HNDIT3012
Object Oriented
Programming

Course Aims

- To develop skills and knowledge required Object oriented concepts and programming by using JAVA programming language for software application development

- Course Code : HNDIT3012 Course
- Title : Object Oriented Programming
- Diploma Program : HNDIT
- Semester : 3 Course
- Status : Compulsory/ GPA
- Number of Credits : 4

Learning Outcomes

- After successful completion of this course the student should be able to:
- LO1: Analyze a solution to a programming problem using tools such as UML.
- LO2: Implement a solution to a problem using correct object-oriented programming concepts.
- LO3: Use the concepts in object orientation such as inheritance, polymorphism, and aggregation
- LO4: Design GUI applications using libraries and other package

Timetable allocation (per week)

- Lectures : 02 hours
- Tutorials /practicals : 04 hours
- Student activities : 07 hours
- Notional hours : 13 hours

Assessment Plan

• Continuous Assessments	40%
• Final Examination (3 hour paper)	60%
• Total	100%

Assignment plan

On-line quizzes 20%

Group Assignment 20%

Final Examination (03 hour paper) 60%

Total 100%

Object Oriented Approch

- In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.
- In analysis phase, OO models are used to fill the gap between problem and solution. It performs well in situation where systems are undergoing continuous design, adaption, and maintenance. It identifies the objects in problem domain, classifying them in terms of data and behavior.

Structured Approach Vs. Object-Oriented Approach

Structured Approach

- Program is divided into number of sub modules or functions.
- Function call is used.
- Problem oriented approach
- Software reuse is not possible.

Object Oriented Approach

- Program is organized by having number of classes and objects.
- Message passing is used.
- Problem domain oriented approach
- Reusability is possible.

UML-introduction

Problem/problem domain

Problem

- Understand problem.
- Consider problem.
- Directly Solve the problem step by step.
- Problem may be changed in future
- Example Add the two marks and find the total and avg (consider marks)
- In design DFD and flow chart

Problem domain

- Understand problem domain.
- Consider problem environment.
- Create the domain and solve the problem step by step.
- Problem domain can't changed changed in future
- Example Add the two marks of student and find the total and avg (consider student))
- In design UML

UML

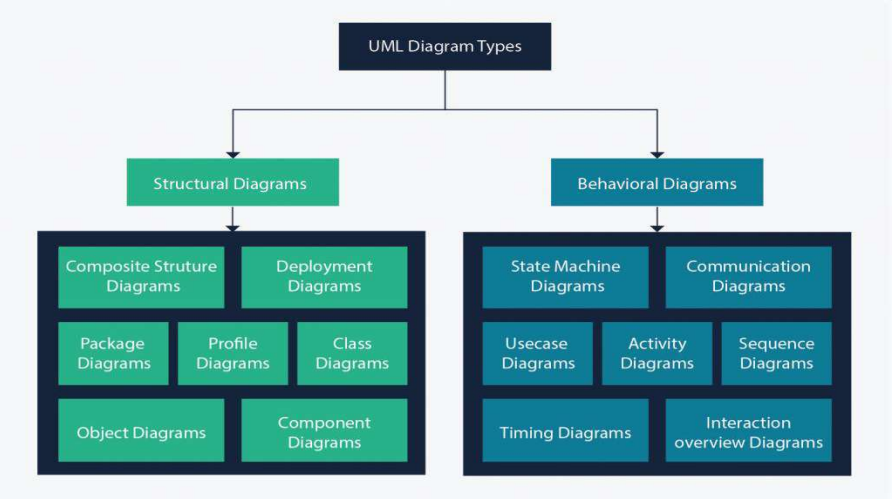
- UML stands for **U**nified **M**odeling **L**anguage. It's a rich language to model software solutions, application structures, system behavior and business processes.
- There are two main categories;

  1. structure diagrams

  2. behavioral diagrams.

Structure Diagrams

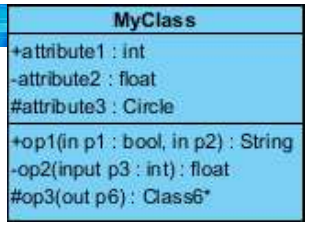
- **Structure diagrams** show the things in the modeled system. In a more technical term, they show different objects in a system.
  - Class Diagram
  - Component Diagram
  - Deployment Diagram
  - Object Diagram
  - Package Diagram
  - Profile Diagram
  - Composite Structure Diagram



Behavioral Diagrams

- show what should happen in a system. They describe how the objects interact with each other to create a functioning system.
  - Use Case Diagram
  - Activity Diagram
  - State Machine Diagram
  - Sequence Diagram
  - Communication Diagram
  - Interaction Overview Diagram
  - Timing Diagram

Class Diagram



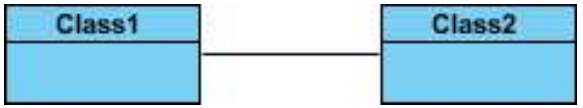
- Class diagrams are the main building block of any object-oriented solution. It shows the classes in a system, attributes, and operations of each class and the relationship between each class.
- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations
- ~ denotes package attributes or operations

Class Relationships

- A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:
  - **Inheritance** (or Generalization)
  - **Simple Association**:
  - **Aggregation**:
  - **Composition**:
  - **Dependency**:

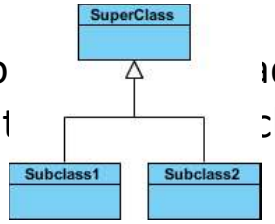
Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2
- A solid line connecting two classes



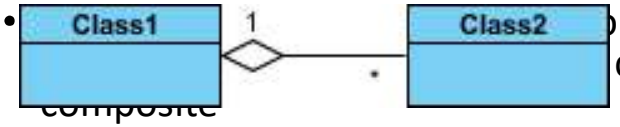
Inheritance (or Generalization)

- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of Super Class.
- A solid line with a ho ad that point from the child t class



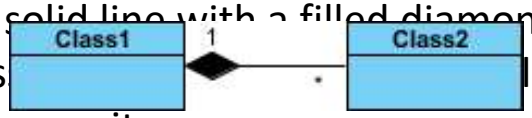
Aggregation.

- A special type of association. It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- nd at the class of composite

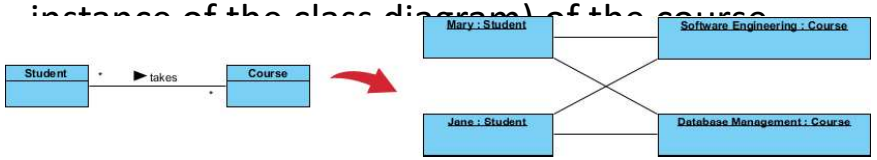


Composition:

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the as lass of composite



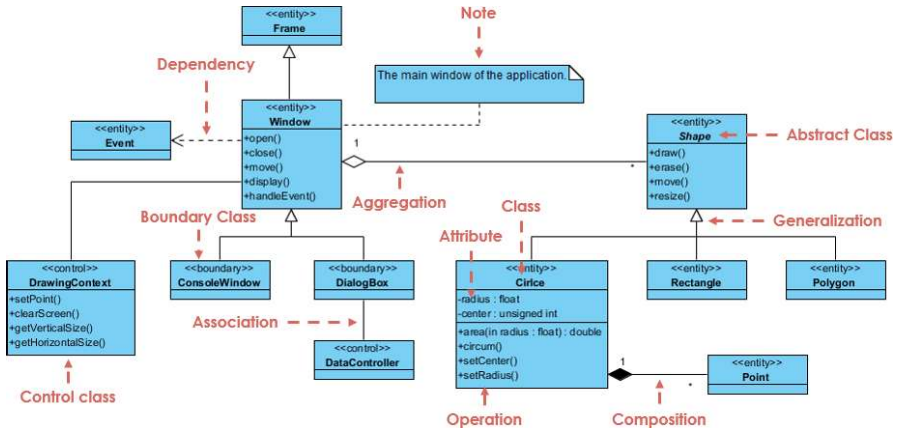
- Requirement: A Student can take many Courses and many Students can be enrolled in one Course.
- In the example below, the **class diagram** (on the left), describes the statement of the requirement above for the static model while the object diagram (on the right) shows the snapshot (an instance of the class diagram) of the course



Multiplicity

How many objects of each class take part in the relationships and multiplicity can be expressed as:

•Exactly one - 1
•Zero or one - 0..1
•Many - 0..* or *
•One or more - 1..*
•Exact Number - e.g. 3..4 or 6
•Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5



What is a use case diagram?

