

Course Specifications Form

University: University of Information Technology & Sciences (UITS)

Faculty: Faculty of Science and Engineering (FSE)

Course specifications

Name of the Program (s): Department of Computer Science and Engineering (CSE)

Types of Course: Core Course

Department offering the course: Department of Computer Science and Engineering

Level/Term: Level-1, Term-2

A- Basic Information

Title of the Course:	Object Oriented Programming Language	Code:	CSE 151
Number of Lectures:	42	Credit:	3.0
Number of Tutorial:	4	Number of Practical:	0
Prerequisite:	CSE 111	Total:	04

B- Detail Information

1. Course Description

Philosophy of Object Oriented Programming (OOP); Advantages of OOP over structured programming; Encapsulation, classes and objects, access specifiers, static and non-static members; Constructors, destructors and copy constructors; Array of objects, object pointers, and object references; Inheritance: single and multiple inheritance; Polymorphism: overloading, abstract classes, virtual functions and overriding; Exceptions; Object Oriented I/O; Template functions and classes; Multi-threaded Programming.

(Prerequisite: CSE 111)

2. Course Objective

- To learn how to implement object-oriented designs with Java.
- To identify Java language components and how they work together in applications.
- To design and program stand-alone Java applications.
- To learn about extending Java classes with inheritance, dynamic binding and gain knowledge about exception handling.

3. Course Outcomes of the course (COs)

- Impart** adequate knowledge and programming skills using the fundamentals and basics of Java Language.
- Analyze** the benefits of object oriented design and understand when it is an appropriate methodology to use.
- Construct** a Java program correctly from the analyzed problems using Object Oriented approach.

4. Mapping of Course CO and PO



Course Outcome (CO) of the Course		Program Outcome (PO)									
CO1	3			5	6	7	8	9	10	11	12
CO2	3										
CO3	3										

5. Contents

Week	Topic	Duration	Lecture	Tutorial/ Practical
1	Data types, Variables and Arrays	150 Minutes	Lecture 1,2,3	Class
2	Operators and Control Statements	150 Minutes	Lecture 4,5,6	Class
3	Introducing Classes and methods- this keyword, garbage collection, finalize method	150 Minutes	Lecture 7,8,9	Class
4	More closer look about Classes and Methods- overloading, constructor & destructor	150 Minutes	Lecture 10,11,12	Class
5	Argument passing, returning objects, Static, Final	150 Minutes	Lecture 13,14,15	Class
6	Inheritance- super, multilevel hierarchy, overriding	150 Minutes	Lecture 16,17,18	Class+ Class Test1
7	Constructors in inheritance, dynamic method dispatching, final keyword	150 Minutes	Lecture 19,20,21	Class
8	Topics Covered up to previous classes	150 Minutes	Lecture 22,23,24	Class
9	Packages and Interfaces-access modifier, define package and class path	150 Minutes	Lecture 25,26,27	Class+Assignment
10	Implementation of interface, nested interface	150 Minutes	Lecture 28,29,30	Class
11	Exception Handling-try-catch	150 Minutes	Lecture 31,32,33	Class

12	Throw, throws, finally, chained exceptions	150 Minutes	Lecture 34,35,36	Class
13	Multithreaded Programming- Creating different threads using inheritance and interface, synchronization among threads	150 Minutes	Lecture 37,38,39	Class+ Class Test2
14	Topics Covered after Class Test -01 up to previous classes	150 Minutes	Lecture 40,41,42	Class

6. Teaching And Learning Methods

- a) Lecture
- b) Presentation
- c) Class Assignments
- d) Class Test / Quiz
- e) Practical Work / Case Study

7. Student Assessment Methods

- a. Formative: Class Participation, Class Activity, Interactive Questions
- b. Summative: Class Assignments, Class Test, Class Quiz, Term Final

8. Assessment Schedule

Assessment 1	Class, Object, Constructor, Modifiers	Class Test 1	Week 6
Assessment 2	Inheritance, Polymorphism	Assignment 1	Week 9
Assessment 3	Exception Handling	Class test 2	Week 13
Total		100%	

9. Weights of Assessments

Assessments	Percentage	CO
Class attendance	10 %	
Class Tests / Assignments	20 %	CO1, CO2
Term Final Examination	70 %	CO1, CO2, CO3, CO4, CO5
Total	100%	

10. List of References

a) Essential books (text books):

1. Java- The Complete Reference 8th Reference by Herbert Schildt.



b) Recommended Reference books:

1. Core Java volume I fundamentals 11th edition by Cay S. Horstmann
2. Java Programming For Beginners - A Simple Start to Java Programming by Scott Sanderson

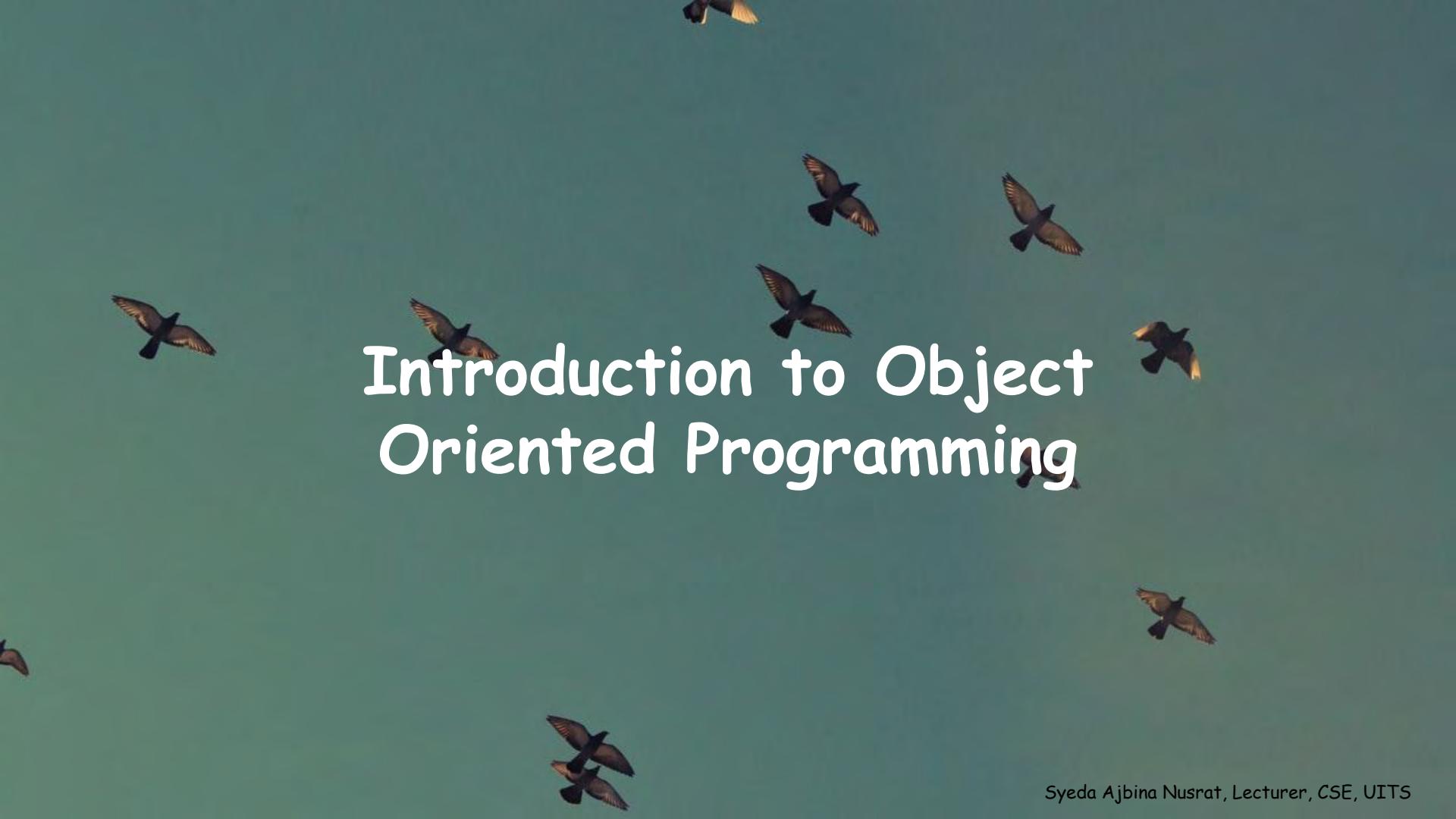
c) Online resource:

<https://www.w3schools.com/java/default.asp>

11. Classroom Facilities

- a. Multimedia Classroom: Sound Systems & Projector.
- b. Internet access from classroom computer
- c. White Board

-----	-----
Course Teacher:	Head of the Department
Date:	Date:



Introduction to Object Oriented Programming

Introduction

Object oriented programming is the principle of design and development of programs using **modular approach**.

- OOP approach provides advantages in creation and development of software for real life application.
- The basic element of OOP is the data.
- The programs are built by combining data and functions that operate on the data.
- Some of the OOP's languages are C++, Java, C#, Smalltalk, Perl, and Python.

Procedural Programming

- The procedural programming focuses on processing of instructions in order to perform a desired computation.
- The top-down concepts to decompose main functions into lower level components for modular coding purpose.
- Therefore it emphasizes more on doing things like algorithms.
- This technique is used in a conventional programming language such as C and Pascal

Object Oriented Programming

- OOP is a concept that combines both the data and the functions that operate both the data and the functions that operate on that data into a single unit called the object.
- An object is a collection of set of data known as member data and the functions that operate on these data known as member function.
- OOP follows **bottom-up design technique**.
- Class is the major concept that plays important role in this approach.
Class is a template that **represents a group of objects** which share common properties and relationships.



Differences

Procedural Programming	Object Oriented Programming
Large programs are divided into smaller programs known as functions	Programs are divided into objects
Data is not hidden and can be accessed by external functions	Data is hidden and cannot be accessed by external functions
Follow top down approach in the program design	Follows bottom-up approach in the program design
Data may communicate with each other through functions	Objects may communicate with each other through functions.
Emphasize is on procedure rather than data	Emphasize is on data rather than procedure

Basic concepts of OOP

The following are the major characteristics of OOP:

- Objects
- Class
- Data abstraction
- Data encapsulation
- Inheritance
- Overloading
- Polymorphism
- Dynamic binding
- Message passing

- a class is a logical construct;
an object has physical reality.

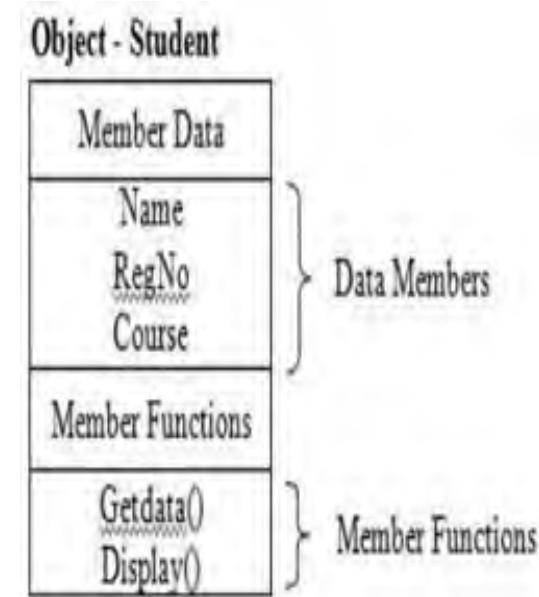
Objects

can be used as parameters in method (7EC7P129,12EC7P142)

- Objects are basic **building blocks** for designing programs.
- An object is a **collection of data members and associated member functions.**
- An object may represent a person, place or a table of data.
- Each object is identified by a unique name.
Each object must be a member of a particular class.
- Example: Apple, orange, mango are the objects of class fruit.

A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality

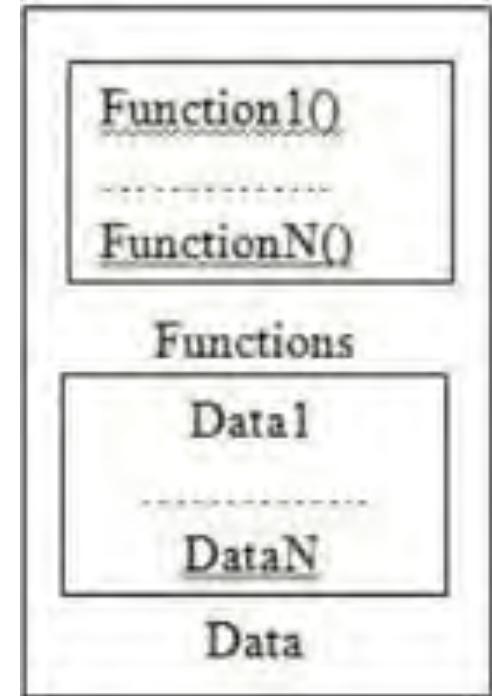
Object - Objects have states and behaviors.
An object is an instance of a class.
Class - A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.



Classes

can be used as parameters in method (7EC7P129,12EC7P142)

- The **objects** can be made **user defined data types** with the help of a class.
- A class is a **collection of objects** that have identical properties, common behavior and shared relationship.
- Once class is defined, any number of objects of that class is created.
- Classes are user defined data types. A class can hold both data and functions.
- Example: Planets, sun, moon are the members of class solar system.



- A process of new data type creation and it contains member variables and methods.
- Classes are user defined data types and work like the built-in type of the programming language.

Data Abstraction, Encapsulation & Data hiding

Data abstraction:

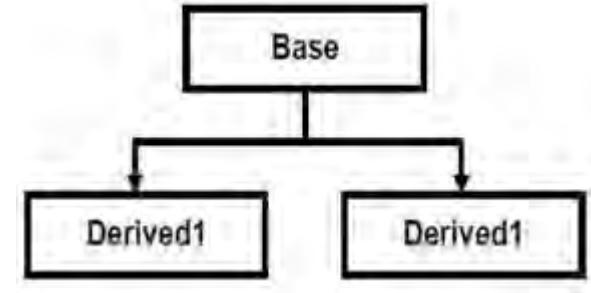
Refers to the process of representing essential features without including background details or explanations.

Data encapsulation:

- The wrapping of data and functions into a single unit(class) is called data encapsulation.
- Data encapsulation enables data hiding and information hiding.
- Data hiding is a method used in object oriented programming to hide information within computer code.

Inheritance

- Inheritance is the process by which one object can acquire and use the properties of another object.
- The existing class is known as base class or super class.
- The new class is known as derived class or sub class.
- The derived class shares some of the properties of the base class. Therefore a code from a base class can be reused by a derived class.



Overloading

- Overloading allows objects to have different meaning depending upon context.
- There are two types of overloading:
 - Operator overloading
 - Function overloading
- When an existing operator operates on new data type is called operator overloading.
- Function overloading means two or more functions have same name, but differ in the number of arguments or data type of arguments.

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

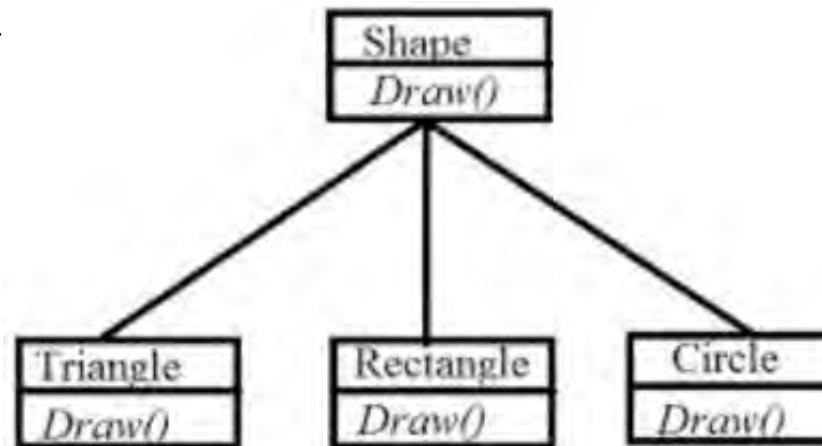
Polymorphism

- The ability of an operator and function to take multiple forms is known as polymorphism.
- The different types of polymorphism are operator overloading and function overloading.

- "Many forms" allow several definitions under a single method name.

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

There are two ways to overload the method in java • By changing number of arguments • By changing the data type



Dynamic binding and Message passing

- Dynamic binding
 - Binding is the process of connecting one program to another.
 - Dynamic binding is the process of linking the procedure call to a specific sequence of code or function at run time or during the execution of the program.
- Message passing
 - In OOP, processing is done by sending message to objects.
 - A message for an object is request for execution of procedure.
 - Message passing involves specifying the name of the object, the name of the function(message) and the information to be sent.



Advantages of OOP

- The programs are modularized based on the principles of classes and objects.
- Linking code and object allows related objects to share common code. This reduces **code duplication** and code reusability.
- Creation and implementation of OOP code is easy and reduces software development time.
- The concept of data abstraction separates object specification and object implementation.
- Data encapsulated along with functions. Therefore external non-member function cannot access or modify data, thus providing data security.
- Easier to develop complex software, because complexity can be minimized through inheritance.
- OOP can communicate through message passing which makes interface description with outside system very simple.



Disadvantages of OOP

- OOP typically involves more lines of code than procedural programs.
- OOP is slower than procedure based programs, as they require more instructions to be executed.
- Not suitable for all types of programs.
- To convert a real world problem into an object oriented model.
- OOP's software development, debugging and testing tools are not standardized.
- Polymorphism and dynamic binding also requires processing time, due to overload of function calls during run time.

Application of OOP

SELF STUDY

References

- Slides by Prof. K. Adisesha

“

YOU WILL NEVER HAVE THIS
DAY AGAIN, SO MAKE IT
COUNT.

@therandomviber

Introduction to JAVA

History of Java

- Java was originally developed by Sun Microsystems starting in 1991
 - James Gosling
 - Patrick Naughton
 - Chris Warth
 - Ed Frank
 - Mike Sheridan
- This language was initially called ***Oak***
- Renamed ***Java*** in 1995

What is Java?

- A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language

----- Sun

- Object-Oriented
 - No free functions
 - All code belong to some class
 - Classes are in turn arranged in a hierarchy or package structure

What is Java?

- Distributed
 - Fully supports IPv4, with structures to support IPv6
 - Includes support for Applets: small programs embedded in HTML documents
- Interpreted
 - The program are compiled into Java Virtual Machine (JVM) code called bytecode
 - Each bytecode instruction is translated into machine code at the time of execution

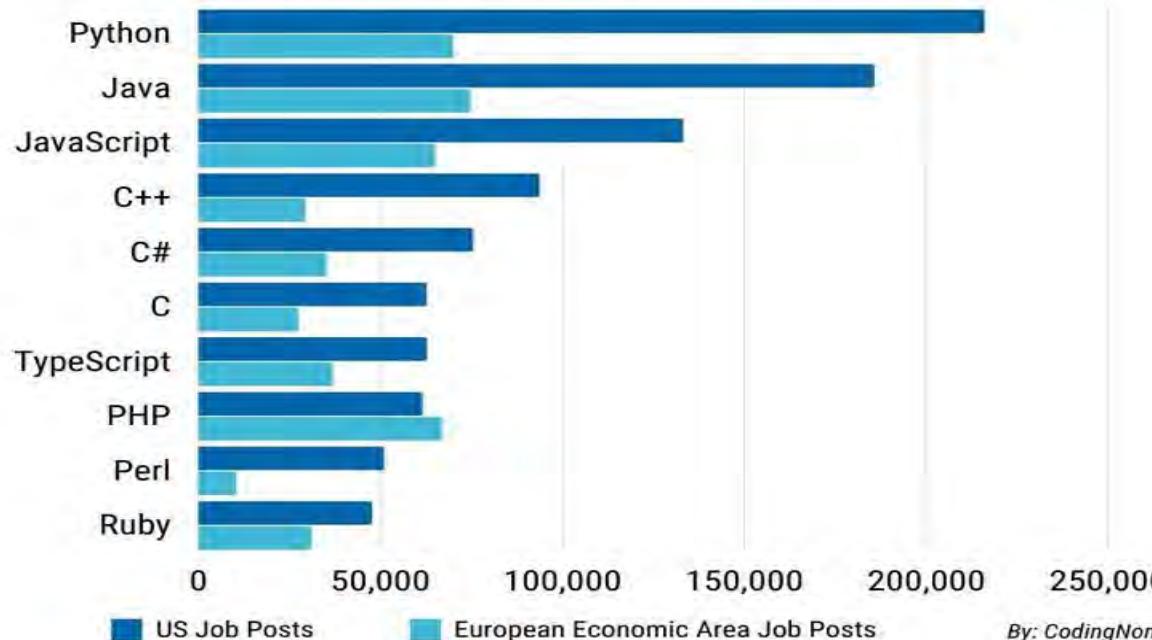
What is Java?

- Robust
 - Java is simple – no pointers/stack concerns
 - Exception handling – try/catch/finally series allows for simplified error recovery
 - Strongly typed language – many errors caught during compilation

JAVA - the popular one

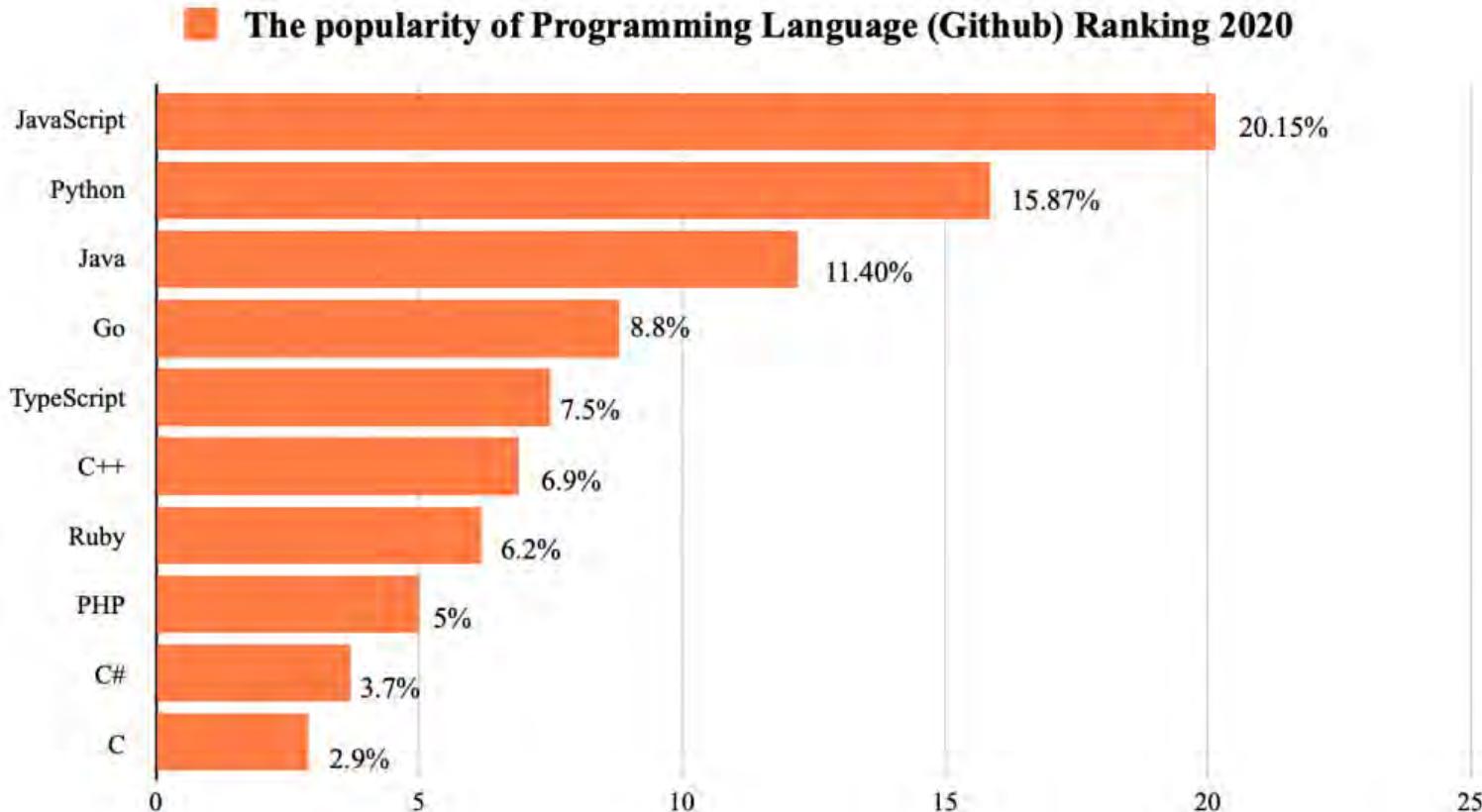
Most in-demand programming languages of 2022

