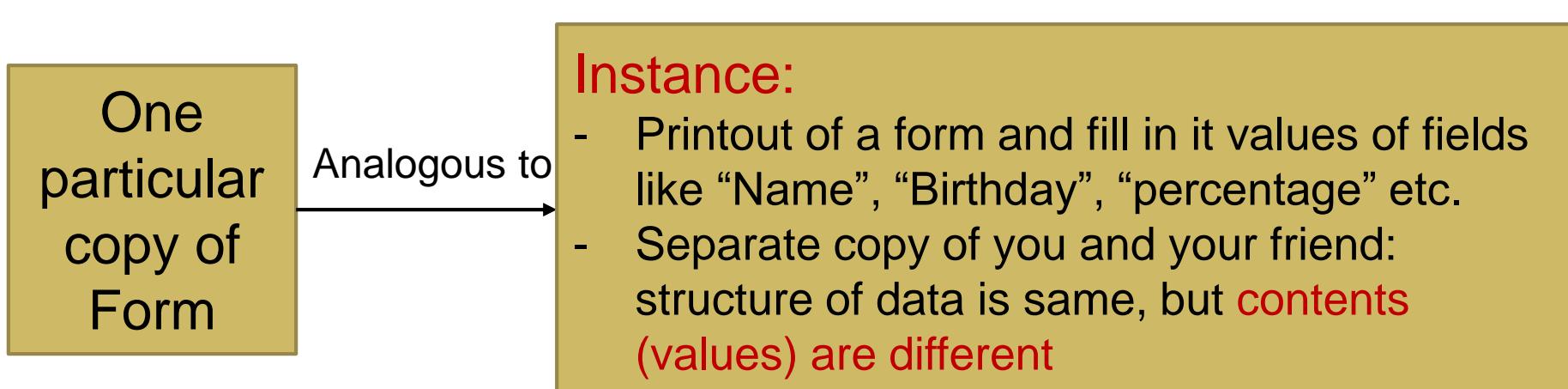
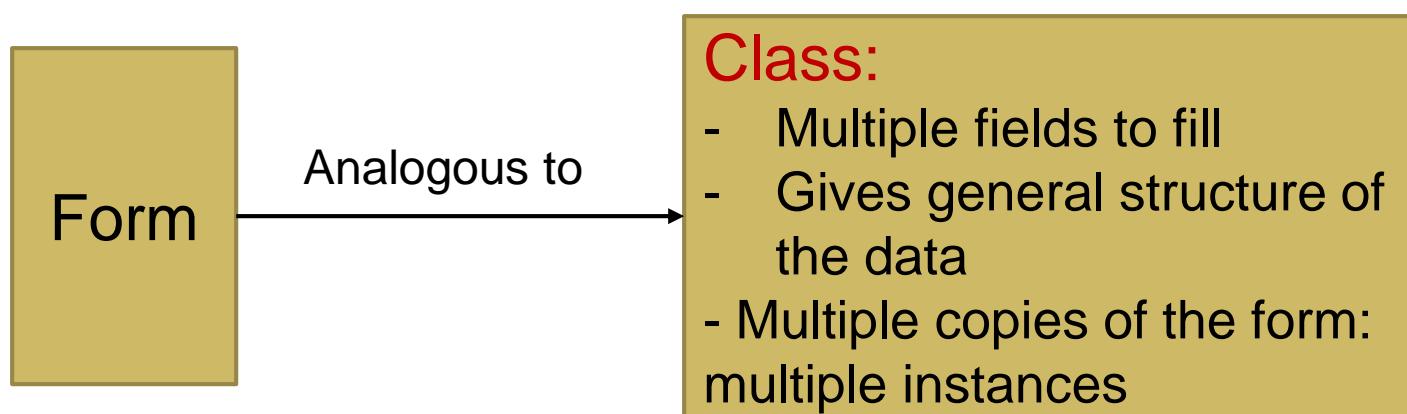


# Object and Class- Analogy

- An analogy for Objects and Class: **Admission Forms (hard copy)**
- The **form itself is like the class.**
  - ▣ It has multiple fields to fill in.
  - ▣ You can print multiple copies of the form, just like you can have multiple instances of a class.
  - ▣ The class gives the general structure of the data, like a form.
- An **instance (object) is one particular copy of the form.**
  - ▣ When you print out a copy and fill it in, you give values to fields like “Name”, “Birthday”, “Percentage” etc.
  - ▣ A single, completed copy of the form is like an instance of the class.
  - ▣ You and your friend could each have a different copy of the form: in that case, the structure of the data would be the same, but the content would be different.

# Object and Class- Analogy

An analogy for Objects and Class: **Admission Forms (hard copy)**



# Object and Class

- **Object:** An **object** is a **custom data structure** that organizes and encapsulates variables and methods into a single data type.
  - It is used near-interchangeably with “instance”.
  - A single set of values of a particular class.
- 
- **Class:** A **custom data type** comprised of multiple variables and/or methods.
  - Instances or objects are created based on the **template provided by the class**.
  - An instance is a set of values for these variables.

# Class

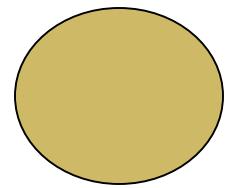
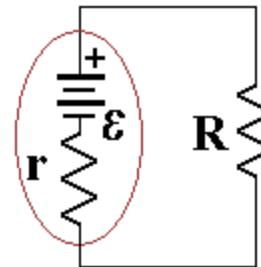
- Example:
  - A **person** would be a single entity that has a lot of variables about them.
    - A first name and a last name.
    - A height, a weight, DOB
    - A hair color, an eye color
    - A phone number, a residential address, an email address
  - These are all variables that we could wrap up into one data type, and we'd call that data type a **person class**.

# Basic Terminology

- **Object:**
  - usually a person, place or thing (a noun)
- **Method:**
  - an action performed by an object (a verb)
- **Attribute:**
  - description of objects in a class
- **Class:**
  - a category of similar objects (such as automobiles)
  - does not hold any values of the object's attributes

# What is an object?

- Tangible Things      as a car, printer, ...
- Roles                  as employee, boss, ...
- Incidents              as flight, overflow, ...
- Interactions          as contract, sale, ...
- Specifications      as colour, shape, ...



# Representing Objects

- An object is represented as rectangle

: Professor

Class Name Only

ProfessorSurati :  
Professor

Class and Object Name

ProfessorSurati

Object Name Only

Class and object



Professor Surati

# What is a Class?

- A class is a **description of a group of objects**
  - ▣ having common properties (attributes), behavior (operations or methods), relationships and semantics
  - ▣ An object is an instance of a class
- A class is an abstraction in that it
  - ▣ Emphasizes relevant characteristics
  - ▣ Suppresses other characteristics

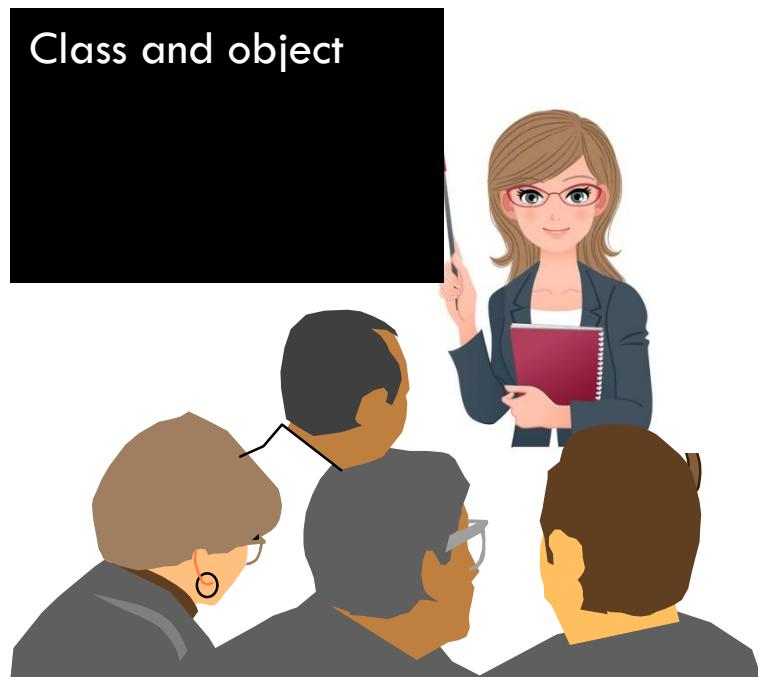
# Example Class

## Class Course

### Properties

- Name
- Location
- Days offered
- Credit hours
- Start time
- End time

Class and object

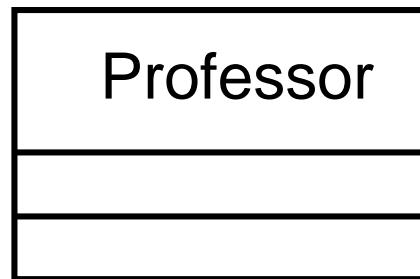


### Behavior

- Add a student
- Delete a student
- Get course syllabus
- Determine if it is full

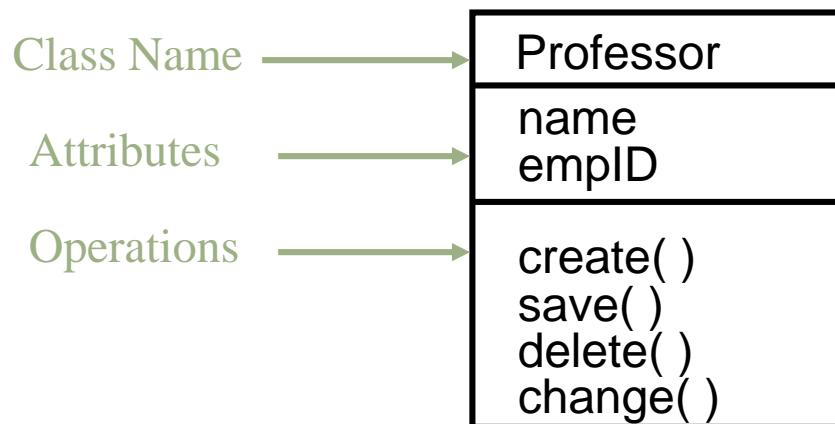
# Representing Classes

- A class is represented using a compartmented rectangle



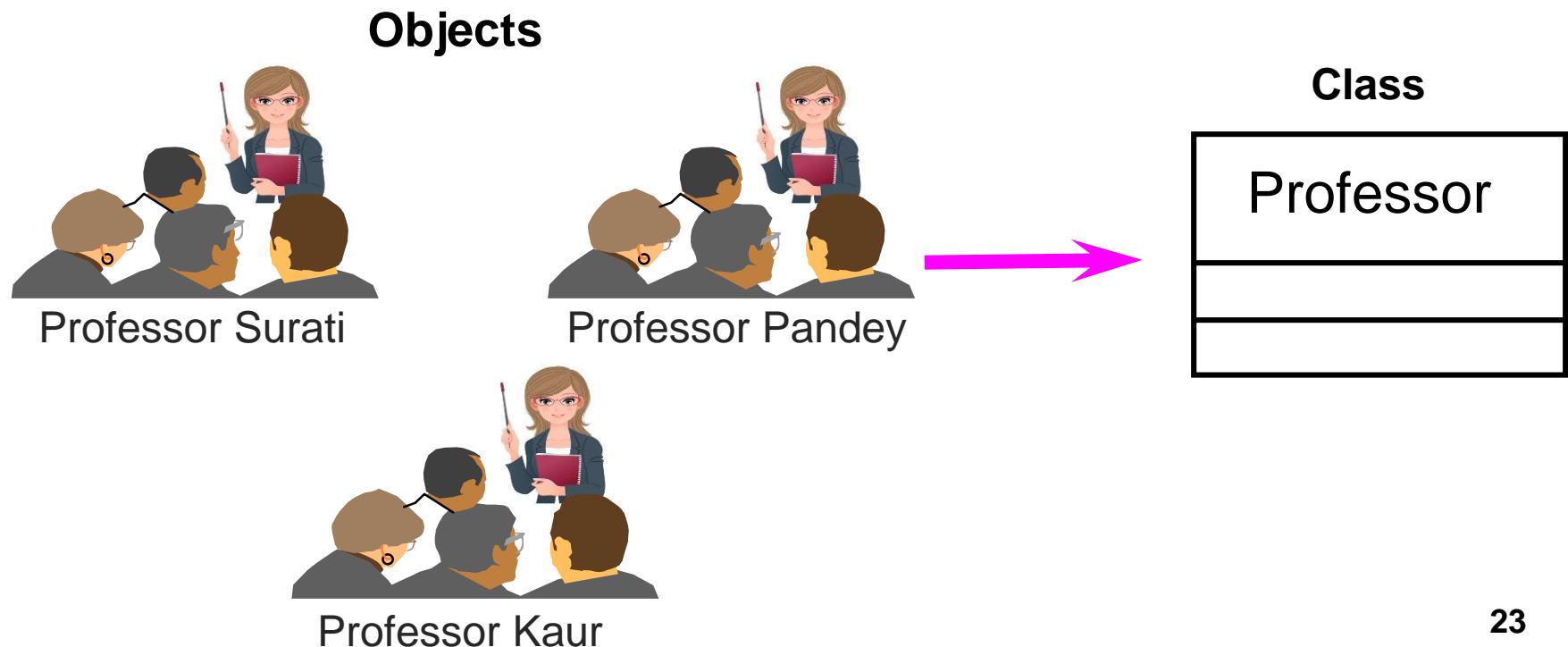
# Class Compartments

- A class is comprised of three sections
  - The first section contains the class name
  - The second section shows the structure (attributes)
  - The third section shows the behavior (operations)

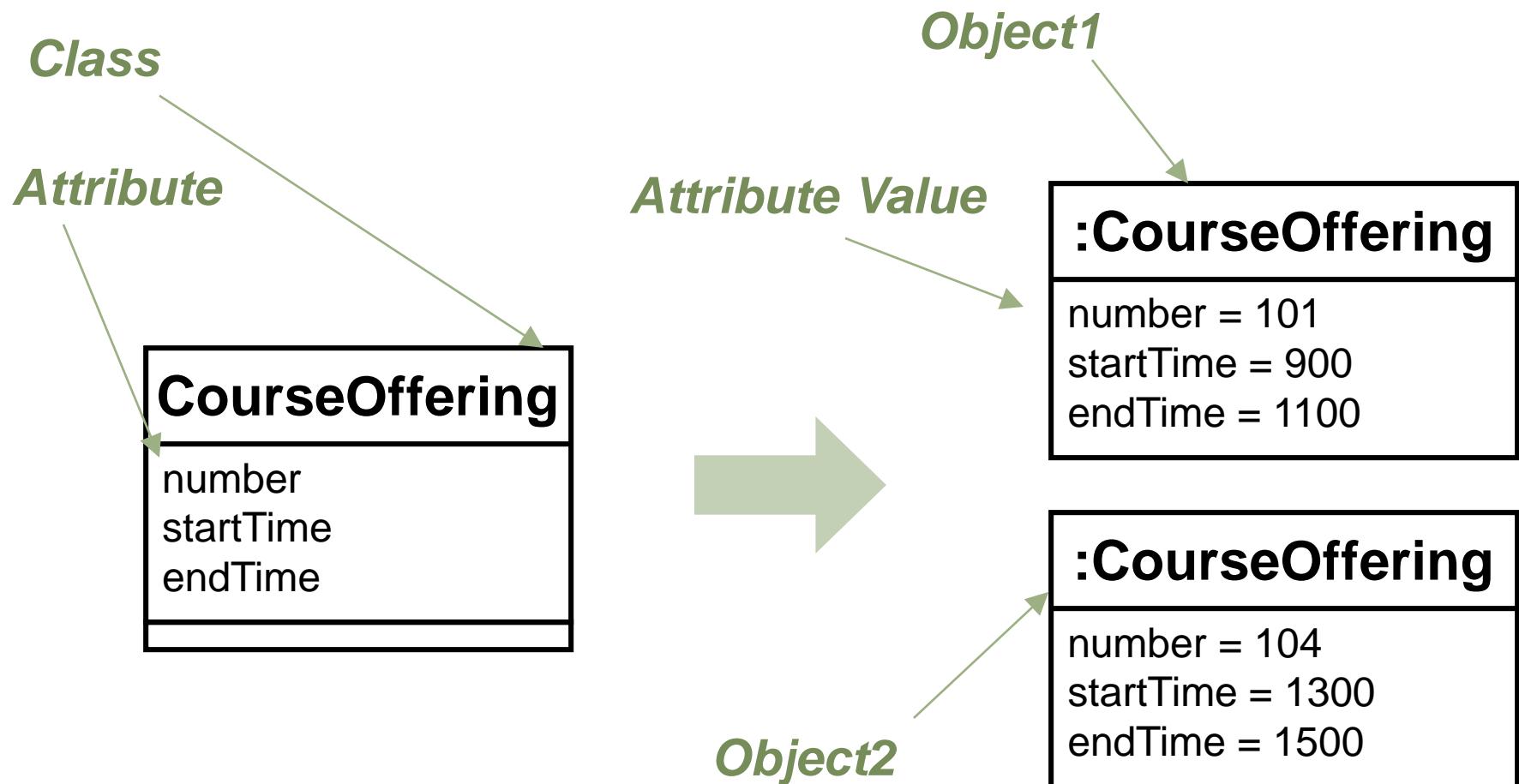


# The Relationship Between Classes and Objects

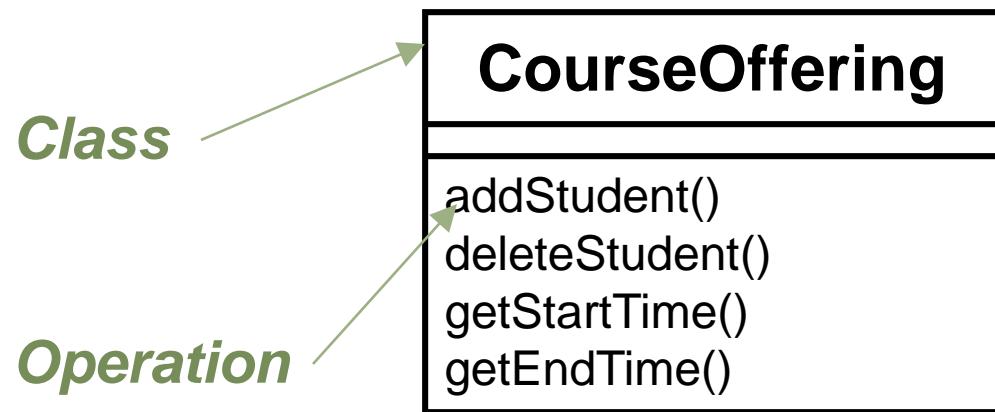
- A class is an **abstract definition of an object**
    - It defines the structure and behavior of each object in the class
    - It serves as a template for creating objects
  - Objects are grouped into classes



# What is an Attribute?

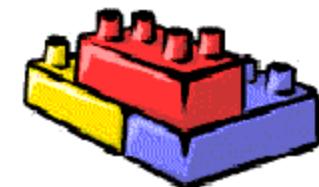


# What is an Operation?



# Why do we care about objects?

- **Modularity** - large software projects can be split up in smaller pieces.
- **Reusability** - Programs can be assembled from pre-written software components.
- **Extensibility** - New software components can be written or developed from existing ones.



# OOP Introduction





Name	Pluto	Scooby Doo	Droopy	Spike
Skin Color	Yellow	brown	white	grey
Ear length	long	short	long	short
Is spotted	no	yes	no	no

Attributes

Values

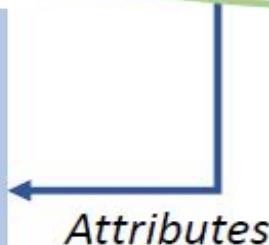
*class: Dog*



## *class: Dog*



- Name
- Skin color
- Ear length
- Is spotted

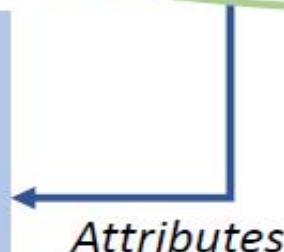


**Class:** A custom data type comprised of multiple variables and/or methods.

*class: Dog*



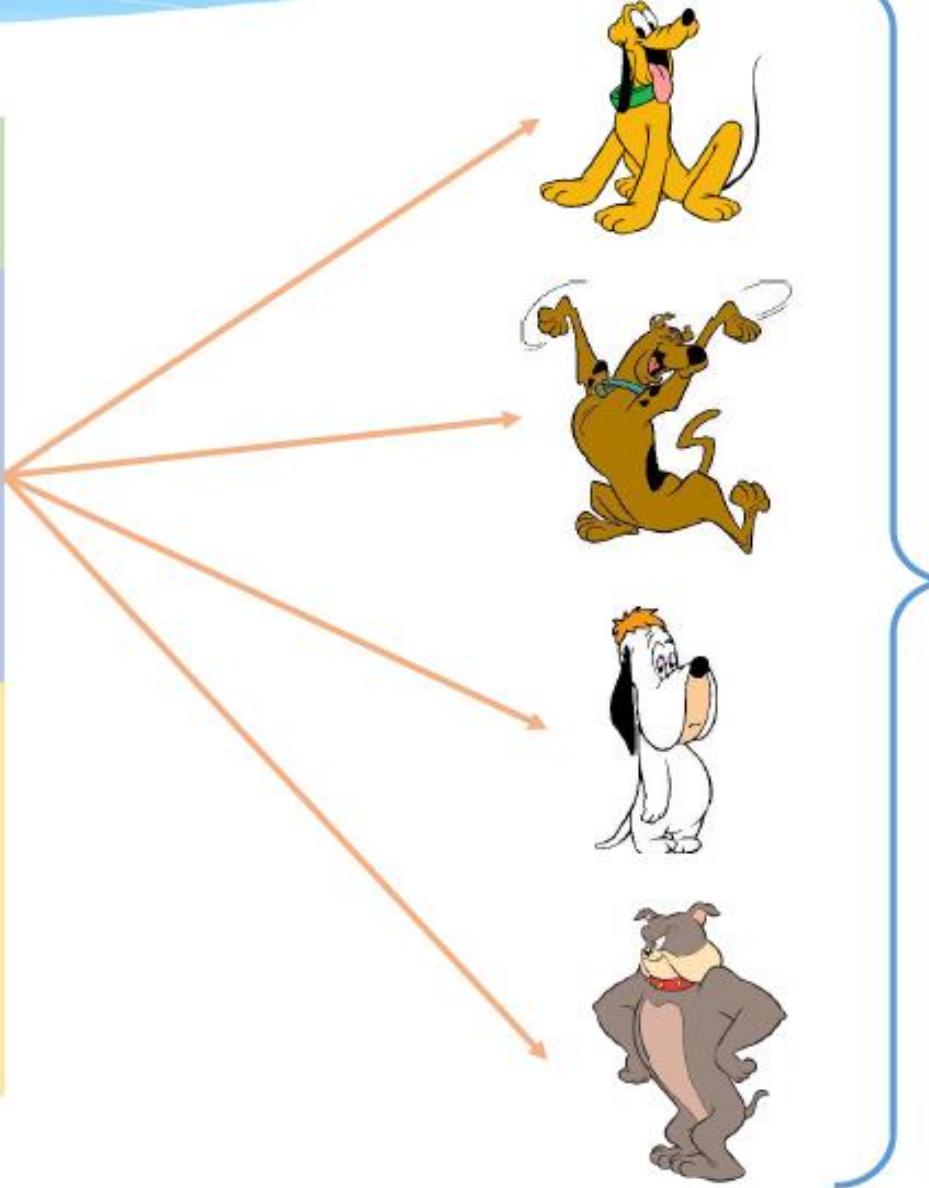
- Name
- Skin color
- Ear length
- Is spotted