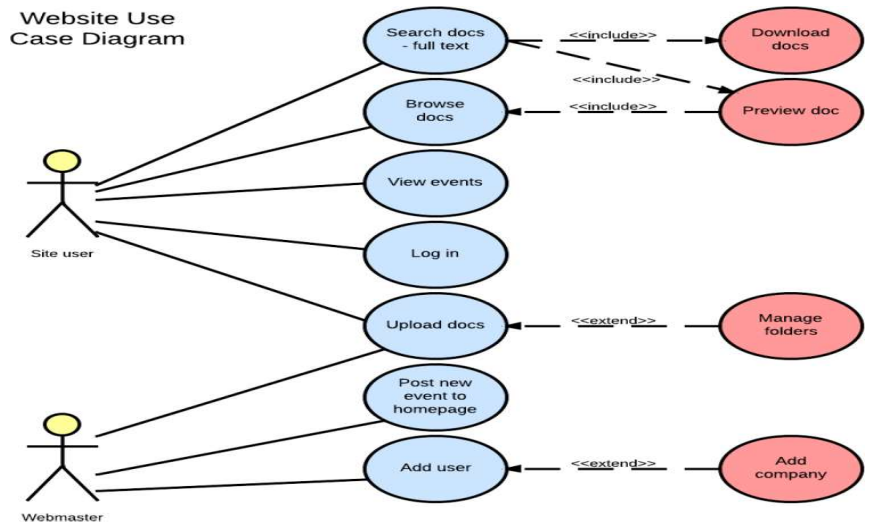
- In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:
  - Scenarios in which your system or application interacts with people, organizations, or external systems
  - Goals that your system or application helps those entities (known as actors) achieve
  - The scope of your system

Use case diagram symbols and notation

- Use cases: Horizontally shaped ovals that represent the different uses that a user might have.
- Actors: Stick figures that represent the people actually employing the use cases.
- Associations: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- System boundary boxes: A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho Killer is outside the scope of occupations in the chainsaw example found below.
- Packages: A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.

Purpose of USE CASE diagrams

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case



Sequence Diagrams

- UML Sequence Diagrams are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when.

Sequence Diagrams at a Glance

- Sequence Diagrams show elements as they interact over time and they are organized according to object (horizontally) and time (vertically):

Purpose of Sequence Diagram

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

Object Dimension

- The horizontal axis shows the elements that are involved in the interaction
- Conventionally, the objects involved in the operation are listed from left to right according to when they take part in the message sequence. However, the elements on the horizontal axis may appear in any order

Time Dimension

- The vertical axis represents time proceedings (or progressing) down the page.

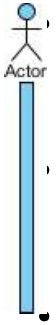
- Note
  - Time in a sequence diagram is all a about ordering, not duration. The vertical space in an interaction diagram is not relevant for the duration of the interaction.

Lifeline

- A lifeline represents an individual participant in the Interaction

Notation

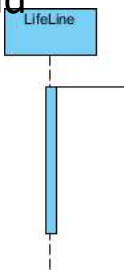
- Actor



- a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)
- external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).
- represent roles played by human users, external hardware, or other subjects.

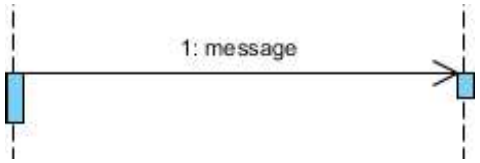
Activations

- A thin rectangle on a lifeline) represents the period during which an element is performing an operation.
- The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively



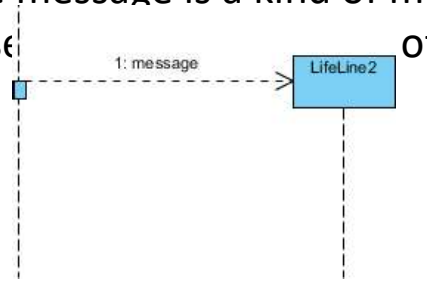
Call Message

- A message defines a particular communication between Lifelines of an Interaction.
- Call message is a kind of message that represents an invocation of operation of target lifeline.



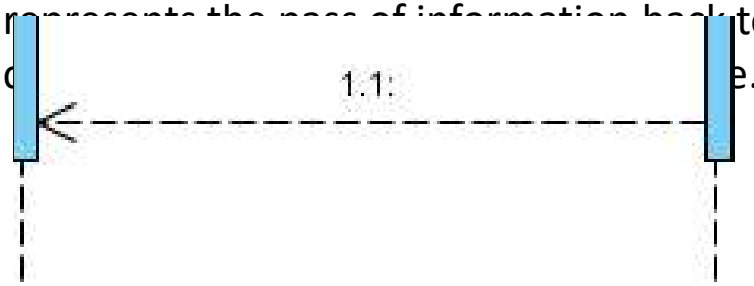
Create Message

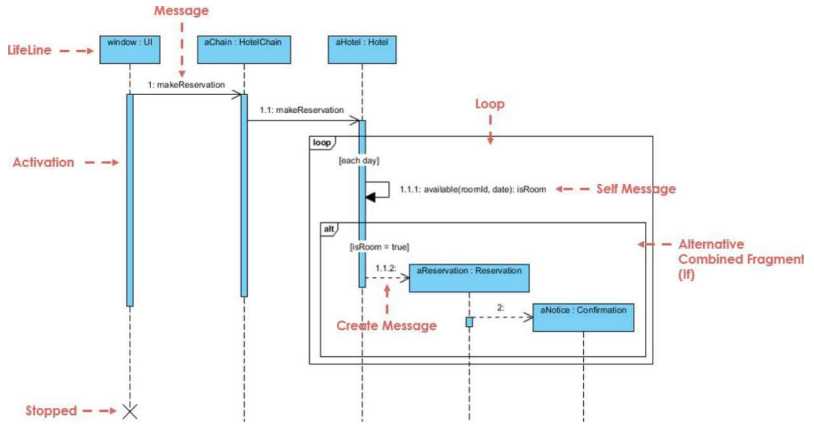
- A message defines a particular communication between Lifelines of an Interaction.
- Create message is a kind of message that represe of (target) lifeline.



Return Message

- A message defines a particular communication between Lifelines of an Interaction.
- Return message is a kind of message that represents the pass of information back to the e.





- Free UML Tool (visual-paradigm.com)

Thank you

HNDIT3012 Object Oriented Programming

Fundermental of java


```
class HelloWorldApp
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!"); // Display the text.
    }
}
```

- Save with **.java** extension
- If a public class is present, the class name should match the file name

Hello World

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword is used to create static method. no need to create object to invoke the static method
- **void** is the return type of the method
- **main** represents startup of the program.
- **String[] args** is used for command line argument.
- **System.out.println()** is used print statement.

Variable

- The Java programming language defines the following kinds of variables:
- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters** You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Data Type

Data Type	Default Value	Description	Range
byte	0	an 8-bit signed two's complement integer	-128 to127
short	0	a 16-bit signed two's complement integer.	32,768 to 32,767
int	0	a 32-bit signed two's complement integer	-2^{31} to $2^{31}-1$.
long	0L	a 64-bit two's complement integer	-2^{63} and a maximum value of $2^{63}-1$
float	0.0f	a single-precision 32-bit IEEE 754 floating point.	
double	0.0d	a double-precision 64-bit IEEE 754 floating point.	
char	'\u0000'	a single 16-bit Unicode character	'\u0000' (or 0) - '\uffff' (or 65,535 inclusive)

Operators

- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators

literals

- boolean result = true;
- char capitalC = 'C';
- byte b = 100;
- short s = 10000;
- int i = 100000

- //The number 26, in decimal
- int decVal = 26;

- // The number 26, in hexadecimal
- int hexVal = 0x1a;

- // The number 26, in binary
- int binVal = 0b11010;
- double d1 = 123.4;

- // same value as d1, but in scientific notation
- double d2 = 1.234e2;
- float f1 = 123.4f;

- Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics) (oracle.com)

- //underscore
- long creditCardNumber = 1234_5678_9012_3456L;
- long socialSecurityNumber = 999_99_9999L;
- float pi = 3.14_15F;
- long hexBytes = 0xFF_EC_DE_5E;
- long hexWords = 0xCAFE_BABE;
- long maxLong = 0x7fff_ffff_ffff_ffffL;
- byte nybbles = 0b0010_0101;
- long bytes = 0b11010010_01101001_10010100_10010010;

Arithmetic operators

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator
++	Increment operator
--	Decrement operator

Operator Precedence

*	/	%
+	-	

When operators of equal precedence appear in the same expression they are evaluated from left to right;

Relational Operators

Operator	Meaning
!=	not equal to
==	is equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to