Based on LinkedIn job postings in the USA & Europe

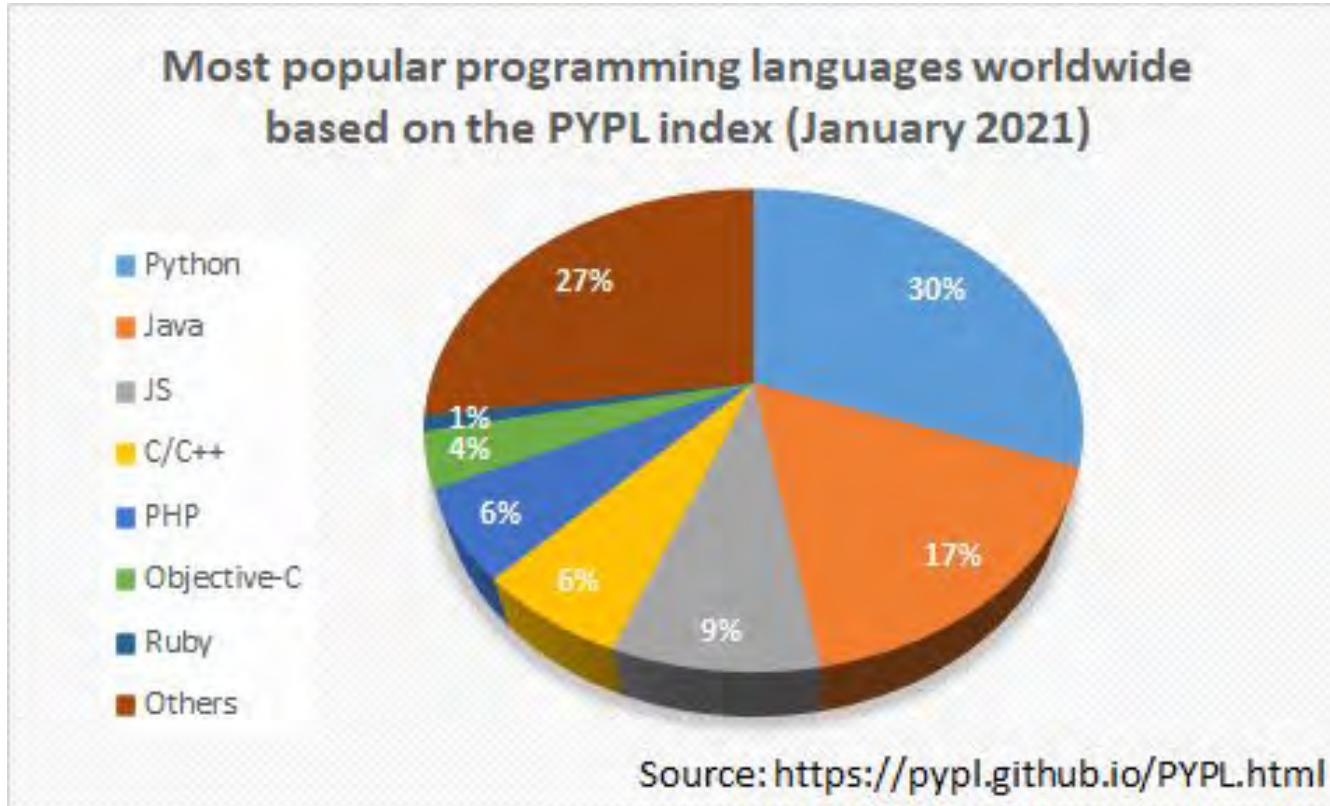


By: CodingNomads

JAVA - the popular one



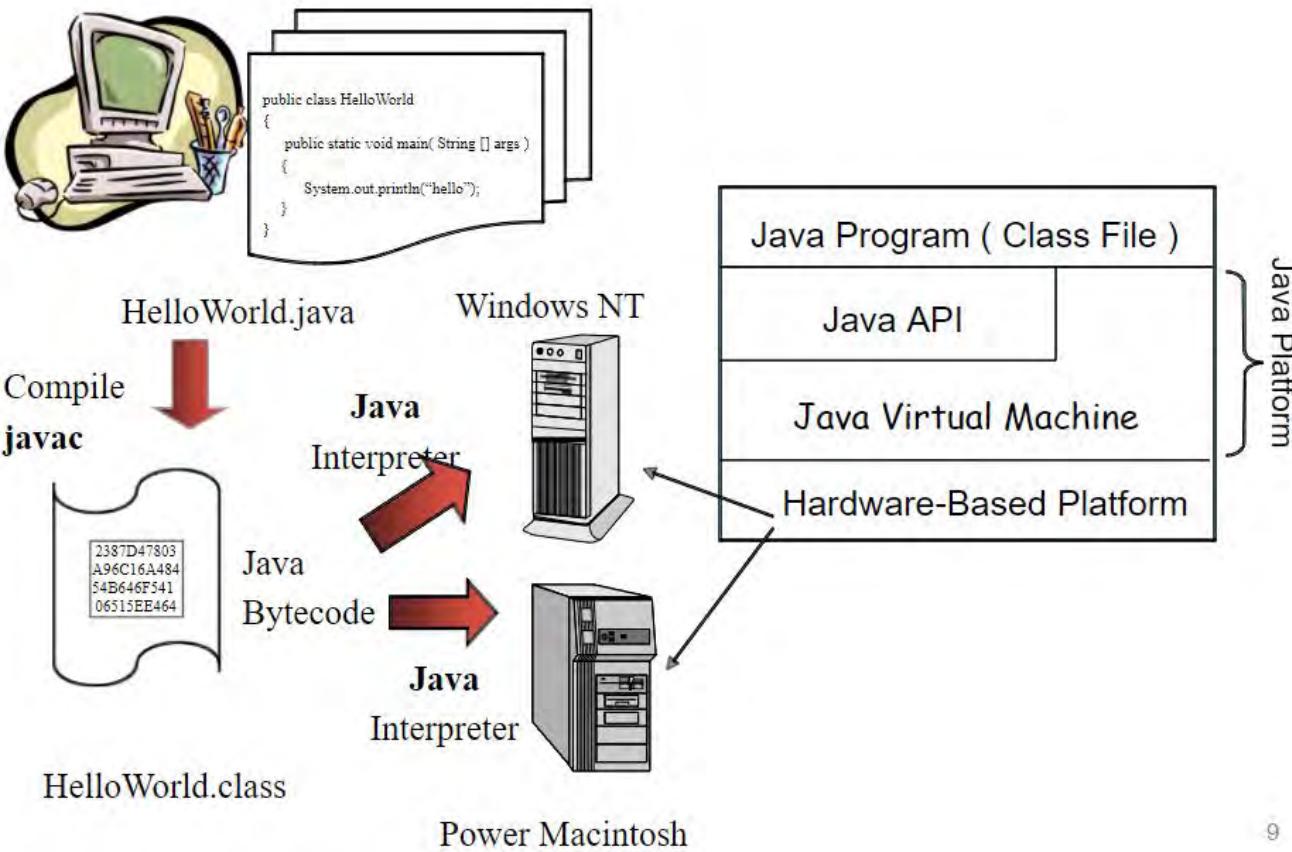
JAVA - the popular one



Java Editions

- Java 2 Platform, Standard Edition (**J2SE**)
 - Used for developing Desktop based application and networking applications
- Java 2 Platform, Enterprise Edition (**J2EE**)
 - Used for developing large--scale, distributed networking applications and Web-based applications
- Java 2 Platform, Micro Edition (**J2ME**)
 - Used for developing applications for small memory-constrained devices, such as cell phones, pagers and PDAs

Java platform



Java Development Environment

- Edit
 - Create/edit the source code
- Compile
 - Compile the source code
- Load
 - Load the compiled code
- Verify
 - Check against security restrictions
- Execute
 - Execute the compiled

Phase 1: Creating a Program

- Any text editor or Java IDE (Integrated Development Environment) can be used to develop Java programs
- Java source--code file names must end with the **.java** extension
- Some popular Java IDEs are
 - NetBeans
 - Eclipse
 - JCreator
 - IntelliJ

Phase 2: Compiling a Java Program

- ***javac Welcome.java***
 - Searches the file in the current directory
 - Compiles the source file
 - Transforms the Java source code into bytecodes
 - Places the bytecodes in a file named **Welcome.class**

Bytecodes

- They are not machine language binary code
- They are independent of any particular microprocessor or hardware platform
- They are **platform--independent instructions**
- Another entity (interpreter) is required to convert the bytecodes into machine codes that the underlying microprocessor understands
- This is the job of the **JVM** (Java Virtual Machine)

JVM (Java Virtual Machine)

- It is a part of the JDK and the foundation of the Java platform
- It can be installed separately or with JDK
- A virtual machine (VM) is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with the VM
- It is the JVM that makes Java a portable language

JVM (Java Virtual Machine)

- The same bytecodes can be executed on any platform containing a compatible JVM
- The JVM is invoked by the java command
 - ***java Welcome***
- It searches the class Welcome in the current directory and executes the main method of class Welcome
- It issues an error if it cannot find the class Welcome or if class Welcome does not contain a method called main with proper signature

Phase 3: Loading a Program

- One of the components of the JVM is the class loader
- The class loader takes the .class files containing the bytecodes and transfers them to RAM
- The class loader also loads any of the .class files provided by Java that our program uses

Phase 4: Bytecode Verification

- Another component of the JVM is the bytecode verifier
- Its job is to ensure that bytecodes are valid and do not violate Java's security restrictions
- This feature helps to prevent Java programs arriving over the network from damaging our system

Phase 5: Execution

- Now the actual execution of the program begins
- Bytecodes are converted to machine language suitable for the underlying OS and hardware
- Java programs actually go through two compilation phases
 - Source code --> Bytecodes
 - Bytecodes --> Machine language

Editing a Java Program

The screenshot shows a Java code editor with a tab labeled "Welcome.java". The code is a simple Java program that prints "Hello Java", "I like Java", and "We study in CSE". The code editor uses color coding for syntax: comments are in gray, strings are in green, and class names and method names are in blue.

```
1  /**
2   * Created by rifat on 20/08/15.
3   */
4  public class Welcome {
5      public static void main(String[] args) {
6          System.out.println("Hello Java");
7          System.out.printf("I like %s\n", "Java");
8          String strDepartment = "CSE";
9          System.out.print("We study in " + strDepartment + "\n");
10     } // end method main
11 } // end class Welcome - NOTE: no semicolon is required here
12
```

Examining Welcome.java

- A Java source file can contain multiple classes, but only one class can be a public class
- Typically Java classes are grouped into packages (similar to namespaces in C++)
- A public class is accessible across packages
- The source file name must match the name of the public class defined in the file with the .java extension

Examining Welcome.java

- In Java, there is no provision to declare a class, and then define the member functions outside the class
- Body of every member function of a class (called method in Java) must be written when the method is declared
- Java methods can be written in any order in the source file
- A method defined earlier in the source file can call a method defined later

Examining Welcome.java

- **public static void main(String[] args)**
 - **main** is the starting point of every Java application
 - **public** is used to make the method accessible by all
 - **static** is used to make main a static method of class Welcome. Static methods can be called without using any object; just using the class name. JVM call main using the **ClassName.methodName** notation
 - **void** means main does not return anything
 - **String args[]** represents an array of String objects that holds the command line arguments passed to the application. *Where is the length of args array?*

Examining Welcome.java

- Think of JVM as a outside Java entity who tries to access the main method of class Welcome
 - main must be declared as a public member of class Welcome
- JVM wants to access main without creating an object of class Welcome
 - main must be declared as static
- JVM wants to pass an array of String objects containing the command line arguments
 - main must take an array of String as parameter

Examining Welcome.java

- ***System.out.println()***
 - Used to print a line of text followed by a new line
 - **System** is a class inside the Java API
 - **out** is a public static member of class System
 - **out** is an object of another class of the Java API
 - **out** represents the standard output (similar to stdout or cout)
 - **println** is a public method of the class of which out is an object

Examining Welcome.java

- **System.out.print()** is similar to **System.out.println()**, but does not print a new line automatically
- **System.out.prinC()** is used to print formatted output like printf() in C
- In Java, characters enclosed by double quotes (" ") represents a String object, where String is a class of the Java API
- We can use the plus operator (+) to concatenate multiple String objects and create a new String object

Compiling a Java Program

- Place the .java file in the bin directory of your Java installation
 - **C:\Program Files\Java\jdk1.8.0_51\bin**
- Open a command prompt window and go to the bin directory
- Execute the following command
 - **javac Welcome.java**
- If the source code is ok, then javac (the Java compiler) will produce a file called Welcome.class in the current directory

Compiling a Java Program

- If the source file contains multiple classes then javac will produce separate .class files for each class
- Every compiled class in Java will have their own .class file
- .class files contain the bytecodes of each class
- So, a .class file in Java contains the bytecodes of a single class only

Executing a Java Program

- After successful compilation execute the following command
 - ***java Welcome***
 - *Note that we have omitted the .class extension here*
- The JVM will look for the class file *Welcome.class* and search for a ***public static void main(String args[])*** method inside the class
- If the JVM finds the above two, ***it will execute the body of the main method***, otherwise it will generate an error and will exit immediately

Another Java Program

```
A.java x
```

```
1  /**
2  * Created by rifat on 21/08/15.
3  */
4  public class A {
5      private int a;
6
7      public A()
8      {
9          this.a = 0;
10     }
11
12     public void setA(int a)
13     {
14         this.a = a;
15     }
16
17     public int getA()
18     {
19         return this.a;
20     }
21
22     public static void main(String args[])
23     {
24         A ob;
25         ob=new A();
26         ob.setA(10);
27         System.out.println(ob.getA());
28     }
29
30 }
```

Examining A.java

- The variable of a class type is called a reference
 - *ob* is a reference to A object
- Declaring a class reference is not enough, we have to use new to create an object
- Every Java object has to be instantiated using keyword **new**
- We access a public member of a class using the dot operator (.)
 - Dot (.) is the only member access operator in Java.
 - Java does not have ::, -->, & and *

“

*If the plan doesn't work,
change the plan.
But never the goal.*

#INSPIRATION

AVEMATEIU.COM

Variable & Types

Variables

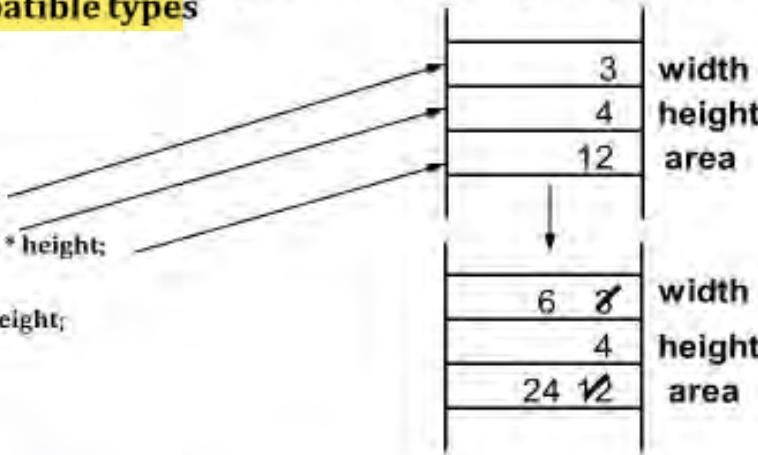
- **What is a variable?**

- The name of some **location of memory** used to hold a data value
- **Different types** of data require **different amounts** of memory. The compiler's job is to reserve sufficient memory
- Variables need to be declared once
- Variables are **assigned** values, and these values may be changed later
- Each variable has a type, and **operations can only be performed between compatible types**

- **Example**

```
int width = 3;  
int height = 4;  
int area = width * height;
```

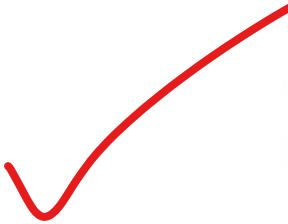
```
width = 6;  
area = width * height;
```





Variable Names

- **Valid Variable Names:** These rules apply to all Java names, or **identifiers**, including methods and class names
 - Starts with: a **letter** (a-z or A-Z), **dollar sign** (\$), or **underscore** (_)
 - Followed by: zero or more **letters**, **dollar signs**, **underscores**, or **digits** (0-9).
 - Uppercase and lowercase are different (total ≠ Total ≠ TOTAL)
 - Cannot be any of the **reserved names**. These are special names (keywords) reserved for the compiler. Examples:
class, float, int, if, then, else, do, public, private, void, ...



Good Variable Names

- **Choosing Good Names** → Not all valid variable names are good variable names
- Some guidelines:
 - Do not use '\$' (it is reserved for special system names.)
 - Avoid names that are identical other than differences in case (total, Total, and TOTAL).
 - Use meaningful names, but avoid excessive length
 - **crItm** → Too short
 - **theCurrentItemBeingProcessed** → Too long
 - **currentItem** → Just right
- **Camel case** capitalization style
- In Java we use camel case
 - Variables and methods start with lower case
 - **dataList2 myFavoriteMartian showMeTheMoney**
 - Classes start with uppercase
 - **String JOptionPane MyFavoriteClass**

Valid/Invalid Identifiers

Valid:

\$\$_

R2D2

INT

okay. "int" is reserved, but case is different here

_dogma_95_

riteOnThru

SchultzieVonWienerschnitzellII

Invalid:

30DayAbs starts with a digit

2 starts with a digit

pork&beans `&' is illegal

private reserved name

C-3PO `-' is illegal

Primitive Data Types

- Java's basic data types:
 - Integer Types:
 - **byte** 1 byte Range: -128 to +127
 - **short** 2 bytes Range: roughly -32 thousand to +32 thousand
 - **int** 4 bytes Range: roughly -2 billion to +2 billion
 - **long** 8 bytes Range: Huge!
 - Floating-Point Types (for real numbers)
 - **float** 4 bytes Roughly 7 digits of precision
 - **double** 8 bytes Roughly 15 digits of precision
 - Other types:
 - **boolean** 1 byte {true, false} (Used in logic expressions and conditions)
 - **char** 2 bytes A single (Unicode) character
 - String is not a primitive data type (they are objects)

Numeric Constants (Literals)

- Specifying constants: (also called **literals**) for primitive data types.

Integer Types:

byte }
short } optional sign and digits (0-9): 12 -1 +234 0 1234567
int
long Same as above, but followed by 'L' or 'l': -1394382953L

Floating-Point Types:

double Two allowable forms:

Avoid this lowercase L. It looks
too much like the digit '1'

Decimal notation: 3.14159 -234.421 0.0042 -43.0

Scientific notation: (use E or e for base 10 exponent)

3.145E5 = $3.145 \times 10^5 = 314500.0$

1834.23e-6 = $1834.23 \times 10^{-6} = 0.00183423$

float Same as double, but followed by 'f' or 'F': 3.14159F -43.2f

Note: By default, integer constants are **int**, unless 'L'/'l' is used to indicate they are **long**. Floating constants are **double**, unless 'F'/'f' is used to indicate they are **float**.

Character and String Constants

- **char constants:** Single character enclosed in single quotes ('...') including:
 - **letters and digits:** 'A', 'B', 'C', ..., 'a', 'b', 'c', ..., '0', '1', ..., '9'
 - **punctuation symbols:** '*', '#', '@', '\$' (except single quote and backslash '\')
 - **escape sequences:** (see below)
- **String constants:** Zero or more characters enclosed in double quotes ("...")
 - (same as above, but may not include a double quote or backslash)
- **Escape sequences:** Allows us to include single/double quotes and other special characters:

\"	double quote	\n	new-line character (start a new line)
\'	single quote	\t	tab character
\\	backslash		

- **Examples:** `char x = '\''` → (x contains a single quote)
`"\"Hi there!\""` → "Hi there!"
`"C:\\WINDOWS"` → C:\\WINDOWS
`System.out.println("Line 1\nLine 2")` prints

Line 1
Line 2

Data Types and Variables

- Java → Strongly-type language
- Strong Type Checking → Java checks that all expressions involve **compatible** types

• int x,y;	// x and y are integer variables
• double d;	// d is a double variable
• String s;	// s is a string variable
• boolean b;	// b is a boolean variable
• char c;	// c is a character variable
• x = 7;	// legal (assigns the value 7 to x)
• b = true;	// legal (assigns the value true to b)
• c = '#';	// legal (assigns character # to c)
• s = "cat" + "bert";	// legal (assigns the value "catbert" to s)
• d = x - 3;	// legal (assigns the integer value $7 - 3 = 4$ to double d)
• b = 5;	// illegal! (cannot assign int to boolean)
• y = x + b;	// illegal! (cannot add int and boolean)
• c = x;	// illegal! (cannot assign int to char)

Numeric Operators

- **Arithmetic Operators:**

- Unary negation: $-x$
- Multiplication/Division: $x * y$ x / y
 - Division between integer types **truncates** to integer: $23 / 4 \rightarrow 5$
 - $x \% y$ returns the **remainder** of x divided by y : $23 \% 4 \rightarrow 3$
 - Division with real types yields a real result: $23.0 / 4.0 \rightarrow 5.75$
- Addition/Subtraction: $x + y$ $x - y$

- **Comparison Operators:**

- Equality/Inequality: $x == y$ $x != y$
- Less than/Greater than: $x < y$ $x > y$
- Less than or equal/Greater than or equal: $x <= y$ $x >= y$

- These comparison operators return a **boolean** value: **true** or **false**.

Common String Operators

- String Concatenation: The '+' operator **concatenates** (joins) two strings.
"von" + "Wienerschnitzel" → "vonWienerschnitzel"

Note: Concatenation does
not add any space

- When a string is concatenated with another type, the other type is first evaluated and **converted** into its string representation

$(8.4) + \text{"degrees"} \rightarrow \text{"32degrees"}$

$(1 + 2) + \text{"5"} \rightarrow \text{"35"}$

- String Comparison: Strings should not be compared using the above operators ($==$, $<=$, $<$, etc). Let **s** and **t** be strings.

- s.equals(t)** → returns true if s equals t
- s.length()** → returns length
- s.compareTo(t)** → compares strings **lexicographically** (dictionary order)
 - result < 0 if s is less than t
 - result == 0 if s is equal to t
 - result > 0 if s is greater than t

References

- This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane, Dept of Computer Science, University of Maryland College Park



Be strong, you
never know who
you are inspiring

Operators

What is an operator?

- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.
- Operators with **one operand** are called **unary** operators
- Operators with two operands are called **binary** operators
- Operators with three operands are called **ternary** operator

Operators available in java

- Assignment operators
- Arithmatic operators
- **InstanceOf operator**
- Relational operators
- Conditional operators
- **Logical operators**

Assignment operator

- The = is called the assignment operator.
- The = operator is used to assign the value present to its right to the operand present to its left.
- eg.,

```
int a=10, b;  
b=a;
```

Arithmatic operators

The basic arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

String concatenation operator

- The + sign can also be used to concatenate two strings.
- Example
- String a ="hello", b="world";
- System.out.println(a + b) will result as **helloworld**

Increment or Decrement operators

- The `++` is the increment operator.
- The `--` is the decrement operator.
- There are two types based on the position of the operator with the operand.
- Postfix : `++` or `--` to the right of the operand
- Prefix : `++` or `--` to the left of the operand

Compound assignment operators

- The most commonly used compound assignment operators are

✓ `+=`

✓ `-=`

✓ `*=`

✓ `/=`

`:=`

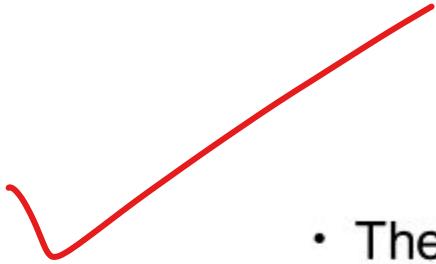
`^=`



Relational operators

- The list of relational operators in java are

<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to



Conditional operator

- The conditional operator is the **ternary operator**.
- The syntax of conditional operator is
`x=(boolean expression)?true:false ;`

InstanceOf operator

- The **instanceof** operator is used for **object reference variables** only.
- It can be used to check whether an object is of a particular type.
- The instanceof operator returns either true or false.
- Example:

```
Classname b = new Classname();
boolean a = b instanceof Classname;
```

The instanceof operator in Java is used to test whether an object is an instance of a specified class or interface. It is also known as the type comparison operator because it compares the instance with a type and returns a boolean value (true or false)

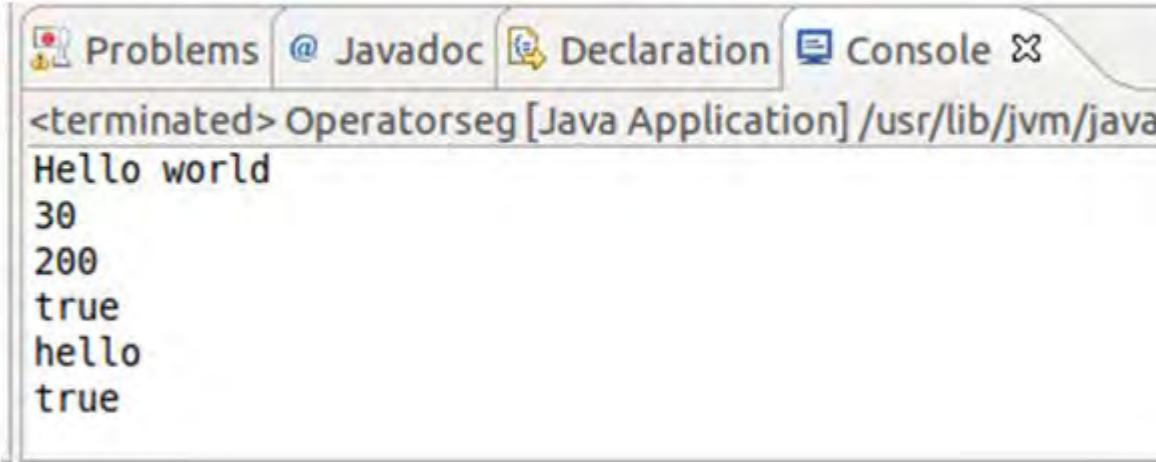
Logical operators

- Bitwise operators
 - &
 - |
 - ^ **XOR**
- Shorthand operators
 - && - logical AND
 - || - logical OR

Sample code for operators

```
package day7;
public class Operatorseg
{
    public static void main(String[] args)
    {
        int a,b,c;
        String str1,str2;
        //assignment operators
        a=10;b=20;c=0;
        str1="Hello";str2="world";
        //Arithmetic operations, concatenation operation,compound assignment
        System.out.println(str1+" "+str2);
        c=a+b;System.out.println(c);
        a*=b;System.out.println(a);
        //relational operators
        boolean x = a>b;System.out.println(x);
        //logical operators
        if(a>b&&b<c)
            System.out.println("hello");
        //conditional operator
        System.out.println(a>b?"true":"false");
    }
}
```

Sample output



The screenshot shows a software interface with a toolbar at the top containing five tabs: Problems, @ Javadoc, Declaration, Console, and a close button. The 'Console' tab is currently selected, indicated by a blue border around its tab area. Below the tabs, the console window displays the output of a Java application named 'Operatorseg'. The output consists of several lines of text: 'Hello world', '30', '200', 'true', 'hello', and 'true'. The text is displayed in black font on a white background.

```
<terminated> Operatorseg [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java -jar Operatorseg.jar
Hello world
30
200
true
hello
true
```

“ If you want to fly, give up everything
that weighs you down. ”

Buddha

Control Statements

Control Statement

✓ Java's control statements can be put into following categories:

- ✓ Sequence
- ✓ Selection statement
- ✓ Iteration statement
- ✓ Jump statement

✓ Three selection statements:

1. if statement
2. switch statement
3. conditional operator statement

The if Statement

- ✓ The *if statement* has the following syntax:

if is a Java reserved word

The *condition* must be a boolean expression.
It must evaluate to either true or false.

```
if (condition)
    statement;
    Statement x;
```

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

The if-else Statement

- ✓ An *else clause* can be added to an if statement to make an *if-else statement*

```
if ( condition )
    statement1;
else
    statement2;
Statement 3;
```

If the *condition* is true, *statement1* is executed; if the condition is false, *statement2* is executed

One or the other will be executed, but not both

Nested if....Else Statements

- ✓ The if..else statement can be contained in another if or else statement.

```
if (test condition1)
{
    if (test condition2)
        statement-1;
    else
        statement-2;
}
else
    statement-3;

statement-x;
```

Nested if....Else Statements

- ✓ An else clause is matched to the last unmatched if (no matter what the indentation implies!)

- ✓ Example:

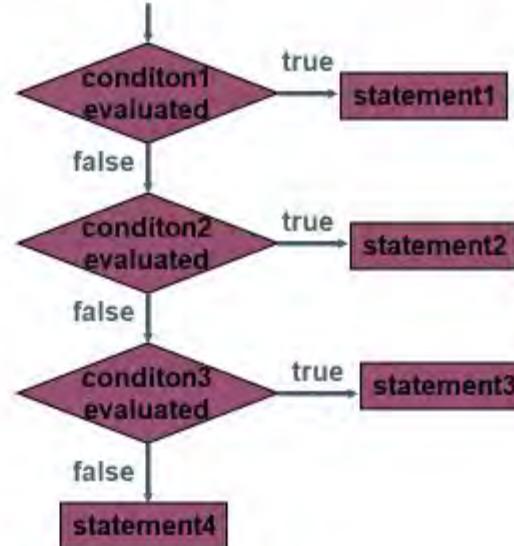
```
if(female)
    if(bal>5000)
        bon = 0.05 * bal;
    else
        bon = 0.02 * bal;
    bal = bal + bon;
```

- ✓ Braces can be used to specify the if statement to which an else clause belongs

The if-else-if ladder

- ✓ Sometime you want to select one option from several alternatives

```
if (condition1)
    statement1;
else if (condition2)
    statement2;
else if (condition3)
    statement3;
else
    statement4;
```



The switch Statement

- ✓ The *switch statement* provides another means to decide which statement to execute next
- ✓ The switch statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- ✓ The expression of a switch statement must result in an *integral type* (byte, short, int, char etc)

Note: JDK 7 allows expression can be of type **String**.