## class: **Dog**

- Name
- Skin color
- Ear length
- Is spotted

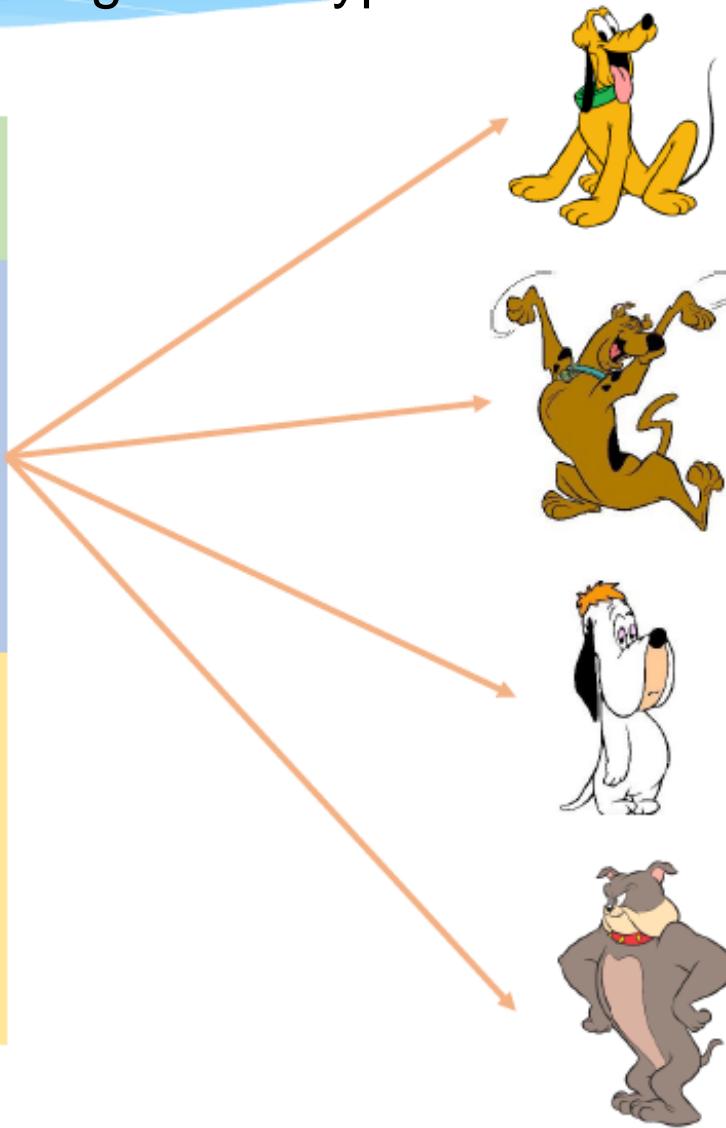
- Walk
- Eat
- Sleep
- Chase cat



*Objects  
of the  
class  
Dog*

**Object:** An object is a custom data structure that organizes and encapsulates variables and methods into a single data type.

class: <i>Dog</i>	
▪ Name	
▪ Skin color	
▪ Ear length	
▪ Is spotted	
➤ <i>Walk</i>	
➤ <i>Eat</i>	
➤ <i>Sleep</i>	
➤ <i>Chase cat</i>	

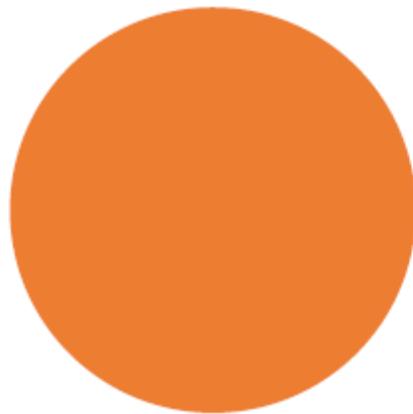


Name	Pluto
Skin Color	Yellow
Ear length	long
Is spotted	no
Name	Scooby Doo
Skin Color	brown
Ear length	short
Is spotted	yes
Name	Droopy
Skin Color	white
Ear length	long
Is spotted	no
Name	Spike
Skin Color	grey
Ear length	short
Is spotted	no

## class: *Dog*

- Name
- Skin color
- Ear length
- Is spotted

- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*



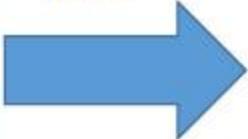
Name	-
Skin Color	-
Ear length	-
Is spotted	-

class: **Dog**

- Name
- Skin color
- Ear length
- Is spotted

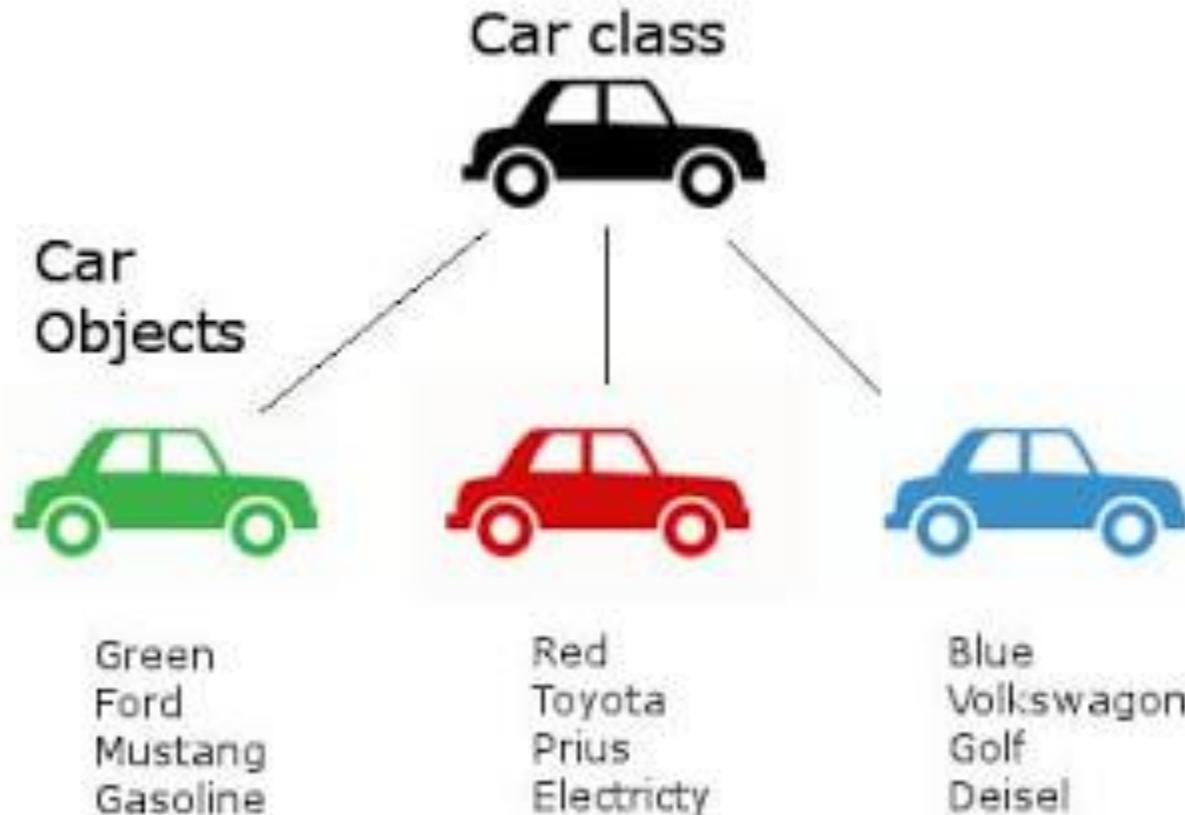
- *Walk*
- *Eat*
- *Sleep*
- *Chase cat*

*New*

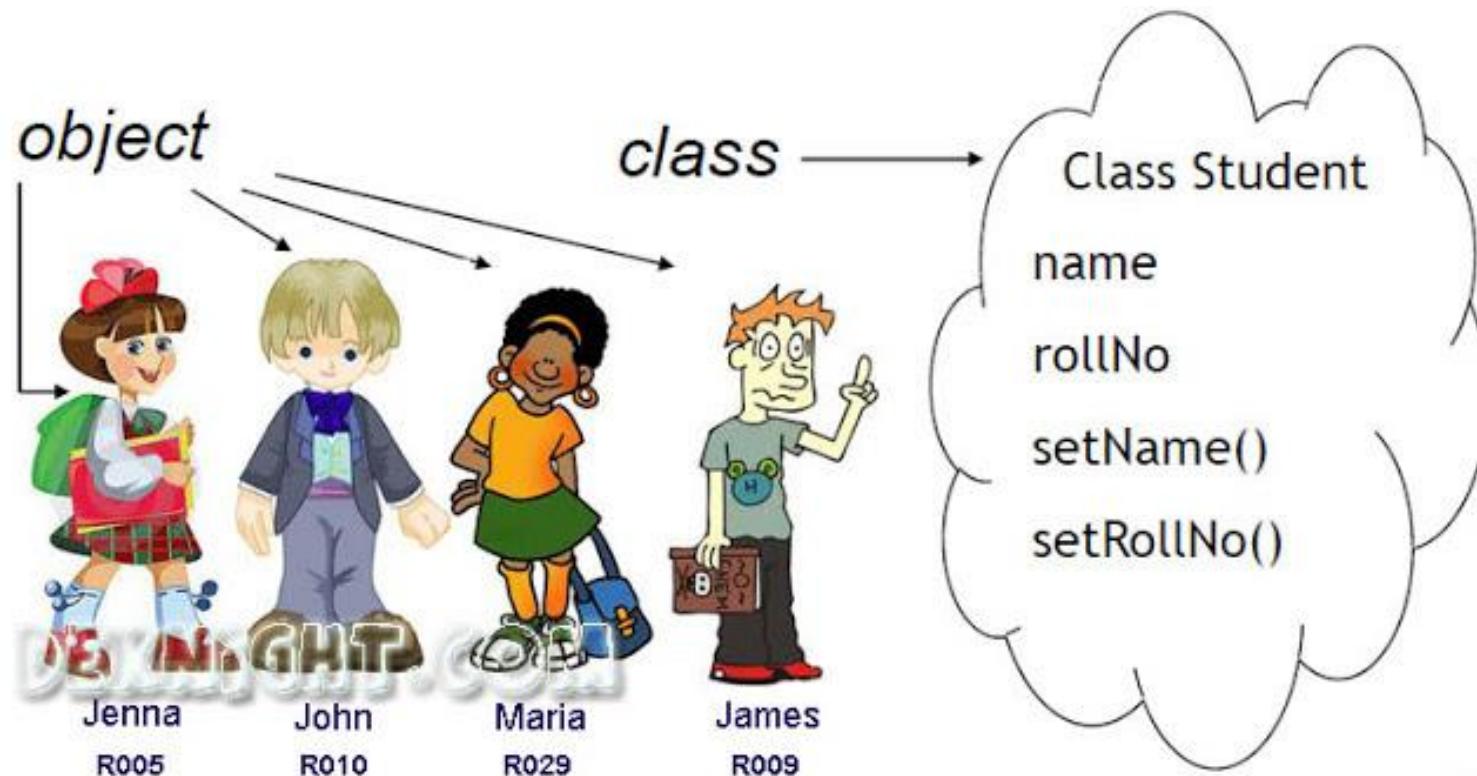


Name	Mr. Vodafone
Skin Color	cream
Ear length	short
Is spotted	no

# Another Example



# Example



# Example

## Class : Cutter

### Objects



COMBINATION PLIERS



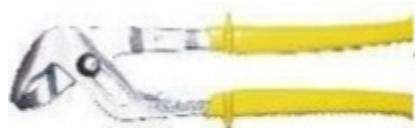
WIRE STRIPPER AND CUTTER



TRIMMING PLIERS



CRIMPING PLIERS



MULTIGRIPS



LONGNOSE PLIERS



LONGNOSE PLIERS

# Example

## Class : Hammer



Raw-hide headed hammer



Lead hammer



Double-headed hammer



Ball peen hammer



Cross peen hammer



Straight peen hammer

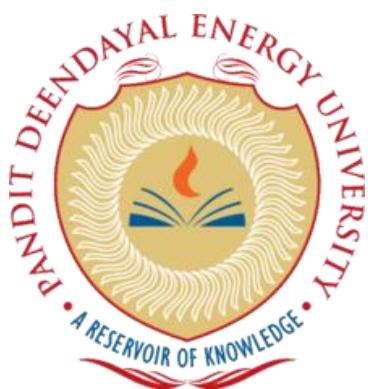
Objects

# Exercise

- Each of the following pairs represent a class-instance pair:
- In each pair, select which option is best described as a **class**?
- **Set 1:**
  - A pet
  - My Cat, Boggle
- **Set 2:**
  - My daughter, Lucy
  - A person
- **Set 3:**
  - A beverage
  - The can of soda, Lucy drinking right now.

```
class copyTest{  
    int a;  
    String name;  
  
    copyTest (){  
        System.out.println("In default constructor");  
    }  
  
    copyTest (int ia, String iName){  
        a=ia;  
        name=iName;  
    }  
    copyTest (copyTest m){  
        System.out.println("In copy constructor");  
        a = m.a;  
        name=m.name;  
    }  
    void printData(){  
        System.out.println("a =" +a+ " ,name=" +name);  
    }  
}
```

```
class mainClass {  
    public static void main(String[] args) {  
  
        copyTest t1 = new copyTest (5, "CE");  
  
        // Reference of an object  
        copyTest t2 = t1;  
        t2.name= "CSE"; //Changes t1.name?  
        System.out.println("Name in t1 is:");  
        t1.printData();  
  
        copyTest t3 = new copyTest (10, "IT");  
        //Creating object by calling copy constructor  
        copyTest t4 = new copyTest(t3); //calls copy  
        constructor  
        t4.name= "ICT"; //Changes t3.name?  
        System.out.println("Name in t3 is:");  
        t3.printData();  
    }  
}
```



# CLASSES AND OBJECTS-PROGRAMS AND CONSTRUCTORS

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Outline

- Default Arguments
- Variable Arguments (Varargs) and examples
- Class- Simple Program
- Creating object of the same class
- Constructors
- Destructor

# Default Arguments

- an argument to a function that a programmer is not required to specify
- Default values can be given
- EX:

```
class defaultArgu{  
    void add(int a, int b=5) {  
        System.out.println(a+b);  
    }  
    public static void main(String[] args) {  
        add(5,10);  
        add(5);  
    }  
}
```

# Variable Arguments (Varargs)

- A method that takes a variable number of arguments
- Syntax of Varargs

```
public static void fun(int ... a) // (data_type ... variable_name)
```

```
{  
    // method body
```

```
}
```

- Internally, the Varargs method is implemented by using the **one dimensional arrays concept.**
- Hence, in the Varargs method, arguments are differentiated by using **Index.**

# Varargs Example

```
class Test1 {  
    // Method that takes variable number of integer arguments.  
    static void fun(int... a)    {  
        System.out.println( "Number of arguments: " + a.length);  
        for (int i : a) // using for each loop to display contents of a  
            System.out.print(i + " ");  
    }  
    public static void main(String args[])    {  
        fun(100);          // one parameter  
        fun(1, 2, 3, 4);  // four parameters  
        fun();             // no parameter  
    }  
}
```

# Varargs...

- A method can have **variable length parameters** with other **parameters** too
- **only one varargs parameter** that should be **written last** in the parameter list of the method declaration

EX: int nums(int a, float b, double ... c)

## Errors:

- Specifying two Varargs in a single method:  
`void method(String... gfg, int... q) //error`
- Specifying Varargs as the first parameter of the method instead of the last one:  
`void method(int... gfg, String q) //error`

# Class- Simple Program

```
Class student{  
    String name;  
    String roll no;  
    int marks;  
    void setData(); //setter  
    void printData();  
}
```

# Creating object of the same class

```
class Temp{  
    int a;  
    public static void main(String[] args) {  
        Temp objTemp = new Temp();  
        objTemp.a=10;  
        System.out.println(objTemp.a);  
        System.out.println(objTemp); //check output  
    }  
}
```

# Constructors

- A special member function
  - to initialize objects with default values unless different values are supplied
  - that takes the same name as the class name
  - that cannot return values (No return type)
  - that is invoked automatically at the time of object creation
  - that can be overloaded
  - The syntax generally is as given below:  
`<class name> {arguments};`

# Constructors...

- Several forms:
  - default constructor (without parameter)
  - parameterized
  - copy constructor
- We can define constructors with **default arguments**
- Unlike methods, **constructors are not considered members** of the class
- Constructor overloading

# Simple Program

```
Class Point{  
    int x, y, z;  
    Point(); //default constructor  
    Point(int, int, int); // parameterized constructor  
    setData();  
    getData();  
    translate();  
    calDistanceOrigin();  
}
```

Lect-7\_prog

# Copy Constructor

- When it is required to create an exact copy of an existing object of the class such that
  - if we have **made any changes in the copy** it should not be **reflected** in the original one and vice-versa.
- **A special type of constructor** that creates an object using another object of the same Java class (**Deep copy**)
- **Parameter- Object** of the same class
- Copies all attributes of first object into second object
- **Returns a duplicate copy** of an existing object of the class
- **EX:**

Copy constructors.doc

# Destructor

- A special member function
  - ▣ To release dynamic allocated memory
  - ▣ Same name as class name
  - ▣ No return type
  - ▣ Cannot be overloaded (only one)
  - ▣ finalize() method:

**protected void finalize()**

{

```
    System.out.println("Object is destroyed by the Garbage Collector");  
}
```

# Questions

- Difference between constructor and method in Java?
- Can we overload main method?
- Difference between VarArgs and method overloading?
- Difference between copy constructor and ob.clone() method?

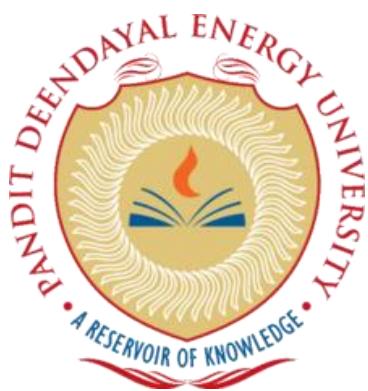
```
class Main {
    public static void main(String[] args) {
        Point p1 = new Point();
        p1.x=10;
        p1.y=20;
        p1.z=30;

        System.out.println(p1.calDistanceOrigin());
        Point p2=new Point();
        p2.setData(5,15,25);

        System.out.println(p2.calDistanceOrigin());
        p2.translate(5, 5, 5);
        System.out.println(p2.x+" "+p2.y+" "+p2.z);
    }
}
```

```
class Point{
    double x,y,z;
    Point(){
        x=y=z=0;
        System.out.println("In default constructor");
    }
    Point(int ix, int iy, int iz){
        x=ix;
        y=iy;
        z=iz;
        System.out.println("In parameterized constructor");
    }
    void setData(int a, int b, int c){
        x=a;
        y=b;
        z=c;
    }
    double calDistanceOrigin(){
        return Math.sqrt(x*x + y*y + z*z);
    }
    void translate(int dx, int dy, int dz){
```

```
    x = x + dx;
    y = y + dy;
    z = z + dz;
}
void printData()
{
    System.out.print("Point is:");
    System.out.println("(" + x + ", " + y + ", " + z + ")");
}
}
```



# STATIC AND REFERENCES

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Outline

- Static Keyword and features
- Static Variable Example
- Static Method and Example
- Restrictions for the static method
- Static-Points to remember
- Call by Value, Call by Reference
- ‘this’ reference in Java
- Explore ‘this’ reference

# Static Keyword

- used for memory management mainly
- used to refer to the **common property of all objects** (not unique for each object)
- **single copy storage** for variables or methods
- The members that are declared with the static keyword inside a class are called static members in java.
- **can be accessed/called even if no instance (object) of the class exists**
  - ▣ not tied to a particular instance
  - ▣ shared across all instances of the class
  - ▣ EX: **main method is static**, so can be called by JVM without creating an object

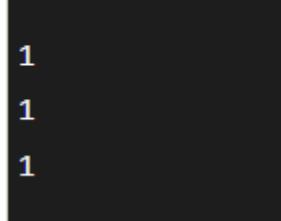
# Features of static keyword

- Can be applied with **variables, methods, inner (nested) classes, and blocks**
- A class cannot be static, **but an inner class can be static**
- Property (attribute or method) of a class, not of an instance (object)
- **Memory is allocated only once** when class is loaded into memory
- the static variable is created and **initialized into the common memory location only once**

# Static Variable Example

```
class Counter{  
    int count=0;          // the instance variable  
    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

# Example...

```
class Counter{  
    static int count=0; // the static variable  
    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  
2  
3
```

# Example...

Static\_prog\_ex.doc

How to change value of static variable?

# Static Method

- If you apply static keyword with any method, it is known as static method.
  - ▣ **belongs to the class** rather than the object of a class
  - ▣ can be invoked **without the need for creating an instance** of a class
  - ▣ can access static data member and **can change the value of it**

# Static method example

Static\_prog\_ex.doc

# 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.
- EX:

```
class A{  
    int a=40;      //non static
```

```
public static void main(String args[]){  
    System.out.println(a);  
}  
}  
  
//error, can't access without object
```

# Static-Points to remember

- Static member cannot call an instance member.
  - Static method can call a static method.
  - Instance method can call a static method.
  - Instance method can call an instance method.
  - Static can be called with object name, but instance can't be called using class name.
- 
- Math class- all static methods
  - String class- all instance methods

# Call by Value, Call by Reference

Reference\_Obj\_prog.doc

# ‘this’ reference in Java

1. Using ‘this’ keyword to refer current class instance variables

```
class Test
```

```
{
```

```
    int a;
```

```
    int b;
```

```
        // Parameterized constructor
```

```
Test(int a, int b)
```

```
{
```

```
    this.a = a;
```

```
    this.b = b;
```

```
}
```

```
}
```

## 2. Using this() to invoke current class constructor

```
class Test
```

```
{
```

```
    int a;
```

```
    int b;
```

```
//Default constructor
```

```
Test()
```

```
{
```

```
    this(10, 20);
```

```
    System.out.println("Inside default constructor \n");
```

```
}
```

```
//Parameterized constructor
Test(int a, int b)
{
    this.a = a;
    this.b = b;
    System.out.println("Inside parameterized constructor");
}

public static void main(String[] args)
{
    Test object = new Test();
}

}
```

# Explore ‘this’ reference

3. Using ‘this’ keyword **to return the current class instance**
4. Using ‘this’ keyword **as method parameter**
5. Using ‘this’ keyword **to invoke current class method**
6. Using ‘this’ keyword **as an argument in the constructor call**

```
//Java Program to demonstrate the use of static variable

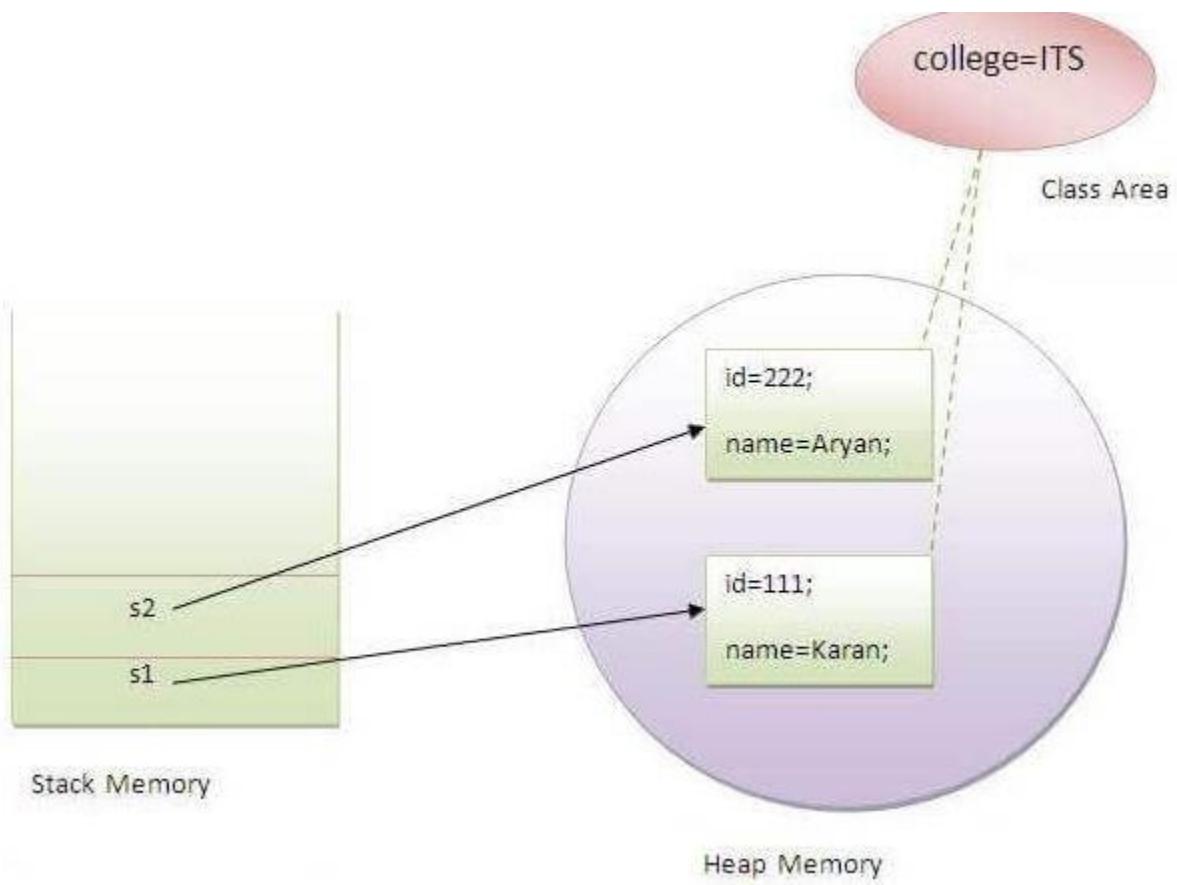
class Student {
    int rollno;
//instance variable
    String name;