Output?

```
public class Arith
 { public static void main(String[] args)
      { //Variables Definition and Initialization
       int number1 = 12, number2 = 4;
       //Addition Operation
       int sum = number1 + number2; System.out.println("Sum is: " +
      sum);
       //Subtraction Operation
       int dif = number1 - number2;
       System.out.println("Difference is : " + dif);
       //Multiplication Operation
       int mul = number1 * number2;
       System.out.println("Multiplied value is : " + mul);
       //Division Operation int div = number1 / number2;
       System.out.println("Quotient is : " + div);
       //Modulus Operation int rem = number1 % number2;
       System.out.println("Remainder is : " + rem);
      }
      }
```

Output?

```
public class Relation
{
 public static void main(String[] args)
 {              int one = 1;
                int two = 2;
                int three = 3;
                int four = 4;
                boolean oneIsOne = one == one;
                boolean res1 = two <= three;
                boolean res2 = two != four;
                boolean res3 = two > four;
                boolean res4 = one == three;


                System.out.println("one == one "+oneIsone);
                System.out.println(" two <= three "+res1);
                System.out.println("two != four  "+res2);
                System.out.println("two > four "+res3);
                System.out.println("one == three "+res4);


}
}
```

Logical Operators

Operator (Boolean Operators)	Operator (Short Circuit Operators Condition Operator)	Meaning
&	&&	Logical AND
|	||	Logical OR
!		Logical NOT

- short circuit logical operators evaluate second expression only if that is needed.

Output?

```
import java.util.Scanner;
public class CheckDescOrder
{
 public static void main(String[] args)
 {        Scanner scanner = new Scanner(System.in);
          int h1 = scanner.nextInt();
          int h2 = scanner.nextInt();
          int h3 = scanner.nextInt();
          boolean descOrdered = (h1 >= h2) && (h2 >= h3);
         System.out.println(descOrdered);

}
}
```


Bitwise Operators

- **Acts on individual bits of an integer**

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Right Shift
<<	Left Shift

Output?

```
public class bitwiseop {
public static void main(String[] args)
{
          //Variables Definition and Initialization
          int num1 = 30, num2 = 6, num3 =0;
          //Bitwise AND
          System.out.println("num1 & num2 = " + (num1 & num2));
          //Bitwise OR
          System.out.println("num1 | num2 = " + (num1 | num2) );
          //Bitwise XOR
          System.out.println("num1 ^ num2 = " + (num1 ^ num2) );
          //Binary Complement Operator
          System.out.println("~num1 = " + ~num1 );
          //Binary Left Shift Operator
          num3 = num1 << 2; System.out.println("num1 << 1 = " + num3 );
          //Binary Right Shift Operator
          num3 = num1 >> 2; System.out.println("num1 >> 1 = " + num3 );
          //Shift right zero fill operator
          num3 = num1 >>> 2; System.out.println("num1 >>> 1 = " + num3 );
}}
```

Operator Precedence

Operators	Precedence
Parentheses	()
unary	negative (-), logical NOT (!)
multiplicative	* / %
additive	+ -
relational	< > <= >=
equality	== !=
logical AND	&&
logical OR	||

SLIATE moodle

Thank you

Object Oriented programing

Week 3 Control Flow Statements

Control flow structures

- Sequential structure :
  - Default mode. Statements are executed from top to bottom of program one by one.
- Selection structure:
  - **Selection structure** or conditional **structure**, is different blocks of code are performed based on whether a boolean condition is true or false
  - If-else, switch-case
- Repetition structure:
  - Same block of code is executed again and again based on whether a boolean condition is true or false
  - while, do-while, for

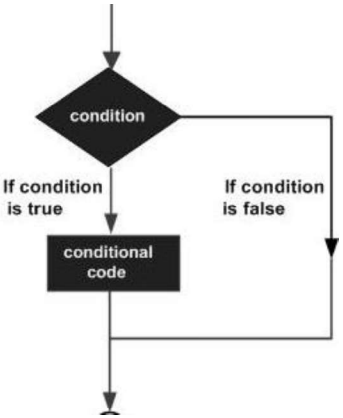
Control flow

control flow (or **flow** of **control**) is the order in which individual statements of an **program** are executed .

If condition

- Syntax

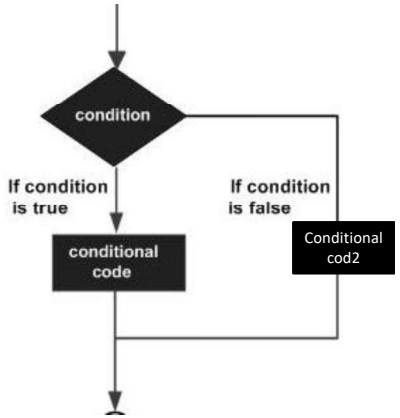
```
If(expression)
{
statement(s)
}
```



If else condition

- Syntax

```
If(expression)
{
Conditional code
}
Else
{
Conditional code 2
}
```



Nested If condition

Syntax

```
If(expression) {    statement(s)} else{statement(s)} }
```

Example

```
public static void main(String[] args)
  {
     int Marks=92;
     if (Marks<40 )
          System.out.print("Result : Fail");
     else
          {
          if (Marks<60 )
                    System.out.print("Result : Simple Pass");
               else
                    System.out.print("Result : Credit Pass");
          }
}
```

If condition

Syntax

```
If(expression) {    statement(s)} else{statement(s)} }
```

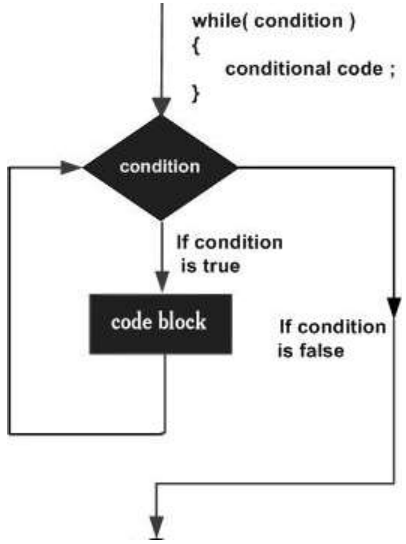
Example

```
public static void main(String[] args)
  {
     int Marks=92;
     if (Marks<40 )
          System.out.print("Result : Fail");
     else
          {
                         System.out.print("Result :  Pass");
                    }
}
```

while loop

- Syntax

```
while (expression)
{
statement(s)
}
```



while loop

Syntax

```
while (expression) {     statement(s)}
```

Example

```
public static void main(String[] args)
  {
        int count = 1;
        while (count < 11)
        {
              System.out.println("Count is: " + count);
              count++;
        }
}
```

do while loop

Syntax

```
do {     statement(s)} while (expression)
```

Example

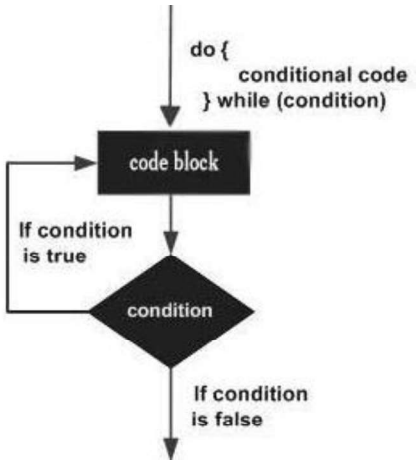
```
public static void main(String[] args)
  {
        int count = 1;
        do
        {
              System.out.println("Count is: " + count);
              count++;
        }
        while (count < 11)
 }
```

statements within the do block are always executed at least once

Do while loop

- Syntax

```
do
 {
statement(s)
}
 while (expression)
```

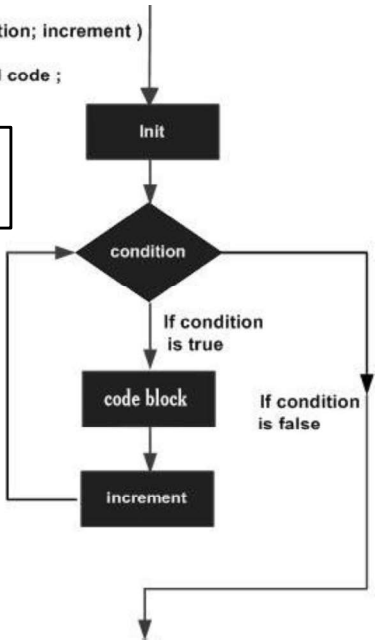


For loop

- Syntax

```
for(initialization; expression; change)
 {}
```

- Change can be increment or decrement



for loop

Syntax

```
for(initialization; expression; change) {}
```

Example

```
public static void main(String[] args)
   {
          for(count=0;count < 10;count++) )
                    {
                     System.out.println("Count is: " + count);
                    }
}
```