- ✓ The flow of control transfers to statement associated with the first value that matches

The switch Statement

- ✓ The general syntax of a switch statement is:

```
switch (expression) {  
    case value1:  
        statement-list1  
        break;  
    case value2:  
        statement-list2  
        break;  
    case value3:  
        statement-list3  
        break;  
    case default:  
        statement-list4  
}
```

switch
and
case
are
reserved
words

If *expression*
matches *value2*,
control jumps
from here

The switch Statement

- A break statement causes control to transfer to the end of the switch statement
- If a break statement is not used, the flow of control will continue into the next case
- Sometimes this can be appropriate, but usually we want to execute only the statements associated with one case

The switch Statement

- ✓ A switch statement can have an optional *default case*
- ✓ The default case has no associated value and simply uses the reserved word `default`
- ✓ If the default case is present, control will transfer to it if no other case value matches
- ✓ If there is no default case, and no other value matches, control falls through to the statement after the switch

Switch example

```
char letter = 'b';
```

```
switch (letter) {  
    case 'a':  
        System.out.println("A");  
        break;  
    case 'b':  
        System.out.println("B");  
        break;  
    case 'c':  
        System.out.println("C");  
        break;  
    case 'd':  
        System.out.println("D");  
        break;  
    default:  
        System.out.println("?");  
}
```

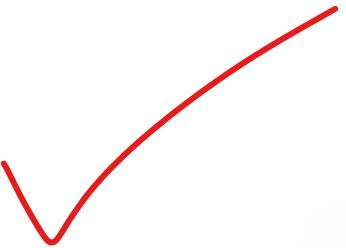
```
char letter = 'b';
```

```
switch (letter) {  
    case 'a':  
        System.out.println("A");  
    case 'b':  
        System.out.println("B");  
    case 'c':  
        System.out.println("C");  
        break;  
    case 'd':  
        System.out.println("D");  
        break;  
    default:  
        System.out.println("?");  
}
```

B

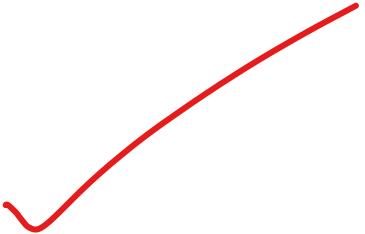
B

C



The Conditional Operator

- ✓ Java has a *conditional operator* that evaluates a boolean condition that determines which of two other expressions is evaluated
- ✓ The result of the chosen expression is the result of the entire conditional operator
- ✓ Its syntax is:
 - ✓ $condition \ ? \ expression1 \ : \ expression2$
- ✓ If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated



The Conditional Operator

- ✓ The conditional operator is similar to an if-else statement, except that it forms an expression that returns a value
- ✓ For example:
 - ✓

```
larger = ((num1 > num2) ? num1 : num2);
```
 - ✓

```
if (num1 > num2)
```



```
    larger = num1;
```



```
else
```



```
    larger = num2;
```
- ✓ The conditional operator is *ternary* because it requires three operands

Iteration Statements

- ✓ *Iteration statements* allow us to execute a statement multiple times until a termination condition is met.
- ✓ Often they are referred to as *loops*
- ✓ Java has three kinds of iteration statements:
 - ✓ the *while loop*
 - ✓ the *do-while loop*
 - ✓ the *for loop*

The while Statement

- ✓ The *while statement* has the following syntax:

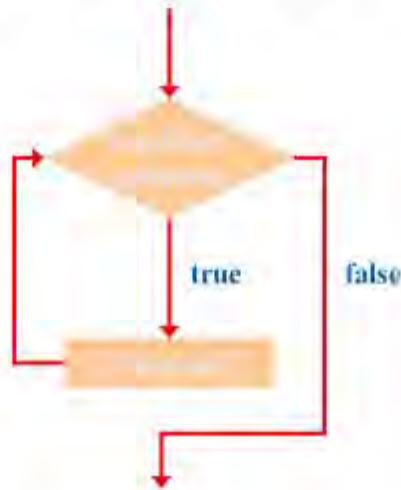
while is a reserved word

while (*condition*)
statement;

If the *condition* is true, the *statement* is executed.
Then the *condition* is evaluated again.

The *statement* is executed repeatedly until
the *condition* becomes false.

Logic of a while Loop



Note that if the condition of a while statement is false initially, the statement is never executed. Therefore, the body of a while loop will execute zero or more times

while Loop Example

```
int LIMIT = 5;  
int count = 1;  
  
while (count <= LIMIT) {  
  
    System.out.println(count);  
    count += 1;  
}  
  
--Null statements are valid in java.
```

Output:

```
1  
2  
3  
4  
5
```

Nested Loops

- ✓ Similar to nested if statements, loops can be nested as well
- ✓ That is, the body of a loop can contain another loop
- ✓ Each time through the outer loop, the inner loop goes through its full set of iterations

The do-while Statement

- ✓ The *do-while statement* has the following syntax:

do and
while are
reserved
words

```
do{  
    statement;  
} while (condition);
```

The *statement* is executed once initially,
and then the *condition* is evaluated

The *statement* is executed repeatedly
until the *condition* becomes false

do-while Example

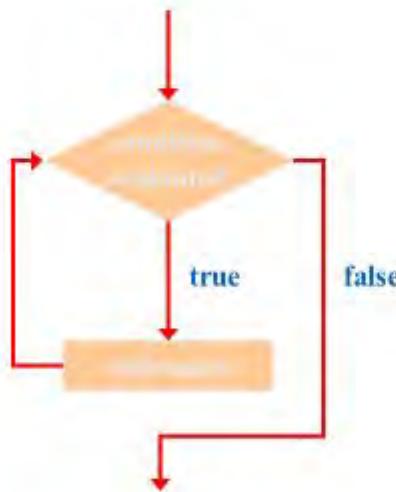
```
int LIMIT = 5;  
int count = 1;  
  
do {  
    System.out.println(count);  
    count += 1;  
} while (count <= LIMIT);
```

Output:

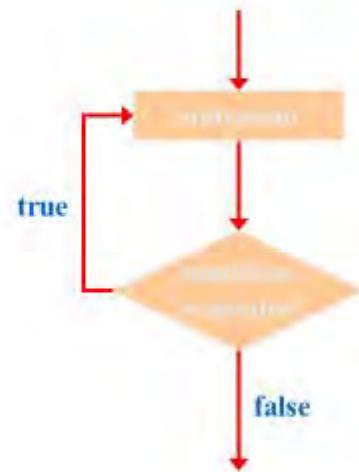
```
1  
2  
3  
4  
5
```

Comparing while and do-while

while loop



Do-while loop



The for Statement

- ✓ The *for statement* has the following syntax:

Reserved word The *initialization* is executed once before the loop begins The *statement* is executed until the *condition* becomes false

```
for (initialization; condition; increment)  
    statement;
```

The *increment* portion is executed at the end of each iteration

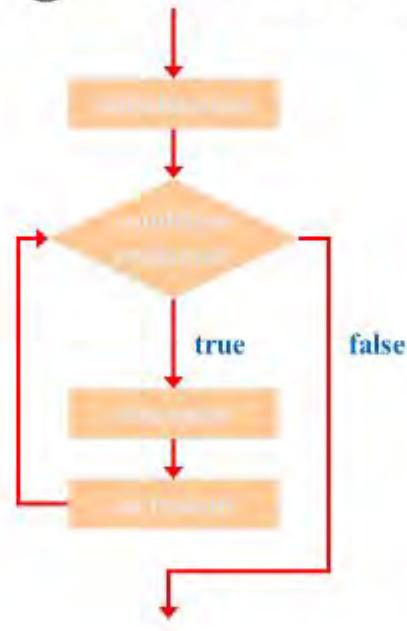
The *condition-statement-increment* cycle is executed repeatedly

The for Statement

- ✓ A for loop is functionally equivalent to the following while loop structure:

```
initialization;
while (condition) {
    statement;
    increment;
}
```

Logic of a for loop



for Example

```
int LIMIT = 5;  
  
for (int count = 1; count <= LIMIT; count++) {  
    System.out.println(count);  
}
```

Output:

1
2
3
4
5

The for Statement

- ✓ Each expression in the header of a for loop is optional
 - ✓ If the *initialization* is left out, no initialization is performed
 - ✓ If the *condition* is left out, it is always considered to be true, and therefore creates an infinite loop
 - ✓ If the *increment* is left out, no increment operation is performed
- ✓ Both semi-colons are always required in the for loop header

Jump Statements-Break

- The break statement has three uses:
 - It terminates a statement sequence in a switch statement.
 - It can be used to exit a loop.
 - It can be used as a civilized form of goto.
- **Civilized Form of Goto Statement**
 - `break label;`
where *label* is the name of the block enclosing the break statement.
 - Labeled break is used to transfer control from a set of nested blocks.

goto

Example-break statement

```
class test_break1
{
    public static void main(String args[])
    {
        boolean t =true;
        first:{  
            second:{  
                third:{  
                    System.out.println("Before the break.");
                    if(t)
                        break second;
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }
            System.out.println("This is after second block.");
        }
    }
}
```

Example-break statement

```
class test_break2
{
    public static void main(String args[])
    {
        one: for(int i=0;i<3;i++)
        {
            System.out.print("Pass: "+i+": ");
        }

        for(int j=0; j<100;j++)
        {
            if(j==10) break one; //Wrong
            System.out.print(j+" ");
        }
    }
}
```

Jump statement-continue

- Continue statement is used to run a loop but stop processing the remainder of the code in its body for a particular iteration.
- In **while** and **do-while** loop, a continue statement causes control to be transferred directly to the conditional expression. In a **for** loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

Example-Continue statement

```
class test_continue                                Output:  
{    public static void main(String args[])  
{        outer: for(int i=0; i<4;i++){  
            for(int j=0;j<4;j++) {  
                if(j>i) {  
                    System.out.println();  
                    continue outer;  
                }  
                System.out.print(" "+(i*j));  
            }  
            System.out.println();  
        }  
    }  
}
```

0
0 1
0 2 4
0 3 6 9

Jump Statement-return

```
class test{
    public static void main (String args[])
    {
        boolean t =true;
        System.out.println("Before the return");
        if(t) return;

        System.out.println("This won't execute.");
    }
}
```

Nested Loop

- Read and practice by yourself.

“

*The day is what you
make it! So why not
make it a great one?*

STEVE SCHULTE

AYERATECH 1999

Classes and Objects in Java

Contents

- Introduce to classes and objects in Java.
- Understand how some of the OO concepts learnt so far are supported in Java.
- Understand important features in Java classes.

Introduction

- Java is a true OO language and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the "state" and "behaviour" of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them. They provide a convenient method for packaging a group of logically related data items and functions that work on them.
- A class essentially serves as a template for an object and behaves like a basic data type "int". It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as encapsulation, inheritance, and polymorphism.

- A class defines a new data type. This new data type is used to create objects of that type.
- A class is a template for an object and an object is an instance of a class.
- Class is user defined type.

Example:

Colored points on the screen

What data goes into making one?

- Coordinates
- Color

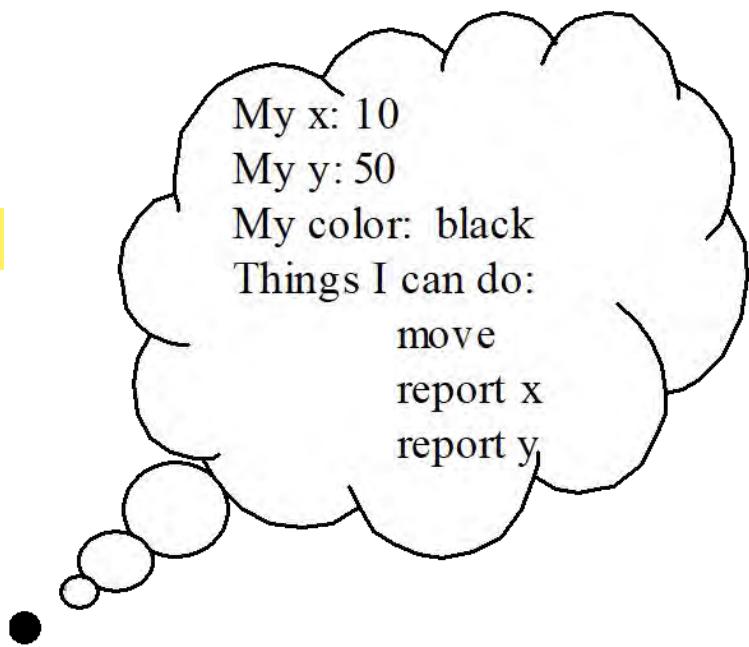
What should a point be able to do?

- Move itself
- Report its position



JAVA Terminology

- ✓ Each point is an *object*
- ✓ Each includes three *data fields*
- ✓ Each has three *methods*
- ✓ Each is an *instance* of the same *class*

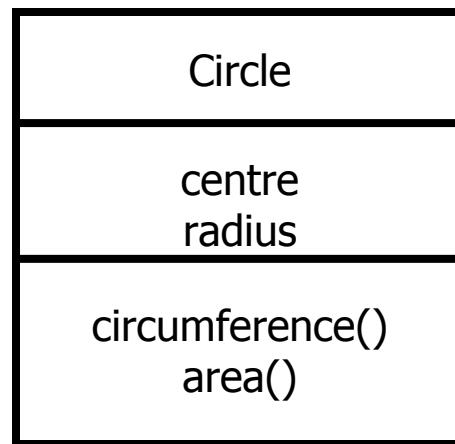


```
class Point {  
    int xCoord;  
    int yCoord;  
    Point() {  
        xCoord = 0;  
        yCoord = 0;  
    }  
    int currentX() {  
        return xCoord;  
    }  
    int currentY() {  
        return yCoord;  
    }  
    void move (int newXCoord, int newYCoord) {  
  
        xCoord = newXCoord;  
        yCoord = newYCoord;  
    }  
}
```

```
class PointProcess{  
    public static void main(String args[]){  
        Point P=new Point();  
        P.move(3,4);  
        System.out.println("X is: "+P.currentX());  
        System.out.println("Y is: "+P.currentY());  
    }  
}
```

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



Classes

- A class is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a **class definition**:

```
class ClassName [extends  
  SuperClassName]  
{  
    [fields declaration]  
    [methods declaration]  
}
```

- Bare bone class – no fields, no methods

```
public class Circle {  
    // my circle class  
}
```

Adding Fields: Class Circle with fields

- Add *fields*

```
public class Circle {  
    public double x, y; // centre coordinate  
    public double r;   // radius of the circle  
}
```

- The fields (data) are also called the *instance* variables.

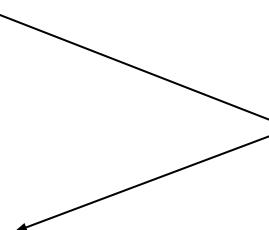
Adding Methods

- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
    Method-body;
}
```

Adding Methods to Class Circle

```
public class Circle {  
  
    public double x, y; // centre of the circle  
    public double r; // radius of circle  
  
    //Methods to return circumference and area  
    public double circumference() {  
        return 2*3.14*r;  
    }  
    public double area() {  
        return 3.14 * r * r;  
    }  
}
```



Method Body

Data Abstraction

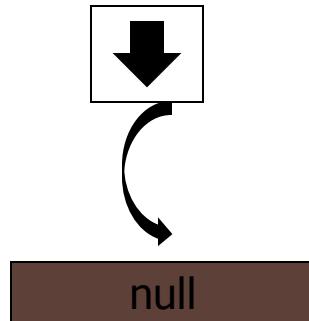
- Declare the Circle class, have created a new data type – Data Abstraction
- Can define variables (objects) of that type:

```
Circle aCircle;  
Circle bCircle;
```

Class of Circle cont.

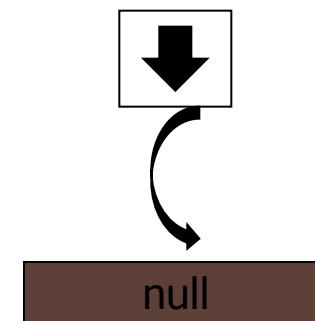
- aCircle, bCircle simply refers to a Circle object, not an object itself.

aCircle



Points to nothing (Null Reference)

bCircle



Points to nothing (Null Reference)

Creating objects of a class

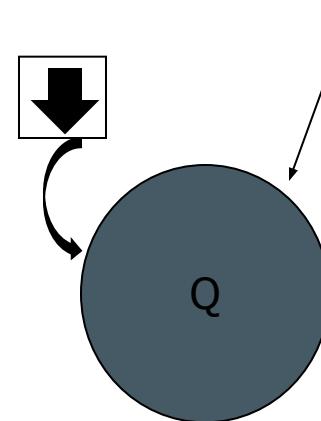
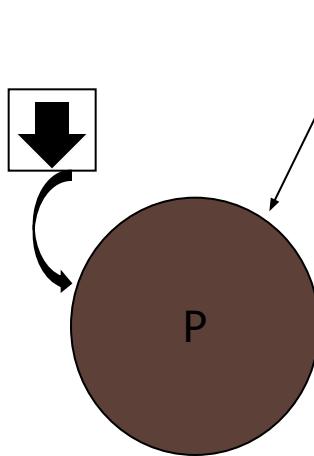
- Objects are created dynamically using the *new* keyword.
- aCircle and bCircle refer to Circle objects

not

Circle aCircle = new Circle();

aCircle = new Circle() ;

bCircle = new Circle() ;



Creating objects of a class

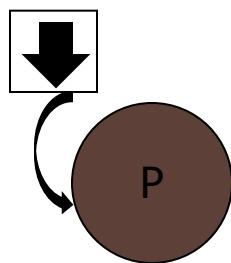
```
aCircle = new Circle();  
bCircle = new Circle() ;  
  
bCircle = aCircle;
```

Creating objects of a class

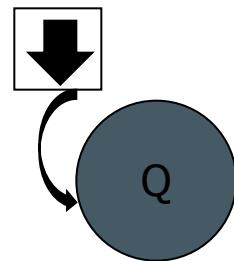
```
aCircle = new Circle();  
bCircle = new Circle() ;  
  
bCircle = aCircle;
```

Before Assignment

aCircle

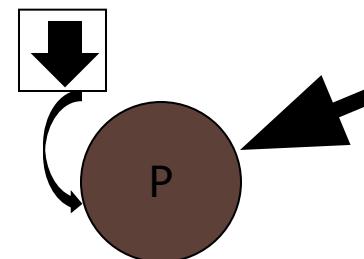


bCircle

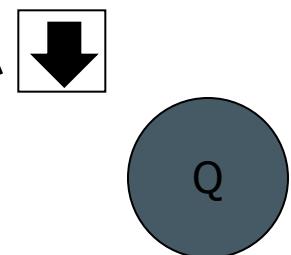


After Assignment

aCircle



bCircle



Accessing Object/Circle Data

- Similar to C syntax for accessing data defined in a structure.

ObjectName.VariableName
ObjectName.MethodName(parameter-list)

```
Circle aCircle = new Circle();
```

```
aCircle.x = 2.0 // initialize center and radius  
aCircle.y = 2.0  
aCircle.r = 1.0
```

Executing Methods in Object/Circle

- Using Object Methods:

sent 'message' to aCircle

```
Circle aCircle = new Circle();
```

```
double area;  
aCircle.r = 1.0;  
area = aCircle.area();
```

Using Circle Class

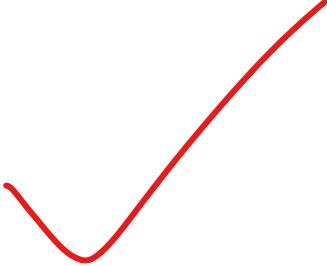
```
// Circle.java: Contains both Circle class and its user class
//Add Circle class code here
class MyMain
{
    public static void main(String args[])
    {
        Circle aCircle; // creating reference
        aCircle = new Circle(); // creating object
        aCircle.x = 10; // assigning value to data field
        aCircle.y = 20;
        aCircle.r = 5;
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.r+" Area="+area);
        System.out.println("Radius="+aCircle.r+" Circumference =" +circumf);
    }
}
```

Radius=5.0 Area=78.5

Radius=5.0 Circumference =31.400000000000002

References

- An reference variable holds the memory address.
- It is similar to the pointer.
- Key difference- cannot manipulate as pointer- It cannot point any arbitrary memory location or it cannot be manipulated like an integer.

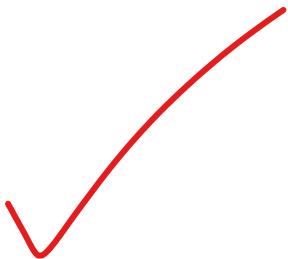


The null reference

- An object reference variable that does not currently point to an object is called a *null reference*
- The reserved word `null` can be used to explicitly set a null reference:

```
name = null;
```

- An object reference variable declared at the class level is automatically initialized to null
- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used

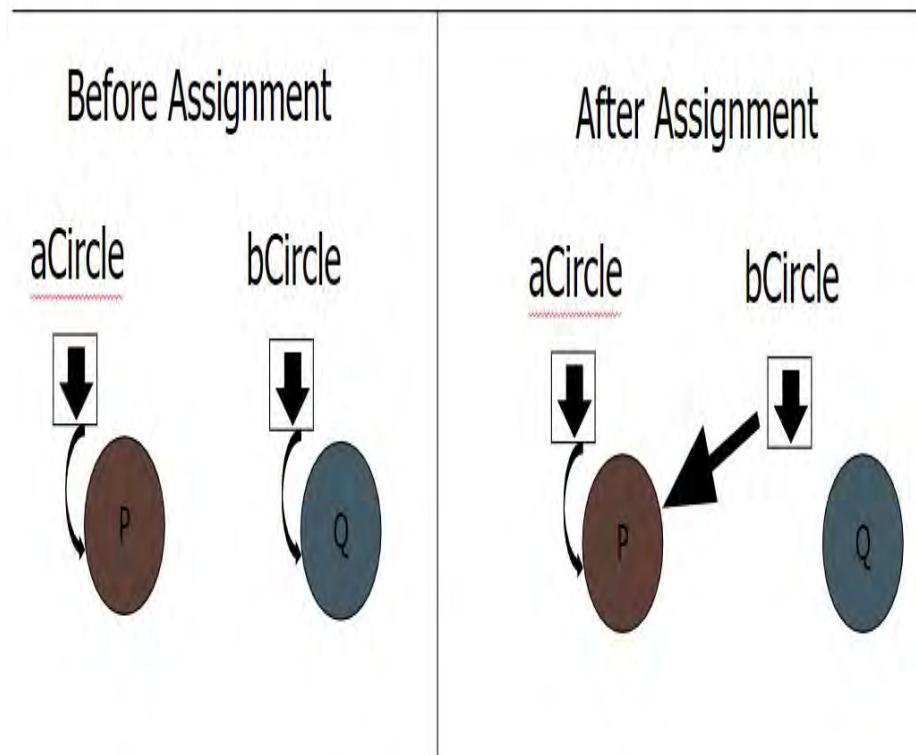


Automatic garbage collection

- Java handles deallocation automatically.
- Garbage collection system automatically deallocates space for the programmer.
- It runs periodically.
- It automatically releases space for an object, when no reference to the object exists.

Automatic garbage collection

- The object  does not have a reference and cannot be used in future.
- The object becomes a candidate for automatic garbage collection.
- Java automatically collects garbage periodically and releases the memory used to be used in the future.



The this keyword

- The this keyword can be used inside any method to refer to the current object.
- Example:

```
Box (double w, double h, double d)
```

```
{
```

```
    this.width = w;
```

```
    this.height = h;
```

```
    this.depth = d;
```

```
}
```

- Application-Instance Variable Hiding

```
Box (double width, double height, double depth)
```

```
{
```

```
    this. width = width;
```

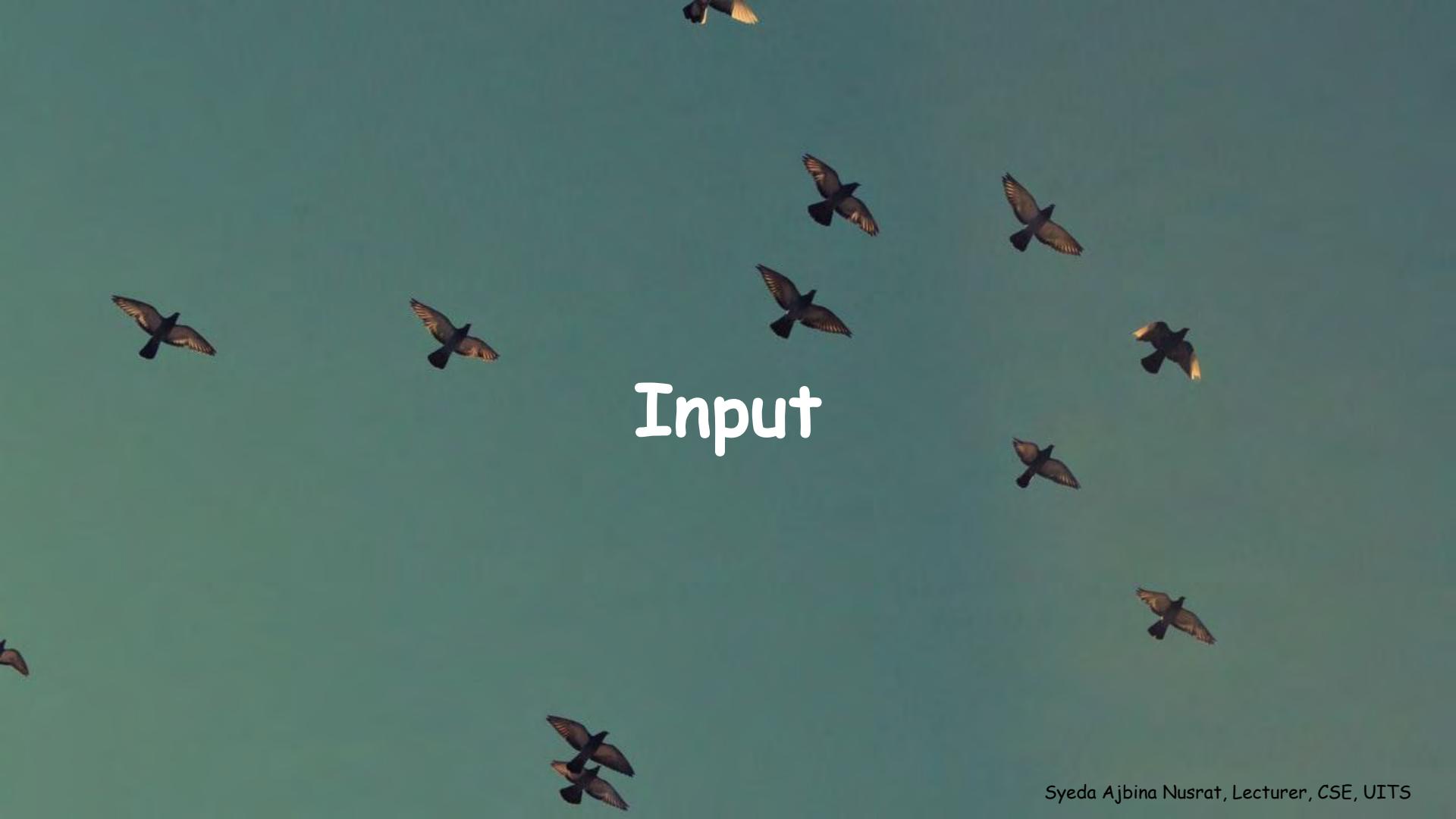
```
    this. heighth = height;
```

```
    this. depth = depth;
```

```
}
```

ALL YOUR DREAMS CAN
COME TRUE IF YOU HAVE THE
COURAGE
TO PURSUE THEM

Walt Disney



Input

Introduction

- You've already seen that output can be displayed to the user using the subroutine **System.out.print**. This subroutine is part of a pre-defined object called **System.out**. The purpose of this object is precisely to display output to the user.

- There is also a corresponding object called **System.in** that exists to read data input by the user, but it provides only very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.
- To use System.in, a Java Scanner is used that will get the inputs.

Scanner Creation/Declaration

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

↑ ↑
Reference variable Object Instantiation

- Scanner is a class which must be instantiated before it can be used. In other words, you must make a new Scanner if you want to use Scanner. A reference must be used to store the location in memory of the Scanner object created.
- System.in is the parameter passed to the Scanner constructor so that Java will know to connect the new Scanner to the keyboard. keyboard is a reference that will store the location of newly created Scanner object.

Scanner Imports

- In order to use Scanner, you must import `java.util.Scanner`.