    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n) {
        rollno = r;
        name = n;
    }
    //method to display the values
    void display () {
        System.out.println(rollno+ " "+name+ " "+college
    );
}
}

//Test class to show the values of objects
```

```
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");

        s1.display();
        s2.display();
    }
}
```



Output:

```
111 Karan ITS  
222 Aryan ITS
```

```
//Java Program to demonstrate the use of a static method.

class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
        college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display values
    void display(){
        System.out.println(rollno+" "+name+" "+college);
    }
}

//Test class to create and display the values of object

public class TestStaticMethod{
    public static void main(String args[]){
        Student.change(); //calling change method
        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
    }
}
```

```
        Student s3 = new Student(333, "Sonoo");
        //calling display method
        s1.display();
        s2.display();
        s3.display();
    }
}
```

Output:111 Karan BBDIT  
222 Aryan BBDIT  
333 Sonoo BBDIT

---

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number
using the static 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);
    }
}
```

```

//Program-1 Call by value
class Tester {
    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b); // Invoke the
        swap method

        swapFunction(a, b); //main is a static method, hence it can call only static
        method

        System.out.println("\n**Now, Before and After swapping values will be same
        here**:");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }

    static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b); // Swap n1
        with n2
        int c = a;
        a = b;
        b = c;
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);
    }
}

//if static void swapFunction(int,int) is in another class, call it by className.swapFunction(a,b);

```

### **// Program-2 Creating reference of an object**

```

public class Main{

    public static void main(String[] args) {
        Point p1 = new Point();
        p1.x=10;
        p1.y=20;
        p1.z=30;

        System.out.println(p1. calDistanceOrigin ());
        Point p2=new Point();
        p2.setData(5,15,25);

        System.out.println(p2. calDistanceOrigin ());
    }
}

```

```

p2.translate(5, 5, 5);
System.out.println(p2.x+" "+p2.y+" "+p2.z);

Point p3=p1; //Does it call constructor? --> No
p3.x=50;
System.out.println(p1.x); // Does p3.x=50 change p1.x? --> Yes

}

}

class Point{
    double x,y,z;
    Point(){
        x=y=z=0;
        System.out.println("In default constructor");
    }
    Point(int ix, int iy, int iz){
        x=ix;
        y=iy;
        z=iz;
        System.out.println("In parameterized constructor");
    }
    void setData(int a, int b, int c){
        x=a;
        y=b;
        z=c;
    }
    double calDistanceOrigin (){
        return Math.sqrt(x * x + y * y + z*z);
    }
    void translate(int dx, int dy, int dz){
        x = x + dx;
        y = y + dy;
        z = z + dz;
    }
    void printData()
    {
        System.out.print("Point is:");
        System.out.println("(" + x + ", " + y + ", " + z + ")");
    }
}

```

```
//Program-3 Call by reference
class Tester {
    int a, b;
    Tester(){
        System.out.println("In default constructor");
    }
    public static void main(String[] args) {
        Scanner s=new Scanner(System.in);
        Tester t = new Tester();

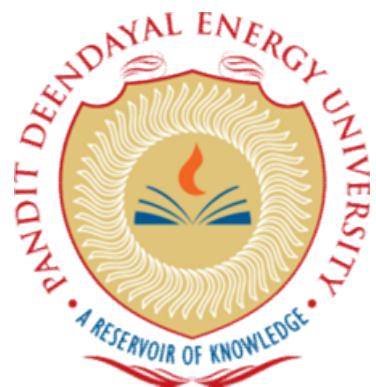
        System.out.println("enter a and b");
        t.a=s.nextInt();
        t.b=s.nextInt();

        System.out.println("Before swapping"); // Invoke the swap method
        t.printData();
        t.swapFunction(t);
        System.out.println("After swapping");
        t.printData();
    }
    void swapFunction(Tester t) { //Will it call default constructor? No
        // swap a with b
        int c = t.a;
        t.a = t.b;
        t.b = c;
    }
    void printData(){
        System.out.println("a =" +a+ " ,b=" +b);
    }
}
```

## //Program-4 Returning objects

Program to implement returning objects.

```
class Rectangle {  
    int length;  
    int breadth;  
    Rectangle(){  
        System.out.println("In default constructor");  
    }  
    Rectangle(int l,int b) {  
        System.out.println("In parameterized constructor");  
        length = l;  
        breadth = b;  
    }  
    Rectangle getRectangleObject() {  
        Rectangle rect = new Rectangle(10,20);  
        return rect; //Will it call default constructor?  
    }  
}  
class RetOb {  
    public static void main(String args[]) {  
        Rectangle ob1 = new Rectangle(40,50);  
        Rectangle ob2; //Will it call default constructor?  
        ob2 = ob1.getRectangleObject();  
        System.out.println("ob1.length : " + ob1.length);  
        System.out.println("ob1.breadth: " + ob1.breadth);  
        System.out.println("ob2.length : " + ob2.length);  
        System.out.println("ob2.breadth: " + ob2.breadth);  
    }  
}
```



# INHERITANCE

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,  
School of Technology,  
Pandit Deendayal Energy University

# Outline

- Final Keyword
- Inheritance
- Why Inheritance?
- Inheritance Example
- Types of Inheritance
- Using Super- two uses
- Method Overriding
- Access modifiers
- Abstract class
- Using Final with Inheritance
- Dynamic Method Dispatch

# Final

- Used to restrict the user.
- Final can be:
  - ▣ variable
  - ▣ method
  - ▣ Class
- If a field declared as final, the copy constructor can change it.

## Java Final Keyword

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

javatpoint.com

# Final variable

- The value of final variable cannot be changed
- Initialize it when it is declared

EX:

```
class carSpeed{  
    final int speed=70; //final variable  
    void changeSpeed(){  
        speed=100; //compile time error  
    }  
    public static void main(String args[]){  
        carSpeed carObj = new carSpeed();  
        carObj.changeSpeed();  
    }  
}
```

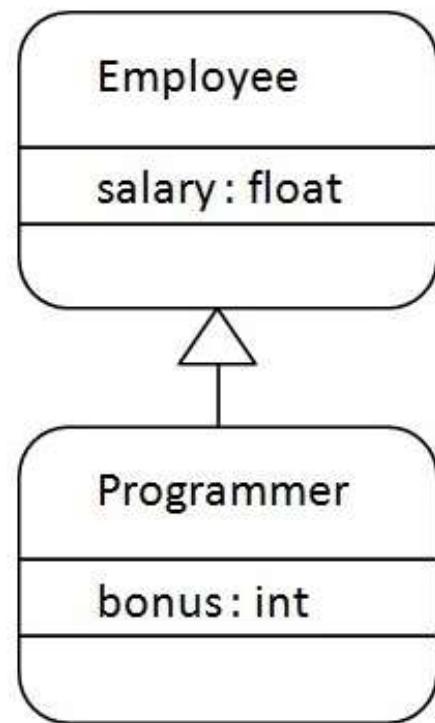
# Inheritance

- A mechanism in which **one object acquires all the properties and behaviors of a parent object**
- Important feature of OOPs
- **Idea:** create new classes that are built upon existing classes
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

# Why Inheritance?

- **Reusability:** to reuse methods and fields of the parent class
- **Method overriding (Run-time polymorphism)**
  
- Define superclass- general aspects of an object
- Inherit superclass to form specialized classes
- Each subclass simply adds its own attributes

# Inheritance Example



- Relationship: Programmer IS-A Employee

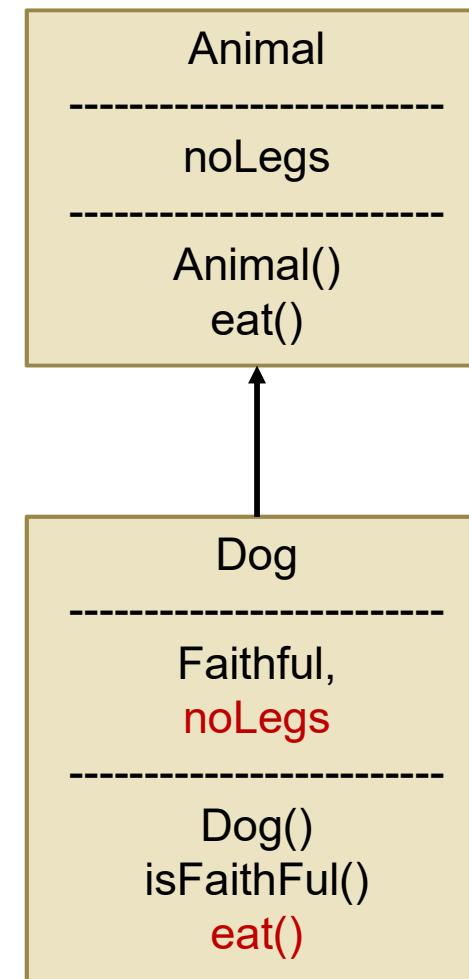
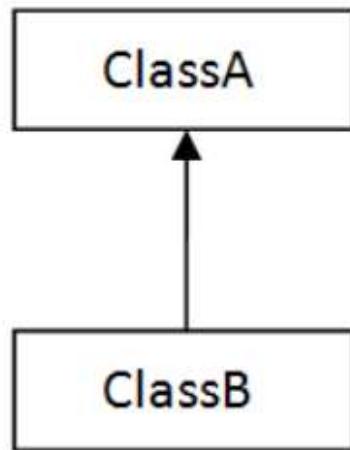
# Simple Program

Inheritance program.doc (**Simple Example**)  
(Refer this file for types of inheritance also).

# Types of Inheritance

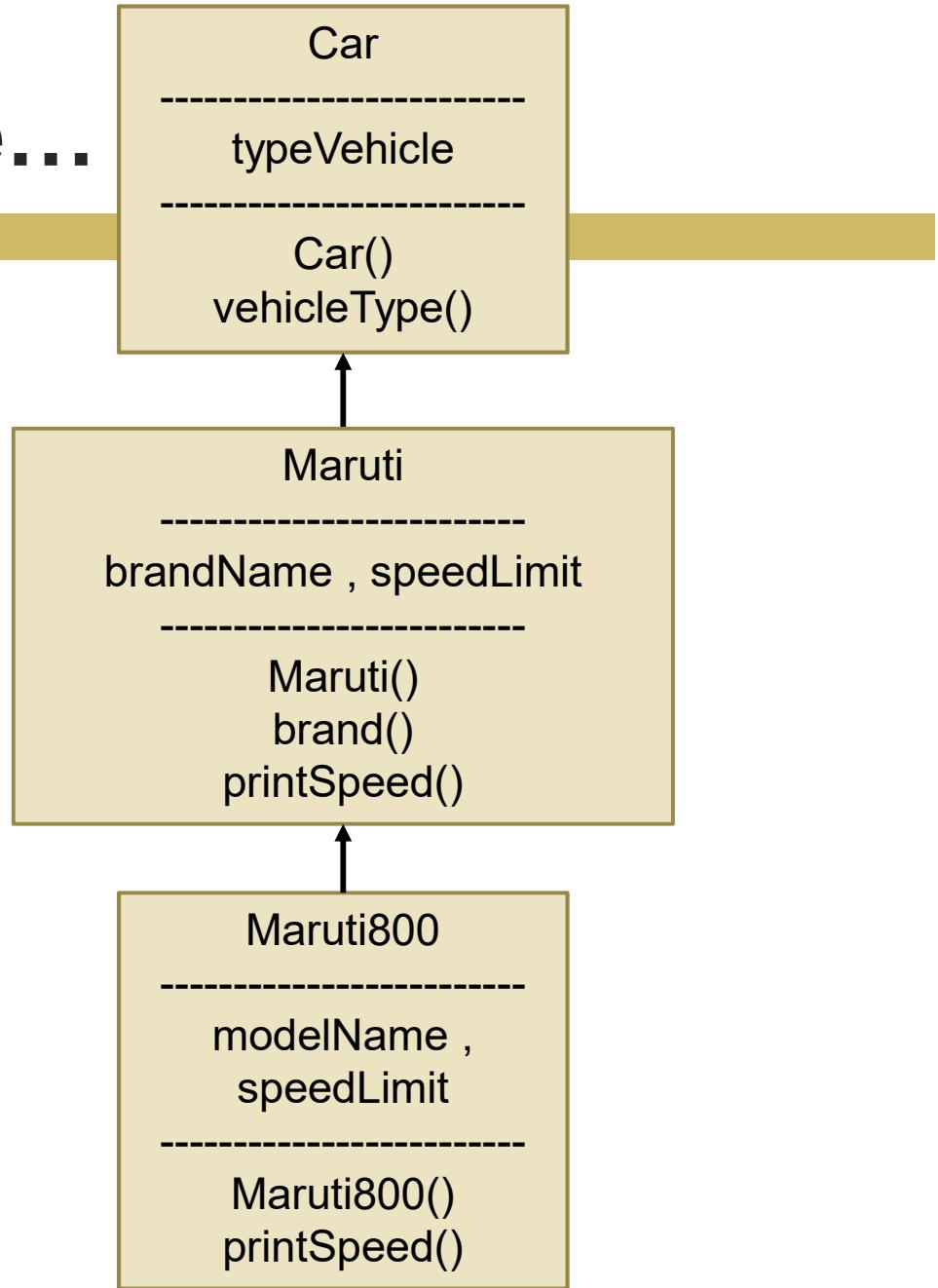
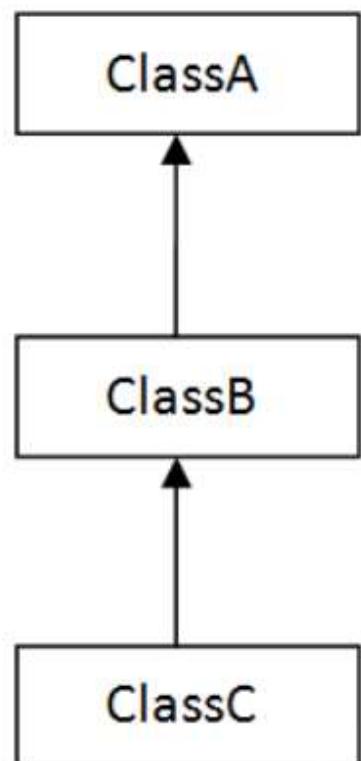
## 1) Single Inheritance

Inheritance program.doc  
**(Single Inheritance)**



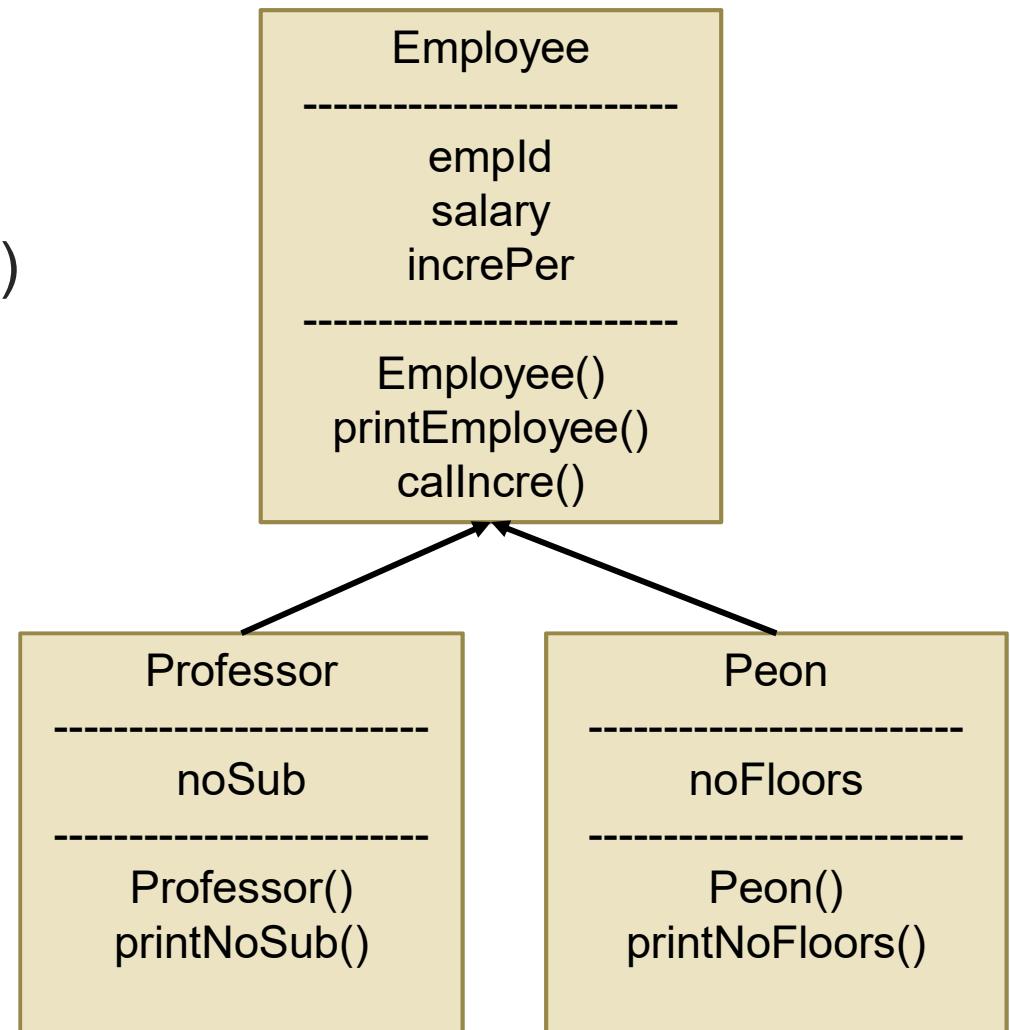
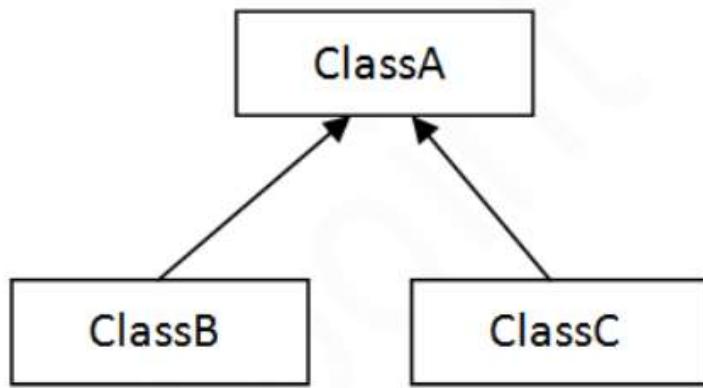
# Types of Inheritance...

2) Multilevel Inheritance  
Inheritance program.doc  
**(Multilevel Inheritance)**



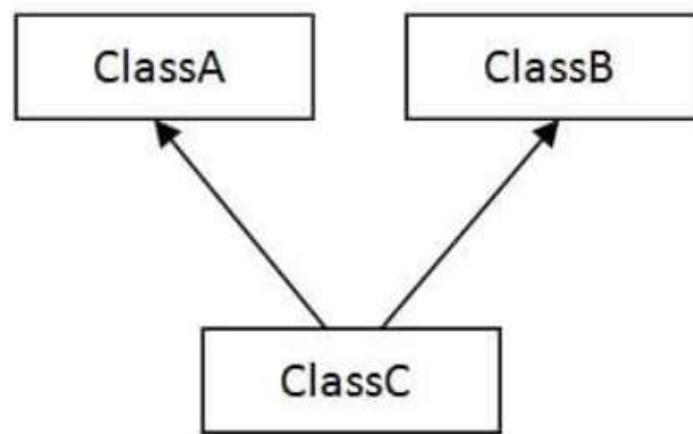
# Types of Inheritance...

3) Hierarchical Inheritance  
Inheritance program.doc  
**(Hierarchical Inheritance)**

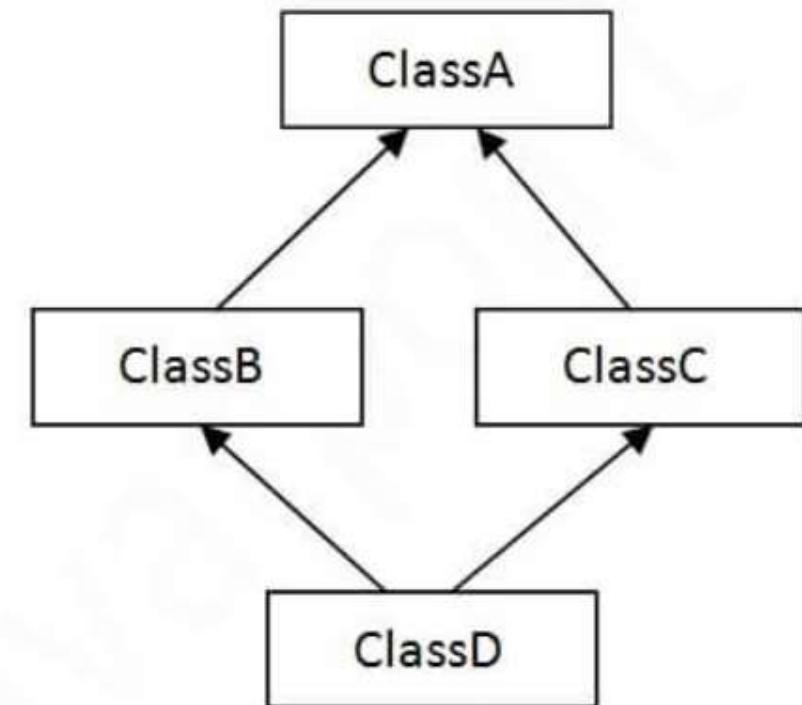


# Types of Inheritance...

## 4) Multiple Inheritance



## 5) Hybrid Inheritance



These two inheritances are **not supported by Java Classes !!**

# Using Super- two uses

(1) To call Superclass Constructors

- Syntax: super(arg-list);

- From previous programs:

Inheritance program.doc (Hierarchical Inheritance, Prog-3, Prog-5)

# Using Super- two uses...

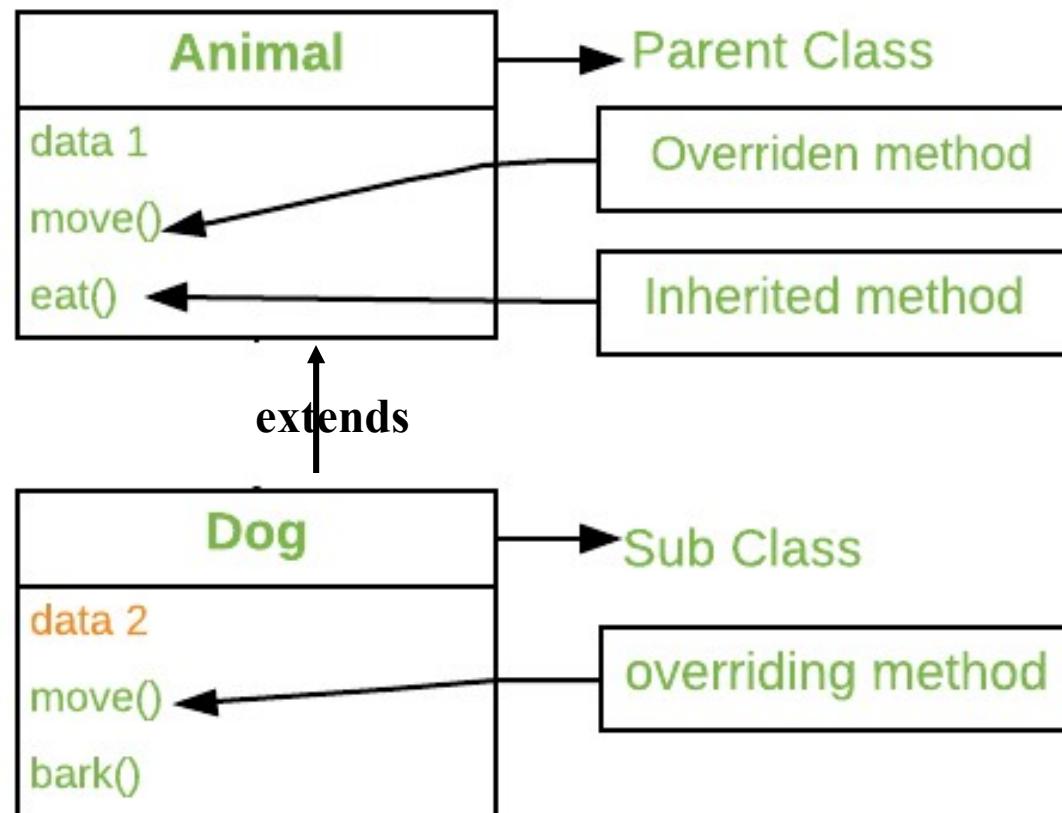
- (2) To refer superclass member (instance variable/method)
  - Syntax: super.member;
  - Used when subclass members hide members by the same name in the superclass
  - Inheritance program.doc (Prog-6)

# Method Overriding

- If subclass (child class) has the same method as declared in the parent class
- Rules for Java Method Overriding
  - ▣ Applicable in Inheritance (IS-A relationship)
  - ▣ 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...