- The ***initialization*** expression initializes the loop; it's executed once, as the loop begins.
- When the ***termination* expression** evaluates to false, the loop terminates.
- The ***change*** expression (increment/decrement) is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value

break statement

Example

```
class Test
 {
 public static void main(String[] args)
   {
          int count ;
          for(count=0;count < 10;count++)
                    {
                    if (count==5)
                     {break;}
                    System.out.println("Counting: " + count);
                    }
   }
 }
```

break statement

- Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
- The break statement has two forms:
  - labeled and unlabeled.
- Unlabeled break
  - Break out from a case
  - Terminate a for, while, or do-while loop

Continue statement

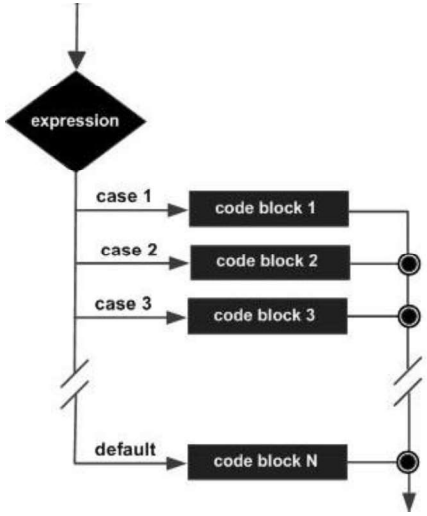
- Continue statement skips the current iteration of a for, while , or do-while loop

```
class Test
  {
  public static void main(String[] args)
       {
        int count ;
        for(count=0;count < 10;count++)
              {
              if (count==5)
              {continue;}
            System.out.println("Counting: " + count);
              }
       }
  }
```

switch Statement

- Syntax

```
switch ( expression )
{
    case value:
          statement(s);
          break;
    case value:
          statement(s);
          break;
    default:
          statement(s);
}
```



- keyword **break** is needed to break out of each case.
- **default** will execute, only if the execution skip from all the cases

THANK YOU

switch Statement

Example

```
public static void main(String[] args) {
       int Day = 4;
      String DayString;
      switch (Day) {
         case 1:  DayString = "Sunday";     break;
         case 2:  DayString = "Monday";     break;
         case 3:  DayString = "Tuesday";    break;
         case 4:  DayString = "Wednesday";  break;
         case 5:  DayString = "Thursday";   break;
         case 6:  DayString = "Friday";     break;
         case 7:  DayString = "Saturday";   break;
         default: DayString = "Invalid Day"; break;
      }
      System.out.println(DayString);
   }
}
```

HNDIT3012
Object Oriented Programming

Object Oriented Programing

- The main ideas behind Java's Object-Oriented Programming, OOP concepts include
  - **Encapsulation**,
  - **Polymorphism**,
  - **Inheritance,**
  - **Abstraction**

  etc

Object Oriented Programing

- Object Oriented Programming (OOP) is a programming paradigm that focuses on the use of objects to represent and manipulate data with real world concepts.
- In OOP, data is encapsulated within objects, and objects are defined by their properties (attributes) and behaviors (methods).
- OOP provides several key concepts that enable developers to write modular, reusable, and maintainable code.

class and object

Vehicle car;

doors	**5**
speed	180
color	white

Vehicle bike;

doors	**0**
speed	90
color	brown

object1

Object 2

Access Modifier Class Name class

```
public class Vehicle {
    private int doors;
    private int speed;
    private String color;

    public void run(){
        //Run method implementation.
    }
}
```

Class Members Instance Variables

Class Members Instance Method

Class Body

Object3

Vehicle racecar;

doors	**2**
speed	**400**
color	red

What Is an Object?

- An object is a software bundle of related state and behavior.
- Software objects are often used to model the real-world objects that you find in everyday life.

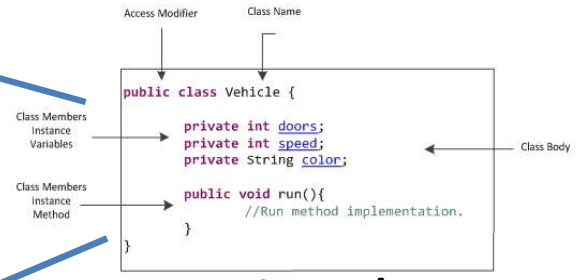
Vehicle car

doors	5
speed	180
color	white

Vehicle bike

doors	0
speed	90
color	brown

class and object



class

Vehicle racecar

doors	2
speed	400
color	red

What Is a Class?

- A class is a blueprint or prototype from which objects are created.
- This section defines a class that models the state and behavior of a real-world object.
- It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

class definition

```
class Vehicle
{ private int door;
  private int speed;
  private string color;
  public void run()
            {
            System.out.println("Vehicle  is running speed is "+speed);
            }
}
```

Object Creation and Usage

```
Class VehicleUse
{
public static void main (String args[])
{
Vehicle car=new Vehicle();
Vehicle bike=new Vehicle();
Vehicle raceCar=new Vehicle();
car.run();
Bike.run();
racecar.run();
}
}
```

Thank you

HNDIT3012
Object Oriented Programming

Types of Java constructors

- There are two types of constructors in Java:
  - Default constructor (no-arg constructor)
  - Parameterized constructor
-

Java Constructors

- A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
- Note that the constructor name must match the class name, and it cannot have a return type (like void).

Types of Java constructors…

```
class Vehicle {
//creating a default constructor
Vehicle()
{
System.out.println(" Vehicle  is created");
}
//main method
public static void main(String args[])
{
//calling a default constructor
Vehicle  b=new  Vehicle();
}
}
```

```
class Vehicle {
//creating a default constructor
Vehicle(int color)
{
String color;
System.out.println(" Vehicle  is created“
+color);
}
//main method
public static void main(String args[])
{
//calling a single parameter  constructor
Vehicle  b=new  Vehicle(“red”);
}
}
```

Difference between constructor and method in Java

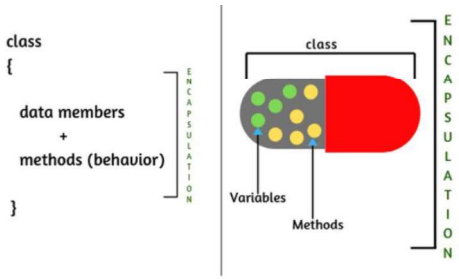
a Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler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

Inheritance

- A class that inherits from another class can reuse the methods and fields of that class.
- Java, Inheritance means creating new classes based on existing ones.
- it is possible to inherit attributes and methods from one class to another

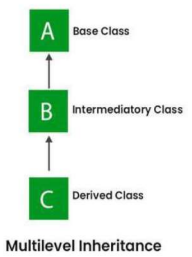
Encapsulation

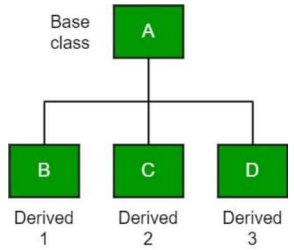
- It refers to the bundling of data with the methods that operate on that data
- Encapsulation in Java is the process by which data (variables) and the code that acts upon them (methods) are integrated as a single unit. By encapsulating a class's variables, other classes cannot access them, and only the methods of the class can access them



"inheritance concept"

- We group the into two categories:
- **subclass** (child/Drieved) - the class that inherits from another class
- **superclass** (parent/Base) - the class being inherited from





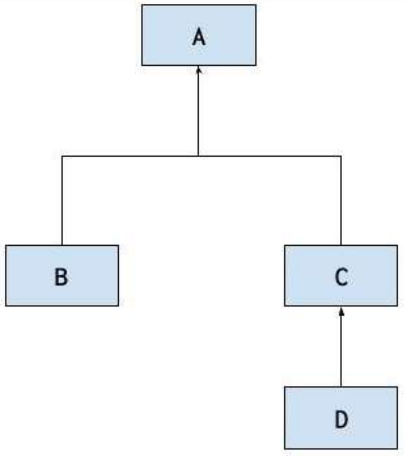
```
class A {

}
class B extends A {

}
class C extends A {

}
class D extends C {

}
```



Hybrid Inheritance - BeginnersBook.com

Child class

```
public class Cat extends Animal{
private String color;
public Cat(boolean veg, String food, int legs)
{ super(veg, food, legs);
  this.color="White"; }
public Cat(boolean veg, String food, int legs,
String color)
{ super(veg, food, legs); this.color=color; }
public String getColor()
{ return color; }
public void setColor(String color)
{ this.color = color; } }
```