```
import java.util.Scanner;
```

Scanner Methods

NAME	USE
nextInt();	Returns the next integer value
nextDouble();	Returns the next double value
nextFloat();	Returns the next float value
nextLong();	Returns the next long value
nextShort();	Returns the next short value
next();	Returns the next one word String value
nextLine();	Returns the next multiple word String value

Reading in Integers

```
Scanner keyboard = new Scanner(System.in);  
System.out.print(" Enter Value");  
int num = keyboard.nextInt();
```

- The `nextInt()` method is used to tell a `Scanner` object to retrieve the next integer value entered.
- In the example, the next integer typed in on the keyboard would be read in and placed in the integer variable `num`.
- `nextInt()` will read up to the first whitespace value entered.

Integers

```
int num = keyboard.nextInt();
```

identifier
↓
↑ data type ↑ reference variable ↑ method call

- The `nextInt()` method will read in the next integer. If a non-integer value is encountered such as a decimal value, the result will be run-time exception.
- `keyboard` is a reference that refers to a `Scanner` object.

Strings

```
String word= keyboard.nextLine();
```

↑ ↑ ↑
identifier reference variable method call
data type

- The `nextLine()` method will read in the next text value entered. A numeric or non-numeric text value will be accepted.
- In the example, the next text entered on the keyboard would be read in and placed in variable `word`.
- The `nextLine()` method would read up to the first whitespace encountered. Whitespace would be any space, any tab, or any enter key.

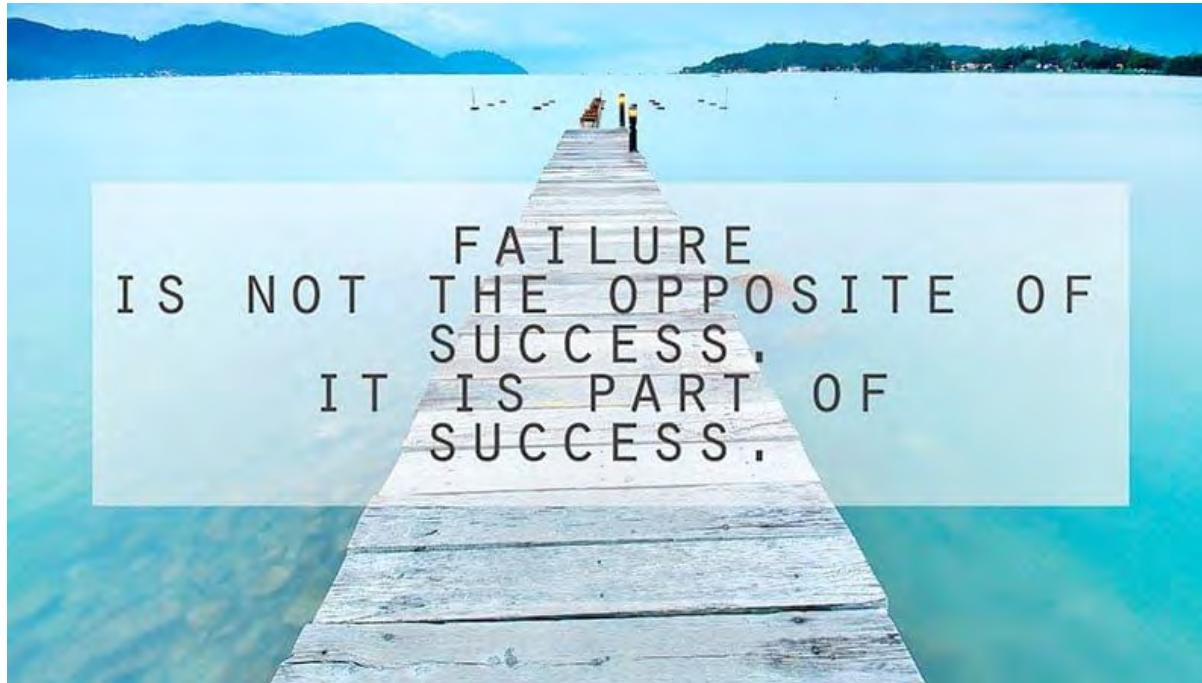
Prompts

```
System.out.print("Enter an integer: ");
```

- When performing input operations, it is a must to use prompts. A prompt is a way of indicating to a user what type of data to enter.
- The prompt above indicates that an integer value is expected.

Reference

Prepared by:
Jean Michael Castor



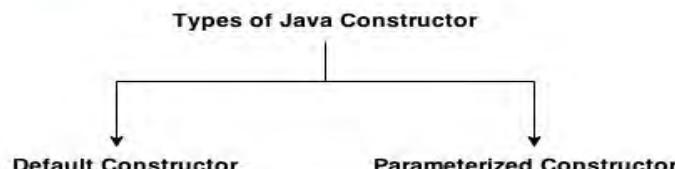
Constructor

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the **same name as the class name**.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different **signature** (i.e., different **input arguments syntax**).

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



```
public class Counter { int  
    CounterIndex;  
  
    // Constructor  
public Counter()  
{  
    CounterIndex = 0;  
}  
  
//Methods to update or access counter  
public void increase()  
{  
    CounterIndex = CounterIndex + 1;  
}  
public void decrease()  
{  
    CounterIndex = CounterIndex - 1;  
}  
int getCounterIndex()  
{  
    return CounterIndex;  
}
```

```
class MyClass {  
    public static void main(String args[])  
    {  
        Counter counter1 = new Counter();  
        counter1.increase();  
        int a = counter1.getCounterIndex();  
        counter1.increase();  
        int b = counter1.getCounterIndex();  
        if (a > b)  
            counter1.increase();  
        else  
            counter1.decrease();  
  
        System.out.println(counter1.getCounterIndex());  
    }  
}
```

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

```
class Student4{  
    int id;  
    String name;  
  
    Student4(int i, String n){  
        id = i;  
        name = n;  
    }  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        Student4 s1 = new Student4(111, "Karan");  
        Student4 s2 = new Student4(222, "Aryan");  
        s1.display();  
        s2.display();  
    }  
}
```

Constructor Overloading

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student5{  
    int id;  
    String name;  
    int age;  
    Student5(int i,String n){  
        id = i;  
        name = n;  
    }  
    Student5(int i,String n,int a){  
        id = i;  
        name = n;  
        age=a;  
    }  
    void display(){System.out.println(id+" "+name+" "+age);}  
  
    public static void main(String args[]){  
        Student5 s1 = new Student5(111,"Karan");  
        Student5 s2 = new Student5(222,"Aryan",25);  
        s1.display();  
        s2.display();  
    }  
}
```

Multiple Constructors

- Sometimes want to initialize in a number of different ways, depending on circumstance.
- This can be supported by having multiple constructors having different input arguments.

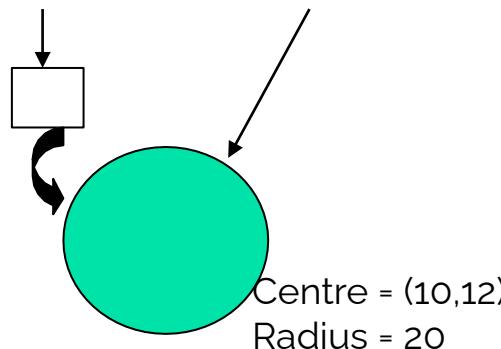
Multiple Constructors

```
public class Circle {  
    public double x,y,r; //instance variables  
    // Constructors  
    public Circle(double centreX, double centreY, double radius) { x = centreX; y =  
        centreY; r = radius;  
    }  
    public Circle(double radius) { x=0; y=0; r = radius; }  
    public Circle() { x=0; y=0; r=1.0; }  
  
    //Methods to return circumference and area  
    public double circumference() { return 2*3.14*r; }  
    public double area() { return 3.14 * r * r; }  
}
```

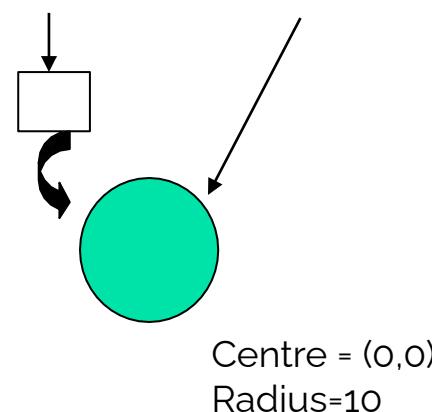
Initializing with constructors

```
public class TestCircles {  
  
    public static void main(String args[]){  
        Circle circleA = new Circle( 10.0, 12.0, 20.0); Circle circleB = new  
        Circle(10.0);  
        Circle circleC = new Circle();  
    }  
}
```

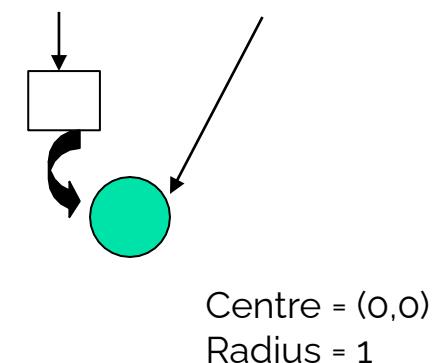
circleA = new Circle(10, 12, 20)



circleB = new Circle(10)



circleC = new Circle()



Java Copy Constructor

- There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.
- There are many ways to copy the values of one object into another in java. They are:
 - By constructor
 - By assigning the values of one object into another
 - By clone() method of Object class

```
class Student6{  
    int id;  
    String name;  
    Student6(int i, String n){  
        id = i;  
        name = n;  
    }  
  
    Student6(Student6 s){  
        id = s.id;  
        name = s.name;  
    }  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        Student6 s1 = new Student6(111, "Karan");  
        Student6 s2 = new Student6(s1);  
        s1.display();  
        s2.display();  
    }  
}
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{  
    int id;  
    String name;  
    Student7(int i, String n){  
        id = i;  
        name = n;  
    }  
    Student7(){  
    }  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        Student7 s1 = new Student7(111, "Karan");  
        Student7 s2 = new Student7();  
        s2.id=s1.id;  
        s2.name=s1.name;  
        s1.display();  
        s2.display();  
    }  
}
```

Difference between constructor and method in java

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Reference

Compiled by:
Vinod Kumar(Asst. prof.)

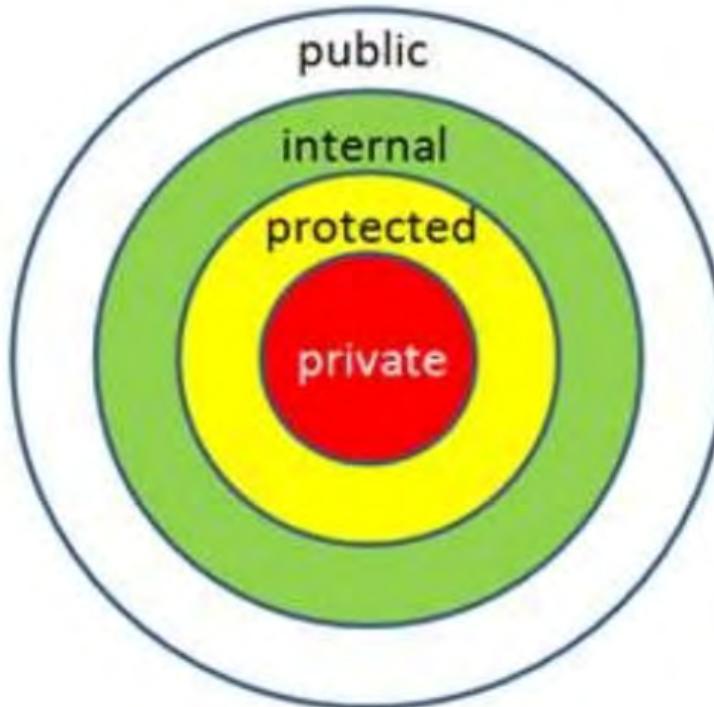


**“Not in doing what
you like, but in liking
what you do is the
secret of happiness.”**

—J.M. BARRIE

Access Modifiers

Access Modifiers in java



Access Modifiers in java

- There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
 - private
 - default
 - protected
 - public
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

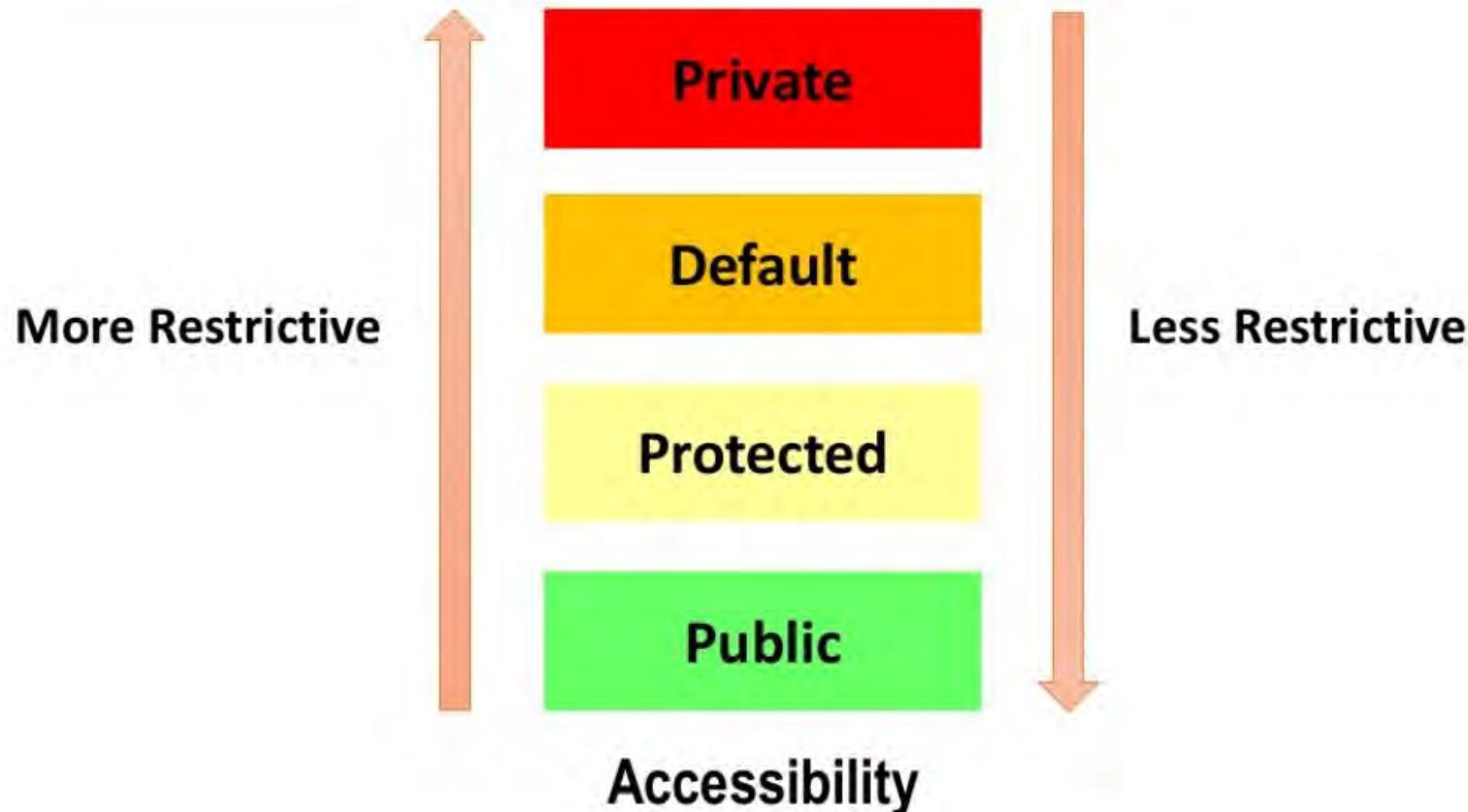
Access Modifiers in java

- **Access Control Modifiers**
- Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are –
 - Visible to the package, the default. No modifiers are needed.
 - Visible to the class only (private).
 - Visible to the world (public).
 - Visible to the package and all subclasses (protected).

Access Modifiers in java

- **Non-Access Modifiers**
- Java provides a number of non-access modifiers to achieve many other functionality.
 - The ***static*** modifier for creating class methods and variables.
 - The ***final*** modifier for finalizing the implementations of classes, methods, and variables.
 - The ***abstract*** modifier for creating abstract classes and methods.
 - The ***synchronized*** and ***volatile*** modifiers, which are used for threads.

Access Modifiers in java



Access Modifiers in java

• 1) private Access Modifier

- The private access modifier is accessible only within class.
- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Private Access Modifier

```
public class Test{  
    private int data=40;  
    public static void main(String args[]){  
        Test ex = new Test();  
        System.out.println("Data is: "+ex.data);  
    }  
}
```

Private Access Modifier

```
public class Test{  
    private int data=40;  
    public static void main(String args[]){  
        Test ex = new Test();  
        System.out.println("Data is: "+ex.data);  
    }  
}
```

Here the *data* variable of the Example class is private and this variable accessed from same class itself

Private Access Modifier

```
public class Test{  
    private int data=40;  
    public static void main(String args[]){  
        Test ex = new Test();  
        System.out.println("Data is: "+ex.data);  
    }  
}
```

Data variable is assess in the
same class where it is defined

Private Access Modifier

```
public class Test{  
    private int data=40;  
    public static void main(String args[]){  
        Test ex = new Test();  
        System.out.println("Data is: "+ex.data);  
    }  
}
```

Output is:

Data is: 40

Private Access Modifier

```
public class Test{  
    private int data;  
  
    public static void main(String args[]){  
        ex.data=40;  
        Test ex = new Test();  
        System.out.println("Data is: "+ex.data);  
    }  
}
```

Same Program as previous but single modification

Now we set the value of the variable here using the object of Example class

Output is:
Data is: 40

Private Access Modifier

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

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

Private Access Modifier

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

This is first class where we declare a private variable and define private method

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

Private Access Modifier

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

This is second class where we try to access private variable and method

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

Private Access Modifier

```
class A{  
    private int data=40; ← Instance variable  
    private void msg() {  
        System.out.println("Hello java");}  
    }  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println(obj.data); //Compile Time Error  
        obj.msg(); //Compile Time Error  
    }  
}
```

Private Access Modifier

```
class A{  
    private int data=40;  
    private void msg() {  
        System.out.println("Hello java");}  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println(obj.data); //Compile Time Error  
        obj.msg(); //Compile Time Error  
    }  
}
```

Private method is here

Private Access Modifier

```
class A{  
    private int data=40;  
    private void msg() {  
        System.out.println("Hello java");}  
    }  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println(obj.data); //Compile Time Error  
        obj.msg(); //Compile Time Error  
    }  
}
```

Here, the data variable of the A class is private, so there's no way for other classes to retrieve or set its value directly.

Private Access Modifier

- So, if we want to make this variable available to the outside world, we defined two public methods:
 - *getter()*, which returns the value of variable, and
 - *setter(parameter)*, which sets its value of the variable.

```
class A {  
    private int data;  
    public int getA() {  
        return this.data;  
    }                                returns  
    public void setA(int data) {  
        this.data=data;  
    }                                assigns  
}
```

Private Access Modifier

```
class A{  
    private int data;  
    public int getA() {  
        return this.data;  
    }  
    public void setA(int data) {  
        this.data=data; }  
    }  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Private Access Modifier

```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Private instance
variable

Private Access Modifier

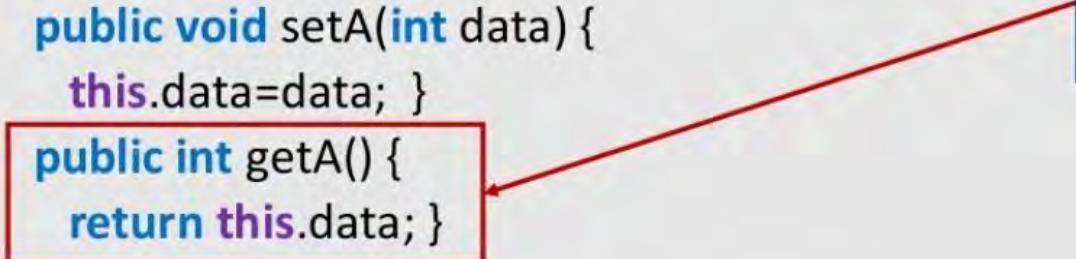
```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Public setter method

Private Access Modifier

```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Public getter method



Private Access Modifier

```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Here we set the value of the private variable

Private Access Modifier

```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        obj.setA(12);  
        System.out.println("Data is: "+obj.getA());  
    }  
}
```

Here we get the
value of the
private variable

Private Access Modifier

```
class A{  
    private int data;  
    public void setA(int data) {  
        this.data=data; }  
    public int getA() {  
        return this.data; }  
}
```

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

```
        A obj=new A();
```

```
        obj.setA(12);
```

```
        System.out.println("Data is: "+obj.getA());
```

```
}
```

Try to avoid using setter and getter methods

Because these methods make your data publicly

Accessing
possible

Output is:

Data is: 12

Role of Private Constructor

- The private modifier when applied to a constructor works in much the same way as when applied to a normal method or even an instance variable.
- Defining a constructor with the private modifier says that only the native class (as in the class in which the private constructor is defined) is allowed to create an instance of the class, and no other caller is permitted to do so.
- There are two possible reasons why one would want to use a private constructor – the first is that you don't want any objects of your class to be created at all, and the second is that you only want objects to be created internally – as in only created in your class.

Role of Private Constructor

- **1. Private constructors can be used in the singleton design pattern**
- Why would you want objects of your class to only be created internally?
- This could be done for any reason, but one possible reason is that you want to implement a singleton. A singleton is a design pattern that allows only one instance of your class to be created, and this can be accomplished by using a private constructor.

Role of Private Constructor

- **2. Private constructors can prevent creation of objects**
- The other possible reason for using a private constructor is to prevent object construction entirely. When would it make sense to do something like that? Of course, when creating an object doesn't make sense – and this occurs when the class only contains static members. And when a class contains only static members, those members can be accessed using only the class name – no instance of the class needs to be created.

Role of Private Constructor

- Java always provides a default, no-argument, public constructor if no programmer-defined constructor exists. Creating a private no-argument constructor essentially prevents the usage of that default constructor, thereby preventing a caller from creating an instance of the class. Note that the private constructor may even be empty.
- Let see an example in the next slide if we make any class constructor private, we cannot create the instance of that class from outside the class

Role of Private Constructor

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

Access Modifiers in java

• 2) public Access Modifier

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
- A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
- However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Note: About package we learn later in detail.. ☺

public Access Modifier

```
class A{  
    public int data=40;  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println("Data is: "+obj.data);  
    }  
}
```

public Access Modifier

```
class A{
```

```
    public int data=40;
```

```
}
```

public instance
variable and
accessible in other
classes

```
public class Test{
```

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

```
        A obj=new A();
```

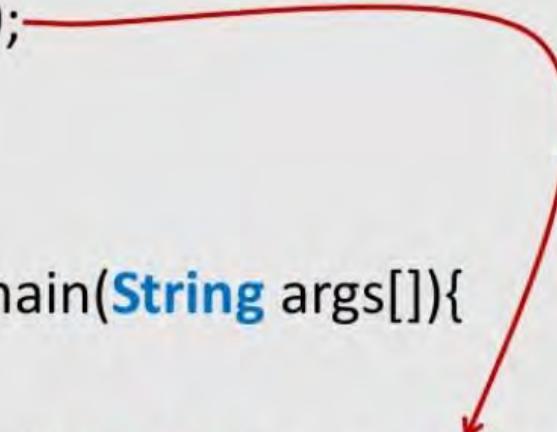
```
        System.out.println("Data is: "+obj.data);
```

```
    }
```

```
}
```

public Access Modifier

```
class A{  
    public int data=40;  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println("Data is: "+obj.data);  
    }  
}
```



Output is:
Data is: 40

Access Modifiers in java

- **3) protected access modifier**

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

protected Access Modifier

```
class A{  
    protected int data=40;  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println("Data is: "+obj.data);  
    }  
}
```

protected Access Modifier

```
class A{
```

```
    protected int data=40;
```

```
}
```

Protected instance
variable

```
public class Test{
```

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

```
        A obj=new A();
```

```
        System.out.println("Data is: "+obj.data);
```

```
    }
```

```
}
```

protected Access Modifier

```
class A{  
    protected int data=40;  
}  
  
public class Test{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println("Data is: "+obj.data);  
    }  
}
```

Access protected instance
variable in other class

Output is:
Data is: 40

protected Access Modifier

```
class AudioPlayer {  
    protected boolean openSpeaker(Speaker sp) {  
        // implementation details  
    }  
}  
  
class StreamingAudioPlayer {  
    boolean openSpeaker(Speaker sp) {  
        // implementation details  
    }  
}
```

Here, if we define `openSpeaker()` method as private, then it would not be accessible from any other class other than `AudioPlayer`. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used `protected` modifier.

Access Modifiers in java

• 4) default access modifier

- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.
- The default modifier is accessible only within package.

Access Modifiers in java

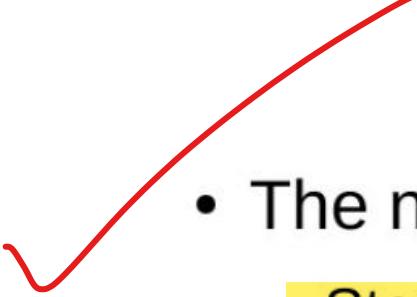
	public	private	protected	< unspecified >
class	allowed	not allowed	not allowed	allowed
constructor	allowed	allowed	allowed	allowed
variable	allowed	allowed	allowed	allowed
method	allowed	allowed	allowed	allowed

	class	subclass	package	outside
private	allowed	not allowed	not allowed	not allowed
protected	allowed	allowed	allowed	not allowed
public	allowed	allowed	allowed	allowed
< unspecified >	allowed	not allowed	allowed	not allowed

Reference

Notes by Adil Aslam

Non Access Modifiers

- 
- The non-access modifiers in java are
 - Static
 - Final
 - Abstract
 - Synchronized
 - Transient
 - volatile

Static modifier

- The static modifier can be used in
 - Variable and
 - methods

Static variable

- The static key word is used to create variables that will exist independently of any instances created for the class.
- Only one copy of the static variable exists regardless of the number of instances of the class.
- Static variables are also known as class variables.
- Local variables cannot be declared static.

Static method

- The static key word is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in.

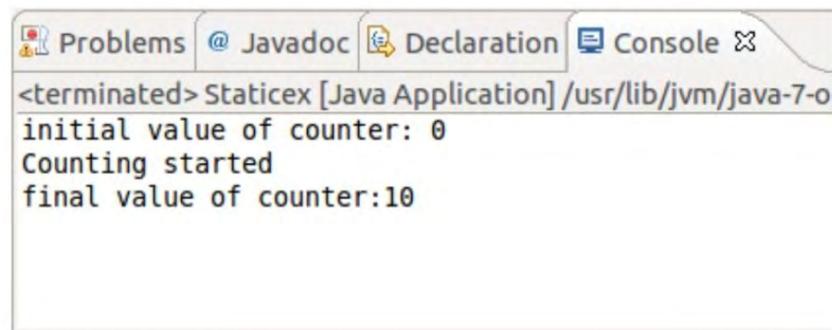
- Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
- Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Example of static modifier

```
package day6;

class Staticmodifier
{
    private static int counter = 0;
    static void addcounter()
    {
        counter++;
    }
    static int getCount()
    {
        return counter;
    }
    Staticmodifier()
    {
        Staticmodifier.addcounter();
    }
}

public class Staticex
{
    public static void main(String args[])
    {
        System.out.println("initial value of counter: "+ Staticmodifier.getCount());
        System.out.println("Counting started");
        for(int i=0;i<10;i++)
        {
            new Staticmodifier();
        }
        System.out.println("final value of counter:"+ Staticmodifier.getCount());
    }
}
```



The screenshot shows an IDE interface with several tabs at the top: Problems, @ Javadoc, Declaration, and Console. The Console tab is active and displays the following text:

```
<terminated> Staticex [Java Application] /usr/lib/jvm/java-7-o
initial value of counter: 0
Counting started
final value of counter:10
```

Final modifier

- The final modifier can be given to
 - Variable and
 - Method
 - Class

Final variable

- A final variable can be explicitly initialized only once.
- A reference variable declared final can never be reassigned to refer to an different object.
- The data within the object can be changed. So the state of the object can be changed but not the reference.

Final method

- A final method cannot be overridden by any subclasses.
- The final modifier prevents a method from being modified in a subclass.

Final class

- The main purpose of using a class being declared as final is to prevent the class from being subclassed.
- If a class is marked as final then no class can inherit any feature from the final class.

Final modifier sample code

```
package day6;
class Final1
{
    final int a =10;
    int out()
    {
        return a;
    }
    @Override
    public String toString()
    {
        return a+"";
    }
    Final1()
    {
        out();
    }
}
public class Finalex
{
    public static void main(String[] args)
    {
        System.out.println(new Final1().toString());
        Final1 f = new Final1();
        //f.a=12;
    }
}
```

The screenshot shows an IDE interface with several tabs at the top: Problems, @ Javadoc, Declaration, and Console. The Console tab is active and displays the output of the Java application. The output shows the value '10' printed to the console, indicating that the final variable 'a' cannot be modified after its initial assignment.