(1) Ex-1:

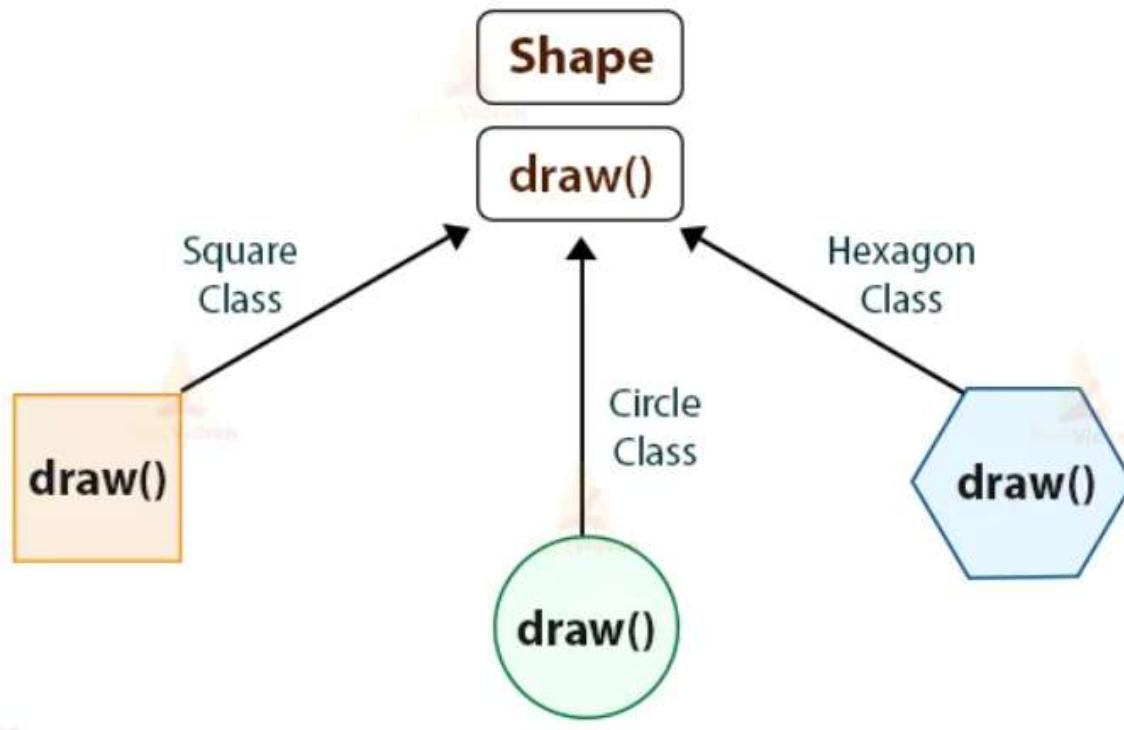


(2) Inheritance programs-1.doc

# Method Overriding...

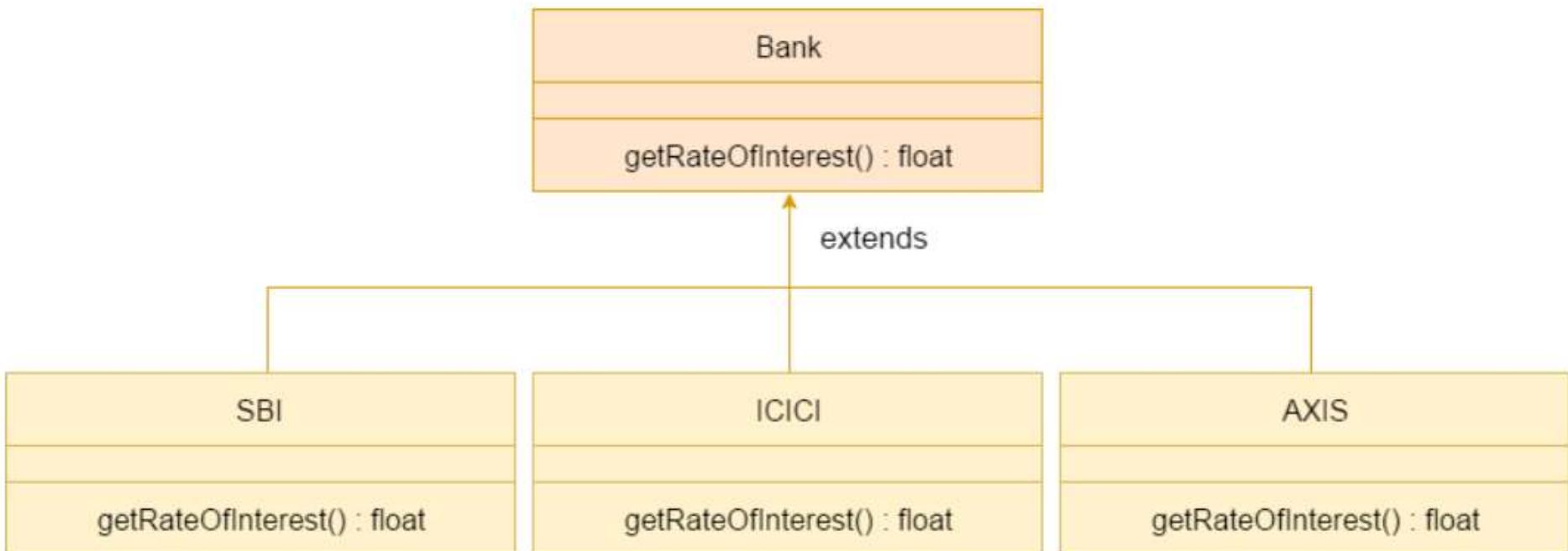
## □ Usage of Java Method Overriding

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



# Method Overriding...

- Try yourself !!
  - ▣ Add the method noLockersAvail() and override it in subclasses.



# Method Overriding...

## □ Rules:

- Static methods can not be overridden
- The methods declared as ‘final’ cannot be overridden
- Constructors cannot be overridden
- Overriding Method must have the same return type (or subtype)
- If lesser access in the subclass than that in the superclass, then we will get a compile-time error

# Access Modifiers

- To restrict the scope of a class, constructor, variable, method, or data member
- Four types of access modifiers in Java:
  - ▣ Default – No keyword required
  - ▣ Private
  - ▣ Protected
  - ▣ Public

# Access Modifiers...

- Access Level of each modifier:

Access Modifier	Access Level	Cannot be accessed from
Default	Only within the package	Outside the package
Private	Only within the class	Outside the class
Protected	Within the package and outside the package through child class	Outside the package without child class
Public	Everywhere (within the class, outside the class, within the package and outside the package)	-

# Access Modifiers...

Access Modifier	Within class	Within package subclass	Within package Non subclass	Outside package by subclass only	Outside package Non subclass
Private	Yes	No	No	No	No
Default	Yes	Yes	Yes	No	No
Protected	Yes	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes	Yes

Inheritance programs-1.doc

# Find outputs

- Inheritance programs-1.doc
  
- Ex-3 and Ex-4 shows use of final with Inheritance

# Abstract Class

- Abstraction
  - ▣ a process of **hiding the implementation details** and showing only functionality to the user
  - ▣ Focus on
    - What the object does
    - Not how it does
- Define a superclass that declares the structure of given abstraction
- Superclass only defines a generalized form shared by all subclasses- subclass will fill in the details
- Determines nature of methods that subclass **must implement**
- Superclass has no meaningful instructions

# Abstract Class...

- A class that is declared abstract
  - ▣ may or may not include abstract methods
  - ▣ must be declared with **an abstract keyword**
- Abstract classes **cannot be instantiated**
- Abstract classes **can be subclassed**
  - ▣ the subclass **usually provides implementations** for all of the abstract methods in its parent class
- It **can have abstract and non-abstract methods**
- It can have constructors and static methods also
- It can have final methods

# Abstract Class...

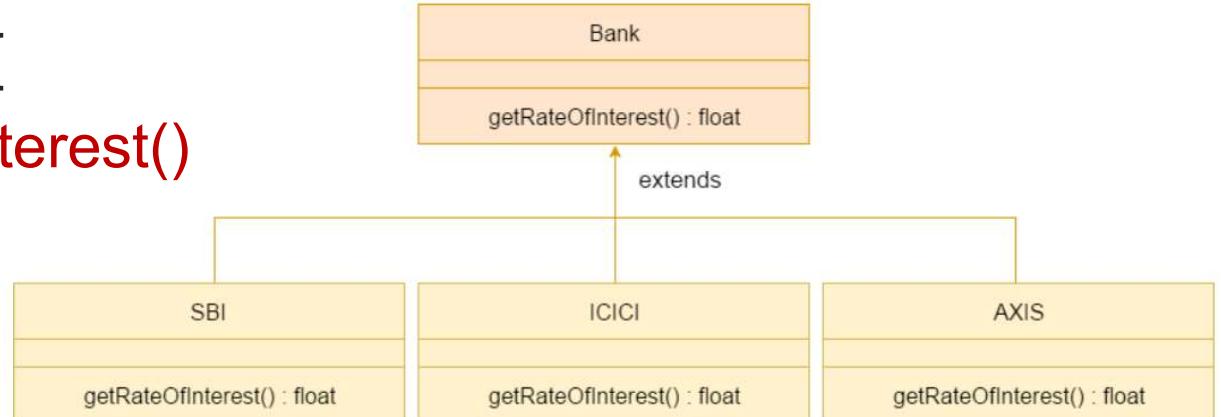
- Syntax of abstract class  
**abstract** class A{}
- Syntax of abstract method  
**abstract** void printData(); //no method body and abstract
- Rule:
  - If there is an abstract method in a class, that class must be abstract

# Abstract Class Example

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

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

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



# Dynamic Method Dispatch

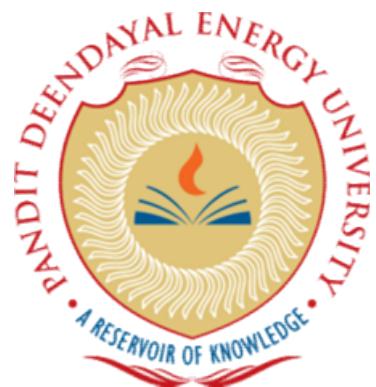
- One of the powerful concepts of Java
- Achieved through **method overriding**
- A **call to an overridden method** is resolved at **run time**
- Implements **run time polymorphism**
- Superclass reference can refer to a subclass object
- Determines which version of that method to execute based upon
  - ▣ **The type of the object being referred to during call**

# Example

□ Inheritance programs-1.doc

# Questions

- Difference between method Overloading and Method Overriding in java?
- What if constructor is made private?



# INTERFACES IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,  
School of Technology,  
Pandit Deendayal Energy University

# Outline

- What is an Interface?
- Contents and Uses
- Relationship between classes and interfaces
- Interface Declaration
- Multiple Inheritance
- Interfaces can be Extended
- Explore

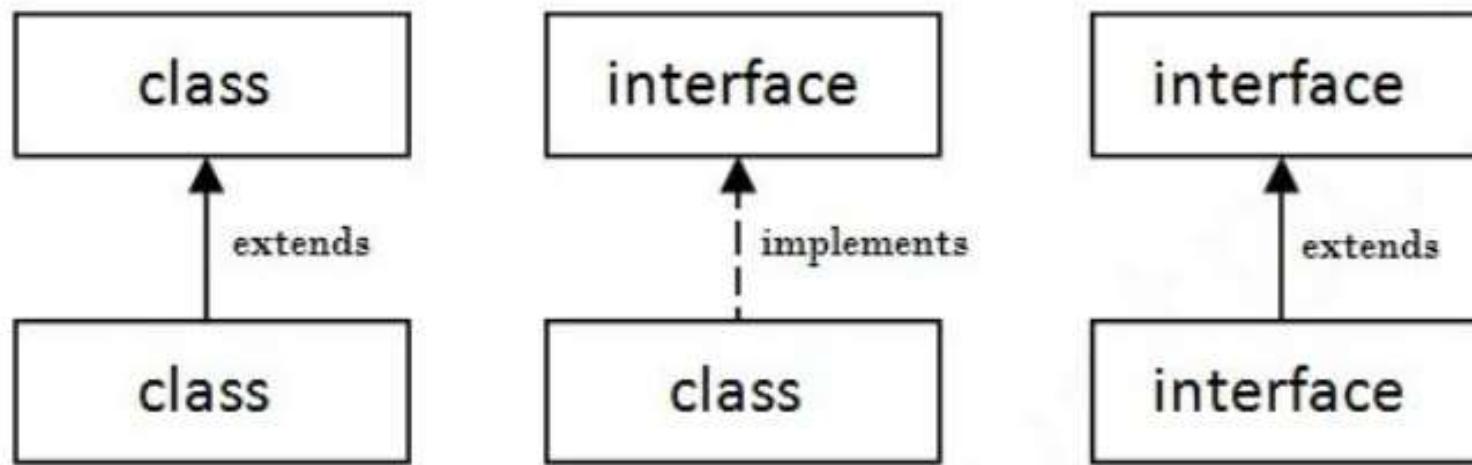
# What is an Interface?

- A blueprint/framework of a class
- A mechanism to achieve abstraction
- Specifies set of methods to be implemented by one or more classes
  - ▣ A class that includes an interface must implement all the methods with public access
  - ▣ A class that includes an interface may have its own methods
- Cannot have a method body
- Represents the IS-A relationship
- Cannot be instantiated

# Contents and Uses

- Interfaces **contain**
  - ▣ Variable (Public, static and final implicitly)- must be initialized
  - ▣ Abstract methods- implicitly public- may have parameters
  
- Interfaces are **used to**
  - ▣ Achieve abstraction
  - ▣ Multiple inheritance

# Relationship between classes and interfaces



- A class can implement more than one interfaces, but can inherit only a single superclass !!
- Any number of classes can implement interface.

# Interface Declaration

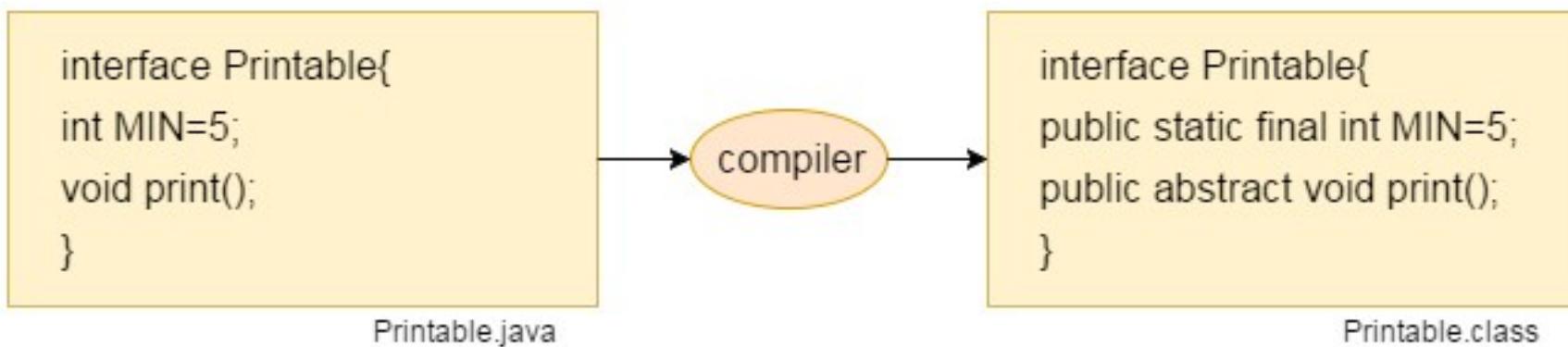
- Interface declaration syntax:

```
interface <interface_name>{  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

- Class that implements interface:

```
class A implements <interface_name>  
{  
    //override all abstract methods  
    //methods must be public  
}
```

# Interface Declaration...



# Interfaces can be Extended

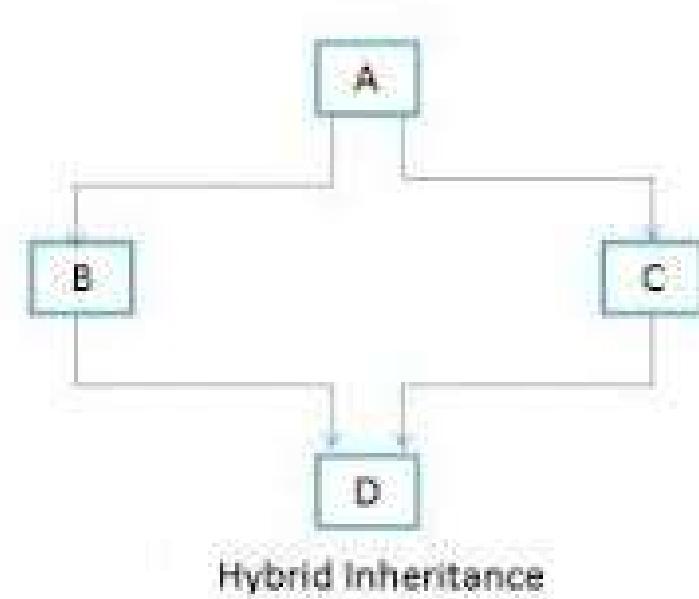
```
interface A {  
    void method1();  
    void method2();  
}  
  
interface B extends A {  
    void method3();  
}  
  
class C implements B { //Must implement all methods of A and B  
    void method1();  
    void method2();  
    void method3();  
}
```

# Multiple Inheritance

- Java class- ambiguity in multiple inheritance
- Use Interfaces for multiple inheritance
- EX:

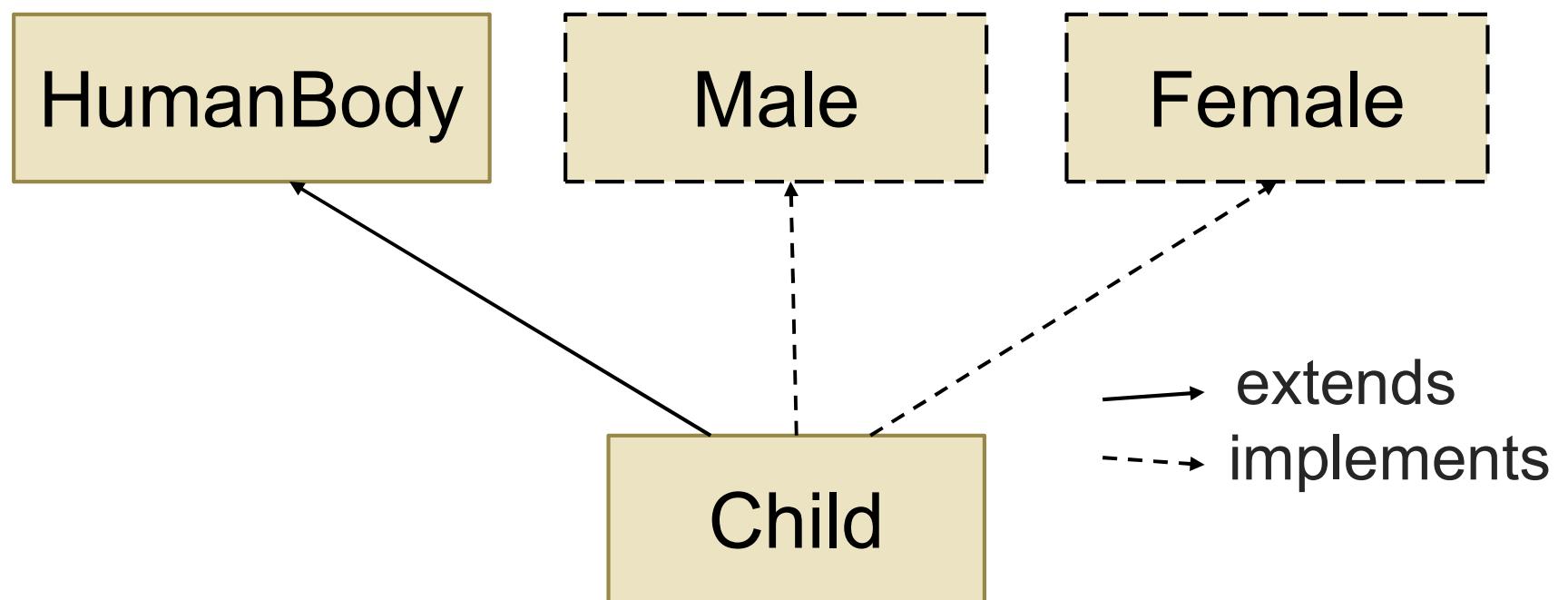
# Hybrid Inheritance

- The composition of two or more types of inheritance



# Hybrid Inheritance can be achieved

- ❑ Single and Multiple Inheritance (not supported but can be achieved through interface)



# Hybrid Inheritance can be achieved...

- Multilevel and Hierarchical Inheritance
- Hierarchical and Single Inheritance
- Multiple and Multilevel Inheritance

# Explore

- Static methods in an interface
- Private methods in an interface
- Default methods in an interface

# Find Output

(1)

```
class P {  
    int a = 30;  
    void display(){System.out.println("in P");}  
}  
  
class Q extends P {  
    int a = 50;  
    void display(){System.out.println("in Q");}  
}
```

# Find Output (1 continue)...