Base class

```
public class Animal
{      private boolean vegetarian;
       private String eats;
       private int noOfLegs;
 public Animal(){}
 public Animal(boolean veg, String food, int legs)
 {this.vegetarian = veg;
  this.eats = food;
  this.noOfLegs = legs; }
 public boolean isVegetarian()
 { return vegetarian; }
 public void setVegetarian(boolean vegetarian)
 { this.vegetarian = vegetarian; }
 public String getEats()
 { return eats; }
 public void setEats(String eats)
 { this.eats = eats; }
 public int getNoOfLegs()
 { return noOfLegs; }
 public void setNoOfLegs(int noOfLegs)
 { this.noOfLegs = noOfLegs; }
}
```

Used class/main

```
public class AnimalUse {
public static void main(String[] args)
{
Cat cat = new Cat(false, "milk", 4, "black");
System.out.println("Cat is Vegetarian?"+
cat.isVegetarian());
System.out.println("Cat eats " + cat.getEats());
System.out.println("Cat has " + cat.getNoOfLegs()+
" legs.");
System.out.println("Cat color is "+
cat.getColor());
}
}
```

Super Keyword in Java

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

super is used to refer immediate parent class instance variable

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class Test1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Usage of Java super Keyword

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

super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class Test2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

super is used to invoke parent class constructor

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

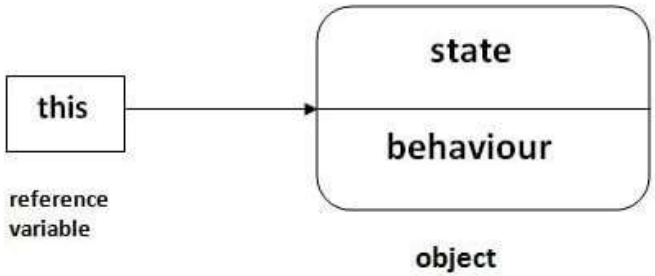
Usage of Java this keyword

Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

this

- here can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



current class instance variable

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

invoke current class method

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

pass as an argument in the method

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

invoke current class constructor

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

pass as argument in the constructor call

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

can be used to return current class instance

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

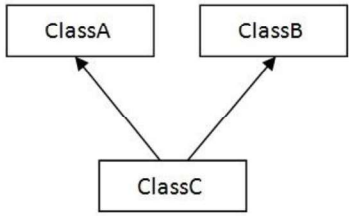
THANK YOU

HNDIT3012 Object Oriented Programming

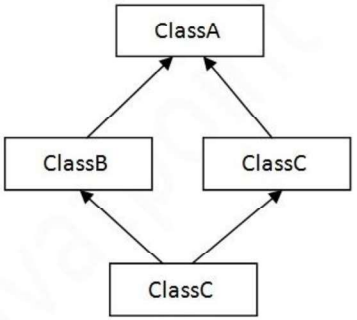
Inheritance

Inheritance

- Multiple inheritance is not supported in java



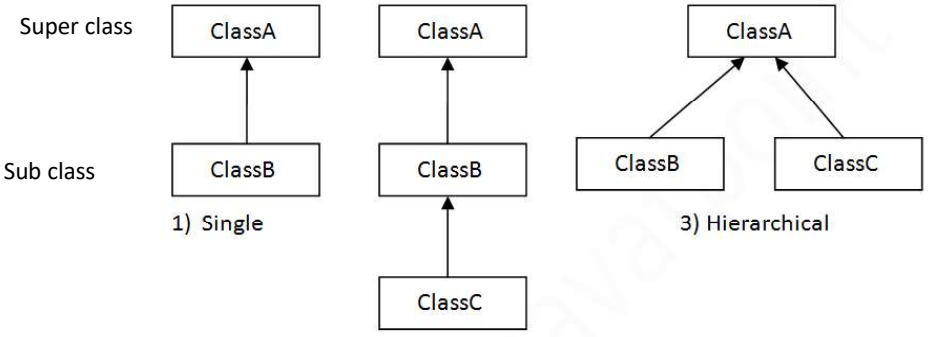
4) Multiple



5) Hybrid

Inheritance

- Forming a new class from an existing class
  - New class includes the old class information

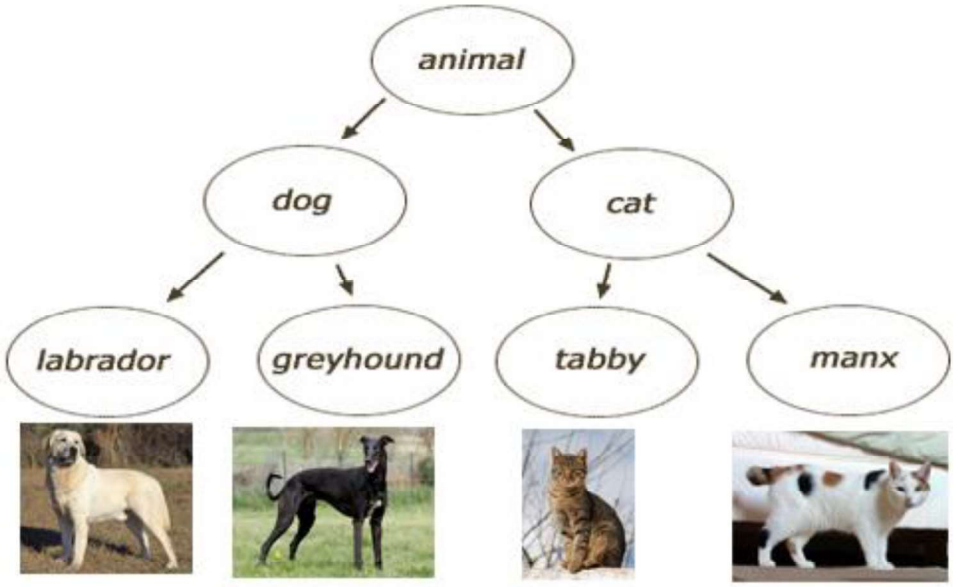


1) Single

2) Multilevel

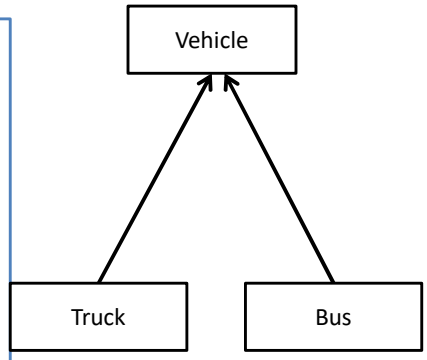
3) Hierarchical

Inheritance



Inheritance

```
class Vehicle
        {
         String Make;
         String fuelType;
         double Price ;
         int Year;
        }
class Truck extends Vehicle
        {
         int cubes=3;
        }
class Bus extends Vehicle
        {
         int no_of_Seats=55;
        }
```



Subclass does not inherit
the private members of its parent class.

super keyword

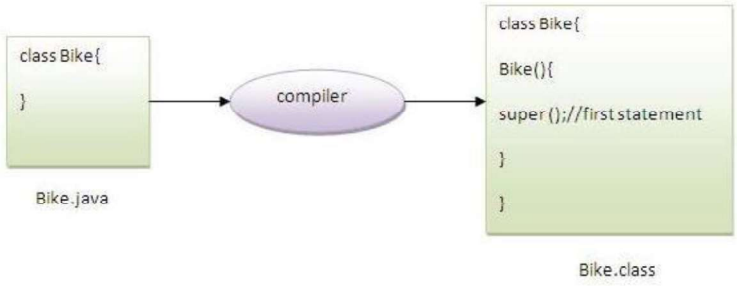
- **super** keyword is a reference variable that is used to refer immediate parent class object.
- super keyword is used to invoke immediate parent class method.