```
<terminated> Finalex [Java Application] /usr/lib/jvm/java-7
10
```

Synchronized modifier

- The synchronized key word used to indicate that a method can be accessed by only one thread at a time.
- The synchronized modifier can be applied with any of the four access level modifiers.

Example

```
public synchronized add()  
{.....}
```

Transient modifier

- An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.
- This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.
- Example

```
transient int a =10; // will remain forever
```

```
int a1 = 20; // will be deallocated after its scope.
```

Volatile modifier

- The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.
- Accessing a volatile variable synchronizes all the cached copies of the variables in the main memory.
- Volatile can only be applied to instance variables, which are of type object or private.
- A volatile object reference can be null.



**“Everything has
beauty, but not
everyone sees it.”**

—CONFUCIUS

Inheritance

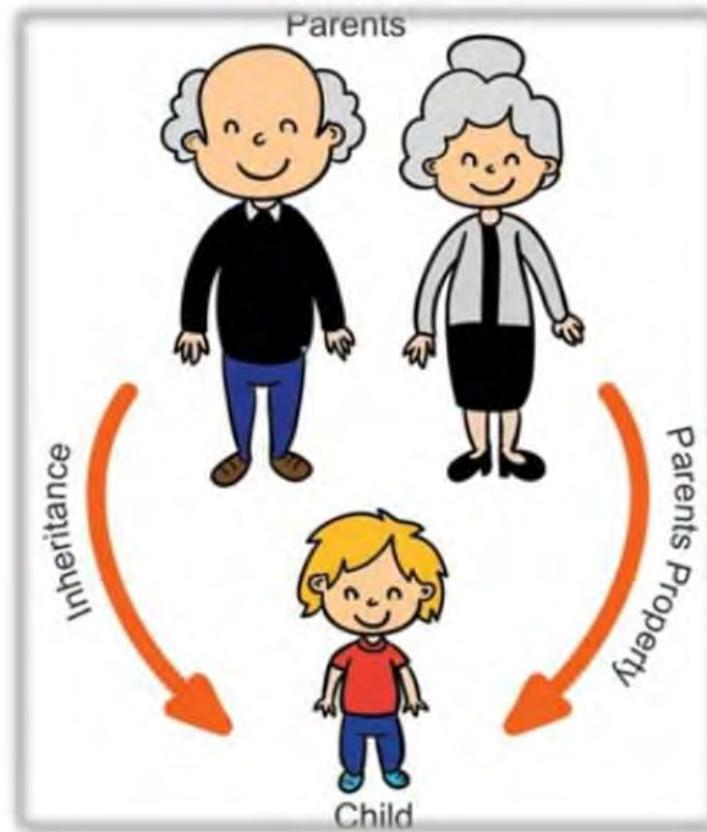
Inheritance in Java

- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.
- When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class(Parent) and Sub class(child) in Java language.
- Inheritance defines is-a relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.

Inheritance in Java

- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
- **Why use Inheritance ?**
- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

Inheritance in Java



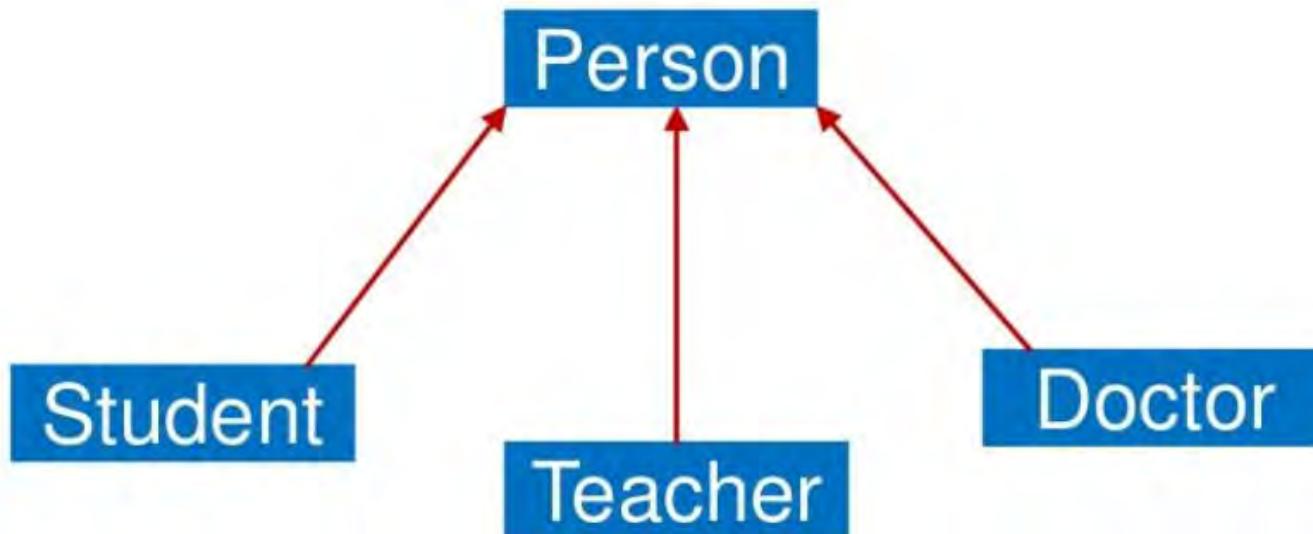
Inheritance in Java

- **Important points**
- In the inheritance the class which is giving data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.
- The data members and methods of a class are known as features.
- The concept of inheritance is also known as re-usability or extendable classes or sub classing or derivation.

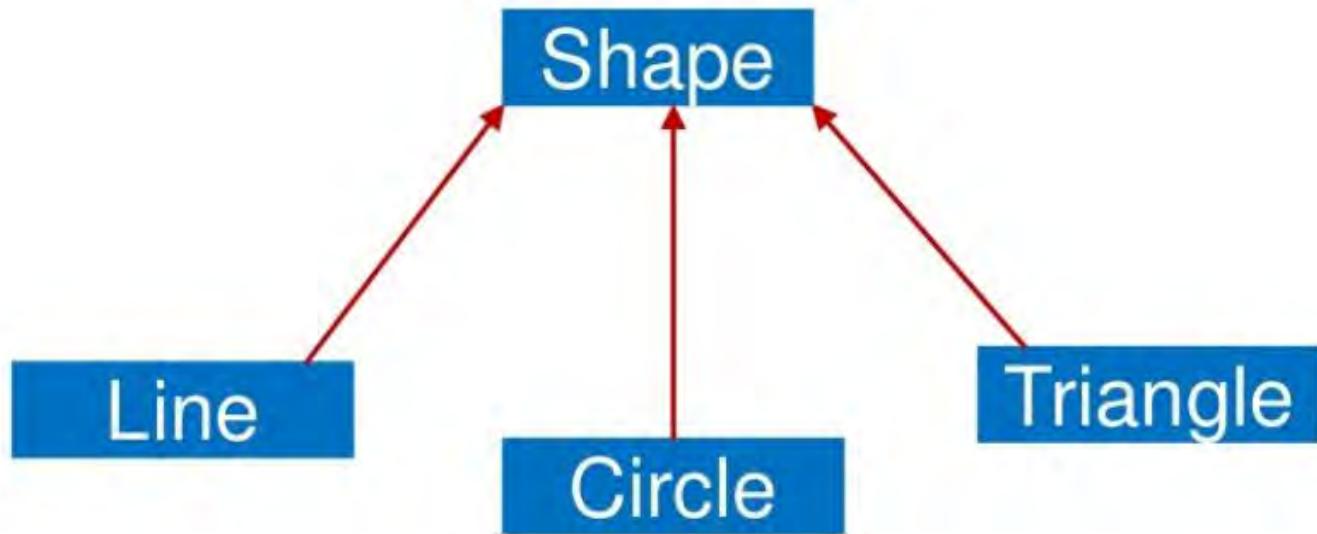
Inheritance in Java

No	Term	Definition
1	Inheritance	Inheritance is a process where one object acquires the properties of another object
2	Subclass	Class which inherits the properties of another object is called as subclass
3	Superclass	Class whose properties are inherited by subclass is called as superclass
4	Keywords Used	extends and implements

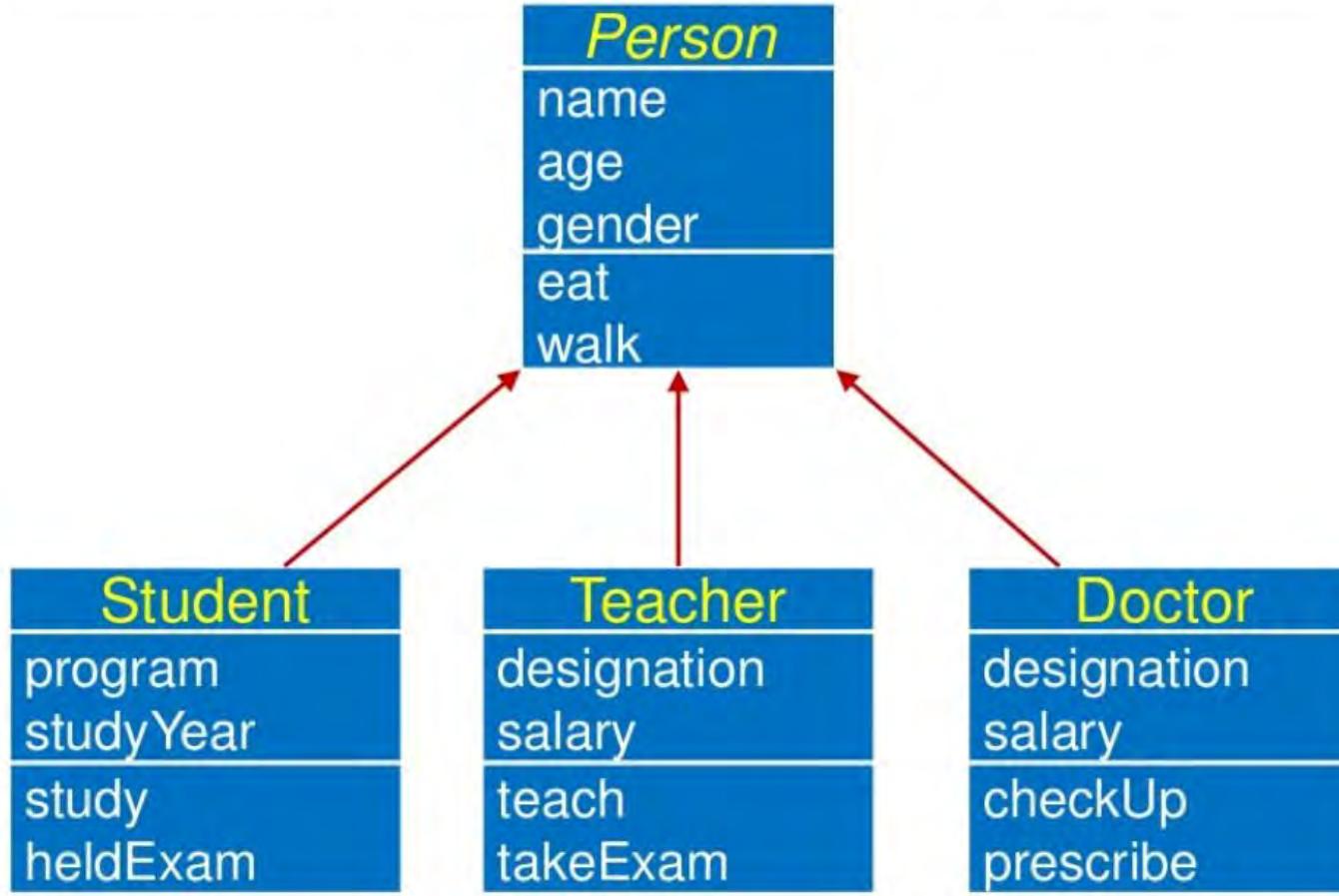
Example – Inheritance



Example – Inheritance



Example – “IS A” Relationship



Concepts Related with Inheritance

- Generalization
- Subtyping (extension)
- Specialization (restriction)

Concepts Related with Inheritance

▪ Generalization

- In OO models, some classes may have common characteristics
- We extract these features into a new class and inherit original classes from this new class
- This concept is known as Generalization

Example – Generalization

Line

color
vertices
length

move
setColor
getLength

Circle

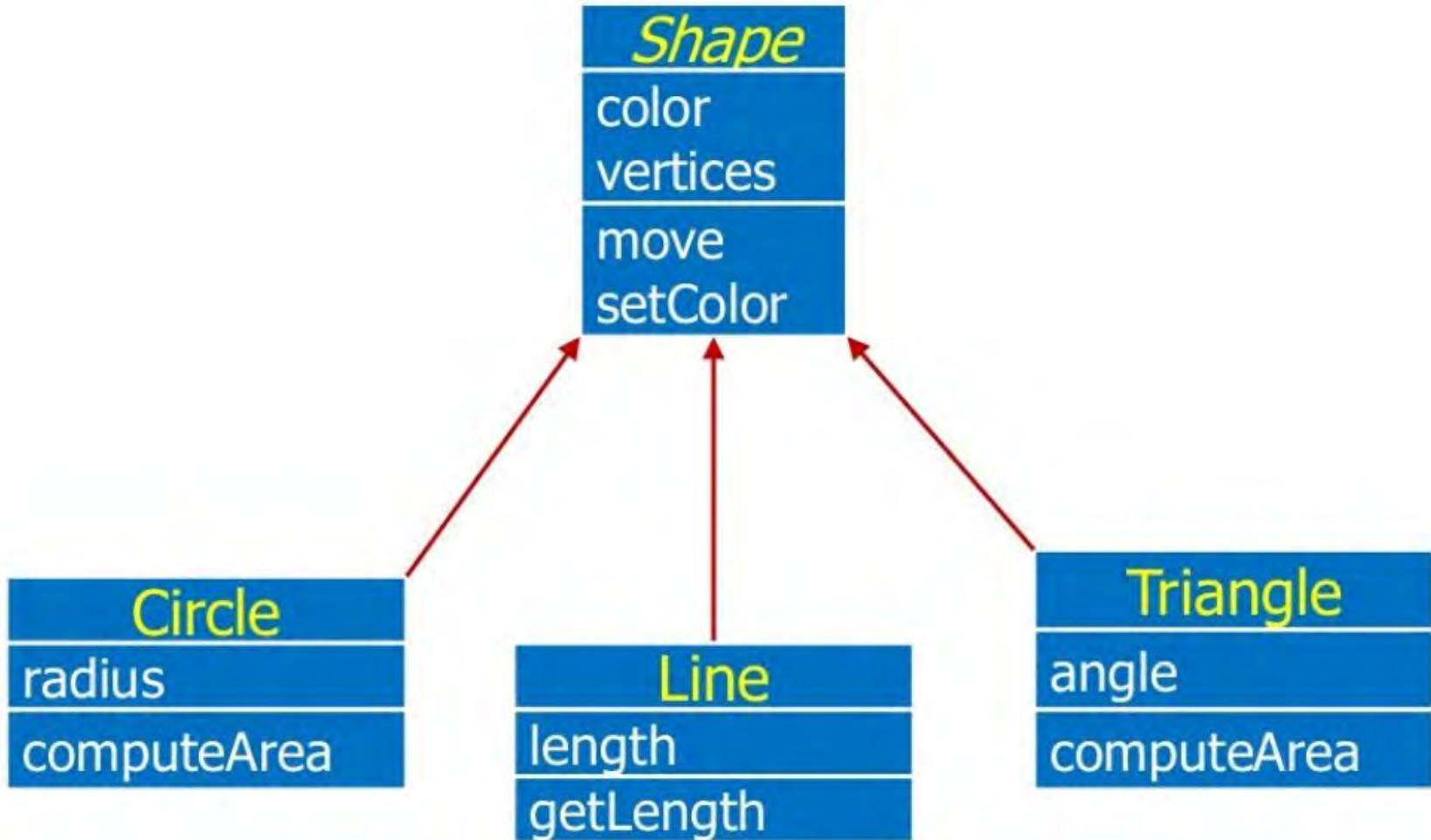
color
vertices
radius

move
setColor
computeArea

Triangle

color
vertices
angle

move
setColor
computeArea



Sub-typing & Specialization

- We want to add a new class to an existing model
- Find an existing class that already implements some of the desired state and behaviour
- Inherit the new class from this class and add unique behaviour to the new class

Sub-typing (Extension)

- Sub-typing means that derived class is behaviourally compatible with the base class
- Behaviourally compatible means that base class can be replaced by the derived class

Example –Sub-typing (Extension)

<i>Person</i>
name
age
gender
eats
walks

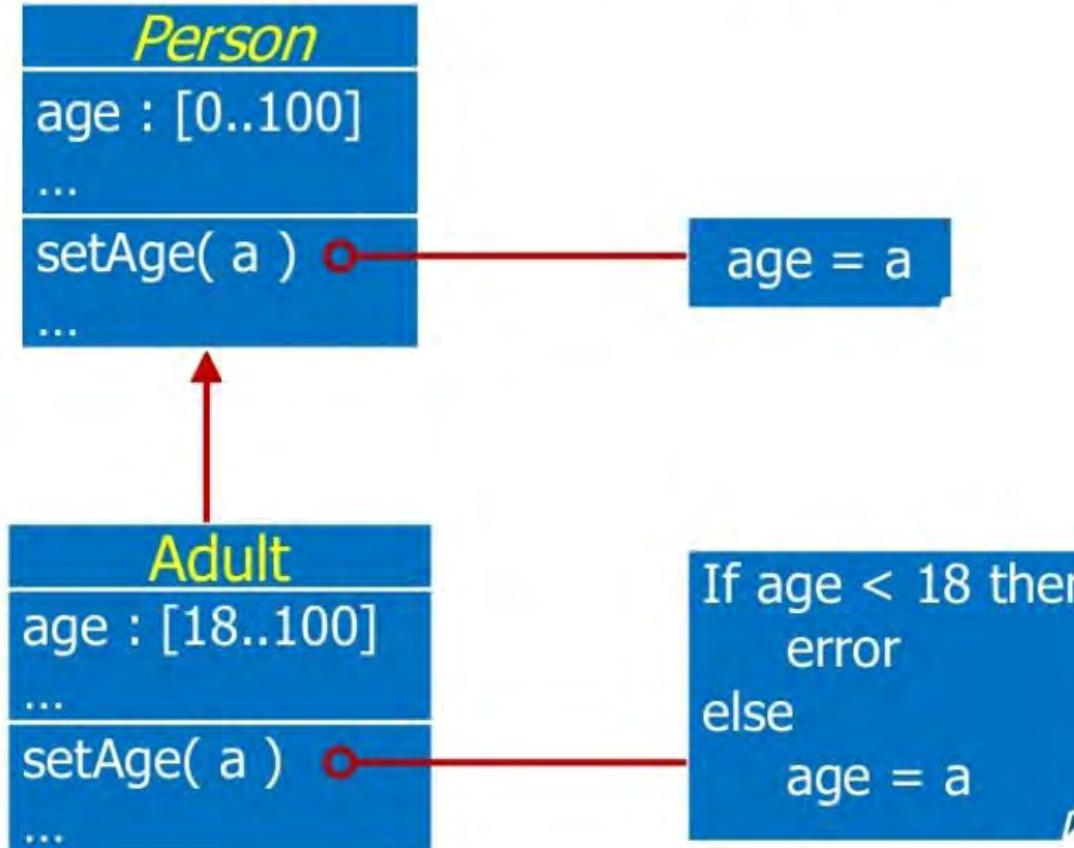


<i>Student</i>
program
studyYear
study
takeExam

Specialization (Restriction)

- Specialization means that derived class is behaviourally incompatible with the base class
- Behaviourally incompatible means that base class can't always be replaced by the derived class

Example – Specialization (Restriction)



Inheritance in Java

- Inheritance in Java
- Inheritance in Java is done using
 - **extends** – In case of Java class and abstract class
 - **implements** – In case of Java interface.
- What is inherited
 - In Java when a class is extended, sub-class inherits all the **public**, **protected** and **default** (**Only if the sub-class is located in the same package as the super class**) methods and fields of the super class.
- What is not inherited
 - **Private** fields and methods of the super class are not inherited by the sub-class and can't be accessed directly by the subclass.
 - Constructors of the super-class are not inherited. There is a concept of constructor chaining in Java which determines in what order constructors are called in case of inheritance.

Inheritance in Java

- **Syntax of Inheritance**

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class.

Inheritance in Java

- Please Note:
- In inheritance Parent Class and Child Class having multiple names, List of the name are below:
 - **Parent Class = Base Class = Super Class**
 - **Child Class = Derived Class = Sub Class**

Simple Example of Inheritance

```
class Parent
{
    public void p1() {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {
    public void c1() {
        System.out.println("Child method");
    }
    public static void main(String[] args) {
        Child cobj = new Child();
        cobj.c1(); //Calling method of Child class
        cobj.p1(); //Calling method of Parent class
    }
}
```

Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
    public static void main(String[] args) {  
        Child cobj = new Child();  
        cobj.c1(); //Calling method of Child class  
        cobj.p1(); //Calling method of Parent class  
    }  
}
```

Super Class/
Parent Class



Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}
```

```
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
    public static void main(String[] args) {  
        Child cobj = new Child();  
        cobj.c1(); //Calling method of Child class  
        cobj.p1(); //Calling method of Parent class  
    }  
}
```

Parent Class
Method

Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
  
    public static void main(String[] args) {  
        Child cobj = new Child();  
        cobj.c1(); //Calling method of Child class  
        cobj.p1(); //Calling method of Parent class  
    }  
}
```

Sub Class/
Child Class



Simple Example of Inheritance

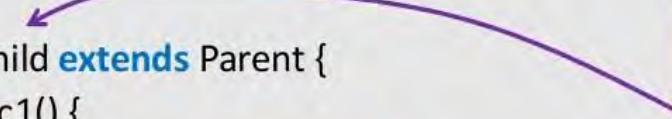
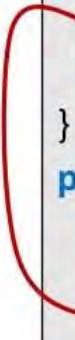
```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
  
    public static void main(String[] args) {  
        Child cobj = new Child();  
        cobj.c1(); //Calling method of Child class  
        cobj.p1(); //Calling method of Parent class  
    }  
}
```

Child Class
Method

Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
}  
  
public static void main(String[] args) {  
    Child cobj = new Child();  
    cobj.c1(); //Calling method of Child class  
    cobj.p1(); //Calling method of Parent class  
}
```

Child Class Inherit
the Properties Of
Parent Class



Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
}  
  
public static void main(String[] args) {  
    Child cobj = new Child(); ←  
    cobj.c1(); //Calling method of Child class  
    cobj.p1(); //Calling method of Parent class  
}
```

Creating An Object
of Child Class

Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
}  
  
public static void main(String[] args) {  
    Child cobj = new Child();  
    cobj.c1(); //Calling method of Child class  
    cobj.p1(); //Calling method of Parent class  
}
```

Calling Child Class and
Parent Class Method
Using Object Of Child
Class

Simple Example of Inheritance

```
class Parent {  
    public void p1() {  
        System.out.println("Parent method");  
    }  
}  
  
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method");  
    }  
}  
  
public static void main(String[] args) {  
    Child cobj = new Child();  
    cobj.c1(); //Calling method of Child class  
    cobj.p1(); //Calling method of Parent class  
}
```

Output is:
Child method
Parent method

Explanation of Previous Program

- When child class inherit the properties of parent class the child class look like this (just for understanding)

```
public class Child extends Parent {  
    public void c1() {  
        System.out.println("Child method"); }  
    public void p1() {  
        System.out.println("Parent method"); }  
    public static void main(String[] args) {  
        Child cobj = new Child();  
        cobj.c1(); //Calling method of Child class  
        cobj.p1(); //Calling method of Parent class  
    }  
}
```

Another example of Inheritance

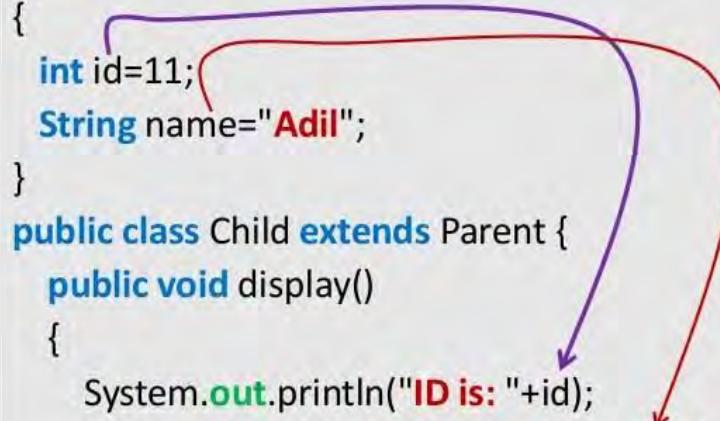
```
class Parent
{
    int id=11;
    String name="Adil";
}

public class Child extends Parent {
    public void display()
    {
        System.out.println("ID is: "+id);
        System.out.println("Name is: "+name);
    }

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

Another example of Inheritance

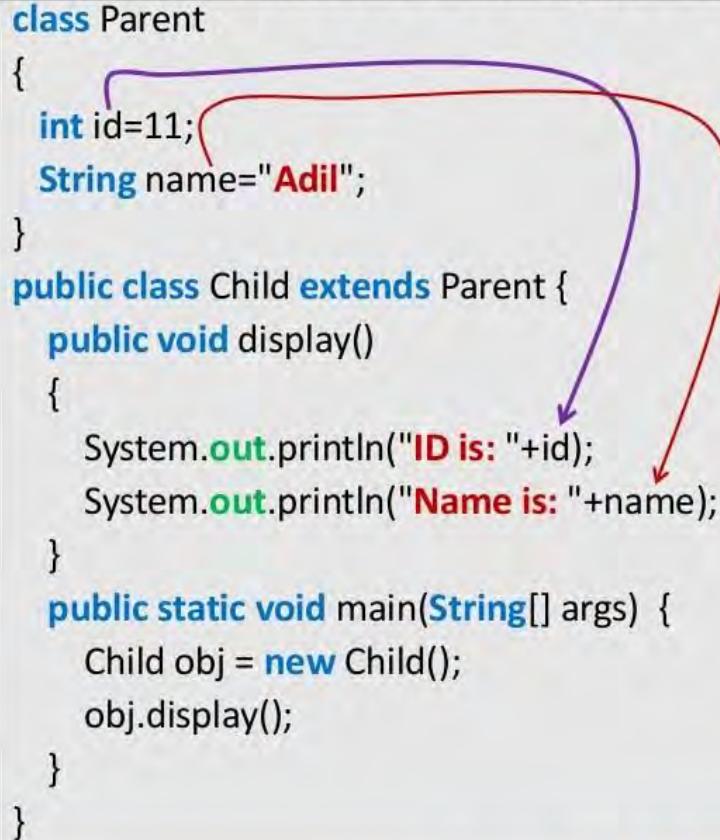
```
class Parent {  
    int id=11;  
    String name="Adil";  
}  
  
public class Child extends Parent {  
    public void display()  
    {  
        System.out.println("ID is: "+id);  
        System.out.println("Name is: "+name);  
    }  
  
    public static void main(String[] args) {  
        Child obj = new Child();  
        obj.display();  
    }  
}
```



Here in Child Class we
Can Inherit Or Access
the Properties of
Parent Class

Another example of Inheritance

```
class Parent {  
    int id=11;  
    String name="Adil";  
}  
  
public class Child extends Parent {  
    public void display()  
    {  
        System.out.println("ID is: "+id);  
        System.out.println("Name is: "+name);  
    }  
  
    public static void main(String[] args) {  
        Child obj = new Child();  
        obj.display();  
    }  
}
```



Here in Child Class we
Can Inherit Or Access
the Properties of
Parent Class

Output is:
ID is: 11
Name is: Adil

Access Control and Inheritance

- A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the members of derived classes should be declared private in the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Access Control and Inheritance

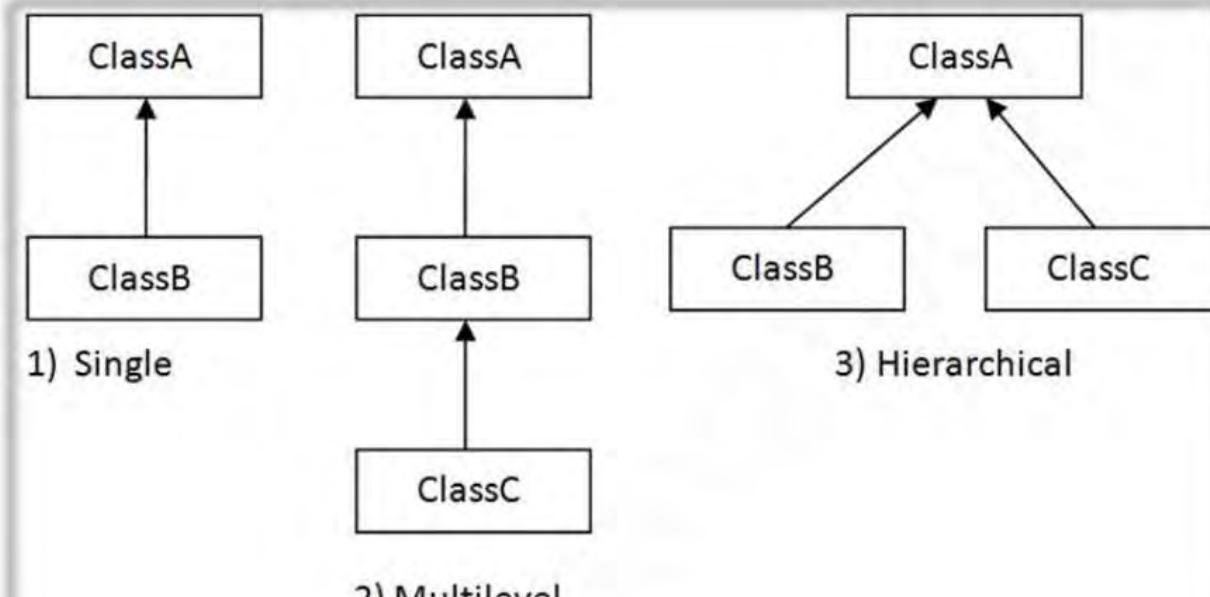
- The following rules for inherited methods are enforced:
 1. Methods declared public in a superclass also must be public in all subclasses.
 2. Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
 3. Methods declared private are not inherited at all, so there is no rule for them.