```
public class Mavenproject1 extends Q {  
    public static void main(String[] args) {  
        Q q = new Q();  
        System.out.println(" Value of a: " +q.a);  
        q.display();  
        P p = new Q();  
        System.out.println("Value of a: " +p.a);  
        p.display();  
    }  
}
```

# Java Collections API

# Collections in Java

- A framework that provides an architecture to store and manipulate the group of objects
- Operations on data:
  - Searching
  - Sorting
  - Insertion
  - Manipulation
  - Deletion

# API :

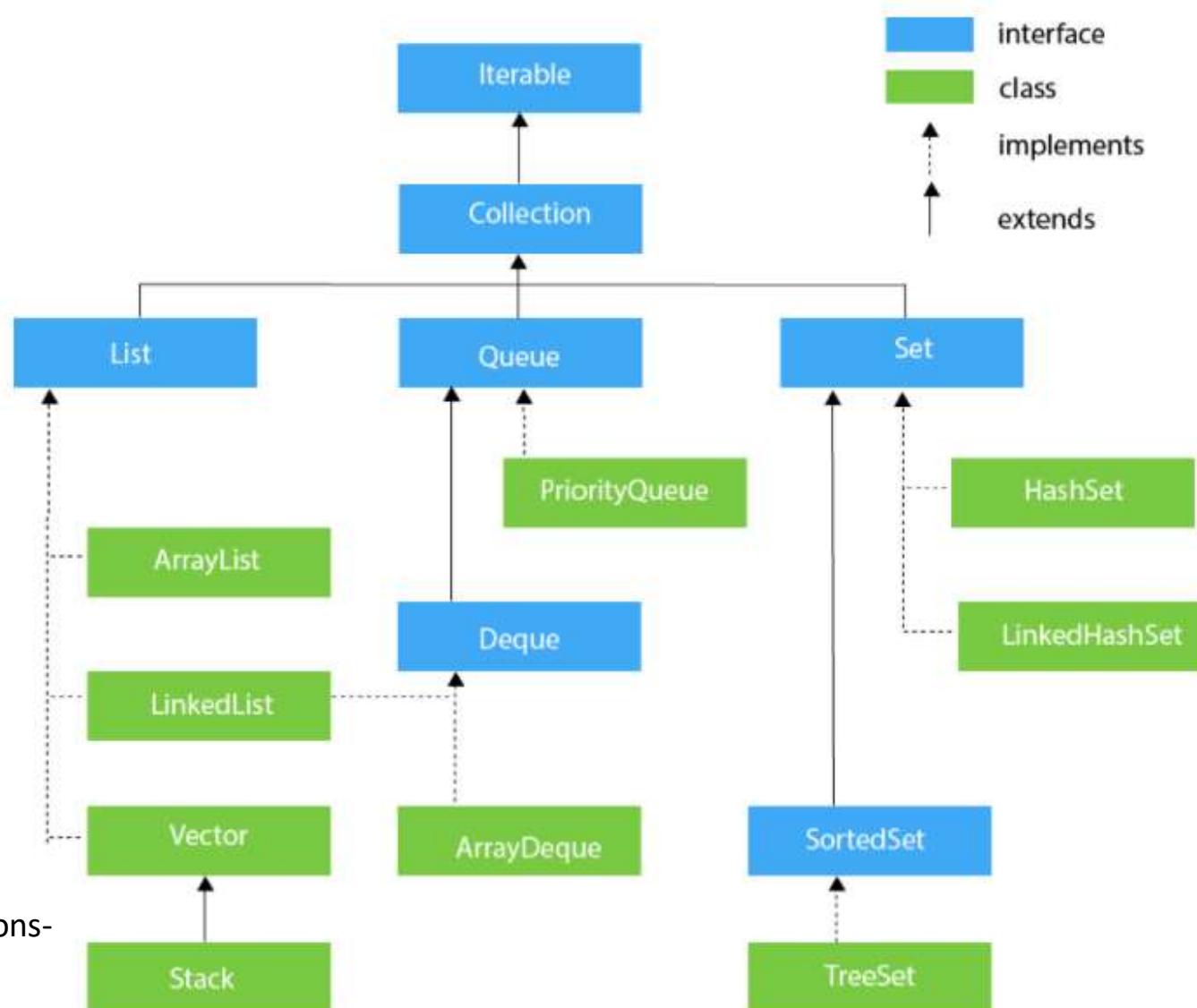


Image source :

<https://www.javatpoint.com/collections-in-java>

## 1. Iterable Interface:

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

## 2. Iterator Interface:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

### 3. Iterable <- Collection <- List <- ArrayList

```
class TestJavaCollection1{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();
        list.add("Name1");//Adding object in arraylist
        list.add("Name2");
        list.add("Name3");
        list.add("Name4");

        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

# Difference between List, Queue and Set

List	Queue	Set
Ordered Collection	Ordered Collection	Unordered Collection
Can have duplicate elements	Can have duplicate elements	Can not have duplicate elements
Add, remove, or update any element directly	FIFO structure	Add, or remove elements but there is no order.

# Iterator Interface

- The object of class which has implemented Iterator interface must have to implement its 3 methods to traverse a collection.

Method	Description
public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
public Object next()	It returns the element and moves the cursor pointer to the next element.
public void remove()	It removes the last elements returned by the iterator.

# ArrayList

- It uses a *dynamic array* for storing the elements.
- Same as an array, but there is *no size limit*.
- Elements can be added or removed anytime. So, it is much more flexible than the traditional array.
- It is in *java.util* package.
- creating old non-generic arraylist : `ArrayList list=new ArrayList();`
- creating new generic arraylist : `ArrayList<String> list=new ArrayList<String>();`

# LinkedList

- Java LinkedList class uses a doubly linked list to store the elements.
- Java LinkedList class can be used as a list, stack or queue.
- Creation of object is similar to ArrayList.

# Vector

- Same as ArrayList.
- But Vector is synchronized, ArrayList is non-synchronized.

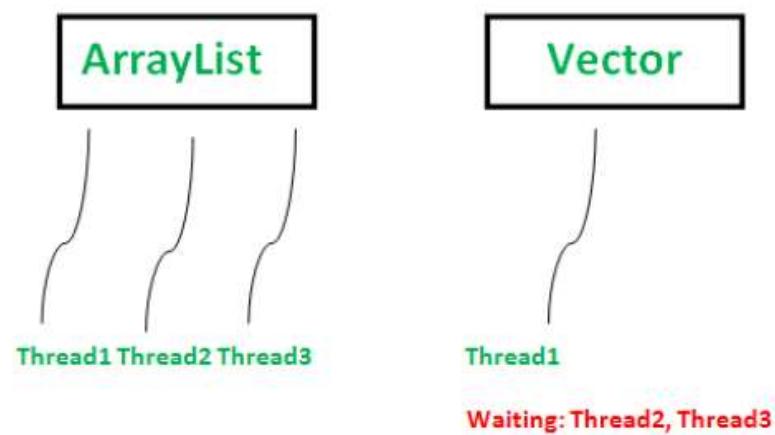


Image Source : <https://www.geeksforgeeks.org/vector-vs-arraylist-java/>

# Difference between ArrayList and Vector

ArrayList	Vector
1) ArrayList is <b>not synchronized</b> .	Vector is <b>synchronized</b> .
2) ArrayList <b>increments 50%</b> of current array size if the number of elements exceeds from its capacity.	Vector <b>increments 100%</b> means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is <b>not a legacy class</b> . It is introduced in JDK 1.2.	Vector is a <b>legacy class</b> .
4) ArrayList is <b>fast</b> because it is non-synchronized.	Vector is <b>slow</b> because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.

# Stack

- linear data structure.
- It is based on **Last-In-First-Out (LIFO)**.
- It has two main methods : push and pop

# PriorityQueue

- Provides the facility of using queue. But it does not orders the elements in FIFO manner.
- The elements are ordered as per their priority.
- In Java to implement priority queue, you require to implement comparable interface.

# Directory

- An abstract class of Java.util package.
- Facilitates to store key-value pairs.

# UNIT 3 – I/O PROGRAMMING

# Java I/O streams

- 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.

# Java IO streams

## Stream

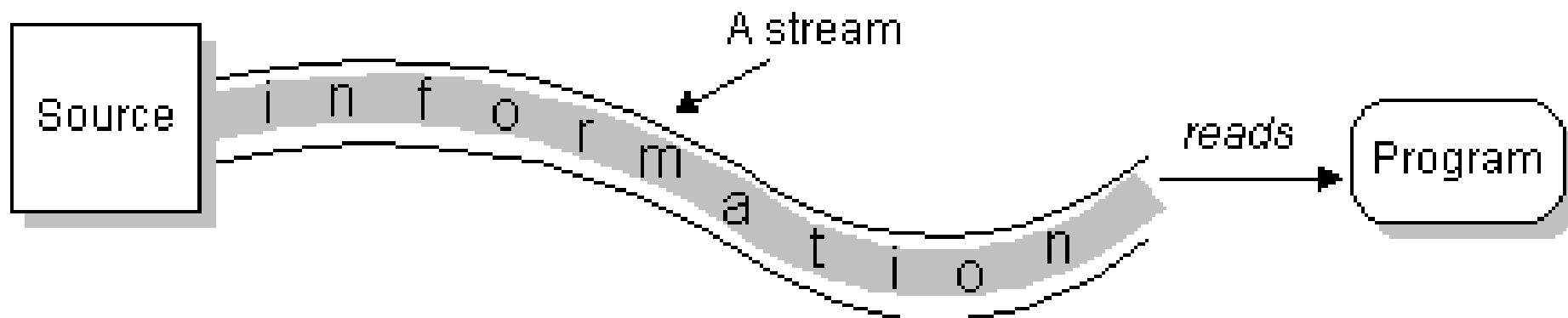
- A stream is a sequence of data.
- In Java, a stream is composed of bytes.
- It's called a stream because it is like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
  1. System.out: standard output stream
  2. System.in: standard input stream
  3. System.err: standard error stream

# Java IO streams

```
int i=System.in.read(); //returns ASCII code of 1st character  
System.out.println((char)i); //will print the character  
System.out.println("simple message"); //print output message to  
console  
System.err.println("error message"); //print an error message to the  
console
```

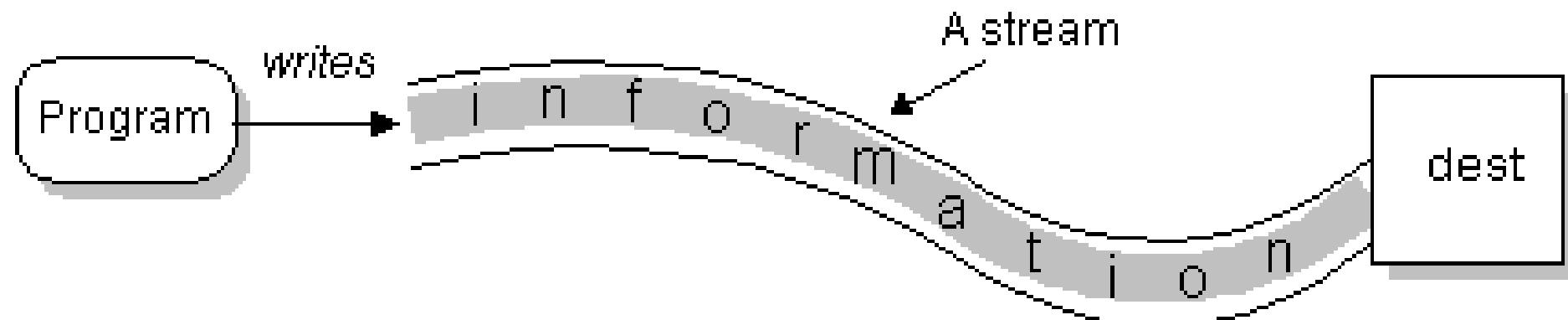
# Java IO streams

- To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:



# Java IO streams

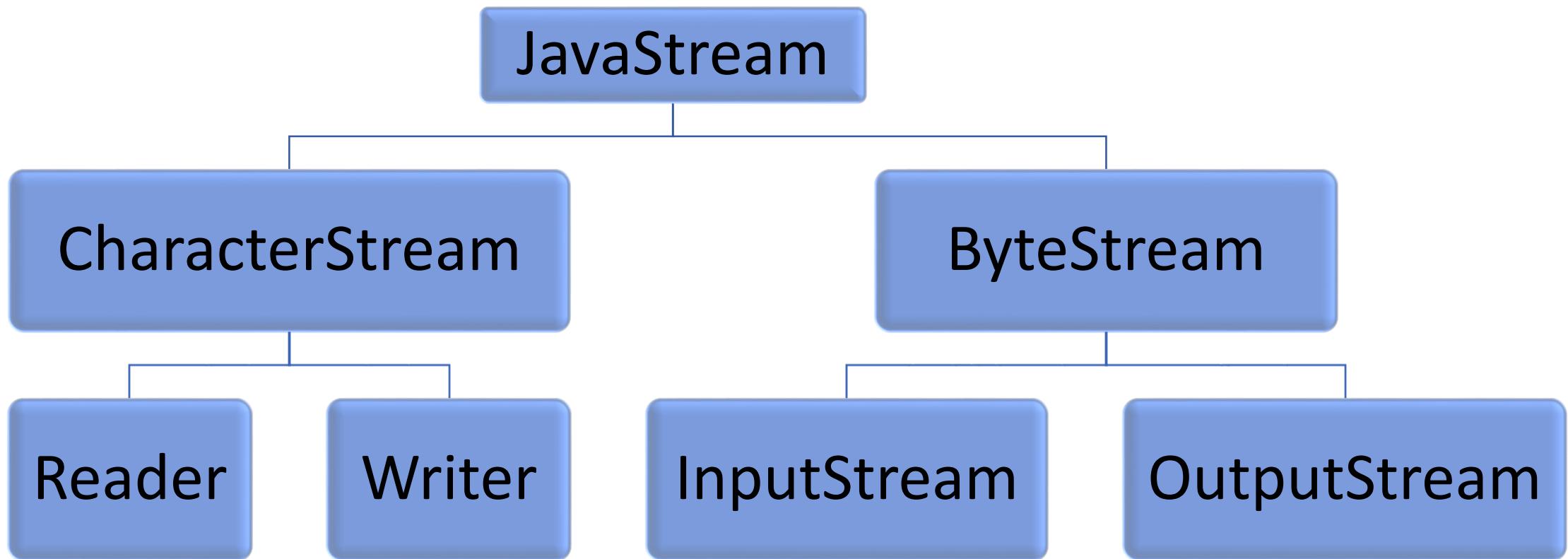
- Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, like this:



# Java IO streams

- No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:
- Reading
  - open a stream
  - while more information
    - read information
  - close the stream
- Writing
  - open a stream
  - while more information
    - write information
  - close the stream

# Java IO streams



# Character and Binary streams

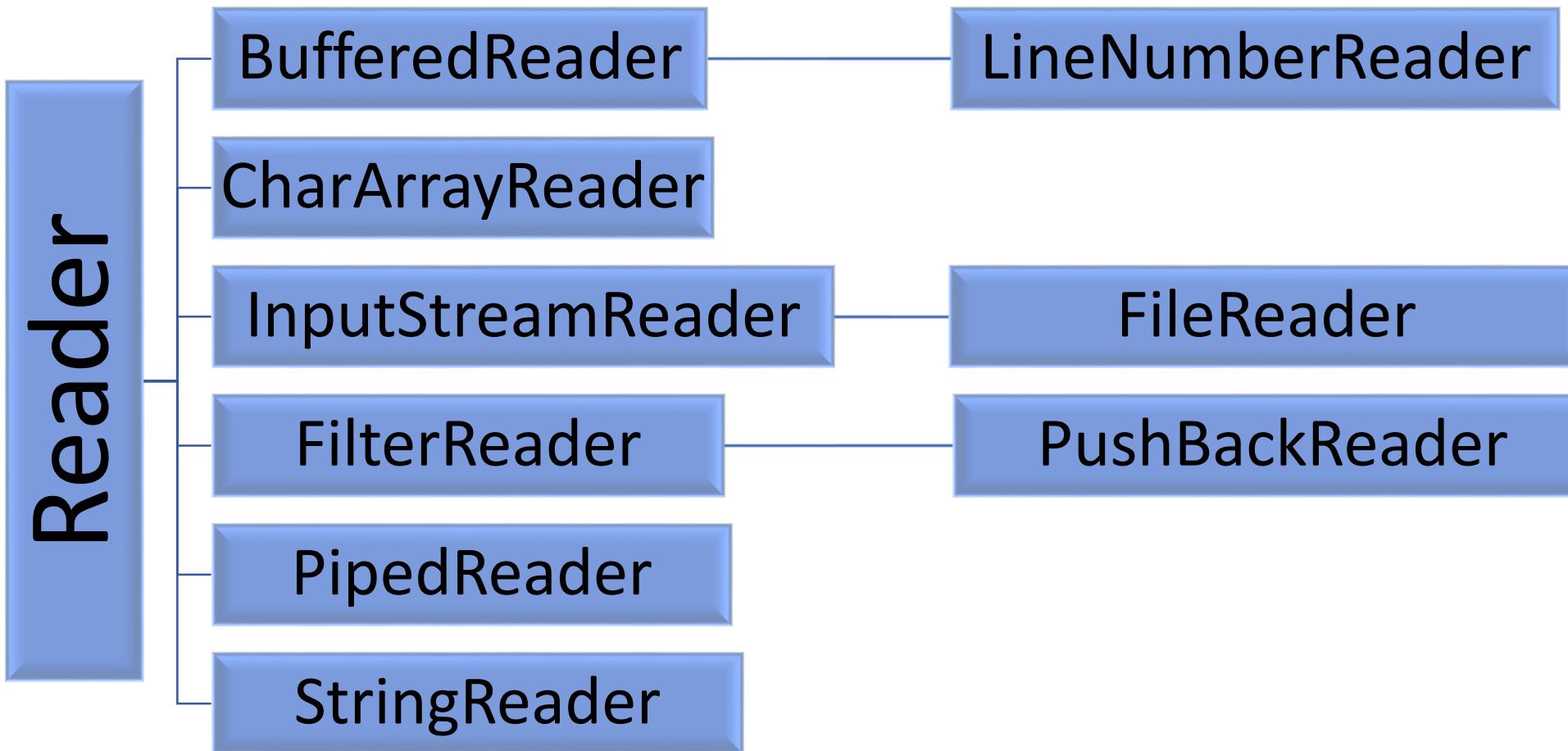
# Character Stream Classes

- Character stream can be used to read and write 16-bit Unicode characters.
- Reader and Writer are the abstract superclasses for character streams in `java.io`.
- Most programs should use readers and writers to read and write textual information. The reason is that they can handle any character in the Unicode character set, whereas the byte streams are limited to ISO-Latin-1 8-bit bytes.

# CharacterStream: Reader

- Reader provides the API and partial implementation for readers-- streams that read 16-bit characters
- Reader stream classes are designed to read character from the files.
- The reader class contains methods that are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implements by the input stream classes.
- The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods to provide higher efficiency, additional functionality, or both.

# CharacterStream: Reader Hierarchy



# CharacterStream: Reader

- Useful methods of InputStream

Modifier and Type	Method	Description
abstract void	close()	It closes the stream and releases any system resources associated with it.
void	mark(int readAheadLimit)	It marks the present position in the stream.
int	read()	It reads a single character.
abstract int	read(char[] cbuf, int off, int len)	It reads characters into a portion of an array.
boolean	ready()	It tells whether this stream is ready to be read.
void	reset()	It resets the stream.
long	skip(long n)	It skips characters.

# CharacterStream: Reader

```
import java.io.*;  
public class ReaderExample {  
    public static void main(String[] args) {  
        try {  
            Reader reader = new FileReader("file.txt");  
            int data = reader.read();  
            while (data != -1) {  
                System.out.print((char) data);  
                data = reader.read();  
            }  
            reader.close();  
        } catch (Exception ex) {  
            System.out.println(ex.getMessage());  
        }  
    }  
}
```

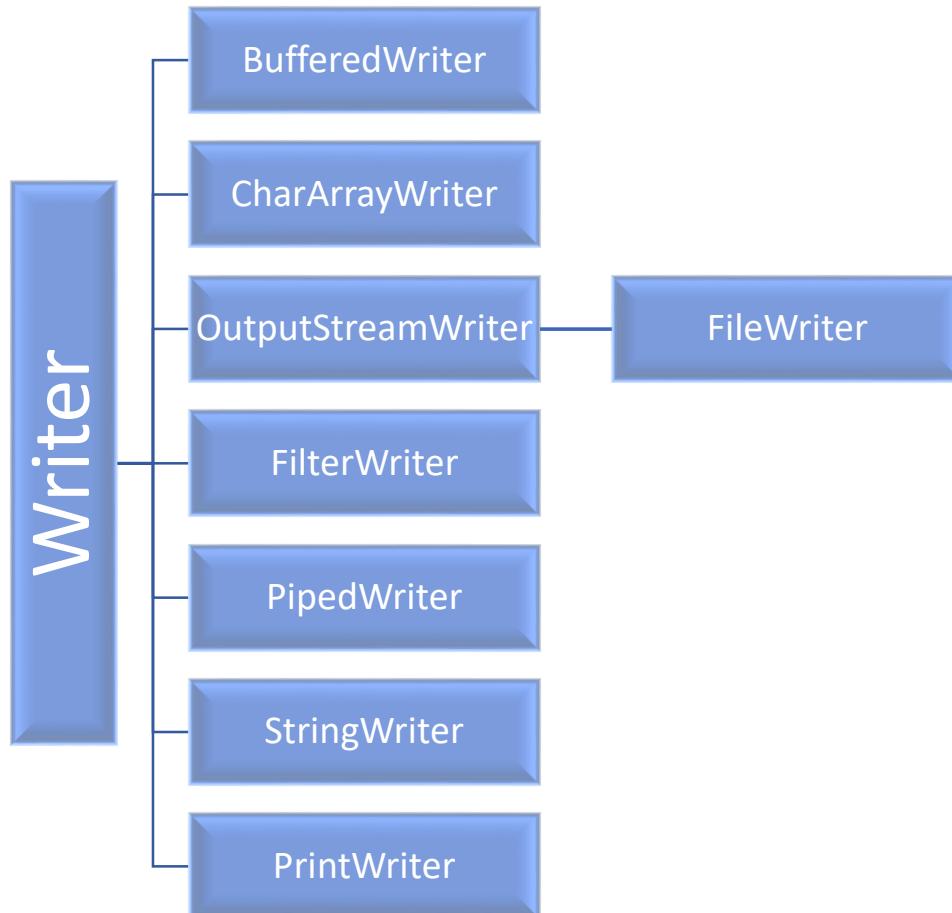
***file.txt***  
***I love my country***

***Output:***  
***I love my country***

# CharacterStream: Writer

- Writer provides the API and partial implementation for writers-- streams that write 16-bit characters.
- It is an abstract class for writing to character streams.
- The methods that a subclass must implement are write(char[], int, int), flush(), and close().
- Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

# CharacterStream: Writer Hierarchy



# CharacterStream: Writer

- Useful methods of InputStream

Modifier and Type	Method	Description
Writer	append(char c)	It appends the specified character to this writer.
Writer	append(CharSequence csq)	It appends the specified character sequence to this writer
Writer	append(CharSequence csq, int start, int end)	It appends a subsequence of the specified character sequence to this writer.
abstract void	close()	It closes the stream, flushing it first.
abstract void	flush()	It flushes the stream.
void	write(int c)	It writes a single character.
void	write(String str, int off, int len)	It writes a portion of a string.