Inheritance

- Advantages
  - Code reusability
  - Reduce duplicate code
  - Reduce development time and effort
  - Provide specialization
  - Extendibility
    - Can extend an already made classes by adding new features
- Disadvantage
  - Tide coupling
  - When the no of inheriting levels are increasing , maintenance is difficult

super()

- **super()** is used to invoke immediate parent class constructor



THANK YOU

HNDIT3012
Object Oriented Programming

polymorphism

Overriding

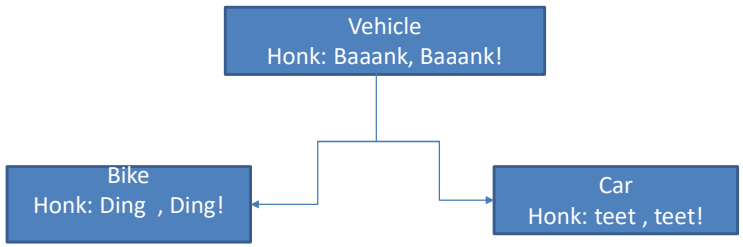
- Known as runtime polymorphism
- If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding
- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**
- A method declared final cannot be overridden
- Static methods are not possible to override

Polymorphism

- **polymorphism** is a feature that allows you to provide a single interface to varying entities of the same type.
- Same name to deferent operations
  - Overloading
  - Overriding

Overriding - Advantages

- Provide specific implementation for object
- Provide specific implementation for method that is already provided by its super class.



```
class Vehicle
{
public void honk() {
 System.out.println("Baaank, Baaank!");
                        }
}


class Bike extends Vehicle
{
public void honk() {
 System.out.println("Ding  , Ding!");
                        }
}
```

```
class Car extends Vehicle
{
public void honk() {
 System.out.println("teet , teet!");
                        }
}


class Main{
 public static void main(String[] args)
                {
 Vehicle myVehicle=new Vehicle();
 myVehicle.honk();
 Car myCar = new Car();
  myCar.honk();
 Bike myBike= new Car();
  myBike.honk();

                }
                }
```

```
class Calculation
 {
            int sum(int a,int b)
            {
            return a+b;
            }
            int sum(int a,int b,int c)
            {
            return a+b+c;
            }
            public static void main(String args[])
            {
            Calculation obj=new Calculation();
            int tot;
            tot=obj.sum(10,10,10);
            System.out.println("Total :"+tot);
            tot=obj.sum(10,10);
            System.out.println("Total :"+tot);
            }
}
```

Overloading

- **Overloading** is a feature that allows a class to have two or more methods having same name, if their argument lists are different.
- Two ways to overloading
  - By changing number of arguments
  - By changing the data type

Advantages of overloading

- Increases the readability
  - call a similar method for different types of data
- providing multiple behavior to same object with respect to attributes of object

Constructor overloading

- A class can have two or more different implementations by having different constructors.

```java
// Java program to illustrate
// Constructor Overloading
class Box {
    double width, height, depth;

    // constructor used when all dimensions
    // specified
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions
    // specified
    Box() { width = height = depth = 0; }

    // constructor used when cube is created
    Box(double len) { width = height = depth = len; }

    // compute and return volume
    double volume() { return width * height * depth; }
}
```

```java
// Driver code
public class Test {
    public static void main(String args[])
    {
        // create boxes using the various
        // constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

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

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

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

THANK YOU

HNDIT3012
Object Oriented Programming

Abstraction and Interfaces

Abstraction

- Two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)

Abstraction

- **Abstraction** is the process of hiding certain details and only show the essential features of the object.
- Hiding the implementation details from the user
- Filter out /hiding unnecessary information
- Advantages
  - Hiding implementation complexity from user
  - Object is easy to used by users

Abstract class

- A class that is declared with abstract keyword
- It cannot be instantiated.
- It needs to be extended (sub classed)
- Its abstract methods need to be implemented.

Syntax

```
abstract class A{}
```

Abstract method

- A method that is declared as abstract
- Does not have implementation
- Abstract methods are implemented in sub classes

Syntax

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

Interface

- An interface is not a class.
- An interface is a collection of abstract methods
- A class implements an interface, thereby inheriting the abstract methods of the interface
- Interface contains behaviors that a class implements.
- Interface is written in a file with a **.java** extension

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

An abstract class can have data member, abstract method, method body, constructor and even main() method

Interface

- Does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface can extend multiple interfaces.
- Why
  - It is used to achieve fully abstraction.
  - By interface, we can support the functionality of multiple inheritance.
  - It can be used to achieve loose coupling.

Interface

- Syntax

```
public interface NameOfInterface
{
//Any number of final, static fields //Any number of abstract
method declarations\
}
```

```
interface Animal
{
public void eat();
public void travel();
}
```

```
Class Dog  implements Animal
{
public void eat()
{System.out.println("eating"); }
public void travel()
{System.out.println("running"); }

}
```

Thank you

HNDIT3012 Object Oriented Programming

Exception Handling

Run time errors

- Occurs at program run time
- Condition that changes the normal flow of execution
  - Program run out of memory
  - File does not exist in the given path
  - Network connections are dropped
  - OS limitations

Program Errors

- Syntax errors
  - Caused by an incorrectly written code such as misspelled keyword or variable, and incomplete code
- Runtime errors
  - Occurs when a statement attempts an operation that is impossible to carry out
  - Ex: division by zero
- Logical errors
  - Occurs when code complies and runs without generating an error, but the result produced is incorrect

Exception

- Unexpected behavior of program
- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

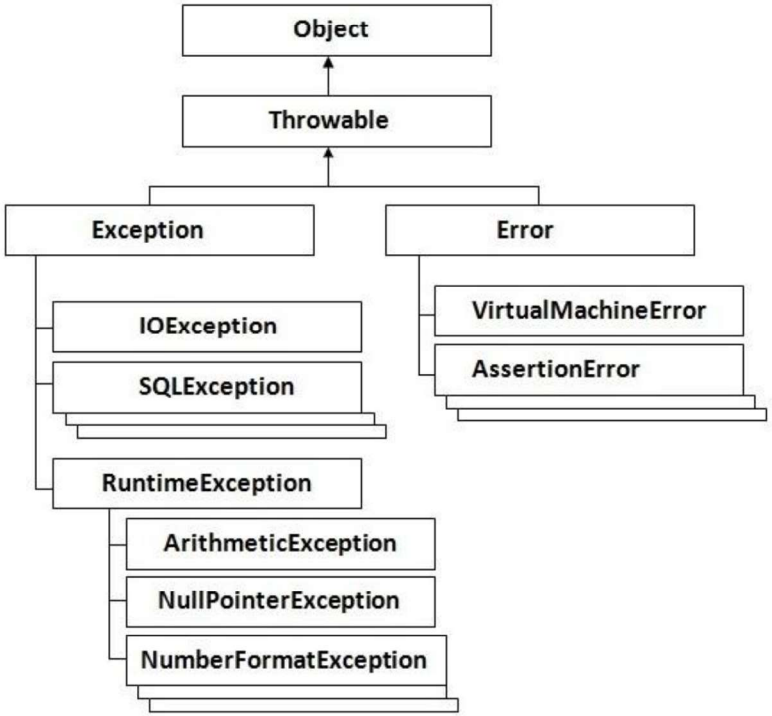
exception object

- When an error occurs within a method, the method creates an object and hands it off to the runtime system
- The object, called an *exception object*
- contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.

Types of Java Exceptions

- Checked exceptions
- Unchecked
  - Runtime exceptions
  - Errors

Hierarchy of Java Exception classes



Types of Exceptions

- Checked exceptions
  - Occurs at the compile time
  - Also called as compile time exceptions.
  - Cannot be ignored at the time of compilation by programmer.
  - All exceptions are checked exceptions, other than Error, RuntimeException, and their subclasses

- IOException
- SQLException
- ClassNotFoundException
- IllegalAccessException

Types of Exceptions

- Runtime exceptions
  - At the time of execution.
  - Never checked
  - These include programming bugs, such as logic errors or improper use of an API.

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- InvalidArgumentException
- NumberFormatException

Exception handling

- **Exception handling in java** is one of the powerful *mechanism to handle the runtime errors*

Types of Exceptions

- Errors
  - Not exceptions at all, but problems that arise beyond the control of the user or the programmer.
  - Errors are typically ignored in your code because you can rarely do anything about an error.
  - For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

OutOfMemoryError
VirtualMachineError
AssertionError

try catch block

```
try
    {
    //Protected code
    }
catch(ExceptionName e1)
    {
    //Catch block
    }
```

try catch block

- Example 1

```
public class Main
{
   public static void main(String[] args) {
      try{
      int a=4, b=0;
      int c=a/b;
      System.out.println(c);
      }
      catch(ArithmeticException E)
      {
      System.out.print("Error"+ E.toString());
      }
   }
}
```

try catch block

- Example 2

```
public class Main
{
   public static void main(String[] args) {
      try{
      int a=4;
      int b=0;
      int c=a/b;
      System.out.println(c);
      }
      catch(ArithmeticException E)
      {
      System.out.print("Error"+ E.toString());
      }
   }
}
```

Multiple catch Blocks

```
try
        { //Protected code
        }
catch(ExceptionType1 e1)
        { //Catch block
        }
catch(ExceptionType2 e2)
        { //Catch block
        }
catch(ExceptionType3 e3)
        { //Catch block
        }
```

finally Block

- A finally block of code always executes, whether or not an exception has occurred.

try catch with finally

- Example 3