Inheritance in Java

• **Types of Inheritance**

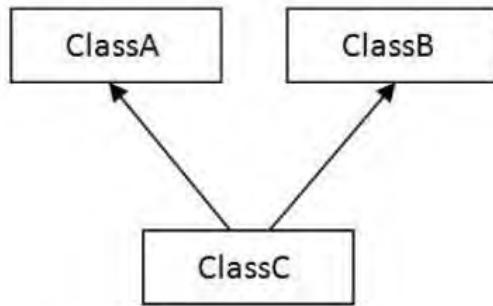
- Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance; they are:
- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

Types of Inheritance

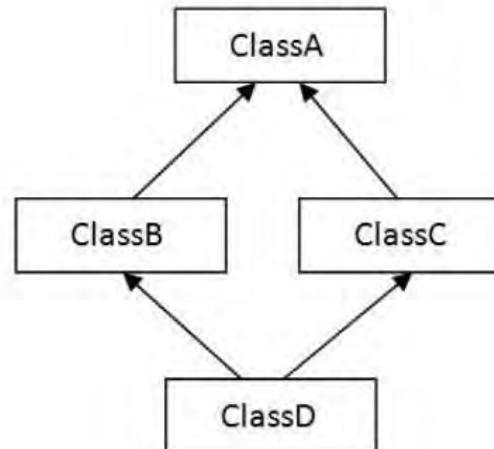


Types of Inheritance

- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



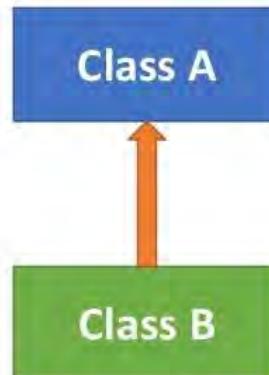
4) Multiple



5) Hybrid

Types of Inheritance

- **1) Single Inheritance**
- **Single inheritance** is easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



Single Inheritance Example

class A

{

.....

}

Class B extends A

{

.....

}

Single Inheritance Example

```
class A<br>{  
    .....  
}  
  
Class B extends A  
  
{  
    .....  
}
```

Parent Class

Child Class

Child Class B
inherit the
Properties of
Parent Class A

Single Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A {  
    public void display()  
    {  
        System.out.println("Data is:"+data);  
    }  
    public static void main(String args[])  
    {  
        B obj = new B();  
        obj.display();  
    }  
}
```

Single Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A {  
    public void display()  
    {  
        System.out.println("Data is:"+data);  
    }  
  
    public static void main(String args[])  
    {  
        B obj = new B();  
        obj.display();  
    }  
}
```

Child Class B
inherit/Access the
data field of Parent
Class A

Single Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A {  
    public void display()  
    {  
        System.out.println("Data is:"+data);  
    }  
  
    public static void main(String args[])  
    {  
        B obj = new B();  
        obj.display();  
    }  
}
```

Child Class B
inherit/Access the
data field of Parent
Class A

Output is:
Data is:10

Another Single Inheritance Example

```
class Faculty {  
    float salary=30000;  
}  
  
class Science extends Faculty {  
    float bonus=2000;  
  
    public static void main(String args[]) {  
        Science obj=new Science();  
        System.out.println("Salary is:"+obj.salary);  
        System.out.println("Bonus is:"+obj.bonus);  
    }  
}
```

Another Single Inheritance Example

```
class Faculty {  
    float salary=30000;  
}  
  
class Science extends Faculty {  
    float bonus=2000;  
  
    public static void main(String args[]) {  
        Science obj=new Science();  
        System.out.println("Salary is:"+obj.salary);  
        System.out.println("Bonus is:"+obj.bonus);  
    }  
}
```

Output is:
Salary is:30000.0
Bonus is:2000.0

Types of Inheritance

• 2) Multilevel Inheritance

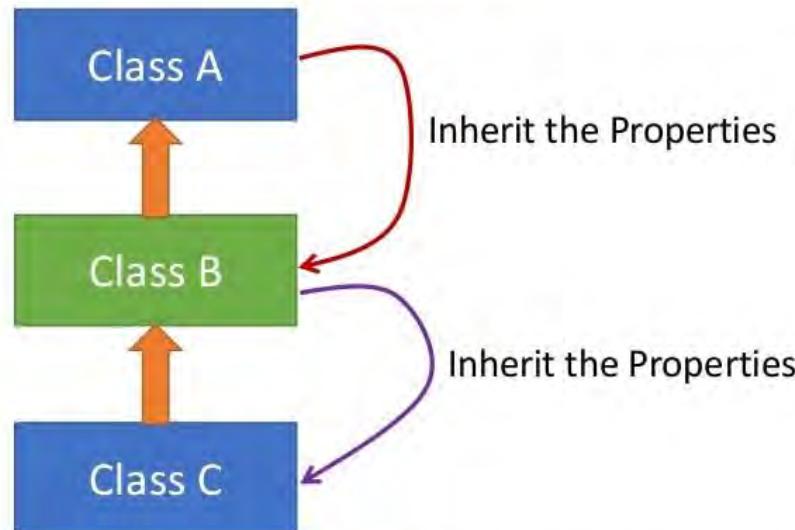
- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.
- **Single base class + single derived class + multiple intermediate base classes.**
- **Intermediate base classes**
- An intermediate base class is one in one context with access derived class and in another context same class access base class.

Multilevel Inheritance



- Here class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that's what is called multilevel inheritance.

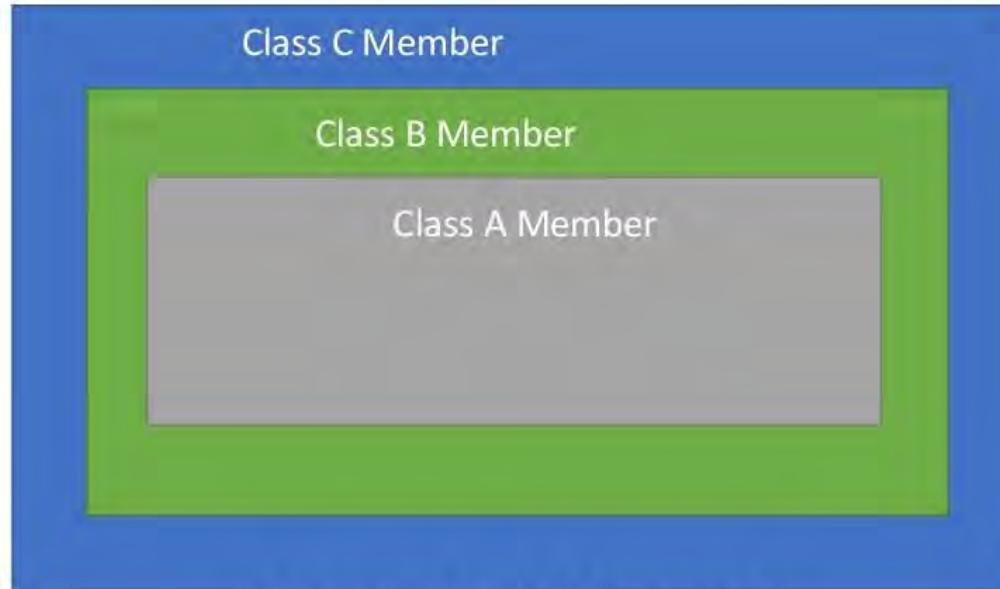
Multilevel Inheritance



- Here class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that's what is called multilevel inheritance.

Multilevel Inheritance

- Class C can inherit the Members of both Class A and B Show below :



C Contains B Which Contains A

Multilevel Inheritance Example

```
class A {
```

```
.....
```

```
}
```

```
Class B extends A {
```

```
.....
```

```
}
```

```
Class C extends B {
```

```
.....
```

```
}
```

A is Parent
Class of B

B is Child Class
of A and Parent
Class of C

C is Child Class
of B

Multilevel Inheritance Example

```
class A {  
    int data=10;  
}  
class B extends A{  
}  
class C extends B {  
    public void display() {  
        System.out.println("Data is:"+data);  
    }  
    public static void main(String args[]) {  
        C obj = new C();  
        obj.display();  
    }  
}
```

Here Class B inherit the properties of Class A and Class C inherit the properties of Class B

Multilevel Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A{  
}  
  
class C extends B {  
    public void display() {  
        System.out.println("Data is:"+data);  
    }  
    public static void main(String args[]) {  
        C obj = new C();  
        obj.display();  
    }  
}
```

Output is:
Data is:10

Another Multilevel Inheritance Example

```
class Faculty {  
    float total_sal=0, salary=1000;  
}  
  
class HRA extends Faculty {  
    float hra=2000;  
}  
  
class DA extends HRA {  
    float da=3000;  
}  
  
class Science extends DA {  
    float bonus=4000;  
}  
  
public static void main(String args[]) {  
    Science obj=new Science();  
    obj.total_sal=obj.salary+obj.hra+obj.da+obj.bonus;  
    System.out.println("Total Salary is:"+obj.total_sal);  
}
```

Here we can create an
Object of Class "Science"

Another Multilevel Inheritance Example

```
class Faculty {  
    float total_sal=0, salary=1000;  
}  
  
class HRA extends Faculty {  
    float hra=2000;  
}  
  
class DA extends HRA {  
    float da=3000;  
}  
  
class Science extends DA {  
    float bonus=4000;  
}  
  
public static void main(String args[]) {  
    Science obj=new Science();  
    obj.total_sal=obj.salary+obj.hra+obj.da+obj.bonus;  
    System.out.println("Total Salary is:"+obj.total_sal);  
}
```

Class C look Like This After inheritance

```
float total_sal=0, salary=1000;  
float hra=2000;  
float da=3000;  
float bonus=4000;
```

Here we can Access the Properties of Class "Faculty", Class "HRA" and Class "DA" using the Object of Class "Science"

obj.total_sal=obj.salary+obj.hra+obj.da+obj.bonus;

System.out.println("Total Salary is:"+obj.total_sal);

Another Multilevel Inheritance Example

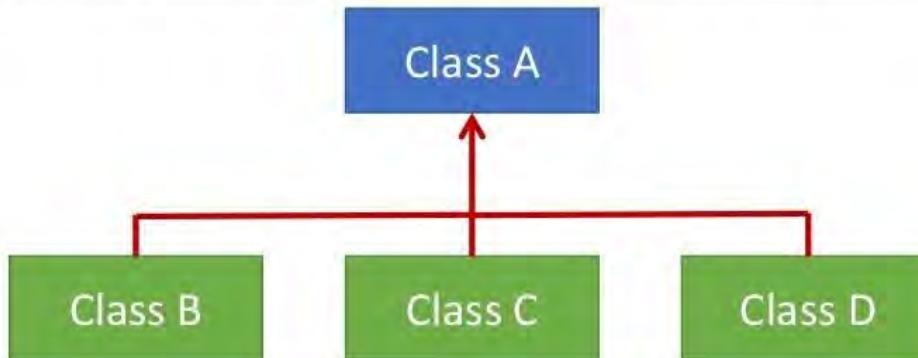
```
class Faculty {  
    float total_sal=0, salary=1000;  
}  
  
class HRA extends Faculty {  
    float hra=2000;  
}  
  
class DA extends HRA {  
    float da=3000;  
}  
  
class Science extends DA {  
    float bonus=4000;  
}  
  
public static void main(String args[]) {  
    Science obj=new Science();  
    obj.total_sal=obj.salary+obj.hra+obj.da+obj.bonus;  
    System.out.println("Total Salary is:"+obj.total_sal);  
}
```

Output is:
Total Salary is:10000.0

Types of Inheritance

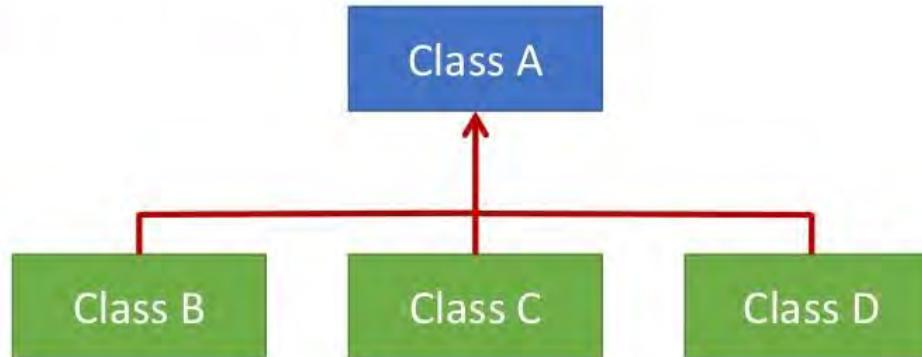
• 3) Hierarchical Inheritance

- In this **inheritance** multiple classes inherits from a **single** class i.e there is one super class and **multiple** sub classes. As we can see from the below diagram when a same class is having more than one sub class (or) more than one sub class has the same parent is called as **Hierarchical Inheritance**.



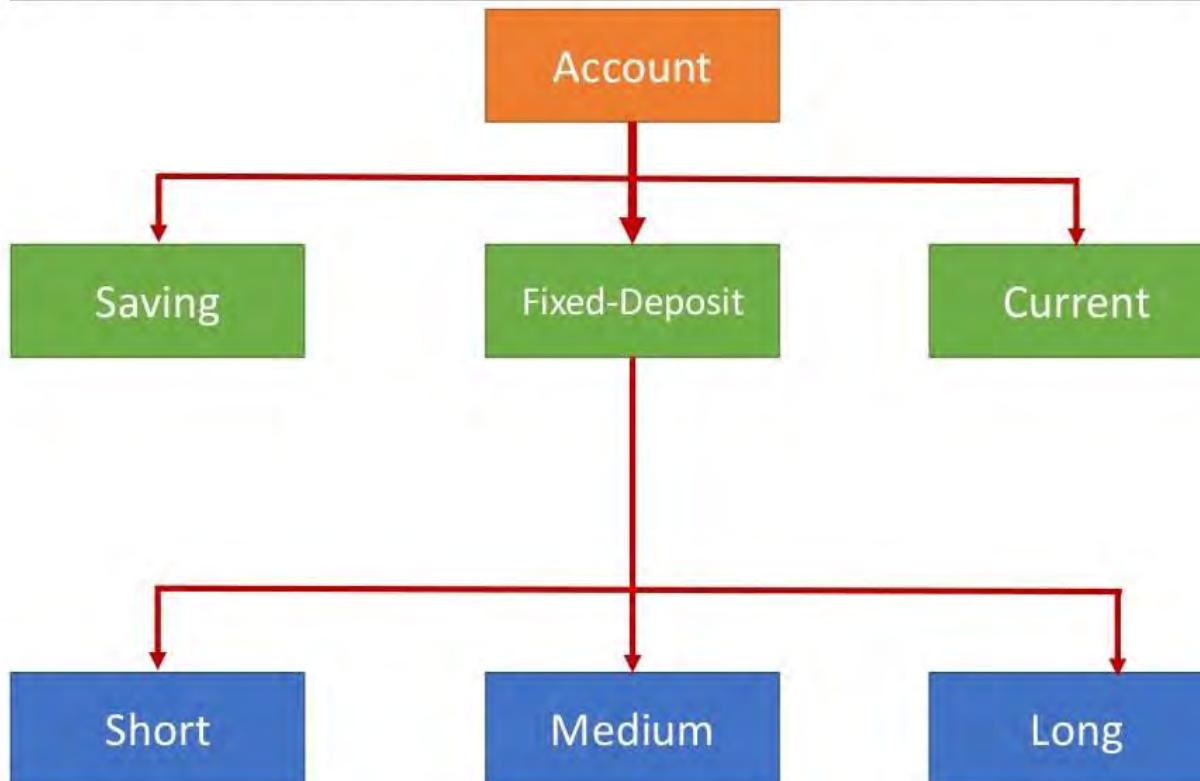
Types of Inheritance

• 3) Hierarchical Inheritance



Here **Class A** acts as the **parent** for sub classes **Class B**, **Class C** and **Class D**

Hierarchical Inheritance(Real time Example)



Hierarchical Inheritance Example

```
class A {
```

.....

```
}
```

```
Class B extends A {
```

.....

```
}
```

```
Class C extends A {
```

.....

```
}
```

```
Class D extends A {
```

.....

```
}
```

A is a Parent Class of Class B, C and D

B is Child Class of Class A

C is Child Class of Class A

D is Child Class of Class A

Hierarchical Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A{  
}  
  
class C extends A {  
}  
  
class D extends A {  
    public static void main(String args[]) {  
        B obj1 = new B();  
        C obj2 = new C();  
        D obj3 = new D();  
  
        System.out.println("Data in Class B is: "+obj1.data);  
        System.out.println("Data in Class C is: "+obj2.data);  
        System.out.println("Data in Class D is: "+obj3.data);  
    }  
}
```

Hierarchical Inheritance Example

```
class A {  
    int data=10;  
}  
  
class B extends A{  
}  
  
class C extends A {  
}  
  
class D extends A {  
    public static void main(String args[]) {  
        B obj1 = new B();  
        C obj2 = new C();  
        D obj3 = new D();  
  
        System.out.println("Data in Class B is: "+obj1.data);  
        System.out.println("Data in Class C is: "+obj2.data);  
        System.out.println("Data in Class D is: "+obj3.data);  
    }  
}
```

Output is:

Data in Class B is: 10
Data in Class C is: 10
Data in Class D is: 10

Another Hierarchical Inheritance Example

```
class A {  
    void methodA() {  
        System.out.println("Method of Class A"); }  
}  
  
class B extends A{ }  
  
class C extends A{ }  
  
class D extends A {  
    public static void main(String args[]) {  
        B obj1 = new B();  
        C obj2 = new C();  
        D obj3 = new D();  
        obj1.methodA();  
        obj2.methodA();  
        obj3.methodA();  
    }  
}
```

Another Hierarchical Inheritance Example

```
class A {  
    void methodA() {  
        System.out.println("Method of Class A"); }  
}  
  
class B extends A{}  
  
class C extends A{}  
  
class D extends A {  
    public static void main(String args[]) {  
        B obj1 = new B();  
        C obj2 = new C();  
        D obj3 = new D();  
        obj1.methodA();  
        obj2.methodA();  
        obj3.methodA();  
    }  
}
```

Output is:

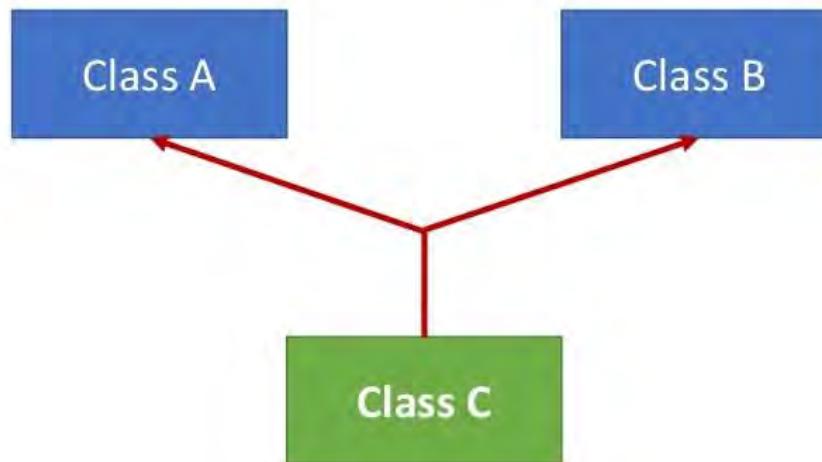
Method of Class A
Method of Class A
Method of Class A

Types of Inheritance

• 4) Multiple Inheritance

- In java programming, multiple and hybrid inheritance is not supported through classes but supported through interface only. We will learn about interfaces later.
- Q) Why multiple inheritance is not supported in java?
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Multiple Inheritance



Here Class A, B and C are three Classes. The C Class inherits A and B classes.in other words Class C inherit the properties of both Class A and Class B.

Multiple Inheritance

- **Why multiple inheritance is not supported in java?**

- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

Multiple Inheritance Example

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

class B{
    void msg(){
        System.out.println("Welcome");
    }
}

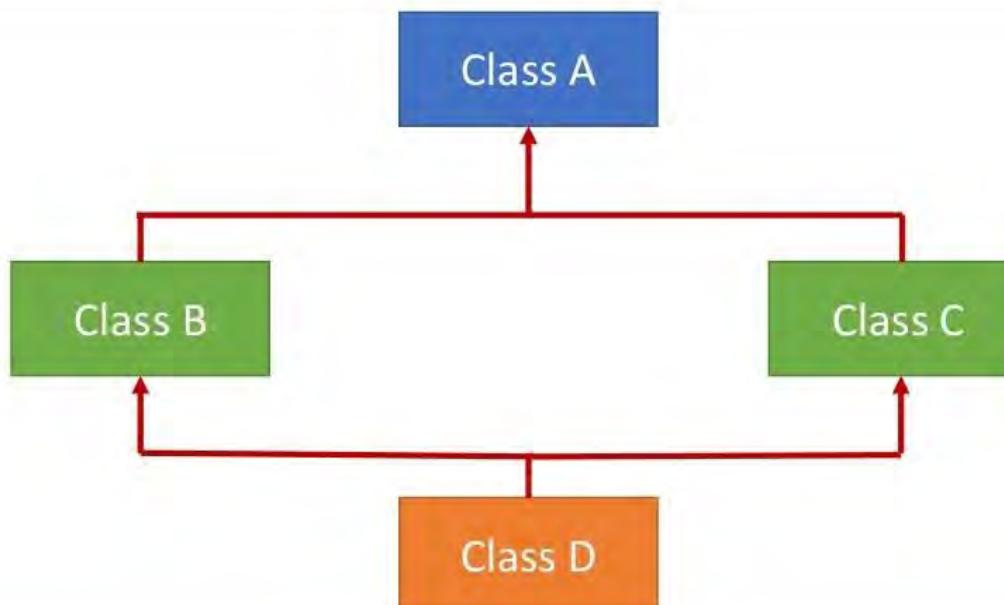
class C extends A,B{ //suppose if it were

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

Types of Inheritance

• 5) Hybrid inheritance

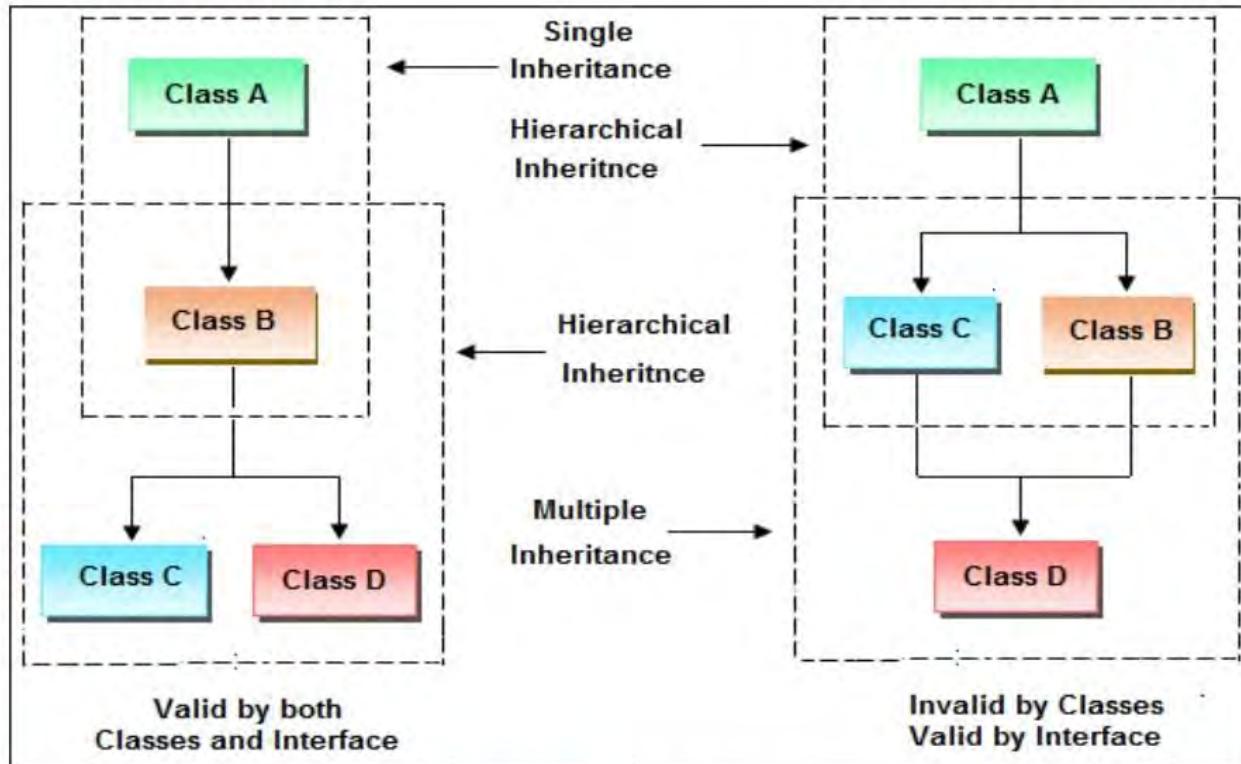
- Any combination of previous three inheritance (single, hierarchical and multi level) is called as hybrid inheritance.



Hybrid Inheritance

- In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple** inheritance
- A typical flow diagram would look like in the previous slide.
- A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right.
- By using **interfaces** we can have multiple as well as **hybrid inheritance** in Java.

Hybrid inheritance



Important Points for Inheritance

- In java programming one derived class can extends only one base class because java programming does not support multiple inheritance through the concept of classes, but it can be supported through the concept of Interface.
- Whenever we develop any inheritance application first create an object of bottom most derived class but not for top most base class.
- When we create an object of bottom most derived class, first we get the memory space for the data members of top most base class, and then we get the memory space for data member of other bottom most derived class.
- Bottom most derived class contains logical appearance for the data members of all top most base classes.

Important Points for Inheritance

- If we do not want to give the features of base class to the derived class then the definition of the base class must be preceded by final hence final base classes are not reusable or not inheritable.
- If we do not want to give some of the features of base class to derived class than such features of base class must be as private hence private features of base class are not inheritable or accessible in derived class.
- An object of base class can contain details about features of same class but an object of base class never contains the details about special features of its derived class (this concept is known as scope of base class object).
- For each and every class in java there exists an implicit predefined super class called `java.lang.Object`. because it provides garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.

Inheritance in Java

• **Advantage of inheritance**

- If we develop any application using concept of Inheritance than that application have following advantages,
- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Inheritance in Java

• Disadvantages of Inheritance

- Inheritance base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes.
- In class hierarchy many data members remain unused and the memory allocated to them is not utilized. Hence affect performance of your program if you have not implemented inheritance correctly.

Reference

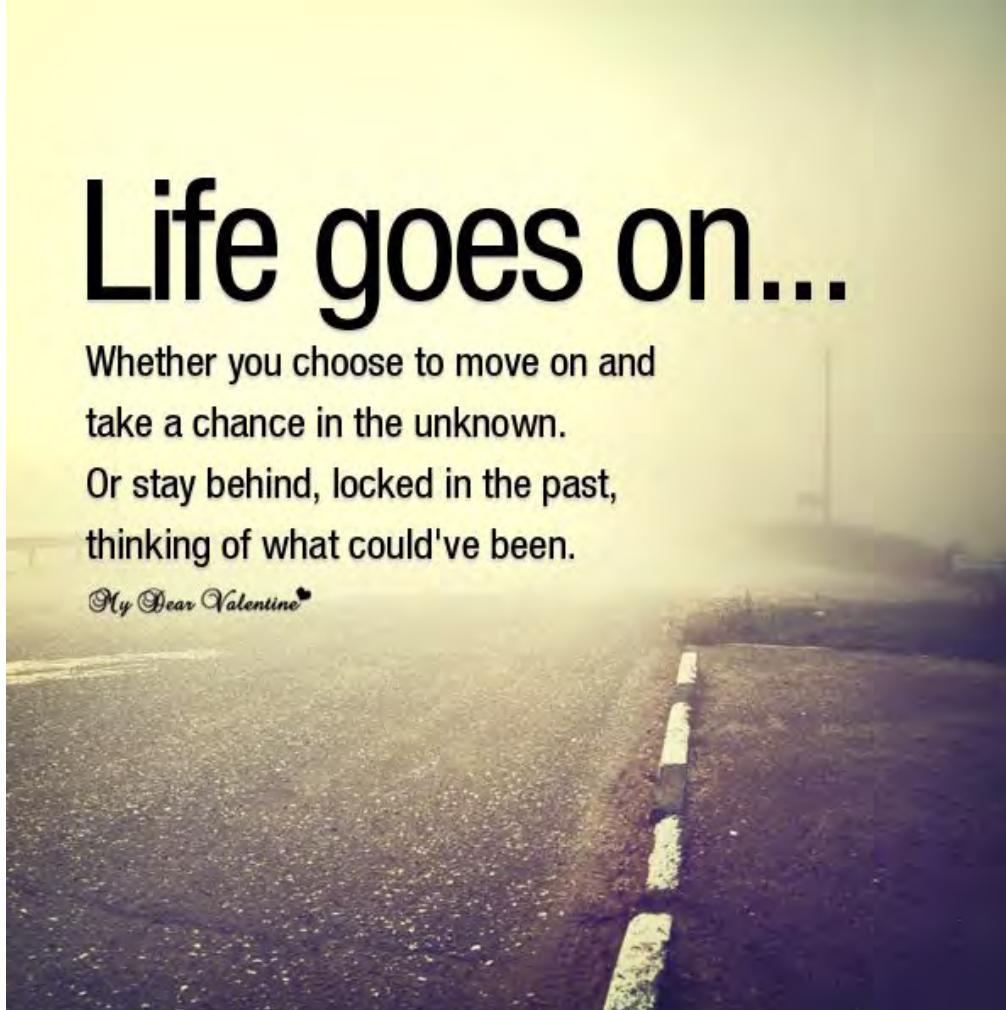
Notes by Adil Aslam

Life goes on...

Whether you choose to move on and
take a chance in the unknown.

Or stay behind, locked in the past,
thinking of what could've been.

My Dear Valentine ❤



Super Keyword

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance **variable**.
2. super can be used to invoke immediate parent class **method**.
3. super() can be used to invoke immediate parent class **constructor**.

- 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the **data member or field** of parent class. It is used if parent class and child class have same fields.

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

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

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

OUTPUT:

black
white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{  
    void eat(){System.out.println("eating...");}  
}  
  
class Dog extends Animal{  
    void eat(){System.out.println("eating bread...");}  
    void bark(){System.out.println("barking...");}  
    void work(){  
        super.eat();  
        bark();  
    }  
}  
  
class TestSuper2{  
    public static void main(String args[]){  
        Dog d=new Dog();  
        d.work(); }}
```