# CharacterStream: Writer

```
import java.io.*;  
public class WriterExample {  
    public static void main(String[] args) {  
        try {  
            Writer w = new FileWriter("output.txt");  
            String content = "I love my country";  
            w.write(content);  
            w.close();  
            System.out.println("Done");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

***Output:  
Done***

***Output.txt  
I love my country***

# Byte Stream Classes

- Byte Stream Classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes.
- Since the streams are unidirectional, they can transmit bytes in only one direction and, therefore, java Provides two kinds of byte stream classes:
  - Input Stream Classes
  - Output Stream Classes

# ByteStream: InputStream

- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.
- Input stream classes are used to read 8-bit bytes.

## InputStream class

- InputStream class is an abstract class. Therefore, we can not create instances of this class rather we must use the subclasses that inherits from this class.
- It is the superclass of all classes representing an input stream of bytes.

# ByteStream: InputStream

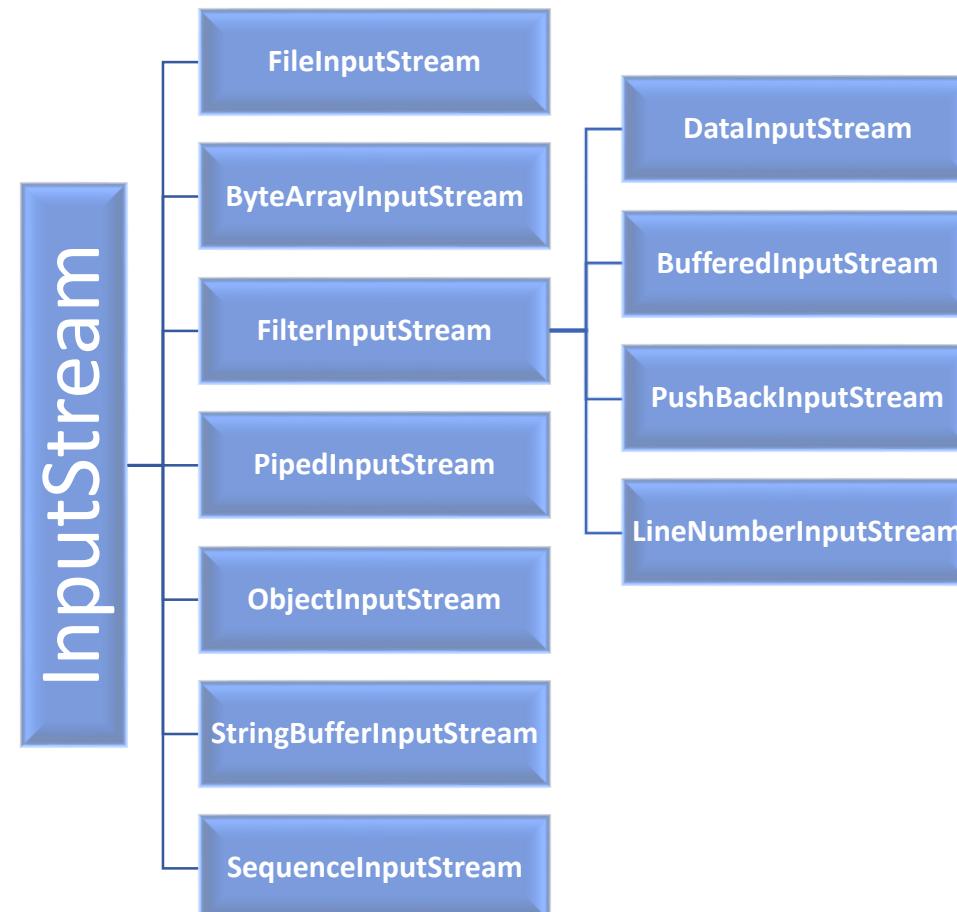
- `InputStream` class defines methods for performing input functions such as:
  - Reading Bytes
  - Closing streams
  - Marking positions in streams
  - Skipping ahead in a stream
  - Finding the number of bytes in a stream

# ByteStream: InputStream

- Useful methods of InputStream

Method	Description
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.
public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
public int skip(n)throws IOException	Skips over n bytes from the input stream
public int reset()throws IOException	Goes back to the beginning of the stream
public void close()throws IOException	is used to close the current input stream.

# ByteStream: InputStream Hierarchy



*Note: DataInputStream extends FilterInputStream and implements interface DataInput. Therefore, the DataInputStream implements the methods described in DataInput in addition to using the methods of InputStream*

# ByteStream: OutputStream

- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

## OutputStream class

- OutputStream class is an abstract class and therefore we cannot instantiate it.
- 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.

# ByteStream: OutputStream

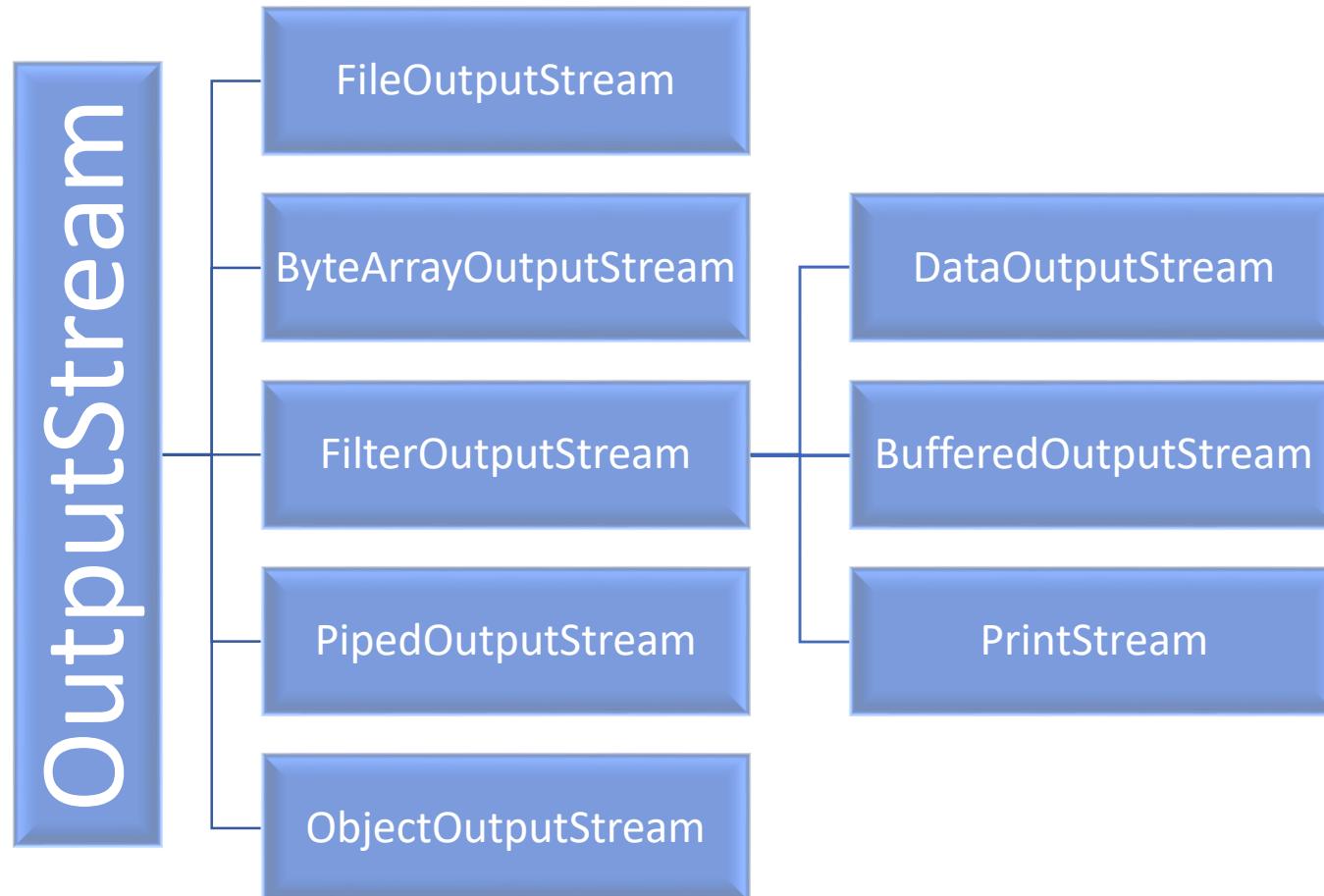
- OutputStream class defines methods for performing input functions such as:
  - Writing Bytes
  - Closing streams
  - Flushing streams

# ByteStream: OutputStream

- Useful methods of OutputStream

Method	Description
public void write(int) throws IOException	is used to write a byte to the current output stream.
public void write(byte[]) throws IOException	is used to write an array of byte to the current output stream.
public void flush() throws IOException	flushes the current output stream.
public void close() throws IOException	is used to close the current output stream.

# ByteStream: OutputStream Hierarchy



*Note: DataOutputStream implements interface DataOutput and, therefore, implements the methods described in DataOutput interface.*

Reading data from and writing  
data to files

# Reading data from files

You can read files using these classes:

- **FileReader class (from CharacterStream)** 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.
- **FileInputStream class (from ByteStream)** 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.

# Reading data from files: FileReader

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

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.

# Reading data from files: FileReader

```
import java.io.FileReader;  
public class FileReaderExample {  
    public static void main(String args[])throws Exception{  
        FileReader fr=new FileReader("D:\\file.txt");  
        int i;  
        while((i=fr.read())!=-1)  
            System.out.print((char)i);  
        fr.close();  
    }  
}
```

*file.txt*  
*I love my country*

*Output:*  
*I love my country*

# Reading data from files: FileInputStream

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 FileDescriptor 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 close the stream.

# Reading data from files: FileInputStream

```
// example to read single character  
import java.io.FileInputStream;  
public class DataStreamExample {  
    public static void main(String args[]){  
        try{  
            FileInputStream fin=new FileInputStream("D:\\file.txt");  
            int i=fin.read();  
            System.out.print((char)i);  
  
            fin.close();  
        }catch(Exception e){System.out.println(e);}  
    }  
}
```



# Reading data from files: FileInputStream

```
// example to read all characters
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("D:\\file.txt"); //Exception-1
            int i=0;
            while((i=fin.read())!=-1){ //Exception-2
                System.out.print((char)i); //Exception-2
            }
            fin.close(); //Exception-3
        }catch(Exception e){System.out.println(e);}
    }
}
```

*file.txt*  
*I love my country*

*Output:*  
*I love my country*

# Writing data to files

You can write files using these classes:

- **FileWriter class (from CharacterStream)** is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.
- **FileOutputStream class (from ByteStream)** 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.

# Writing data to files: FileWriter

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.
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.

# FileWriter

```
import java.io.FileWriter;
public class FileWriterExample {
    public static void main(String args[]){
        try{
            FileWriter fw=new FileWriter("D:\\file.txt");
            fw.write("Welcome to Java");
            fw.close();
        }catch(Exception e){
            System.out.println(e);
            System.out.println("Success...");
        }
    }
}
```

***Output:***

***Success...***

***file.txt***

***Welcome to Java***

# Writing data to files: FileOutputStream

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 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.

# Writing data to files: FileOutputStream

```
//example to write byte  
import java.io.FileOutputStream;  
public class FileOutputStreamExample {  
    public static void main(String args[]){  
        try{  
            FileOutputStream fout=new FileOutputStream("D:\\file.txt");  
            fout.write(65);  
            fout.close();  
            System.out.println("success...");  
        }catch(Exception e){System.out.println(e);}  
    }  
}
```

*Output:*  
*Success...*

*file.txt*  
**A**

# Writing data to files: FileOutputStream

```
//example to write string  
import java.io.FileOutputStream;  
public class FileOutputStreamExample {  
    public static void main(String args[]){  
        try{  
            FileOutputStream fout=new FileOutputStream("D:\\file.txt");  
            String s="Welcome to Java";  
            byte b[]={s.getBytes()};//converting string into byte array  
            fout.write(b);  
            fout.close();  
            System.out.println("success...");  
        }catch(Exception e){System.out.println(e);}  
    }  
}
```

***Output:  
Success...***

***file.txt  
Welcome to Java***

# Reading data from files: BufferedReader

- Data can be read line by line using BufferedReader class
  - to read the text from a character-based input stream
- It uses buffer-makes the performance fast
- It inherits **Reader class**
- requires reader object as argument

# Java BufferedReader class constructors

Constructor	Description
BufferedReader( Reader rd)	It is used to create a buffered character input stream that uses the default size for an input buffer.
BufferedReader( Reader rd, int size)	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 array.
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 stream 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.

# Java BufferedReader Example

```
import java.io.*;
public class BufferedReaderExample {
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        BufferedReader br=new BufferedReader(fr);

        int i;
        while((i=br.read())!=-1){
            System.out.print((char)i);
        }
        br.close(); //sequence is reverse
        fr.close();
    }
}
```

# Writing data into files: BufferedWriter

- Used to provide buffering for Writer instances
- It makes the performance fast
- It inherits **Writer class**

# Java BufferedWriter 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.

# Java BufferedWriter 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

# Java BufferedWriter Example

```
import java.io.*;
public class BufferedWriterExample {
    public static void main(String[] args) throws Exception {
        FileWriter writer = new FileWriter("D:\\testout.txt");
        BufferedWriter buffer = new BufferedWriter(writer);
        buffer.write("Welcome to JavaTpoint.");
        buffer.close();
        System.out.println("Success");
    }
}
```

# Explore

- Reading data from console by InputStreamReader and BufferedReader

# Serializable interface

- Serialization:
- To convert the state of an object into a byte stream –stores metadata in binary form
- done using ObjectOutputStream
- a mechanism of converting the state of an object into a byte stream
- Deserialization:
- The reverse process of serialization
- Byte stream is used to recreate the actual Java object in memory

# Serializable interface

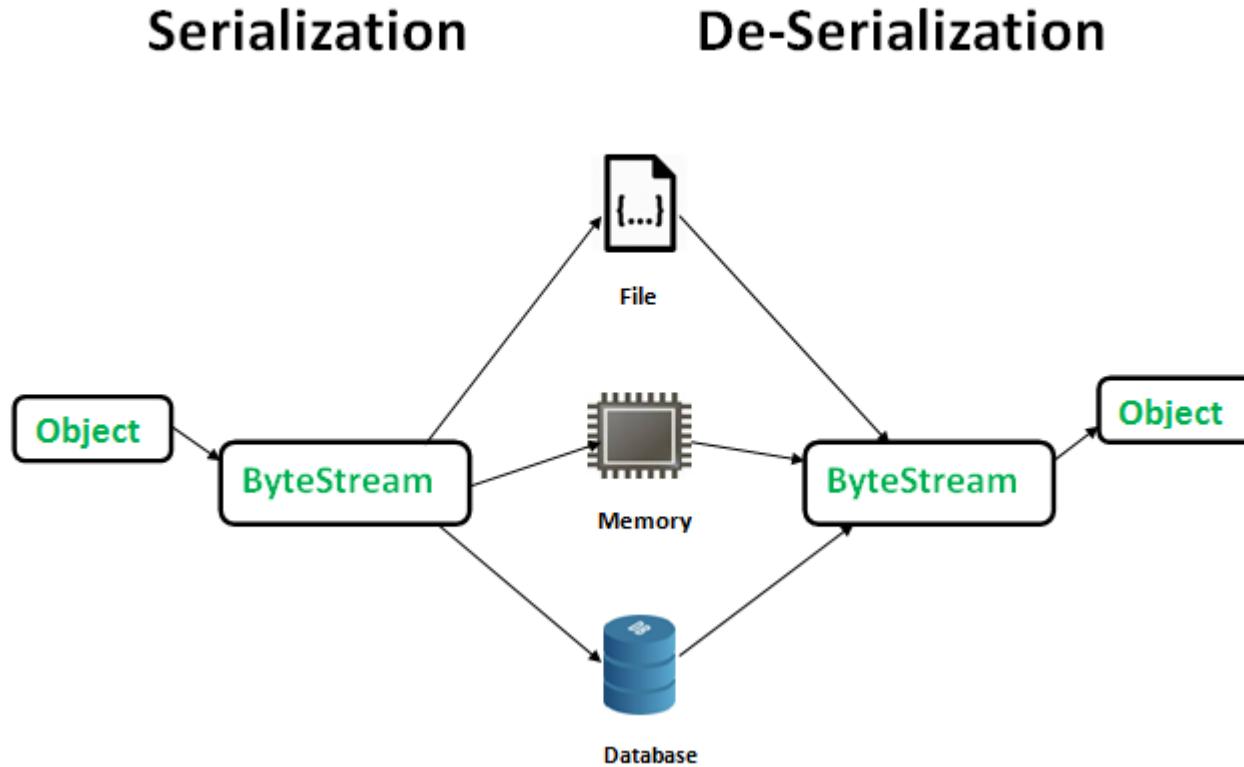


Image source: <https://www.geeksforgeeks.org/serializable-interface-in-java/>

# Reading data from and Writing data to files on random Positions

## RandomAccessFile

- This class is used for reading and writing to random access file.
- A random access file behaves like a large array of bytes.
- There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations.
- If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

# Reading data from and Writing data to files

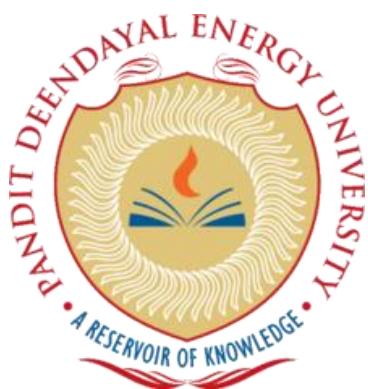
```
import java.io.IOException;
import java.io.RandomAccessFile;
public class RandomAccessFileExample {
    static final String FILEPATH
="myFile.TXT";
    public static void main(String[] args) {
        try {
            System.out.println(new
String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I love my
country and my people", 31);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
private static byte[] readFromFile(String
filePath, int position, int size)
throws IOException {
    RandomAccessFile file = new
RandomAccessFile(filePath, "r");
    file.seek(position);
    byte[] bytes = new byte[size];
    file.read(bytes);
    file.close();
    return bytes;
}
```

```
private static void writeToFile(String filePath,
String data, int position)
throws IOException {
    RandomAccessFile file = new
RandomAccessFile(filePath, "rw");
    file.seek(position);
    file.write(data.getBytes());
    file.close();
}
```

**Output:**

*The myFile.TXT contains text "This class is used for reading and writing to random access file."  
after running the program it will contains  
This class is used for reading I love my country and my people.*



# EXCEPTION HANDLING IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# Outline

- Error
- Exception
- Error vs exception
- Exception handling in Java
- Exception handling hierarchy
- Types of Exception handling
- Keywords
- Syntax
- Multiple catch clauses
- Nested try statements
- Methods generating exceptions
- Custom exceptions

# Error

- “Error” is a **critical condition** that cannot be handled by the code of the program.
- An Error “indicates serious problems that a reasonable application should not try to catch.”
- Errors are the conditions which **cannot get recovered** by any handling techniques.
- It surely cause termination of the program abnormally.
- Errors belong to unchecked type and mostly occur at runtime.
- Some of the **examples** of errors are Out of memory error or a System crash error.

# Error Example

```
class StackOverflow {  
    public static void test(int i) {  
        // No correct as base condition leads to non-stop  
        recursion.  
        if (i == 0) return;  
        else test(i++);  
    } }  
public class ErrorEg {  
    public static void main(String[] args) {  
        StackOverflow.test(5); // eg of StackOverflowError  
    }  
}
```

**Output:**

```
Exception in thread "main"  
java.lang.StackOverflowError  
at StackOverflow.test(ErrorEg.java:7)  
at StackOverflow.test(ErrorEg.java:7)  
at StackOverflow.test(ErrorEg.java:7)  
at StackOverflow.test(ErrorEg.java:7)  
at StackOverflow.test(ErrorEg.java:7)
```

# Exception

- An Exception “indicates conditions that a reasonable application **might want to catch** (handled by the code of the program).”
- Exceptions are the **conditions that occur at runtime** and may cause the termination of program. But they are recoverable using try, catch and throw keywords.
- Exceptions are divided into two categories : checked and unchecked exceptions.
- Checked exceptions like IOException known to the compiler at compile time while unchecked exceptions like ArrayIndexOutOfBoundsException known to the compiler at runtime.
- It is mostly caused by the program written by the programmer (due to bad data provided by user).

# Error Vs Exception

BASIS	ERROR	EXCEPTION
Basic	An error is caused due to lack of system resources.	An exception is caused because of the code.
Recovery	An error is irrecoverable.	An exception is recoverable.
Keywords	There is no means to handle an error by the program code.	Exceptions are handled using three keywords "try", "catch", and "throw".
Consequences	As the error is detected the program will terminated abnormally.	As an exception is detected, it is thrown and caught by the "throw" and "catch" keywords correspondingly.
Types	Errors are classified as unchecked type.	Exceptions are classified as checked or unchecked type.
Package	In Java, errors are defined "java.lang.Error" package.	In Java, an exceptions are defined in "java.lang.Exception".
Example	OutOfMemory, StackOverFlow.	Checked Exceptions : NoSuchMethod, ClassNotFoundException. Unchecked Exceptions : NullPointerException, IndexOutOfBoundsException.

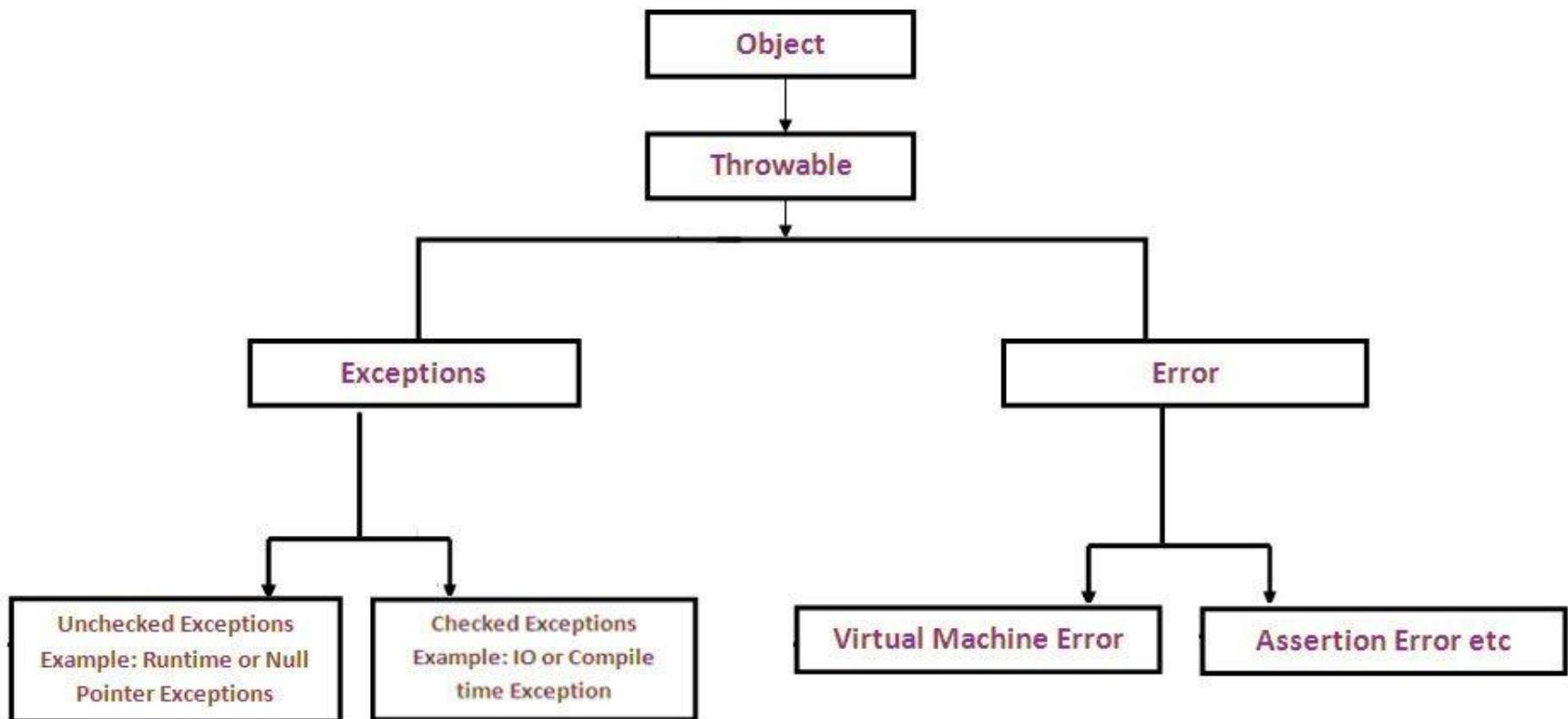
# Exception handling in Java

- An Exception is an **unwanted event** that interrupts the normal flow of the program.
- System's exception handler
  - ▣ a system generated **error message** is shown to the user
  - ▣ program **execution gets terminated**
- The Exception Handling in Java is one of the powerful mechanism
  - ▣ to handle the runtime errors
  - ▣ While **maintaining normal flow of the application**
  - ▣ provide a **meaningful message (user friendly warning)** to the user about the issue rather than a system generated message-easy to understand
  - ▣ Let the users correct the error

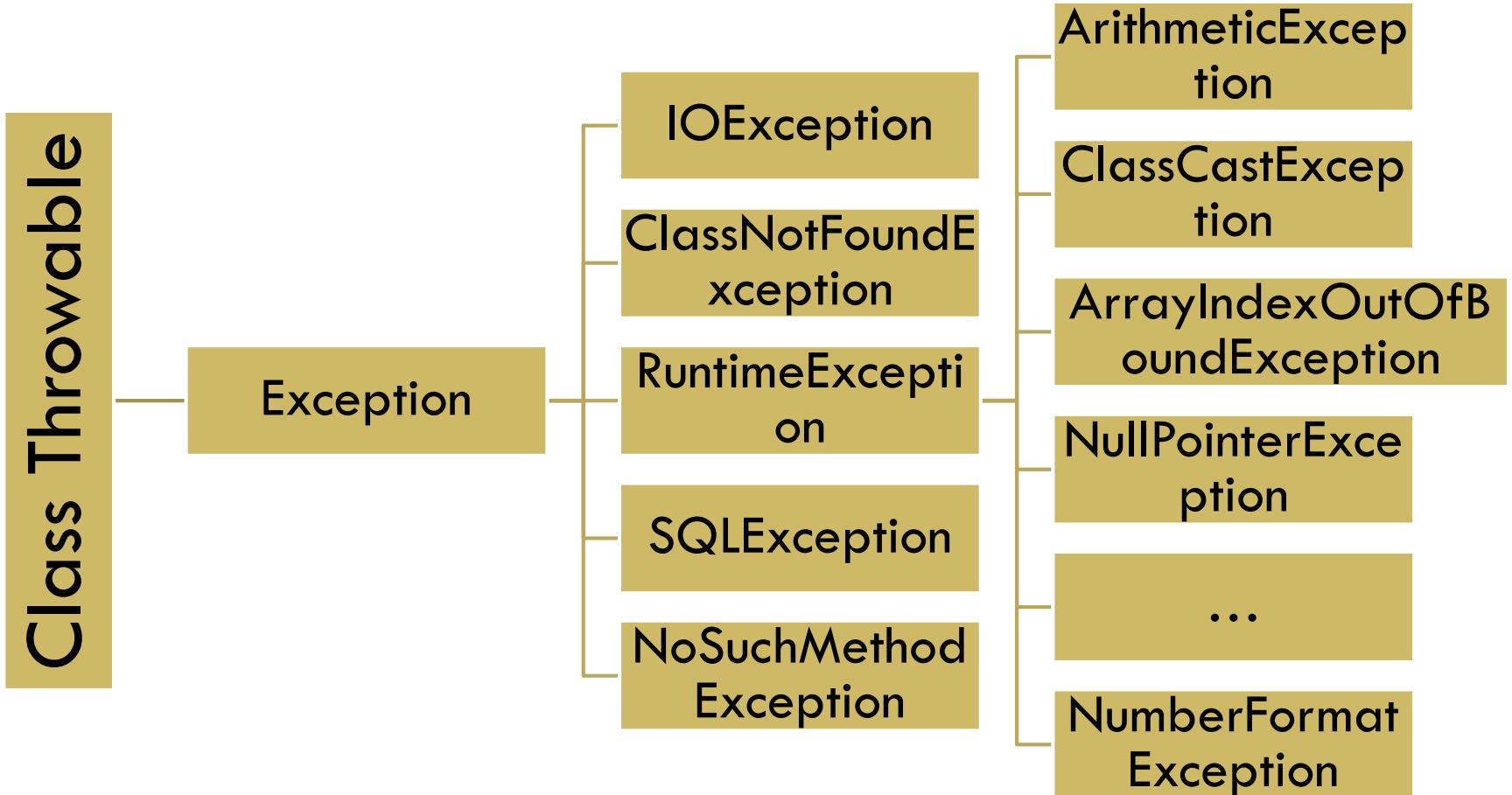
# Exception handling- Advantages

- Ensures that the **flow of the program doesn't break when** an exception occurs.
- Custom exceptions can be defined in applications

# Exception Handling Hierarchy



# Exception Handling Hierarchy...



# Types of 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:
  - ▣ Checked Exception
  - ▣ Unchecked Exception
  - ▣ Error

# Checked Exceptions

- 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**.

# 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**.

# Keywords in Exception handling

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.

# Exception handling in Java: Syntax