```
public class ExcepTest
{   public static void main(String[] args)
    {
     try{
        int a=4;
        int b=0;
        int c=a/b;
        System.out.println(c);
         }
     catch(ArithmeticException E)
      {       System.out.print("Error"+ E.toString());        }
     finally
      {       System.out.print("End");       }
   }
}
```

Nesting try blocks

```
public class Demo
  {
  public static void main(String args[])
    {
    try
      {
      try
        {
        System.out.println("going to divide");
        int b =39/0;
        }
      catch(ArithmeticException e)
        {          System.out.println(e);          }
      System.out.println("other statement");
      }
    catch(Exception e){System.out.println("handeled");}
    System.out.println("normal flow..");
    }
  }
```

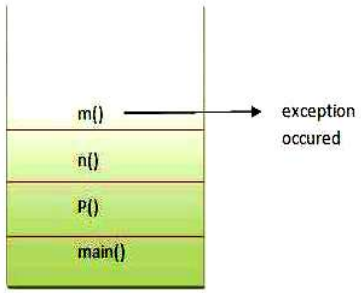
Throwable

- Superclass of all errors and exceptions
- All exceptions and errors extend from a common java.lang.Throwable parent class

Exception Propagation

```
class Demo
{
  void m(){   int data=50/0;  }
  void n(){    m();   }
  void p()
       {
     try{
          n();
        }
     catch(Exception e)
        {System.out.println("exception handled");}
     }
  public static void main(String args[])
       {
     Demo obj=new Demo();
     obj.p();
     System.out.println("normal flow...");
     }
}
```

An exception is first thrown from the top of the stack is not caught, it drops down to the previous method in the call stack.



throw

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

```
throw new ArithmeticException("not valid");
```

- Refer **Lab8**

```
class TestVar
{
   static void fun() throws IllegalAccessException
   {
      System.out.println("Inside fun(). ");
      throw new IllegalAccessException("demo");
   }
   public static void main(String args[])
   {
      try
      {          fun();      }
      catch(Exception e)
      {
         System.out.println("Caught in main"+e.toString());
      }
   }
}
```

throws

- Keyword in the signature of method to indicate that method might throw one of the listed type exceptions
- Use throws keyword to delegate the responsibility of exception handling to the caller (method or JVM), then caller method is responsible to handle that exception.
- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.

questions

HNDIT3012 Object Oriented Programming

Swing and Event Handling

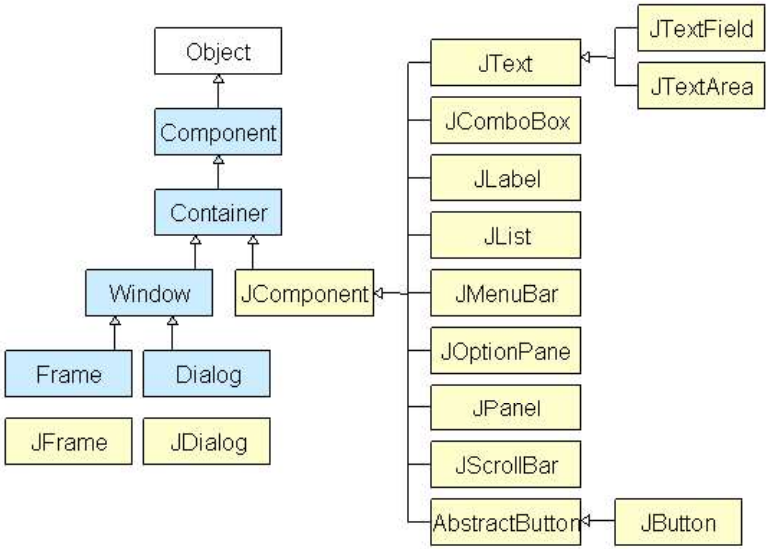
Swing features

- **Light Weight** -
  - Swing component are independent of native Operating System's API
  - Swing API controls are pure JAVA code instead of underlying operating system calls.
- **Rich controls** -
  - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, table controls
- **Highly Customizable** -
  - Swing controls can be customized easily as visual appearance is independent of internal representation.
- **Pluggable look-and-feel**- SWING
  - based GUI Application look and feel can be changed at run time

Swing

- Swing API is set of extensible GUI Components to create JAVA Front End/ GUI Applications

SWING - Controls



SWING - Controls

Class	Description
Component	A Container is the abstract base class for the non menu user-interface controls of SWING. Component represents an object with graphical representation
Container	A Container is a component that can contain other SWING components.
Jcomponent	A JComponent is a base class for all swing UI components. In order to use a swing component that inherits from JComponent, component must be in a containment hierarchy whose root is a top-level Swing container.

Control	Description
JLabel	A JLabel object is a component for placing text in a container.
JButton	This class creates a labeled button.
JColorChooser	A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.
JCheck Box	JCheckBox is a graphical component that can be in either an **on**(true) or **off** (false) state.
JRadioButton	A graphical component that can be in either an **on** (true) or **off** (false) state. in a group.
Jlist	A JList component presents the user with a scrolling list of text items.
JComboBox	A JComboBox component presents the user with a to show up menu of choices.
JTextField	A JTextField object is a text component that allows for the editing of a single line of text.
JPasswordField	A JPasswordField object is a text component specialized for password entry.
JTextArea	A JTextArea object is a text component that allows for the editing of a multiple lines of text.
ImageIcon	A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
JScrollbar	A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
JOptionPane	provides set of standard dialog boxes that prompt users for a value or informs
JFileChooser	A JFileChooser control represents a dialog window from which the user can select a file.
JProgressBar	As the task progresses, the progress bar displays the task's percentage of completion.
JSlider	Lets the user graphically select a value by sliding a knob within a bounded interval.
JSpinner	a single line input field that lets the user select a number or an object value from an ordered sequence.

SWING - Containers

Container	Description
Panel	JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.
Frame	A JFrame is a top-level window with a title and a border
Window	A JWindow object is a top-level window with no borders and no menubar.

Frame

```
import javax.swing.*;
class testSwing
{
public static void main(String[]args)
    {
    JFrame f=new JFrame(); //creating instance of JFrame
    f.setSize(400,500);//400 width and 500 height
    f.setLayout(null);//using no layout managers
    f.setVisible(true);//making the frame visible
    }
}
```

```
import javax.swing.*;
class testSwing
{
public static void main(String[]args)
        {
        JFrame w=new JFrame("First window");
        JButton b=new JButton("click");
        b.setBounds(130,100,100, 40);
        w.add(b);
        w.setSize(400,500);
        w.setLayout(null);
        w.setVisible(true);
        }
}
```

Layout Manager Classes

LayoutManager	Description
BorderLayout	Arranges the components to fit in the five regions: east, west, north, south and center.
CardLayout	The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
FlowLayout	The FlowLayout is the default layout. It layouts the components in a directional flow.
GridLayout	The GridLayout manages the components in form of a rectangular grid.
GridBagLayout	This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.
GroupLayout	The GroupLayout hierarchically groups components in order to position them in a Container.
SpringLayout	A SpringLayout positions the children of its associated container according to a set of constraints.

Layouts

- Layout means the arrangement of components within the container
- Layouting the controls is automatically done by the Layout Manager

JOptionPane

- Display a message (through the use of the showMessageDialog method)
- Ask for user's confirmation (using the showConfirmDialog method)
- Obtain the user's input (using the showInputDialog method)
- Do the combined three above (using the showOptionDialog method)

JOptionPane

```
import javax.swing.JOptionPane;
 public class Addition
{
  public static void main( String args[ ] )
  {
    String firstNumber =
    JOptionPane.showInputDialog( "Enter first integer" );
    String secondNumber =
    JOptionPane.showInputDialog( "Enter second integer" );
    int number1 = Integer.parseInt( firstNumber );
    int number2 = Integer.parseInt( secondNumber );
    int sum = number1 + number2; // add numbers
    JOptionPane.showMessageDialog( null, "The sum is " + sum,
      "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
  }
}
```

Java Event Model

- **Source**
  - Object which event occurs.
  - Source providing information of the occurred event
- **Listener**
  - It is also known as event handler.
  - Listener is object generating response to an event.

Event

- Change in the state of an object is known as event
- Event driven programming – develop application respond to user generated events
  - Button clicks
  - Mouse move
  - Keystrokes
- Events is represented in Java by a class that extends the AWTEvent class

java.awt.event

- Package defines classes and interfaces used for event handling in the AWT and Swing.

java.awt.event

- ActionListener
  - Interface for receiving action events.
  - Object created with is registered with a component, using the component‘ saddActionListener() method
  - When the action event occurs, that object's actionPerformed() method is invoked