Output:
eating...
barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

- 3) super is used to invoke parent class constructor.

In Java, constructor of base class with no argument gets automatically called in derived class constructor.

```
// filename: Main.java

class Base {

    Base() {
        System.out.println("Base Class Constructor Called ");
    }
}

class Derived extends Base {

    Derived() {
        System.out.println("Derived Class Constructor Called ");
    }
}
```

```
public class Main {

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

But, if we want to call parameterized constructor of base class, then we can call it using super(). The point to note is **base class constructor call must be the first line in derived class constructor**. For example, in the following program, super(_x) is first line derived class constructor.

// filename: Main.java

```
class Base {  
    int x;  
    Base(int _x) {  
        x = _x;  
    }  
}
```

```
class Derived extends Base {  
  
    int y;  
  
    Derived(int _x, int _y) {  
        super(_x);  
        y = _y;  
    }  
  
    void Display() {  
        System.out.println("x = "+x+", y = "+y);  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Derived d = new Derived(10, 20);  
        d.Display();  
    }  
}
```

Output:

x = 10, y = 20



you'll grow beautifully in your own way.

- D h i m a n

Polymorphism

What is polymorphism in programming?

- **Polymorphism** is the capability of a method to do different things based on the object .
- In other words, **polymorphism** allows you define one interface and have multiple implementations.
 1. It is a feature that allows one interface to be used for a general class of actions.
 2. An operation may show different behavior in different instances.
 3. The behavior depends on the types of data used in the operation.
 4. It plays an important role in allowing objects having different internal structures to share the same external interface.
 5. **Polymorphism** is extensively used in implementing inheritance.

Why we use polymorphism?

- Polymorphism allow you to “program in the general” rather than “program in the specific.” .
- Polymorphism allow you to write programs that process objects that share the same superclass .
- With polymorphism, we can design and implement systems that are easily extensible .
- New classes can be added with little or no modification to the general portions of the program .

Types of polymorphism in java

- There are two types of polymorphism in java :
 1. Runtime polymorphism (Dynamic polymorphism)
 2. Compile time polymorphism (static polymorphism).

Runtime Polymorphism(or Dynamic polymorphism)

[Method overriding](#) is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of class X can hold object of class X or an object of any sub classes of class X. For e.g. if class Y extends class X then both of the following statements are valid:

```
Y obj = new Y();
//Parent class reference can be assigned to child object
X obj = new Y();
```

Lets see the below example to understand it better :

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}

public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}

public class Z
{
    public static void main (String args [])
    {
        X obj1 = new X(); // Reference and object X
        X obj2 = new Y(); // X reference but Y object
        obj1.methodA();
        obj2.methodA();
    }
}
```

Output:

```
hello, I'm methodA of class X
hello, I'm methodA of class Y
```

Compile time Polymorphism(or Static polymorphism)

- Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both.

Lets see the below example to understand it better :

```
class X
{
    void methodA(int num)
    {
        System.out.println ("methodA:" + num);
    }
    void methodA(int num1, int num2)
    {
        System.out.println ("methodA:" + num1 + "," + num2);
    }
    double methodA(double num) {
        System.out.println("methodA:" + num);
        return num;
    }
}
```

```
class Y
{
    public static void main (String args [])
    {
        X Obj = new X();
        double result;
        Obj.methodA(20);
        Obj.methodA(20, 30);
        result = Obj.methodA(5.5);
        System.out.println("Answer is:" + result);
    }
}
```

Output:

```
methodA:20
methodA:20,30
methodA:5.5
Answer is:5.5
```

When we use polymorphism ?

- You are programming in Java? Then its not a choice, you can't avoid using **polymorphism**. The biggest benefit of **polymorphism** is that it allows for extensible programs. You can have an API which deals with base types, and requires no knowledge at the time of writing what extra types might be required in the future .
- Because the API uses **polymorphism** it means if I create a new class by extending an existing one or implementing an existing interface I know that my new object will work seamlessly with the existing API. It also means I can keep things general in my code.



kindness always comes back

k.tolnoe

Abstract Classes and Interfaces



Java is “safer” than Python

- Python is very *dynamic*—classes and methods can be added, modified, and deleted as the program runs
 - If you have a call to a function that doesn’t exist, Python will give you a **runtime** error *when you try to call it*
- In Java, everything has to be defined before the program begins to execute
 - If you have a call to a function that doesn’t exist, the compiler marks it as a **syntax error**
 - Syntax errors are far better than runtime errors
 - Among other things, they won’t make it into distributed code
 - To achieve this, Java requires some additional kinds of classes



Abstract methods

- You can *declare* an object without *defining* it:
`Person p;`
- Similarly, you can declare a *method* without defining it:
`public abstract void draw(int size);`
 - Notice that the body of the method is missing
- A method that has been declared but not defined is an **abstract method**



Abstract classes I

- Any class containing an abstract method is an **abstract class**
- You must declare the class with the keyword **abstract**:
`abstract class MyClass {...}`
- An abstract class is *incomplete*
 - It has “missing” method bodies
- You cannot **instantiate** (create a new instance of) an abstract class



Abstract classes II

- You can extend (subclass) an abstract class
 - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated
 - If the subclass does *not* define all the inherited abstract methods, it too must be abstract
- You can declare a class to be **abstract** even if it does not contain any abstract methods
 - This prevents the class from being instantiated



Why have abstract classes?

- Suppose you wanted to create a class **Shape**, with subclasses **Oval**, **Rectangle**, **Triangle**, **Hexagon**, etc.
- You don't want to allow creation of a “Shape”
 - Only *particular* shapes make sense, not *generic* ones
 - If **Shape** is abstract, you can't create a **new Shape**
 - You *can* create a **new Oval**, a **new Rectangle**, etc.
- Abstract classes are good for defining a general category containing specific, “concrete” classes



An example abstract class

- ```
public abstract class Animal {
 abstract int eat();
 abstract void breathe();
}
```
- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the `eat()` and `breathe()` methods



# Why have abstract methods?

---

- Suppose you have a class **Shape**, but it *isn't* abstract
  - **Shape** should *not* have a **draw()** method
  - Each subclass of **Shape** *should* have a **draw()** method
- Now suppose you have a variable **Shape figure**; where **figure** contains some subclass object (such as a **Star**)
  - It is a *syntax error* to say **figure.draw()**, because the Java compiler can't tell in advance what kind of value will be in the **figure** variable
  - A class "knows" its superclass, but doesn't know its subclasses
  - An object knows its class, but a class doesn't know its objects
- **Solution:** Give **Shape** an *abstract* method **draw()**
  - Now the class **Shape** is abstract, so it can't be instantiated
  - The **figure** variable cannot contain a (generic) **Shape**, because it is impossible to create one
  - Any object (such as a **Star** object) that *is* a (kind of) **Shape** *will* have the **draw()** method
  - The Java compiler can depend on **figure.draw()** being a legal call and does not give a syntax error



# A problem

---

- `class Shape { ... }`
- `class Star extends Shape {`  
  `void draw() { ... }`  
  `...`  
}
- `class Crescent extends Shape {`  
  `void draw() { ... }`  
  `...`  
}
- `Shape someShape = new Star();`
  - This is legal, because a Star *is* a Shape
- `someShape.draw();`
  - This is a syntax error, because *some Shape* might not have a `draw()` method
  - Remember: *A class knows its superclass, but not its subclasses*



## A solution

---

- ```
abstract class Shape {  
    abstract void draw();  
}  
▪ class Star extends Shape {  
    void draw() { ... }  
    ...  
}  
▪ class Crescent extends Shape {  
    void draw() { ... }  
    ...  
}  
▪ Shape someShape = new Star();  
    ▪ This is legal, because a Star is a Shape  
    ▪ However, Shape someShape = new Shape(); is no longer legal  
▪ someShape.draw();  
    ▪ This is legal, because every actual instance must have a draw() method
```



Interfaces

- An **interface** declares (describes) methods but does not supply bodies for them

```
interface KeyListener {  
    public void keyPressed(KeyEvent e);  
    public void keyReleased(KeyEvent e);  
    public void keyTyped(KeyEvent e);  
}
```

- All the methods are implicitly **public** and **abstract**
 - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
 - An **interface** is like a *very* abstract class—*none* of its methods are defined
- An interface may also contain constants (**final** variables)



Designing interfaces

- Most of the time, you will use Sun-supplied Java interfaces
- Sometimes you will want to design your own
- You would write an interface if you want classes of various types to all have a certain set of capabilities
- For example, if you want to be able to create animated displays of objects in a class, you might define an interface as:
 - ```
public interface Animatable {
 install(Panel p);
 display();
}
```
- Now you can write code that will display *any* **Animatable** class in a **Panel** of your choice, simply by calling these methods



# Implementing an interface I

---

- You **extend** a class, but you **implement** an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like
- Example:

```
class MyListener
 implements KeyListener, ActionListener { ... }
```



## Implementing an interface II

---

- When you say a class **implements** an interface, you are promising to *define* all the methods that were *declared* in the interface
- Example:

```
class MyKeyListener implements KeyListener {
 public void keyPressed(KeyEvent e) {...};
 public void keyReleased(KeyEvent e) {...};
 public void keyTyped(KeyEvent e) {...};
}
```

- The “...” indicates actual code that you must supply
- Now you can create a new **MyKeyListener**



## Partially implementing an Interface

---

- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener {
 public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword **abstract**
- You can even *extend* an interface (to add methods):
  - `interface FunkyKeyListener extends KeyListener { ... }`



## What are interfaces for?

---

- **Reason 1:** A class can only **extend** one other class, but it can **implement** multiple interfaces

- This lets the class fill multiple “roles”
  - In writing Applets, it is common to have one class implement several different listeners
  - Example:

```
class MyApplet extends Applet
 implements ActionListener, KeyListener {
 ...
}
```

- **Reason 2:** You can write methods that work for more than one kind of class



# How to use interfaces

---

- You can write methods that work with more than one class
- ```
interface RuleSet { boolean isLegal(Move m, Board b);
                    void makeMove(Move m); }
```

 - Every class that implements `RuleSet` must have these methods
- ```
class CheckersRules implements RuleSet { // one implementation
 public boolean isLegal(Move m, Board b) { ... }
 public void makeMove(Move m) { ... }
}
```
- `class ChessRules implements RuleSet { ... } // another implementation`
- `class LinesOfActionRules implements RuleSet { ... } // and another`
- `RuleSet rulesOfThisGame = new ChessRules();`
  - This assignment is legal because a `rulesOfThisGame` object is a `RuleSet` object
- `if (rulesOfThisGame.isLegal(m, b)) { makeMove(m); }`
  - This statement is legal because, whatever kind of `RuleSet` object `rulesOfThisGame` is, it must have `isLegal` and `makeMove` methods



## Interfaces, again

---

- When you implement an interface, you promise to define *all* the functions it declares
- There can be a *lot* of methods

```
interface KeyListener {
 public void keyPressed(KeyEvent e);
 public void keyReleased(KeyEvent e);
 public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?



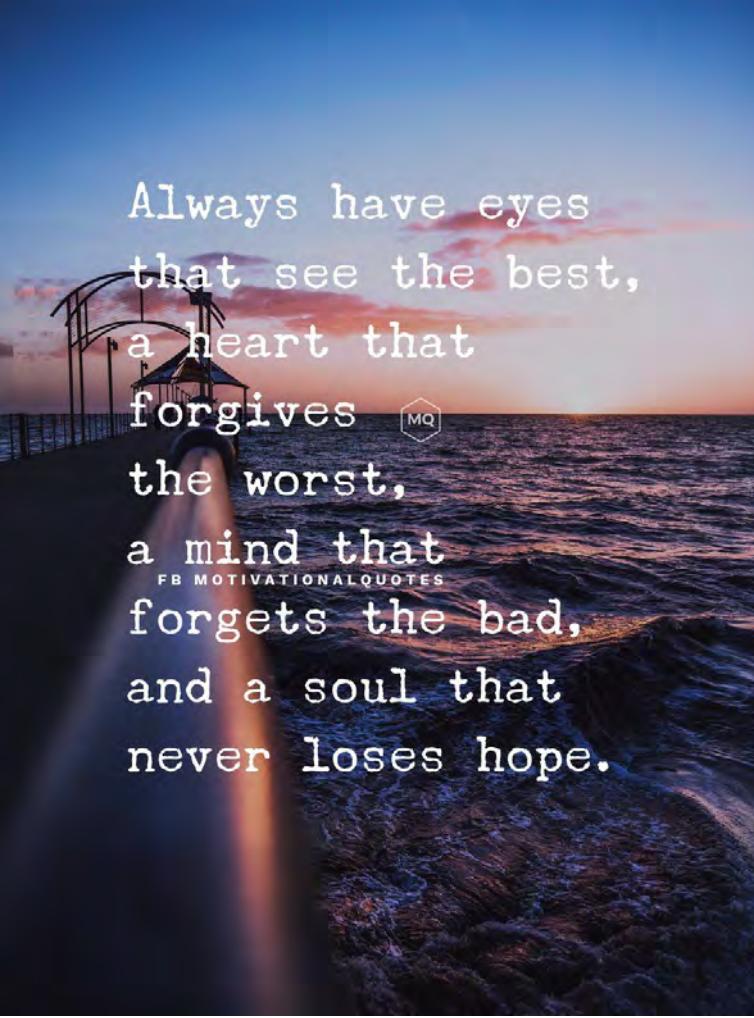
## Adapter classes

---

- Solution: use an adapter class
- An **adapter class** implements an interface and provides empty method bodies

```
class KeyAdapter implements KeyListener {
 public void keyPressed(KeyEvent e) { };
 public void keyReleased(KeyEvent e) { };
 public void keyTyped(KeyEvent e) { };
}
```

- You can override only the methods you care about
- This isn't elegant, but it does work
- Java provides a number of adapter classes

A photograph of a sunset over a body of water. In the foreground, the dark silhouette of a bridge or pier extends from the left towards the horizon. The sky is a gradient of orange, yellow, and blue. The water reflects these colors. A small, semi-transparent hexagonal logo with the letters "MQ" is positioned in the upper right area of the image.

Always have eyes  
that see the best,  
a heart that  
forgives MQ  
the worst,  
a mind that  
FB MOTIVATIONALQUOTES  
forgets the bad,  
and a soul that  
never loses hope.

# Packages & Inbuilt classes

## Package :

Packages are java's way of grouping a variety of classes and/or interfaces together.

## Java API packages :

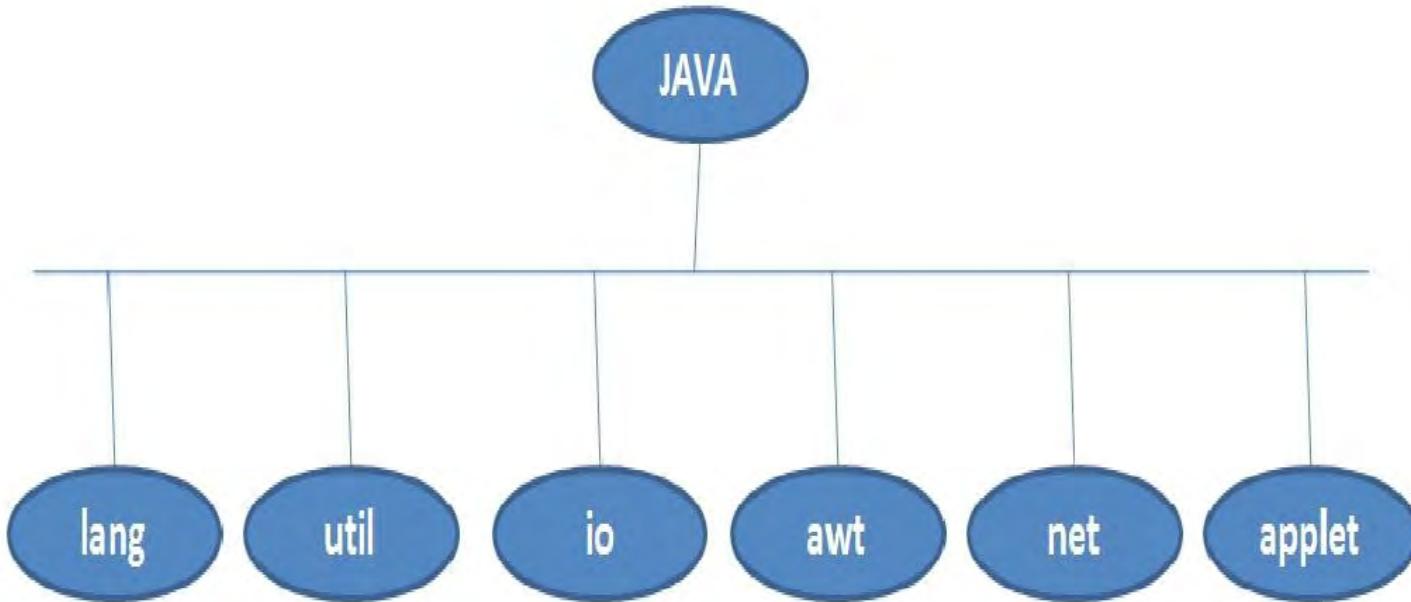
java API provides a large number of classes grouped into different packages according to functionality.

Frequently used API packages are [java2all](#).



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS



| Package Name | Contents                                                                                                                                                                                                             |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Java.lang    | Language support classes. These are classes that java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions. |
| Java.util    | Language utility classes such as vectors, hash tables, random numbers, date, etc.                                                                                                                                    |
| Java.io      | Input/output support classes. They provide facilities for the input and output of the data.                                                                                                                          |
| Java.awt     | Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.                                                                                         |
| Java.net     | Classes for networking. They include classes for communicating with local computers as well as with internet servers.                                                                                                |
| Java.applet  | Classes for creating and implementing  <a href="http://www.java2all.com">http://www.java2all.com</a>                             |

## Syntax for import packages is as under

**import packagename.classname;**

or

**import packagename.\*;**

These are known as **import statements** and must appear at the top of the file, before any file declaration as you can see in our few examples. Here **import** is a keyword.

The **first statement** allows the specified class in the **specified package** to be imported.

**EX :**

```
import java.awt.Color;
double y = java.lang.Math.sqrt(x);
```

Here lang is a package, Math is a class and sqrt is a method.

For create new package you can write...

```
package firstPackage;
```

```
public class Firstclass
```

```
{
```

.....

body of class

.....

```
}
```



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

## Vector class :

The Vector class is one of the most important in all of the Java class libraries. We cannot expand the size of a **static array**.

We may think of a **vector as a dynamic array** that **automatically expands** as more elements are added to it.

All vectors are created with some **initial capacity**.



When space is needed to accommodate more elements, the capacity is **automatically increased**. That is why vectors are commonly used in java programming.

This class provides the following

**constructors:**

`Vector()`

`Vector(int n)`

`Vector(int n, int delta)`

The first form creates a vector with an initial capacity of ten elements.

The second form creates a vector with an initial capacity of  $n$  elements.



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

The third form creates a vector with an initial capacity of  $n$  elements that increases by  $\delta$  elements each time it needs to expand.



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
import java.util.*;
public class Vector_Demo
{
 public static void main(String args[])
 {
 int i;
 Vector v = new Vector();
 v.addElement(new Integer(10));
 v.addElement(new Float(5.5f));
 v.addElement(new String("Hi"));
 v.addElement(new Long(2500));
 v.addElement(new Double(23.25));
 System.out.println(v);
 String s = new String("Bhagirath");
 v.insertElementAt(s,1);
 System.out.println(v);
 v.removeElementAt(2);
 System.out.println(v);
 for(i=0;i<5;i++)
 {
 System.out.println(v.elementAt(i));
 }
 }
}
```

## Output :

Bhagirath

Hi

2500

23.25



<http://www.java2all.com>

The **Random** class allows you to generate random **double**, **float**, **int**, or **long** numbers.

This can be very helpful if you are building a simulation of a real-world system.

This class provides the following constructors.

**Random()**

**Random(**long start**)**

Here, start is a value to initialize **the random number generator**.



| Method                   | Description                                                                                                                                                     |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Double<br>nextDouble()   | Returns a random double value.                                                                                                                                  |
| Float nextFloat()        | Returns a random float value.                                                                                                                                   |
| Double<br>nextGaussian() | Returns a random double value. Numbers obtained from repeated calls to this method have a Gaussian distribution with a mean of 0 and a standard deviation of 1. |
| Int nextInt()            | Returns a random int value.                                                                                                                                     |
| Long nextLong()          | Returns a random long value.                                                                                                                                    |
| Method                   | Description                                                                                                                                                     |

```
import java.util.*;
public class Random_Demo
{
 public static void main(String args[])
 {
 Random ran = new Random();
 for(int i = 0; i<5;i++)
 {
 System.out.println(ran.nextInt());
 }
 }
}
```

### Output :

64256704  
1265771787  
-1962940029  
1372052



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

## Date class :

The Date classes encapsulate information about a specific **date and time**.

It provides the following constructors.

**Date()**

**Date(long msec)**

Here, the first form returns an object that represent the current date and time.

The second form returns an object that represents the date and time msec in milliseconds after



| <b>Method</b>               | <b>Description</b>                                                                           |
|-----------------------------|----------------------------------------------------------------------------------------------|
| Boolean<br>after(Date d)    | Returns true if d is after the current date.<br>Otherwise, returns false.                    |
| Boolean<br>before(Date d)   | Returns true if d is before the current date.<br>Otherwise, returns false.                   |
| Boolean<br>equals(Date d)   | Returns true if d has the same value as the current date. Otherwise, returns false.          |
| Long getTime()              | Returns the number of milliseconds since the epoch.                                          |
| Void setTime<br>(long msec) | Sets the date and time of the current object to represent msec milliseconds since the epoch. |
| String<br>toString()        | Returns the string equivalent of the date.                                                   |

```
import java.util.*;
public class Date_Demo
{
 public static void main(String args[])
 {
 Date dt = new Date();
 System.out.println(dt);

 Date epoch = new Date(0);
 System.out.println(epoch);
 }
}
```

## Output :

Fri May 25 00:04:06 IST 2012  
Thu Jan 01 05:30:00 IST 1970



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
import java.util.Date;
public class date2
{
 public static void main(String[] args)
 {
 Date d1 = new Date();

 try
 {
 Thread.sleep(1000);
 }
 catch(Exception e){}

 Date d2 = new Date();
 System.out.println("First Date : " + d1);
 System.out.println("Second Date : " + d2);
 System.out.println("Is second date after first ? : " + d2.after(d1));
 }
}
```

## Output :

First Date : Fri May 25 00:06:46 IST 2012  
Second Date : Fri May 25 00:06:47 IST  
2012

Is second date after first ? : true



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
import java.util.*;
public class date3
{
 public static void main(String args[])
 {
 Date date = new Date();
 System.out.println("Date is : " + date);
 System.out.println("Milliseconds since January 1, 1970, 00:00:00 GMT : " + date.getTime());
 Date epoch = new Date(0);
 date.setTime(10000);
 System.out.println("Time after 10 second " + epoch);
 System.out.println("Time after 10 second " + date);
 String st = date.toString();
 System.out.println(st);
 }
}
```

## Output :

Date is : Fri May 25 00:12:52 IST 2012

Milliseconds since January 1, 1970, 00:00:00 GMT :

**1337884972842**

Time after 10 second Thu Jan 01 05:30:00 IST 1970

**Time after 10 second Thu Jan 01 05:30:10 IST 1970**

Thu Jan 01 05:30:10 IST 1970



The Calendar class allows you to interpret date and time information.