```
try {  
    // block of code  
}  
catch ( ExceptionType1 e) {  
    // Exception handling routine for ExceptionType1 (optional)  
}  
catch (ExceptionType2 e ) {  
    // Exception handling routine for ExceptionType2 (optional)  
}  
..  
catch (ExceptionType_n e) {  
    // Exception handling routine for ExceptionType_n (optional)  
}  
finally {  
    // Program code of exit (optional)    }  
}
```

# Exceptions

```
import java.util.Random;
public class Mavenproject1{
    public static void main(String[] args)  {
        int a = 0, b = 0;
        Random r=new Random();
        for(int i=0;i<100;i++){
            b=r.nextInt(10);
            System.out.println("b= "+b);
            a=12345/b;
            System.out.println("a= "+a); // will not be printed if exception raised
        }
    }
}
```

# Output

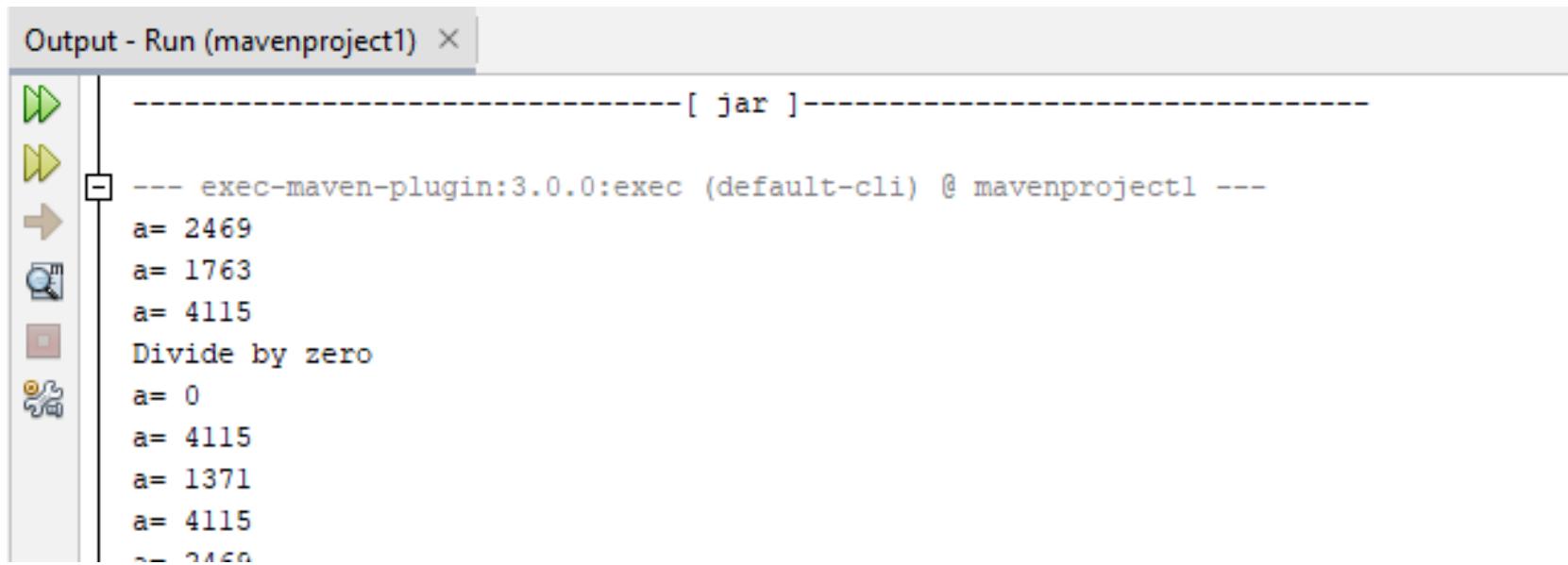
Output - Run (mavenproject1) ×

The screenshot shows the 'Output' window from an IDE, specifically for a Maven project named 'mavenproject1'. The window title is 'Output - Run (mavenproject1)'. The content is organized into several sections:

- Building mavenproject1 1.0-SNAPSHOT**: A tree view showing the build process. It includes a 'jar' section and an 'exec-maven-plugin' section.
- [ jar ]**: Shows the creation of a jar file.
- exec-maven-plugin:3.0.0:exec (default-cli) @ mavenproject1 ---**: The execution phase of the Maven build.
- b= 7**, **a= 1763**, **b= 0**: Variable assignments during the execution phase.
- Exception in thread "main" java.lang.ArithmetricException: / by zero**: A Java exception message.
- at com.mycompany.mavenproject1.Mavenproject1.main(Mavenproject1.java:20)**: The stack trace of the exception, pointing to line 20 of the main method in the Mavenproject1 class.
- Command execution failed.**: An error message indicating the failure of the command execution.
- org.apache.commons.exec.ExecuteException: Process exited with an error: 1 (Exit value: 1)**: The final error message from the Apache Commons Exec library.

```
import java.util.Random;
public class Mavenproject1{
    public static void main(String[] args)  {
        int a = 0, b = 0;
        Random r=new Random();
        for(int i=0;i<100;i++){
            try{
                b=r.nextInt(10);
                a=12345/b;
            }
            catch(ArithmeticException e){
                System.out.println("Divide by zero");
                a=0;
            }
            System.out.println("a= "+a);          } }}}
```

# Output



The screenshot shows the Eclipse IDE's Output view for a Maven project named "mavenproject1". The title bar reads "Output - Run (mavenproject1) X". The left sidebar contains several icons: a green play button, a yellow play button, a brown arrow, a magnifying glass, a brown square, and a gear. The main pane displays the following log output:

```
[ jar ]  
--- exec-maven-plugin:3.0.0:exec (default-cli) @ mavenproject1 ---  
a= 2469  
a= 1763  
a= 4115  
Divide by zero  
a= 0  
a= 4115  
a= 1371  
a= 4115  
a= 2469
```

# Exception handling in Java

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

**Output:**  
?

# Different Exception handling

- Given some scenarios where unchecked exceptions may occur. They are as follows:
  1. A scenario where **ArithmetiException** occurs
    - If we divide any number by zero, there occurs an ArithmetiException.
    - `int a=50/0;//ArithmetiException`
  2. A scenario where **NullPointerException** occurs
    - If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.
    - `String s=null;`
    - `System.out.println(s.length());//NullPointerException`

# Different Exception handling...

3. A scenario where **NumberFormatException** occurs
  - ❑ The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.
  - ❑ String s="abc";
  - ❑ int i=Integer.parseInt(s);//NumberFormatException
4. A scenario where **ArrayIndexOutOfBoundsException** occurs
  - ❑ If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:
  - ❑ int a[]=new int[5];
  - ❑ a[10]=50; //ArrayIndexOutOfBoundsException

# Multiple Catch Clauses

- Two or more catches- each handles a different type of exception
- Each catch is inspected in order
- The first one whose type matches- execute it
- The remaining exceptions are bypassed
- Exception subclasses must come before any of their superclasses

- Try this code:

```
int a[ ]={10};  
i=1;  
a[ i ] = i / (i-1);
```

Two catch statements in sequence:

**ArrayIndexOutOfBoundsException**

**ArithmaticException**

??

# Nested try statements

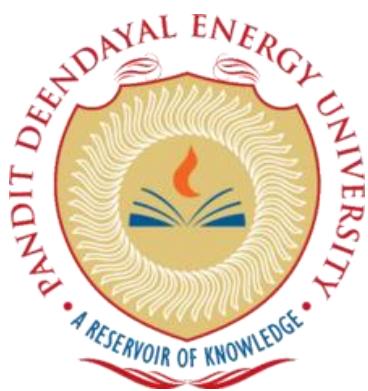
- A try statement can be inside the block of another try

# Throw and throws

- In previous examples, Java runtime exceptions
- Throw ThrowableInstance
- Mobile no program
  
- Method throws exception

# Questions

- Difference between try-catch and if-else? With examples.
- Customized or user-defined exceptions.



# MULTITHREADING IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

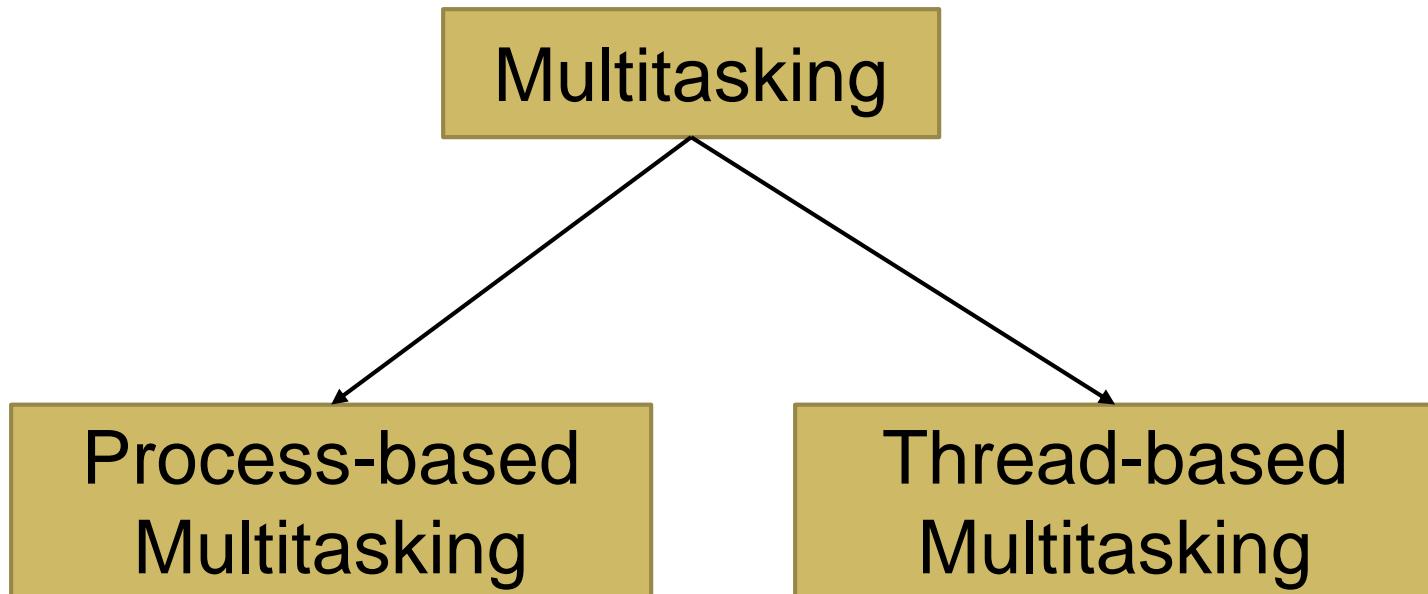
Pandit Deendayal Energy University

# Outline



# Multitasking

- Performing more than one tasks concurrently



# Multitasking...

- Process-based
  - ▣ Allows computer to run two or more programs concurrently
  - ▣ The smallest unit of code that can be dispatched- **Program**
  - ▣ Ex- run Java compiler, visit a website, use a text editor
- Thread-based
  - ▣ Allows a single program to perform two or more tasks simultaneously
  - ▣ The smallest unit of code that can be dispatched- **Thread**
  - ▣ Ex- text editor – formatting the text file, printing of that file- these two processes by threads

# Multitasking...

- Processes are **heavyweight** tasks – require separate address space, Interprocess Communication is expensive
- Threads are **lightweight** processes- executed in same address space of a single process, Interthread Communication is inexpensive

# Main Thread

```
public class ThreadsJava {  
  
    public static void main(String[] args) {  
        Thread thOb=Thread.currentThread();  
        System.out.println("current thread: "+thOb);  
  
        thOb.setName("Created thread");  
        System.out.println("After      Name      Change:  
"+thOb);  
    }  
}
```

# Creating a Thread

- (1) By extending Thread class and override run() method
- (2) By implementing Runnable interface

# Creating a Thread...

- (1) By extending Thread class and override run() method.

Some of the methods:

getName(), setName(), start(), sleep(), join(),  
run(), isalive(), getPriority(), getId()

# Creating a Thread-Example

- (1) By extending Thread class and override run() method ([Prog-1](#), [Prog-2](#))

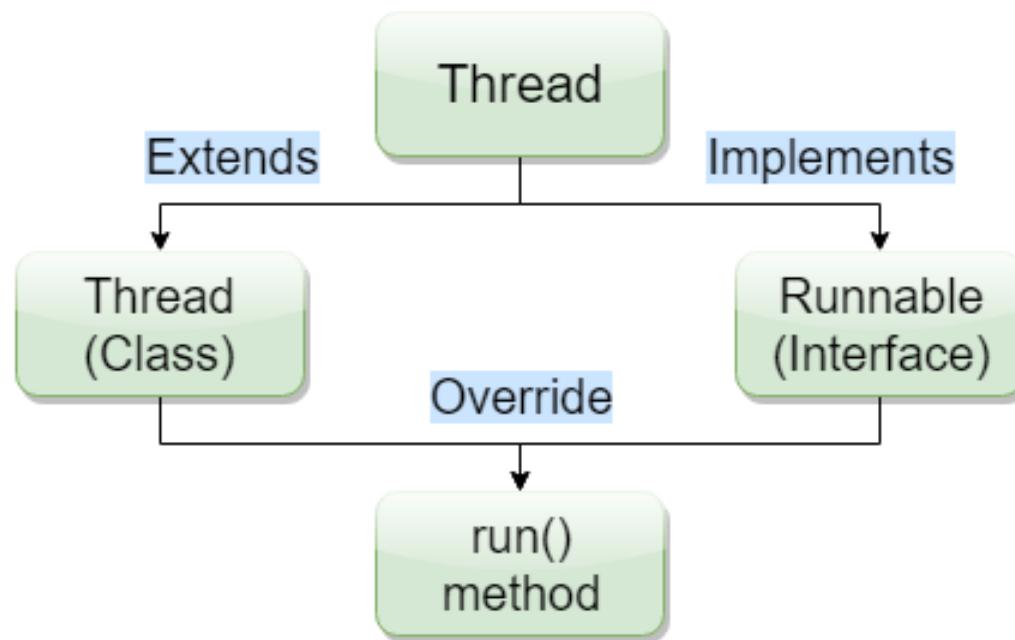
```
public class ThreadsTest extends Thread{  
    public void run(){ //overrides run() method  
        System.out.println("New thread");  
    }  
    public static void main(String[] args) {  
        ThreadsTest thOb=new ThreadsTest();  
        thOb.start();  
    } }
```

# Creating a Thread...

- (2) By implementing Runnable interface (**Prog-3**, **Prog-4**, **Prog-5**)

```
public class ThreadsTest implements Runnable {  
    public void run(){ //overrides run() method  
        System.out.println("New thread");  
    }  
    public static void main(String[] args) {  
        ThreadsTest thOb=new ThreadsTest();  
        new Thread(thOb).start();    }}  
}
```

# Extends Thread v/s Implements Runnable

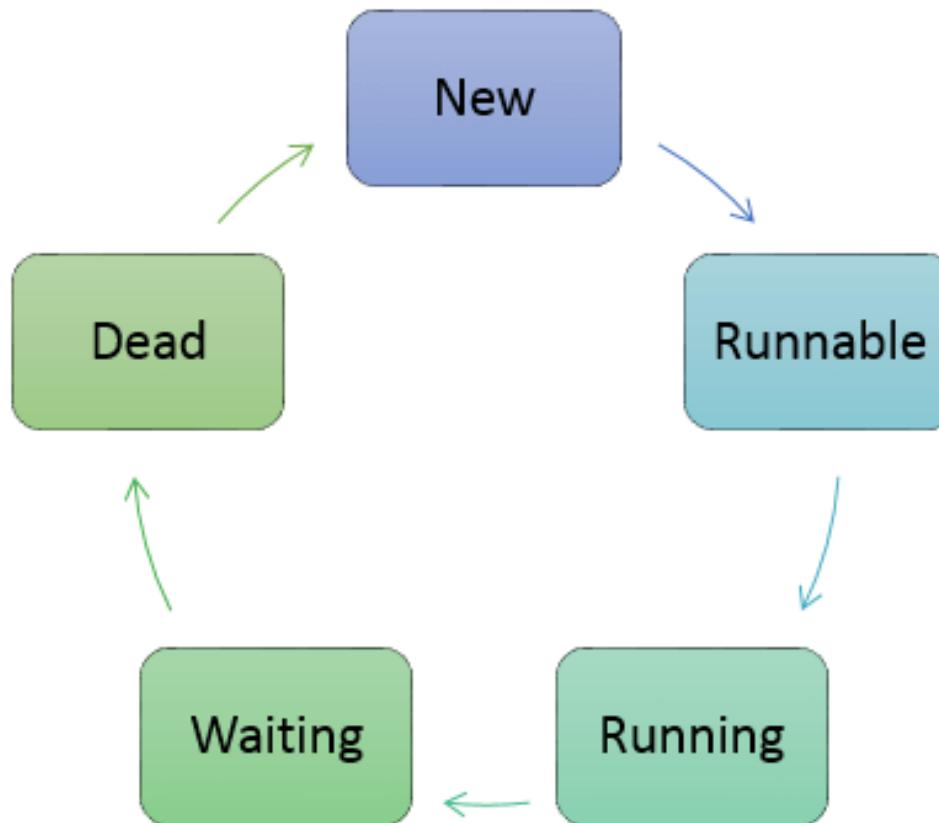


Source: <https://www.javatpoint.com/runnable-interface-in-java>

# Extends Thread v/s Implements Runnable

Extends Thread	Implements Runnable
The class inherits all methods of parent Thread class – <b>overhead of additional methods</b> (consume excess or indirect memory, computation time, or other resources).	The class needs to <b>override only abstract method run()</b> provided by Runnable interface- less memory requirements
The <b>class cannot extend (inherit)</b> any other classes	The <b>class can extend (inherit)</b> the other class EX- class A extends B implements Runnable
The code will only be in a thread environment	More flexible code, so preferable for various executor services, EX- N/W programming ( <b>sockets</b> )
Every thread creates a unique object and associates with it.	Multiple threads can share the same objects.

# Thread Life Cycle



# Thread Life Cycle...

- **New:** In this phase, the thread is created using class “Thread class”. It remains in this state till the program **starts** the thread. It is also known as born thread.
- **Runnable:** In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
- **Running:** When the thread starts executing, then the state is changed to “running” state. The scheduler selects one thread from the thread pool, and it starts executing in the application.

# Thread Life Cycle...

- **Waiting:** This is the state when a thread has to wait. As multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
- **Dead:** This is the state when the thread is terminated. The thread is in running state and as soon as it completes the processing, it goes in “dead state”.

# Creating Multiple Threads

- Prog-6: Creating multiple threads
- Prog-7: Creating multiple threads- with extending threads

# isAlive() and join() method

- isAlive() method:
  - ▣ Checks whether the thread is still running or not
  - ▣ Returns true if the thread is running
- join() method:
  - ▣ the current thread **stops its execution**
  - ▣ the current thread **goes into the wait state**
  - ▣ until the thread on which the join() method is invoked has achieved its dead state
  - ▣ **Calling thread waits until the specified thread joins it**

# Thread Priorities

- `setPriority()` method
- `getPriority()` method

# Synchronization

- To control the access of multiple threads to any shared resource
- To allow only one thread to access the shared resource
- To prevent **thread interference**
- To prevent **consistency problem**
- Can be achieved through-
  - ▣ Mutual Exclusive
  - ▣ Inter-thread communication (Co-operation)

# Synchronization through Mutual Exclusive

- To prevent interfering of the other threads while one thread uses shared resource
- Concept of **Lock or monitor**
- Every object has a lock with them
- Thread needs to acquire the object's lock to access it
- Only one thread is executing inside the object at a time- **Synchronized Block**
- **Prog-8**

# Inter-thread Communication

- A mechanism in which **a thread** is paused running in its critical section
- **Another thread is allowed to enter** (or lock) in the same critical section for execution
- **Polling**- The process of testing a condition repeatedly till it becomes true
  - ▣ Usually implemented with the help of **loops**
  - ▣ Wastes many CPU cycles and makes the implementation inefficient
  - ▣ EX- Producer-Consumer

# Producer-Consumer Problem

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

- **Problem**

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

# Solution with multithreading

- **wait():** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- **notify():** It wakes up one single thread called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
- **notifyAll():** It wakes up all the threads called wait() on the same object.
- **Prog-9**

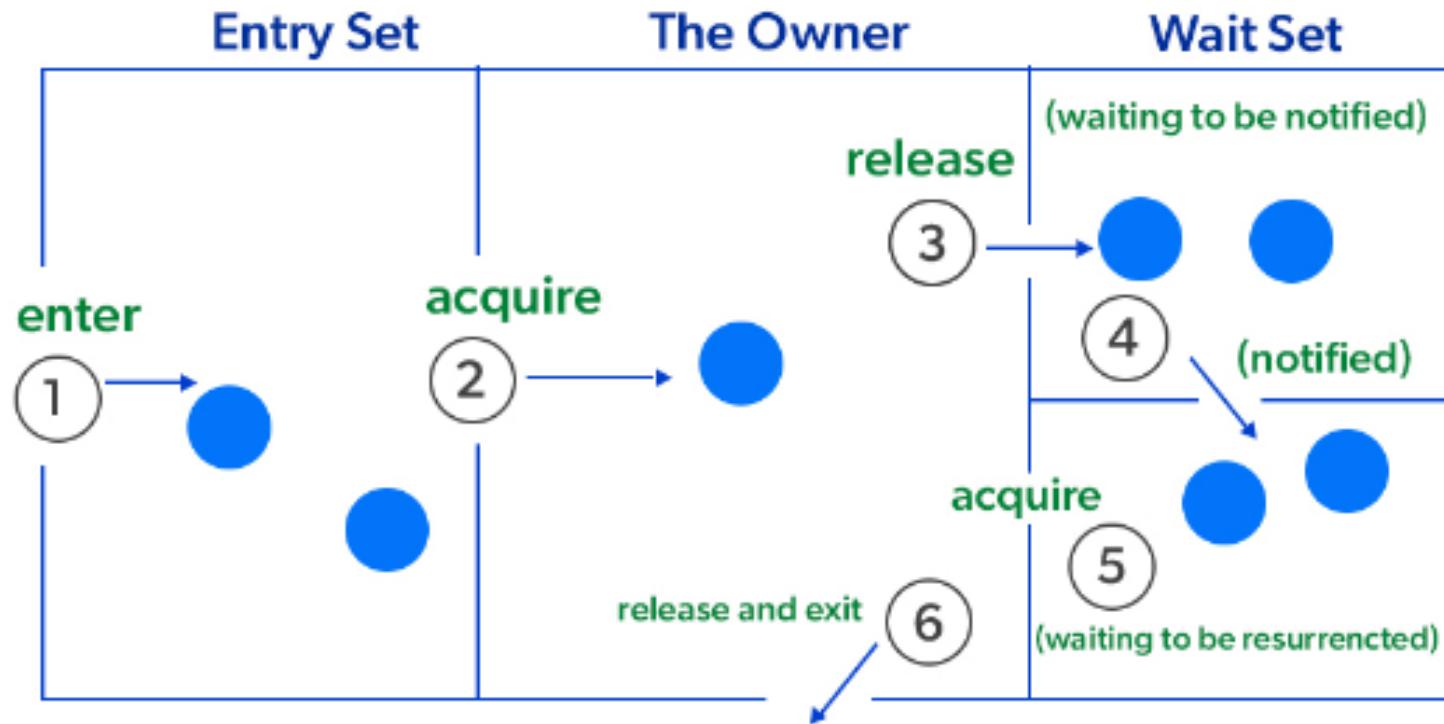


Image source: <https://www.geeksforgeeks.org/inter-thread-communication-java/>

## **List of Thread Programs in Java**

1. Check the details of current thread (id, name, priority, setName etc.)
2. Create single thread by extending class thread.
3. Create a single thread by implementing runnable interface:
  - a. Take thread object in main
  - b. Take thread object in class that is implementing runnable
4. Create multiple threads by extending thread class and assign same task to all the threads.
5. Create multiple threads by implementing runnable and assign different task to each thread (prime number, whether given number is Armstrong or not).
6. Write a program to demonstrate priorities among multiple threads.
7. Use of synchronization.
8. Implement producer consumer problem (all four)  
producer : p  
consumer : c
  - a. single p single c
  - b. single p multiple c
  - c. multiple p single c
  - d. multiple p multiple c

## Synchronization Program in Java