```
import java.awt.*;
import java.awt.event.*;
public class tstEvnt implements ActionListener
        {
        private Frame F;
        private TextField T;
        private Button B;
        public static void main(String[] args)
                {
                tstEvnt T= new tstEvnt();
                T.setGUI();
                }
        public void setGUI()
                {
                 F = new Frame("Window Title");
                T= new TextField(20);
                B = new Button("Click me");
                F.setSize(350,100);
                F.setVisible(true);
                F.setLayout(new FlowLayout());
                F.add(T);
                F.add(B);
                B.addActionListener(this);//component registration
                }
         public void actionPerformed(ActionEvent e)
                {
                 T.setText("Button Clicked " );
                }
        }
```


```
import java.awt.*;
import java.awt.event.*;
public class tstEvnt implements ActionListener
        {
        private Frame F;
        private TextField T;
        private Button B;
        public static void main(String[] args)
                {
                tstEvnt T= new tstEvnt();
                T.setGUI();
                }
        public void setGUI()
                {
                 F = new Frame("Window Title");
                T= new TextField(20);
                B = new Button("Click me");
                F.setSize(350,100);
                F.setVisible(true);
                F.setLayout(new FlowLayout());
                F.add(T);
                F.add(B);
                B.addActionListener(this);//component registration
                }
         public void actionPerformed(ActionEvent e)
                {
                 T.setText("Button Clicked " );
                }
        }
```


Tahnk you

HNDIT3012 Object Oriented Programming

Input Output

Java Streams

- **System.out:** standard output stream

```
System.out.println("simple message");
```

- **System.in:** standard input stream

```
int i=System.in.read();
//returns ASCII code of 1st character
```

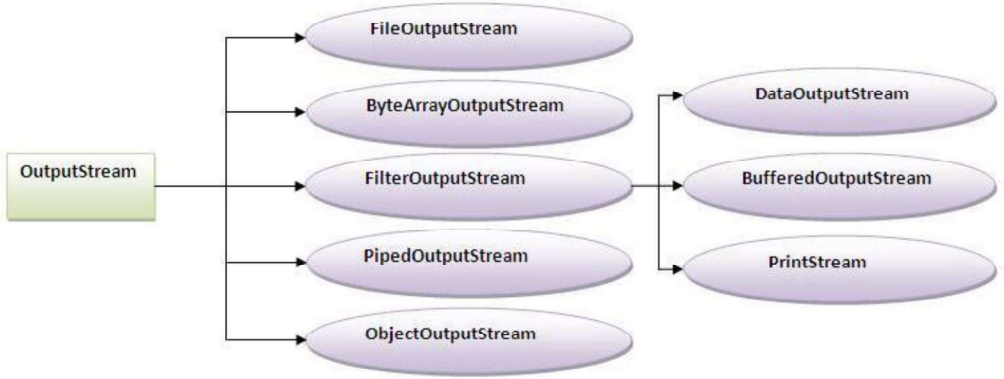
- **System.err:** standard error stream

```
System.err.println("error message");
```

Stream

- Java uses the concept of stream to make I/O operation fast.
- A stream is a sequence of data

OutputStream



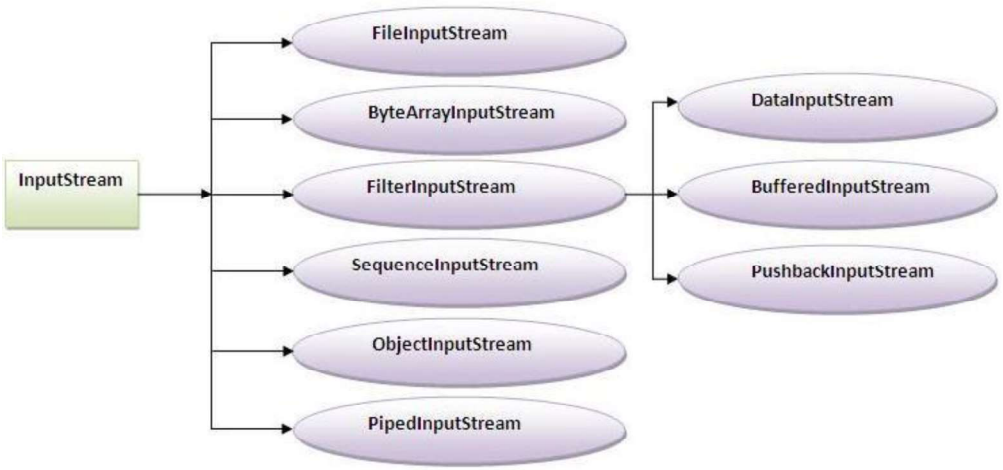
FileOutputStream class

```
import java.io.*;
class Test{
  public static void main(String args[])
{
   try
      {
       FileOutputStream fout=new FileOutputStream("abc.txt");
       String s="Java is my favourite ";
       byte b[]=s.getBytes();//converting string into byte array
       fout.write(b);
       fout.close();
       System.out.println("success...");
       }
    catch(Exception e){System.out.println(e);}
  }
}
```

FileInputStream class

```
import java.io.*;
class Demo
    {
    public static void main(String args[])
            {
            try         {
                        FileInputStream fin=new FileInputStream("abc.txt");
                        int i=0;
                        while((i=fin.read())!=-1)
                                    {
                                    System.out.println((char)i);
                                    }
                        fin.close();
                        }
            catch(Exception e)
                        {
                        System.out.println(e);
                        }
    } }
```

InputStream

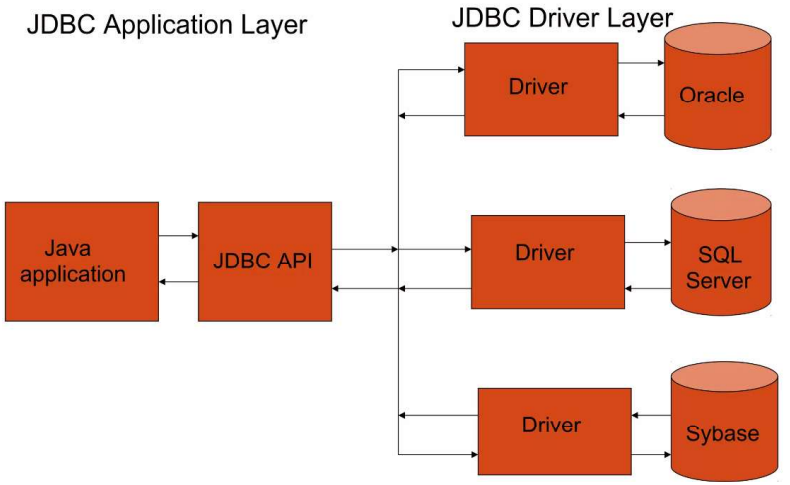


THANK YOU

HNDIT3012 Object Oriented Programming

JDBC

The following figure shows the JDBC architecture:



JDBC

- JDBC stands for Java Database Connectivity.
- JDBC is a Java API to connect and execute the query with the database.
- It is a part of JavaSE (Java Standard Edition).
- JDBC API uses JDBC drivers to connect with the database.
- We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database.

What is API

- API (Application programming interface) is a document that contains a description of all the features of a product or software.
- It represents classes and interfaces that software programs can follow to communicate with each other.
- An API can be created for applications, libraries, operating systems, etc.

JDBC API

- A list of popular *interfaces* of JDBC API are given below:
  - Driver interface
  - Connection interface
  - Statement interface
  - PreparedStatement interface
  - CallableStatement interface
  - ResultSet interface
  - ResultSetMetaData interface
  - DatabaseMetaData interface
  - RowSet interface
- A list of popular *classes* of JDBC API are given below:
  - DriverManager class
  - Blob class
  - Clob class
  - Types class

JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database.
- There are 4 types of JDBC drivers:
  - JDBC-ODBC bridge driver
  - Native-API driver (partially java driver)
  - Network Protocol driver (fully java driver)
  - Thin driver (fully java driver)

JDBC API

- We can use JDBC API to handle database using Java program and can perform the following activities:
- Connect to the database
- Execute queries and update statements to the database
- Retrieve the result received from the database

Java Database Connectivity

- There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

Java Database Connectivity with MySQL

- we need to know following informations for the mysql database:
  - **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
- **Connection URL:**
  - The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.
- **Username:**
  - The default username for the mysql database is **root**.
- **Password:**
  - It is the password given by the user at the time of installing the mysql database

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(  "jdbc:mysql://localhost:3306/st","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from student");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

first create a table in the mysql database

- create database stu;
- create table student(id **int**,name varchar(10), marks **int)**;

Tahnk you