This class defines several **integer constants** that are used when you get or set components of the calendar. These are listed here.



|             |                      |              |
|-------------|----------------------|--------------|
| AM          | AM_PM                | APRIL        |
| AUGUST      | DATE                 | DAY_OF_MONTH |
| DAY_OF_WEEK | DAY_OF_WEEK_IN_MONTH | DAY_OF_YEAR  |
| DECEMBER    | DST_OFFSET           | ERA          |
| FEBRUARY    | FIELD_COUNT          | FRIDAY       |
| HOUR        | HOUR_OF_DAY          | JANUARY      |
| JULY        | JUNE                 | MARCH        |
| MAY         | MILLISECOND          | MINUTE       |



|           |               |                                                                                                                                                                                                                                                                       |
|-----------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MONDAY    | MONTH         | NOVEMBER                                                                                                                                                                                                                                                              |
| OCTOBER   | PM            | SATURADAY                                                                                                                                                                                                                                                             |
| SECOND    | SEPTEMBER     | SUNDAY                                                                                                                                                                                                                                                                |
| THURSDAY  | TUESDAY       | UNDERIMBER                                                                                                                                                                                                                                                            |
| WEDNESDAY | WEEK_OF_MONTH | WEEK_OF_YEAR                                                                                                                                                                                                                                                          |
| YEAR      | ZONE_OFFSET   | <p>The Calendar class does not have public constructors. Instead, you may use the static getInstance() method to obtain a calendar initialized to the current date and time.</p>  |



One of its forms is shown here:

**Calendar getInstance()**



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
import java.util.Calendar;
public class Cal1
{
 public static void main(String[] args)
 {
 Calendar cal = Calendar.getInstance();

 System.out.println("DATE is : " + cal.get(cal.DATE));
 System.out.println("YEAR is : " + cal.get(cal.YEAR));
 System.out.println("MONTH is : " + cal.get(cal.MONTH));
 System.out.println("DAY OF WEEK is : " + cal.get(cal.DAY_OF_WEEK));
 System.out.println("WEEK OF MONTH is : " + cal.get(cal.WEEK_OF_MONTH));
 System.out.println("DAY OF YEAR is : " + cal.get(cal.DAY_OF_YEAR));
 System.out.println("DAY OF MONTH is : " + cal.get(cal.DAY_OF_MONTH));
 System.out.println("WEEK OF YEAR is : " + cal.get(cal.WEEK_OF_YEAR));
 System.out.println("HOUR is : " + cal.get(cal.HOUR));
 System.out.println("MINUTE is : " + cal.get(cal.MINUTE));
 System.out.println("SECOND is : " + cal.get(cal.SECOND));
 System.out.println("DAY OF WEEK IN MONTH is : " +
cal.get(cal.DAY_OF_WEEK_IN_MONTH));
 System.out.println("Era is : " + cal.get(cal.ERA));
 System.out.println("HOUR OF DAY is : " + cal.get(cal.HOUR_OF_DAY));
 System.out.println("MILLISECOND : " + cal.get(cal.MILLISECOND));
 System.out.println("AM_PM : " + cal.get(cal.AM_PM)); // Returns 0 if AM and 1 if PM
 }
}
```



## **Output :**

**Date is : Fri May 25 00:21:14 IST 2012**

**Milliseconds since January 1, 1970, 00:00:00 GMT :  
1337885474477**

**Time after 10 second Thu Jan 01 05:30:00 IST 1970**

**Time after 10 second Thu Jan 01 05:30:10 IST 1970**

**Thu Jan 01 05:30:10 IST 1970**



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

The GregorianCalendar class is a subclass of Calendar.

It provides the logic to **manage date and time information** according to the rules of the Gregorian calendar.

This class provides following constructors:

GregorianCalendar()

GregorianCalendar(**int year, int month, int date**)

GregorianCalendar(**int year, int month, int date, int hour, int minute, int sec**)



**GregorianCalendar(int year, int month, int date, int hour, int minute)**

The first form creates an object initialized with the current date and time.

The other forms allow you to specify how various date and time components are initialized.

The class provides all of the method defined by Calendar and also adds the **isLeapYear()** method shown here:

Boolean **isLeapYear()** This method returns true if the current year is a leap year. Otherwise, it returns false.



```
import java.util.*;
public class gcal1
{
 public static void main(String[] args)
 {
 GregorianCalendar c1 = new GregorianCalendar();

 System.out.println("DATE is : " + c1.get(c1.DATE));
 System.out.println("YEAR is : " + c1.get(c1.YEAR));
 System.out.println("MONTH is : " + c1.get(c1.MONTH));
 System.out.println("DAY OF WEEK is : " + c1.get(c1.DAY_OF_WEEK));
 System.out.println("WEEK OF MONTH is : " + c1.get(c1.WEEK_OF_MONTH));
 System.out.println("DAY OF YEAR is : " + c1.get(c1.DAY_OF_YEAR));
 System.out.println("DAY OF MONTH is : " + c1.get(c1.DAY_OF_MONTH));
 System.out.println("WEEK OF YEAR is : " + c1.get(c1.WEEK_OF_YEAR));
 System.out.println("HOUR is : " + c1.get(c1.HOUR));
 System.out.println("MINUTE is : " + c1.get(c1.MINUTE));
 System.out.println("SECOND is : " + c1.get(c1.SECOND));
 System.out.println("DAY OF WEEK IN MONTH is : " +
c1.get(c1.DAY_OF_WEEK_IN_MONTH));
 System.out.println("Era is : " + c1.get(c1.ERA));
 System.out.println("HOUR OF DAY is : " + c1.get(c1.HOUR_OF_DAY));
 System.out.println("MILLISECOND : " + c1.get(c1.MILLISECOND));
 System.out.println("AM_PM : " + c1.get(c1.AM_PM)); // Returns 0 if AM and 1 if PM */
 }
}
```

## Output :

Era is : 1

HOUR OF DAY is : 0

MILLISECOND : 780

AM\_PM : 0



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

## Math Class :

For scientific and engineering calculations, a variety of **mathematical** functions are required.

Java provides these functions in the Math class available in **java.lang** package.

The methods defined in Math class are given following:



| Method                   | Description                                         |
|--------------------------|-----------------------------------------------------|
| Double<br>sin(double x)  | Returns the sine value of angle x in radians.       |
| Double<br>cos(double x)  | Returns the cosine value of the angle x in radians  |
| Double<br>tan(double x)  | Returns the tangent value of the angle x in radians |
| Double<br>asin(double x) | Returns angle value in radians for arcsin of x      |
| Double<br>acos(double x) | Returns angle value in radians for arccos of x      |
| Double<br>atan(double x) | Returns angle value in radians for arctangent of x  |

|                                      |                                                                 |
|--------------------------------------|-----------------------------------------------------------------|
| Double<br>exp(double x)              | Returns exponential value of x                                  |
| Double<br>log(double x)              | Returns the natural logarithm of x                              |
| Double<br>pow(double x,<br>double y) | Returns x to the power of y                                     |
| Double<br>sqrt(double x)             | Returns the square root of x                                    |
| Int abs(double<br>n)                 | Returns absolute value of n                                     |
| Double<br>ceil(double x)             | Returns the smallest whole number greater than or<br>equal to x |
| Double<br>floor(double x)            | Returns the largest whole number less than or equal<br>to x     |

|                                   |                                          |
|-----------------------------------|------------------------------------------|
| Int max(int n, int m)             | Returns the maximum of n and m           |
| Int min(int n, int m)             | Returns the minimum of n and m           |
| Double<br>rint(double x)          | Returns the rounded whole number of x    |
| Int round(float x)                | Returns the rounded int value of x       |
| Long<br>round(double x)           | Returns the rounded int value of x       |
| Double<br>random()                | Returns a random value between 0 and 1.0 |
| Double<br>toRadians(double angle) | Converts the angle in degrees to radians |
| Double<br>toDegrees(double angle) | Converts the angle in radians to degrees |

```
public class Angles
{
 public static void main(String args[])
 {
 double theta = 120.0;
 System.out.println(theta + " degrees is " + Math.toRadians(theta) + " radians.");
 theta = 1.312;
 System.out.println(theta + " radians is " + Math.toDegrees(theta) + " degrees.");
 }
}
```

## Output :

120.0 degrees is 2.0943951023931953

radians.

1.312 radians is 75.17206272116401

degrees.



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Hashtable is a part of the **java.util** library and is a concrete implementation of a dictionary.

(Dictionary is a class that represents a **key/value** storage repository. Given a **key and value**, you can store the value in a **Dictionary object**. Once the value is stored, you can retrieve it by using its **key**.)

Hashtable stores **key/value** pairs in a **hash table**. When using a Hashtable, you specify an object that is used as a key, and the value that you want to link to that key.



The key is then **hashed**, and the resulting hash code is used as the index at which the value is stored within the table.

The Hashtable constructors are shown here:

Hashtable()

Hashtable(**int size**)

The first constructor is the default constructor.

The second constructor creates a hash table that has an initial size specified by size.

The methods available with Hashtable are



<http://www.java2all.com>

Syeda Ajrina Nusrat, Lecturer, CSE, UITS

| <b>Method</b>                       | <b>Description</b>                                                                                                          |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Void clear()                        | Resets and empties the hash table.                                                                                          |
| Boolean containsKey(Object key)     | Returns true if some key equals to key exists within the hash table. Returns false if the key isn't found.                  |
| Boolean containsValue(Object value) | Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.             |
| Enumeration elements( )             | Returns an enumeration of the values contained in the hash table.                                                           |
| Object get(Object key)              | Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned. |
| Boolean isEmpty( )                  | Returns true if the hash table is empty; Returns false if it contains at least one key.                                     |

|                                           |                                                                                                                               |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Enumeration<br>keys( )                    | Returns an enumeration of the keys contained in the hash table.                                                               |
| Object<br>put(Object key<br>Object value) | Inserts a key and a value into the hash table.                                                                                |
| Object<br>remove(Object<br>key)           | Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned. |
| Int size( )                               | Returns the number of entries in the hash table.                                                                              |
| String toString( )                        | Returns the string equivalent of a hash table.                                                                                |
| Enumeration<br>keys( )                    | Returns an enumeration of the keys contained in the hash table.                                                               |
| Object<br>put(Object key<br>Object value) | Inserts a key and a value into the hash table.                                                                                |

```
import java.util.*;
public class hash1
{
 public static void main(String args[])
 {
 Hashtable marks = new Hashtable();
 Enumeration names;
 Enumeration emarks;
 String str;
 int nm;

 // Checks wheather the hashtable is empty
 System.out.println("Is Hashtable empty " + marks.isEmpty());
 marks.put("Ram", 58);
 marks.put("Laxman", 88);
 marks.put("Bharat", 69);
 marks.put("Krishna", 99);
 marks.put("Janki", 54);

 System.out.println("Is Hashtable empty " + marks.isEmpty());
 // Creates enumeration of keys
 names = marks.keys();
 while(names.hasMoreElements())
 {
 str = (String) names.nextElement();
 System.out.println(str + ": " + marks.get(str));
 }
 }
}
```



```
/*
nm = (Integer) marks.get("Janki");
marks.put("Janki", nm+15);
System.out.println("Janki's new marks: " + marks.get("Janki"));

// Creates enumeration of values
emarks = marks.elements();
while(emarks.hasMoreElements())
{
 nm = (Integer) emarks.nextElement();
 System.out.println(nm);
}

// Number of entries in a hashtable
System.out.println("The number of entries in a table are " + marks.size());

// Checking wheather the element available

System.out.println("The element is their " + marks.containsValue(88));
*/
// Removing an element from hashtable
```

```
System.out.println("=====");
marks.remove("Bharat");
names = marks.keys();
while(names.hasMoreElements())
{
 str = (String) names.nextElement();
 System.out.println(str + ": " + marks.get(str));
}
// Returning an String equivalent of the Hashtable

System.out.println("String " + marks.toString());
// Emptying hashtable

marks.clear();
System.out.println("Is Hashtable empty " + marks.isEmpty());
}
```

### Output :

Laxman: 88

Janki: 54

Ram: 58

String {Krishna=99, Laxman=88, Janki=54,

Ram=58}

Is Hashtable empty true

Everyone's  
journey is  
different. Don't  
compare your path  
to anyone else's.

# Exception Handling

Exception is a run-time error which arises during the execution of java program. The term exception in java stands for an “**exceptional event**”.

So Exceptions are nothing but some abnormal and typically an event or conditions that arise during the execution which may interrupt the normal flow of program.

An exception can occur for many different reasons, including the following:



A user has entered invalid data.

A file that needs to be opened cannot be found.

A network connection has been lost in the middle of communications, or the JVM has run out of memory.

“If the exception object is not handled properly, the interpreter will display the error and will terminate the program.



Now if we want to continue the program with the remaining code, then we should write the part of the program which generate the error in the **try{ }** block and catch the errors using **catch()** block.

Exception turns the direction of normal flow of the program control and send to the related catch() block and should display error message for taking proper action. This process is known as.”

## **Exception handling**



The purpose of exception handling is to detect and report an exception so that proper action can be taken and prevent the program which is automatically terminate or stop the execution because of that exception.

Java exception handling is managed by using five keywords: **try, catch, throw, throws and finally**.

**Try:** Piece of code of your program that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.

**Catch:** Catch block can catch this exception and handle it in some logical manner.

**Throw:** System-generated exceptions are automatically thrown by the Java run-time system. Now if we want to manually throw an exception, we have to use the throw keyword.

**Throws:** If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException. A throws clause lists the types of exceptions that a method might throw.



This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

**Finally:** Any code that absolutely must be executed before a method returns, is put in a finally block.

**General form:**



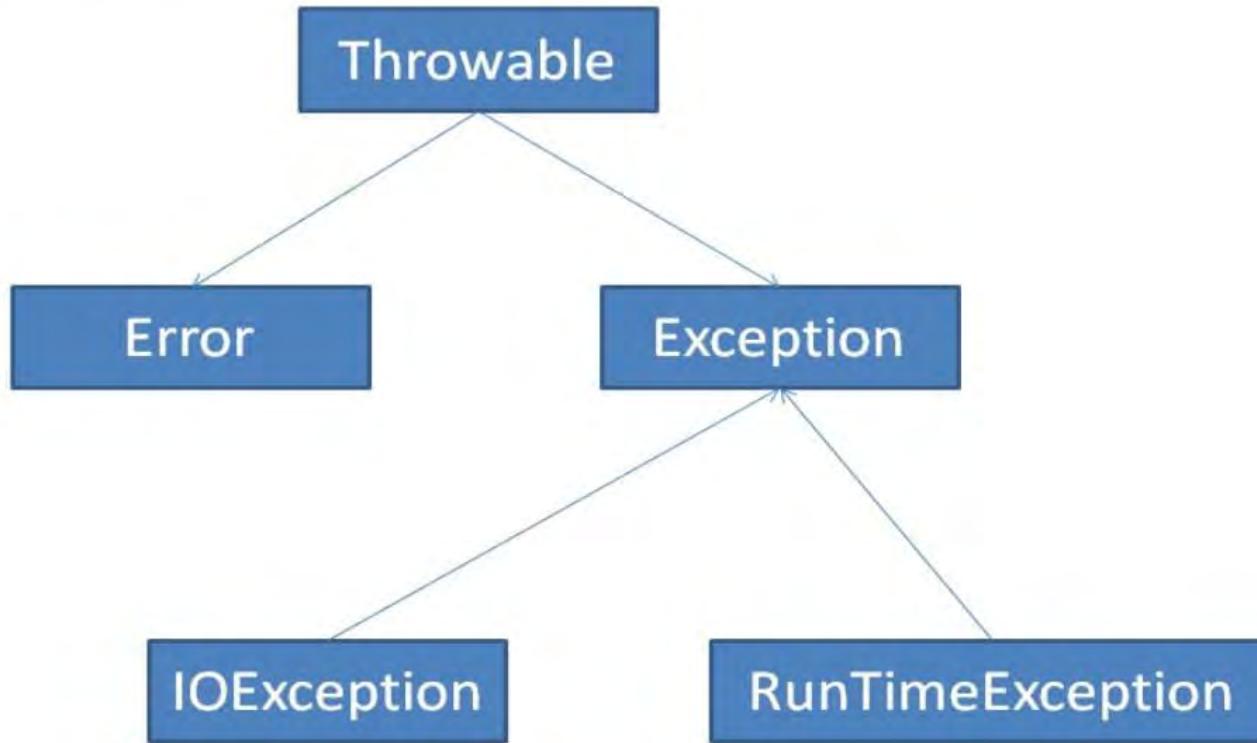
```
try {
 // block of code to monitor for errors
}

catch (ExceptionType1 e1) {
 // exception handler for ExceptionType1
}

catch (ExceptionType2 e2) {
 // exception handler for ExceptionType2
}
// ...
finally {
 // block of code to be executed before try block
ends
}
```



## Exception Hierarchy:



All exception classes are subtypes of the `java.lang.Exception` class.

The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class.

### **Errors :**

These are not normally trapped form the Java programs.

Errors are typically ignored in your code because you can rarely do anything about an error.



These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment.

### **For Example :**

- (1)** JVM is out of Memory. Normally programs cannot recover from errors.
  
- (2)** If a stack overflow occurs then an error will arise. They are also ignored at the time of compilation.



The Exception class has two main subclasses:

- (1) IOException or Checked Exceptions class and
- (2) RuntimeException or Unchecked Exception class

**(1) IOException or Checked Exceptions :**

Exceptions that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.



For example, if a file is to be opened, but the file cannot be found, an exception occurs.

These exceptions cannot simply be ignored at the time of compilation.

## **Java's Checked Exceptions Defined in `java.lang`**



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

| Exception                  | Meaning                                                                            |
|----------------------------|------------------------------------------------------------------------------------|
| ClassNotFoundException     | Class not found.                                                                   |
| CloneNotSupportedException | Attempt to clone an object that does not implement the <b>Cloneable</b> interface. |
| IllegalAccessException     | Access to a class is denied.                                                       |
| InstantiationException     | Attempt to create an object of an abstract class or interface.                     |
| Exception                  | Meaning                                                                            |
| ClassNotFoundException     | Class not found.                                                                   |
| CloneNotSupportedException | Attempt to clone an object that does not implement the <b>Cloneable</b> interface. |



|                       |                                                    |
|-----------------------|----------------------------------------------------|
| InterruptedException  | One thread has been interrupted by another thread. |
| NoSuchFieldException  | A requested field does not exist.                  |
| NoSuchMethodException | A requested method does not exist.                 |



## (2) **RuntimeException or Unchecked Exception :**

Exceptions need not be included in any method's **throws** list. These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

## **Java's Unchecked RuntimeException Subclasses**



| Exception                      | Meaning                                                           |
|--------------------------------|-------------------------------------------------------------------|
| ArithmaticException            | Arithmatic error, such as divide-by-zero.                         |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds.                                     |
| ArrayStoreException            | Assignment to an array element of an incompatible type.           |
| ClassCastException             | Invalid cast.                                                     |
| IllegalMonitorStateException   | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException          | Environment or application is in incorrect state.                 |
| IllegalThreadStateException    | Requested operation not compatible with current thread state.     |



|                                 |                                                     |
|---------------------------------|-----------------------------------------------------|
| IndexOutOfBoundsException       | Some type of index is out-of-bounds.                |
| NegativeArraySizeException      | Array created with a negative size.                 |
| NullPointerException            | Invalid use of a null reference.                    |
| NumberFormatException           | Invalid conversion of a string to a numeric format. |
| SecurityException               | Attempt to violate security.                        |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of a string.    |
| UnsupportedOperationException   | An unsupported operation was encountered.           |

# Try And Catch



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

We have already seen introduction about try and catch block in java exception handling.

Now here is the some examples of try and catch block.

**EX :**

```
public class TC_Demo
{
 public static void main(String[] args)
 {
 int a=10; int b=5,c=5; int x,y;
 try
 {
 x = a / (b-c);
 }
 catch(ArithmetcException e){
 System.out.println("Divide by zero");
 }
 y = a / (b+c);
 System.out.println("y = " + y);
 }
}
```

**Output :**

Divide by zero  
y = 1



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Note that program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and continues the execution, as if nothing has happened.

If we run same program without try catch block we will not gate the y value in output. It displays the following message and stops without executing further statements.

**Exception in thread "main"**

**java.lang.ArithmetricException: / by zero**  
**at Thrw\_Excp.TC\_Demo.main(TC\_Demo.java:10)**



Here we write ArithmeticException in catch block because it caused by math errors such as divide by zero.

## Now how to display description of an exception ?

You can display the description of thrown object by using it in a println() statement by simply passing the exception as an argument. For example;

```
catch (ArithmaticException e)
{
 system.out.println("Exception:" +e);
}
```



## Multiple catch blocks :

It is possible to have multiple catch blocks in our program.

**EX :**

```
public class MultiCatch
{
 public static void main(String[] args)
 {
 int a [] = {5,10}; int b=5;
 try
 {
 int x = a[2] / b - a[1];
 }
 catch(ArithmaticException e)
 {
 System.out.println("Divide by zero");
 }
 catch(ArrayIndexOutOfBoundsException e)
 {
 System.out.println("Array index error");
 }
 catch(ArrayStoreException e)
 {
 System.out.println("Wrong data type");
 }
 int y = a[1]/a[0];
 System.out.println("y = " + y);
 }
}
```

**Output :**

Array index error  
y = 2



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Note that array element a[2] does not exist.  
Therefore the index 2 is outside the array boundary.

When exception in try block is generated, the java treats the multiple catch statements like cases in switch statement.

The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

When you are using multiple catch blocks, it is important to remember that exception subclasses must come before any of their superclasses.



This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass will never be reached if it comes after its superclass. And it will result into syntax error.

**// Catching super exception before sub**

**EX :**



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
class etion3
{
 public static void main(String args[])
 {
 int num1 = 100;
 int num2 = 50;
 int num3 = 50;
 int result1;

 try
 {
 result1 = num1/(num2-num3);
 System.out.println("Result1 = " + result1);
 }

 catch (Exception e)
 {
 System.out.println("This is mistake. ");
 }
 catch(ArithmeticException g)
 {
 System.out.println("Division by zero");
 }
 }
}
```

## Output :

Array index error  
y = 2



## Output :

If you try to compile this program, you will receive an error message because the exception has already been caught in first catch block.

Since ArithmeticException is a subclass of Exception, the first catch block will handle all exception based errors,

including ArithmeticException. This means that the second catch statement will never execute.

To fix the problem, reverse the order of the catch statement.



## Nested try statements :

The try statement can be nested.

That is, a try statement can be inside a block of another try.

Each time a try statement is entered, its corresponding catch block has to entered.

The catch statements are operated from corresponding statement blocks defined by try.

## EX :

```
public class NestedTry
{
 public static void main(String args[])
 {
 int num1 = 100;
 int num2 = 50;
 int num3 = 50;
 int result1;
 try {
 result1 = num1/(num2-num3);
 System.out.println("Result1 = " + result1);
 try {
 result1 = num1/(num2-num3);
 System.out.println("Result1 = " + result1);
 }
 catch(ArithmeticException e)
 {
 System.out.println("This is inner catch");
 }
 }
 catch(ArithmeticException g)
 {
 System.out.println("This is outer catch");
 }
 }
}
```

## Output :

This is outer catch



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

# Finally



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.

We can put finally block after the try block or after the last catch block.

The finally block is executed in all circumstances. Even if a try block completes without problems, the finally block executes.



## EX :

```
public class Finally_Demo
{
 public static void main(String args[])
 {
 int num1 = 100;
 int num2 = 50;
 int num3 = 50;
 int result1;

 try
 {
 result1 = num1/(num2-num3);
 System.out.println("Result1 = " + result1);
 }
 catch(ArithmetricException g)
 {
 System.out.println("Division by zero");
 }
 finally
 {
 System.out.println("This is final");
 }
 }
}
```

### Output :

Division by zero  
This is final



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

# Throw



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

We saw that an exception was generated by the JVM when certain run-time problems occurred. It is also possible for our program to explicitly generate an exception.

This can be done with a throw statement. Its form is as follows:

### **Throw object;**

Inside a catch block, you can throw the same exception object that was provided as an argument.



This can be done with the following syntax:

```
catch(ExceptionType object)
{
 throw object;
}
```

Alternatively, you may create and throw a new exception object as follows:

Throw new ExceptionType(args);



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Here, exceptionType is the type of the exception object and args is the optional argument list for its constructor.

When a throw statement is encountered, a search for a matching catch block begins and if found it is executed.

## EX :

```
class Throw_Demo
{
 public static void a()
 {
 try
 {
 System.out.println("Before b");
 b0;
 }
 }
}
```



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
catch(ArrayIndexOutOfBoundsException j) //manually thrown object catched here
{
 System.out.println("J :" + j) ;
}
}

public static void b()
{
 int a=5,b=0;
 try
 { System.out.println("We r in b");
 System.out.println("*****");
 int x = a/b;
 }
 catch(ArithmeticException e)
 {
 System.out.println("c :" + e);
 throw new ArrayIndexOutOfBoundsException("demo"); //throw from here
 }
}
```



```
public static void main(String args[])
{
 try
 {
 System.out.println("Before a");
 a0;
 System.out.println("*****");
 System.out.println("After a");
 }
 catch(ArithmaticException e)
 {
 System.out.println("Main Program : " + e);
 }
}
```

## Output :

Before a  
Before b  
We r in b  
\*\*\*\*\*

C :  
java.lang.ArithmaticException: / by zero

J :  
java.lang.ArrayIndexOutOfBoundsException: demo  
\*\*\*\*\*

After a



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

## Throwing our own object :

If we want to throw our own exception, we can do this by using the keyword throw as follow.

**throw new Throwable\_subclass;**

**Example :** **throw new ArithmeticException( );**  
**throw new NumberFormatException( );**

**EX :**

```
import java.lang.Exception;
class MyException extends Exception
{
 MyException(String message)
 { super(message);
 }
}
```



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
class TestMyException
{
 public static void main(String[] args)
 {
 int x = 5, y = 1000;
 try
 {
 float z = (float)x / (float)y;
 if(z < 0.01)
 {
 throw new MyException("Number is too small");
 }
 }
 catch(MyException e)
 {
 System.out.println("Caught MyException");
 System.out.println(e.getMessage());
 }
 finally
 {
 System.out.println("java2all.com");
 }
 }
}
```

## Output :

Caught MyException  
Number is too small  
**java2all.com**



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Here The object e which contains the error message "Number is too small" is caught by the catch block which then displays the message using **getMessage( )** method.

## **NOTE:**

Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class. An object of a class that extends Throwable can be thrown and caught.



# Throws



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException.

A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error or RuntimeException**, or any of their subclasses.



All other exceptions that a method can throw must be declared in the throws clause.

If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws
exception-list
{
// body of method
}
```



Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Throw is used to actually throw the exception, whereas throws is declarative statement for the method. They are not interchangeable.

## EX :

```
class NewException extends Exception
{
 public String toS()
 {
 return "You are in NewException ";
 }
}
```



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

```
class customexception
{
 public static void main(String args[])
 {
 try
 {
 doWork(3);
 doWork(2);
 doWork(1);
 doWork(0);
 }
 catch (NewException e)
 {
 System.out.println("Exception : " + e.toS());
 }
 }
 static void doWork(int value) throws NewException
 {
 if (value == 0)
 {
 throw new NewException();
 }
 else
 {
 System.out.println("****No Problem.****");
 }
 }
}
```

## Output :

\*\*\*\*No Problem.\*\*\*\*

\*\*\*\*No Problem.\*\*\*\*

\*\*\*\*No Problem.\*\*\*\*

Exception : You are in  
NewException



<http://www.java2all.com>

Syeda Ajbina Nusrat, Lecturer, CSE, UITS

Put your own quote here!