```
//First implement the program without synchronized kw  
//Implement by making credit() method synchronized  
//Implement by using synchronized block  
  
  
class account{  
    int balance;  
    account(){  
        balance=5000;  
    }  
    void credit(int n) throws InterruptedException{  
        System.out.println(" Balance before credit is: "+balance);  
        int b=balance;  
        Thread.sleep(500);  
        balance=b+n;  
        System.out.println("Balance after credit is: "+balance);  
    }  
    void disBal(){  
        System.out.println("Final balance is: "+balance);  
    }  
}  
  
  
class Thread1 implements Runnable{  
    Thread t;  
    account a;
```

```
int c;

String thName;

Thread1(String thName,account a,int c){

    this.thName=thName;

    this.a=a;

    this.c=c;

    t=new Thread(this);

}

public void run(){

    System.out.println("In Thread: "+thName);

    try{

        a.credit(c);

    }

    catch(InterruptedException e){

        System.out.println("Main thread interrupted");

    }

}

}

class ThreadsTest{

    public static void main(String [] args){

        account ob=new account();

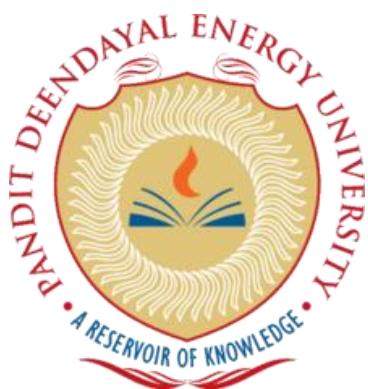
        Thread1 ob1=new Thread1("One", ob, 2000);

        Thread1 ob2=new Thread1("Two",ob, 5000);

    }

}
```

```
ob1.t.start();
ob2.t.start();
try{
    Thread.sleep(2000);
}
catch(InterruptedException e){
    System.out.println("Main thread interrupted");
}
ob.disBal();
}
}
```



# EVENT HANDLING AND GUI IN JAVA

Presented by:

**Dr. Shivangi K. Surati**

Assistant Professor,

Department of Computer Science and Engineering,

School of Technology,

Pandit Deendayal Energy University

# AWT classes

- Abstract Window Toolkit (AWT)- defines a basic set of controls, windows and dialog boxes
- Limited graphical interface
- **Uses platform-specific equivalents** or peers for execution
- Look and feel of the components are **defined by platform, not Java**
- **Heavyweight** because they use native code resources
- No longer widely used

# Problems in AWT

- Native peers-
  - ▣ Variations between OS, a component might look, or even act, differently on different platforms
  - ▣ May not support philosophy of Java- Write once, run anywhere
  - ▣ Look and feel of each component is fixed

# Swing in Java

- part of Java Foundation Classes (JFC)- a set of GUI components for Desktop applications
- used to create window-based applications
- built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java
- provides platform-independent and lightweight components
- provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

# Swing in Java...

- It does not replace AWT, only eliminates limitations
- Key features:
  - ▣ Swing components are lightweight
    - Written entirely in Java
    - No mapping with platform-specific peers
    - More efficient and flexible
  - ▣ Swing supports a Pluggable Look and Feel (PLAF)
    - Look and Feel components are under control of Swing
    - Possible to change the way that a component is rendered
    - “Plug in” a new component without any side effects

# MVC connection

The components implicitly contains three parts:

- **Model:** The state information associated with the component- checkbox checked or unchecked
  - **View:** How the component is displayed on the screen
  - **Controller:** How the component reacts to the user- click the checkbox-change the model to reflect user's choice
- 
- Swing uses a modified version of MVC- combines view and controller into a single logical entity – UI delegate
  - Swings follows **Model-Delegate architecture** or Separable Model architecture

# AWT vs Swing

Java AWT	Java Swing
AWT components are <b>platform-dependent</b> .	Java swing components are <b>platform-independent</b> .
AWT components are <b>heavyweight</b> .	Swing components are <b>lightweight</b> .
AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
AWT provides <b>less components</b> than Swing.	Swing provides <b>more powerful components</b> such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT <b>doesn't follows MVC</b> (Model View Controller)	Swing <b>follows MVC</b> .

# Hierarchy of Java Swing classes

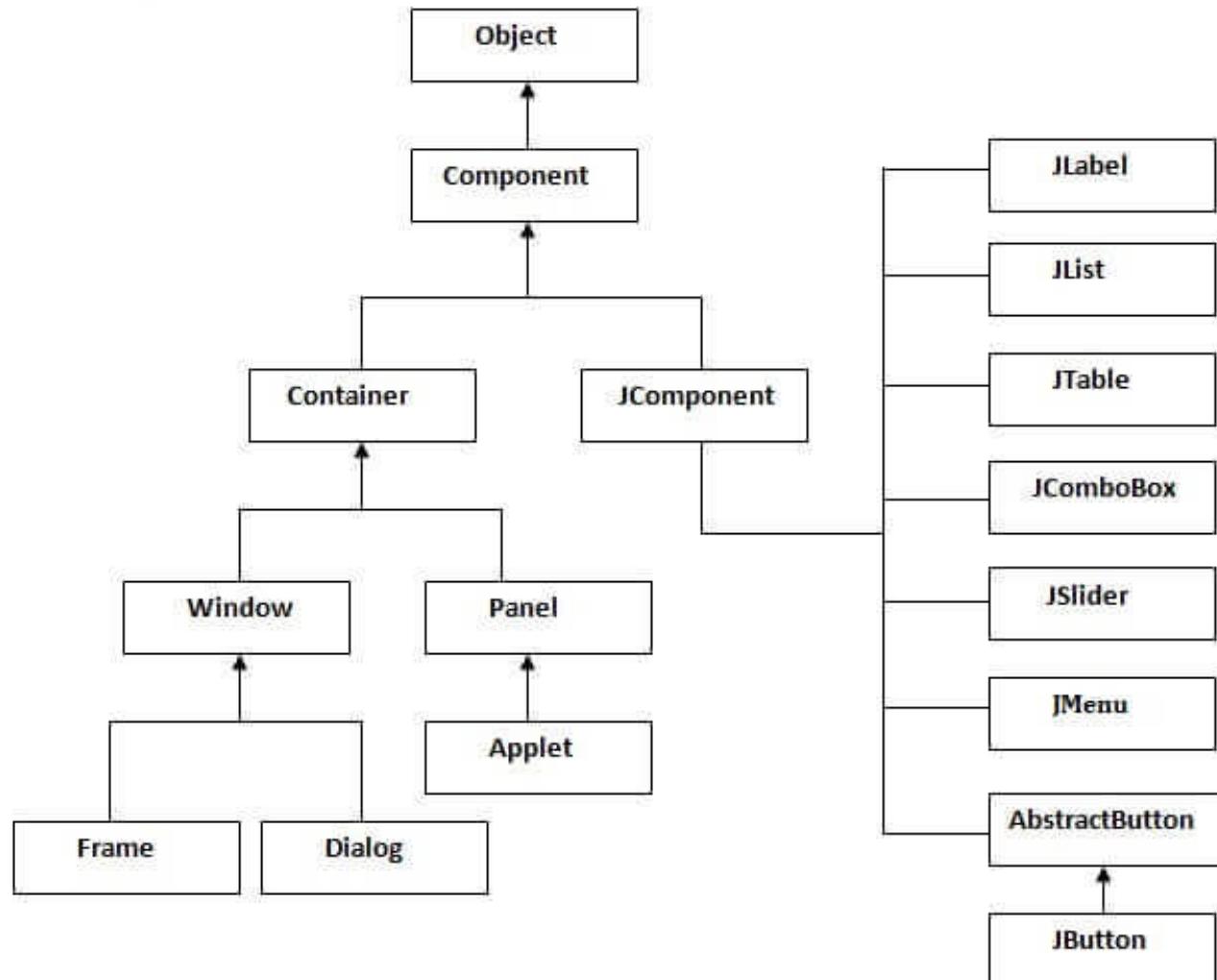


Image source: <https://www.javatpoint.com/java-swing>

# Components and Containers

- Component- an independent visual control- button, slider
- Container- holds a group of components
  
- For component to be displayed, it must be within a container
- All Swing GUIs have at least one container
- Containers are components, they can also hold other containers
- Top-level containers: JFrame, Japplet, Jwindow, JDialog

# Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width, int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

# Top-Level Container Panes

- Each top level container defines a set of panes
- At top level of hierarchy- JRootPane- manages the other panes
- The other panes that comprise the root pane are-
  - ▣ Glass Pane: top-level panel, covers all other panes, transparent instance of JPanel
  - ▣ Content Pane: Mostly application interact with this pane, add visual components
  - ▣ Layered Pane: Instance of JLayeredPane, gives a depth value to components

# Java Swing Examples

- There are two ways to create a frame:
  - ▣ By creating the object of Frame class (association)
  - ▣ By extending Frame class (inheritance)
- Different Layouts:
  - ▣ BorderLayout, CardLayout, FlowLayout, GridLayout, GridBagLayout

# Event Handling

- Response to user input
- Handles the events generated by those interactions
- For example, click on button, dragging mouse
- Uses **delegation event model** approach
- Example programs
  - ▣ Register the component with the Listener -Many classes to register the component with the Listener
  - ▣ put the event handling code into one of the following places- within class, other class and anonymous class

# Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

<b>Name of the course: Object Oriented Programming with Java</b>	<b>Course Code: 20CP204T</b>
<b>Program: B. Tech.</b> <b>Branch: CE</b>	<b>Semester: 3<sup>rd</sup></b> <b>Academic Year: 2022-23</b>

## Tutorial- Find Output/Error

### Q-1:

```

import java.util.Scanner;

class Point
{
    float x,y,z;
}

class test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        Point P[];
        int n;

        System.out.println("How many Points do you want: ");
        n = sc.nextInt();

        P=new Point[n];

        for ( int i = 0 ; i < n ; i++ )
        {
            System.out.println("Enter x,y and z for "+(i+1)+"th Point:");
            P[i].x= sc.nextFloat();
            P[i].y= sc.nextFloat();
            P[i].z= sc.nextFloat();
        }
    }
}

```

### Q-2

```

class Automobile
{
    private String drive()
    {

```

```

        return "Driving vehicle";
    }
}
class Car extends Automobile
{
    protected String drive()
    {
        return "Driving car";
    }
}

public class test extends Car
{
    public final String drive()
    {
        return "Driving Electric car";
    }
    public static void main(String[] args)
    {
        final Car car = new test();
        System.out.println(car.drive());
    }
}

```

### Q-3

```

class Super
{
    int i=15;
}
class Sub extends Super
{
    int i=10;
}
public class test
{
    public static void main(String[] args)
    {
        Super s1 = new Sub();

        System.out.println(s1.i);
    }
}

```

## Q-4

```
abstract class Car
{
    static
    {
        System.out.print("1");
    }
    public Car(String name)
    {
        super();
        System.out.print("2");
    }
    {
        System.out.print("3");
    }
}
public class BlueCar extends Car
{
    {
        System.out.print("4");
    }
    public BlueCar()
    {
        super("blue");
        System.out.print("5");
    }
    public static void main(String[] args)
    {
        new BlueCar();
    }
}
```

## Q-5

```
public class test
{
    public void print(Integer i)
    {
        System.out.println("Integer");
    }
    public void print(int i)
    {
        System.out.println("int");
    }
    public void print(long i)
```

```
{  
    System.out.println("long");  
}  
  
public static void main(String[] args)  
{  
    test T1=new test();  
  
    T1.print(10);  
}  
}
```

### **Q-6:**

```
class A  
{  
    public A(String s)  
    {  
        System.out.print("A");  
    }  
}
```

```
public class B extends A  
{  
    public B(String s)  
    {  
        System.out.print("B");  
    }  
    public static void main(String[] args)  
    {  
        new B("C");  
        System.out.println(" ");  
    }  
}
```

### **Q-7:**

```
class A  
{  
}
```

```
class B extends A  
{  
}
```

```
class C extends B
```

```

{
}

public class MainClass
{
    static void overloadedMethod(A a)
    {
        System.out.println("ONE");
    }

    static void overloadedMethod(B b)
    {
        System.out.println("TWO");
    }

    static void overloadedMethod(Object obj)
    {
        System.out.println("THREE");
    }

    public static void main(String[] args)
    {
        C c = new C();

        overloadedMethod(c);
    }
}

```

## **Q-8:**

```

public class P
{
    static void m1()
    {
        System.out.println("Class P");
    }
}

public class Q extends P
{
    static void m1()
    {
        System.out.println("Class Q");
    }
}

```

### **Q-9:**

```
public class Test{
    public static void main(String[] args){
        System.out.println("main method");
    }
    public static void main(String args){
        System.out.println("Overloaded main method");
    }
}
```

### **Q-10:**

```
class X
{
    public X(int i)
    {
        System.out.println(1);
    }
}
```

```
class Y extends X
{
    public Y()
    {
        System.out.println(2);
    }
}
```

### **Q-11:**

```
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1 == arr2)
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

### **Q-12:**

```
package inheritancePractice;
class P {
    int a = 30;
}
class Q extends P {
```

```

        int a = 50;
    }
public class Test extends Q {
    public static void main(String[] args) {
        Q q = new Q();
        System.out.println(" Value of a: " +q.a);
        P p = new Q();
        System.out.println("Value of a: " +p.a);
    }
}

```

### **Q-13:**

```

final class Complex {
    private double re, im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    Complex(Complex c)
    {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(c1);
        Complex c3 = c1;
        System.out.println(c2);
    }
}

```

### **Q-14:**

```

public class A {
    public static void main(String[] args)
    {
        System.out.println('j' + 'a' + 'v' + 'a');
    }
}

```

Q-15:

```
class demo
```

```

{
    int a, b;
    demo()
    {
        a = 10;
        b = 20;
    }

    public void print()
    {
        System.out.println ("a = " + a + " b = " + b + "n");
    }
}
class Test
{

    public static void main(String[] args)
    {
        demo obj1 = new demo();
        demo obj2 = obj1
        obj1.a += 1;
        obj1.b += 1;
        System.out.println ("Values of obj1 : ");
        obj1.print();
        System.out.println ("Values of obj2 : ");
        obj2.print();

    }
}

```

## **Q-16:**

```

// Find num1 & num2
public class Main
{
    static int findNum1(int a, int b, int c){
        int num1 = a;
        boolean b1 = (num1<b) && ((num1=b)>0);
        b1 = (num1<c) && ((num1=c)>0);
        return num1;
    }
    static int findNum2(int a, int b, int c){
        int num2 = a;
        boolean b1 = (num2>b) && ((num2=b)>0);
        b1 = (num2>c) && ((num2=c)>0);
        return num2;
    }
}
```

```
public static void main(String[] args) {  
    System.out.println("num1: "+findNum1(11,-16,12));  
    System.out.println("num2: "+findNum2(11,-16,12));  
}  
}
```

### **Q-17:**

```
public class Code  
{  
    public static void main(String args[])  
    {  
        int y = 08;  
        y = y + 2;  
        System.out.println(y);  
    }  
}
```

### **Q-18:**

```
class Exercise1b {  
    public static void main(String [] args) {  
        int x = 1;  
        while ( x < 10 ) {  
            if ( x > 3) {  
                System.out.println("big x");  
            }  
        }  
    }  
}
```

### **Q-19:**

```
public static void main(String [] args) {  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3) {  
            System.out.println("small x");  
        }  
    }  
}
```

### **Q-20.**

```
class TapeDeck {
```

```
    boolean canRecord = false;
```

```
void playTape() {  
    System.out.println("tape playing");  
}  
  
void recordTape() {  
    System.out.println("tape recording");  
}  
}  
  
class TapeDeckTestDrive {  
    public static void main(String [] args) {  
  
        t.canRecord = true;  
        t.playTape();  
  
        if (t.canRecord == true) {  
            t.recordTape();  
        }  
    }  
}
```

## Q-21.

```
class DVDPlayer {
```

```
    boolean canRecord = false;  
  
    void recordDVD() {  
        System.out.println("DVD recording");  
    }  
}
```

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

```
        DVDPlayer d = new DVDPlayer();  
        d.canRecord = true;
```

```
d.playDVD();  
  
if (d.canRecord == true) {  
    d.recordDVD();  
}  
}  
}
```

## **Q-22.**

```
class Books {  
    String title;  
    String author;  
}
```

```
class BooksTestDrive {  
    public static void main(String [] args) {  
  
        Books [] myBooks = new Books[3];  
        int x = 0;  
        myBooks[0].title = "The Grapes of Java";  
        myBooks[1].title = "The Java Gatsby";  
        myBooks[2].title = "The Java Cookbook";  
        myBooks[0].author = "bob";  
        myBooks[1].author = "sue";  
        myBooks[2].author = "ian";
```

```
        while (x < 3) {  
            System.out.print(myBooks[x].title);  
            System.out.print(" by ");  
            System.out.println(myBooks[x].author);  
            x = x + 1;  
        }  
    }  
}
```

## **Q-23.**

What is the output of the following code snippet?

```
int five = 5;  
int two = 2;
```

```
int total = five + (five > 6 ? ++two : --two);
```

### **Q-24:**

```
public static void main(String... args) {  
    String car, bus = "petrol";  
    car = car + bus;  
    System.out.println(car);  
}
```

#### **Options:**

- a. petrol
- b. petrolpetrol
- c. compilation error
- d. runtime error

### **Q-25.**

```
class A  
{  
    public A(String s)  
    {  
        System.out.print("A");  
    }  
}  
  
public class B extends A  
{  
    public B(String s)  
    {  
        System.out.print("B");  
    }  
    public static void main(String[] args)  
    {  
        new B("C");  
        System.out.println(" ");  
    }  
}
```

### **Q-26. class Clidder**

```
{  
    private final void flipper()  
    {  
        System.out.println("Clidder");  
    }  
}
```

```
public class Clidlet extends Clidder
```

```

{
    public final void flipper()
    {
        System.out.println("Clidlet");
    }
    public static void main(String[] args)
    {
        new Clidlet().flipper();
    }
}

```

### **Q-27:**

Will this code compile successfully? If yes, what is output? If no, identify the errors.

```

package pack1;
public class A
{
    private int x = 50;
    protected int y = 100;
    int z = 200;
}
package pack2;
import pack1.A;
public class B extends A {

}
import pack2.B;
public class Test {
    public static void main(String[] args)
    {
        B b = new B();
        System.out.println(b.x);

        System.out.println(b.y);
        System.out.println(b.z);
    }
}

```

### **Q-28:**

```

class Base {
    public void show() {
        System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
        System.out.println("Derived::show() called");
    }
}

```

```

}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();
        b.show();
    }
}

```

## **Q-29:**

```

package overridingPrograms;
public class X
{
void draw(int a, float b) throws Throwable
{
System.out.println("Circle");
}
}
public class Y extends X
{
@Override
void draw(int a, float b)
{
System.out.println("Rectangle");
}
}
public class Z extends Y
{
@Override
void draw(int a, float b) throws ArithmeticException
{
System.out.println("Square");
}
}
public class Test
{
public static void main(String[] args) throws Throwable
{
X x = new Y();
x.draw(20, 30.5f);
Y y = (Y)x;
y.draw(10,2.9f);
Z z = (Z)y;
z.draw(20, 30f);
}
}

```

### **Q-30:**

```
class Automobile {  
    private String drive() {  
        return "Driving vehicle";  
    }  
}  
  
class Car extends Automobile {  
    protected String drive() {  
        return "Driving car";  
    }  
}  
  
public class ElectricCar extends Car {  
  
    @Override  
    public final String drive() {  
        return "Driving electric car";  
    }  
  
    public static void main(String[] wheels) {  
        final Car car = new ElectricCar();  
        System.out.print(car.drive());  
    }  
}
```

- A. Driving vehicle
- B. Driving electric car
- C. Driving car
- D. The code does not compile

### **Q-31:**

```
class Building {  
    Building() {  
        System.out.println("pdeu's-Building");  
    }  
  
    Building(String name) {  
        this();  
        System.out.println("pdeu's-building: String Constructor" + name);  
    }  
}  
  
public class House extends Building {  
    House() {  
        System.out.println("pdeu's-House ");
```

```

    }

House(String name) {
    this();
    System.out.println("pdeu's-house: String Constructor" + name);
}

public static void main(String[] args) {
    new House(" pdeu");
}
}

```

### **Q-32:**

```

class Test
{
    final int MAXIMUM = m1();

private int m1()
{
    System.out.println(MAXIMUM);
    return 1500;
}

public static void main(String[] args)
{
    Test t = new Test();

    System.out.println(t.MAXIMUM);
}
}

```

- a) Compilation error
- b) Runtime error
- c) 0
- 1500
- d) 1500
- 1500

### **Q-33:**

```

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

    // final with for-each statement
    for (final int i : arr)

```

```
        System.out.print(i + " ");
    }
}
a) Compilation error
b) Runtime error
c) 1 2 3
```

### **Q-34:**

```
class Test {
public
    static void main(String[] args)
    {
        int x = 20;
        System.out.println(x);
    }
static
{
    int x = 10;
    System.out.print(x + " ");
}
}
Option
A) 10 20
B) 20 10
C) 10 10
D) 20 20
```

### **Q-35:**

```
public class Test {
    public static void main(String[] args) {
        method(null);
    }

    public static void method(Object o) {
        System.out.println("Object method");
    }

    public static void method(String s) {
        System.out.println("String method");
    }
}
```

PROGRAMS:

- 1) Create a class Person using constructors that has a single variable age. Such that when the object person1 is created, it gets initialized to default age 20 and when person2 is created the user input his choice of age.
- 2) Write a program to print the area of two rectangles having sides (4,5) and (5,8) respectively by creating a class named 'Rectangle' with a method named 'Area' which returns the area and length and breadth passed as parameters to its constructor. Construct a class to find the volume of a cuboid, cube and cylinder using the concept of overloading. The formulas are given below:

Shapes	Volume Formula	Variables
Rectangular Solid or Cuboid	$V = l \times w \times h$	$l = \text{Length}$ $w = \text{Width}$ $h = \text{Height}$
Cube	$V = a^3$	$a = \text{Length of edge or side}$
Cylinder	$V = \pi r^2 h$	$r = \text{Radius of the circular base}$ $h = \text{Height}$