Classes and Objects

TEKNOTURF

What is an Object?

- Any real world entity with well defined properties is called an object.
- Object will have a state, behaviour and a unique identity
- Objects can be tangible (physical entity) or intangible (cannot be touched or felt)

A wooden chair

A chemical process

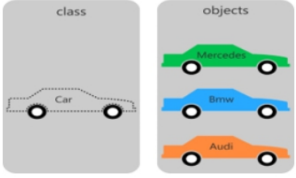
A transaction

Classes and Objects

TEKNOTURF

What is a class?

- A *class* is a template or *blue print* that describes the behaviors/states that an object of its type supports
- A class defines its
  - object's properties and
  - object's behavior through methods



Classes and Objects

Class is a template for a collection of objects that share a common set of attributes and behaviour.

In the perspective of OO programming

- Class is a description of the object
  - It describes the data that each object possesses
  - It also describes the various behaviors of the object

Object is an instance of a class

Classes and Objects

Class: Employee

Employee: Object

Employee ID : 101
Name : Rohith
Salary : 25000.00

```
class Employee
{
  //Attributes
   int empID;
   String name;
   double salary;

   //Methods
   void punchCard()
   {
      //do something
   }
   void  doProject()
   {
      //do something
   }
}

}
```

Classes and Objects

Car Object

Attributes :
model = “Mustang”
Color = “Yellow”
numOfPassengers = 0
amtOfGas = 16.5

Behaviour:
Add / Remove Passenger
Get Tank Filled
Report when out of Gas

```
class Car
{
  //Attributes
  String model;
   String color;
   int numOfPassengers;
   float amtOfGas;

   //Methods
   void addPassenger()
   {
     // do something
   }
```

```
void removePassenger()
   {
     // do something
   }
 void fillTank()
   {
     // do something
   }
 void report_OutOfGas()
   {
     // do something
   }
}
```

Class Declaration

TEKNOTURF

Declaring a class

```
<modifier>* class <class_name> {
        <attribute_declaration>*
        <constructor_declaration>*
        <method_declaration>*
}
```

Example:

```
public class Employee {
    private int empId;
    private String name;
    private  double salary;
    public void setEmpId(int  id) {
      empId = id;
    }
}
```

Attribute Declaration

TEKNOTURF

Attributes/Fields are used to declare the properties of a class. These attributes are called instance variables.

They can be intrinsic types (int, boolean…) or user-defined type.

Basic syntax of an attribute :

- <visibility>* <modifier>* <type> <name> [= <initial_value>]

Example: private double salary;

Method Declaration

TEKNOTURF

Methods describe the responsibility of the class. These methods are common for all the objects. Hence they are known as **instance methods.**

Method will have a name and a return type.

Methods can accept parameters.

Basic syntax of a method:

```
<visibility>* <modifier>* <return_type> <name> ( <argument>* ) {
        <statement>*
}
```

Example :

```
public void calculateArea(int side) {
        int area = side * side;
    System.out.println(area);
    }
```

```
public boolean isAgeValid( ) {
    int age = 15;
    if(age>0 && age <=100)
            return true;
    else
            return false;
}
```

Create Object

TEKNOTURF

Creation of Object

- Objects are created for a class using the keyword new.
- In Java, all Objects are created at run time in the heap area.

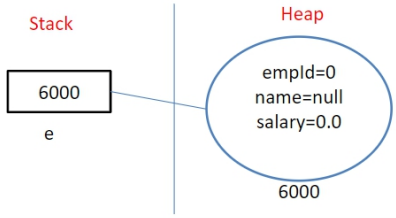
Syntax

- class_name reference_variable = new class_name(<parameter list>)

Example

- Employee e=new Employee();

Here variable 'e' is the name of the object. 'e' is the reference variable that holds the address of the Object created in Heap.



Note : Instance variables will be assigned with default values.

Create Object and Access Members

TEKNOTURF

- The attributes and methods of a class can be accessed using the dot operator as <object>.<member>

```
//object creation
    Employee empObj=new Employee();

    //assign meaningful state
    empObj.empId(101);
    empObj.name("Rohit");
    empObj.salary(25000);

    //retrieve values
    System.out.println("Emp ID  "+empObj.empId);
    System.out.println("Name  "+empObj.name);
    System.out.println("Salary  "+empObj.salary);
```

class Example

TEKNOTURF

```
public class Employee {

  //Attributes
       int empId;
       String name;
       double salary;

  //Methods

   public void calculateGrossPay(int bonus)
  {
     return salary + bonus
  }
}
```

```
public class EmployeeMain
{
   public static void main(String a[])
   {
      //object creation
        Employee empObj=new Employee();

      //assign meaningful state
         empObj.empId=101;
         empObj.name="Rohit";
         empObj.salary=25000;
     //retrieve values
          System.out.println("Emp ID  "+empObj.empId);
          System.out.println("Name  "+empObj.name);
          System.out.println("Salary  "+empObj.salary);
    }
}
```

Accessors and Mutators

Encapsulation is an important OO Concept.

How is encapsulation implemented in Java?

- By wrapping up of data and methods, that operate on that data into a single unit, class
- Data inside a class needs to be secure. Declare the data in a class as private.
- No outside class can access private data members of other class.
- Outside classes should be able to read and update the private data members only by using the public methods.
- These methods are called Accessors or Getters and Mutators or Setters.
- Data Hiding is not possible without Encapsulation.

Accessors and Mutators

Accessor

- Method used to return the value of a private field
- Does not change the state of the Object
- Example :

```
public int getSalary() {
    return salary;
}
```

Mutator

- Method used to set a value of a private field
- Changes the state of the Object
- Example :

```
public void setSalary(double  sal) {
    salary = sal;
}
```

Create Object and Access Members

```
//object creation
Employee empObj=new Employee();

//assign meaningful state using Setters
empObj.setEmpId(101);
empObj.setName("Rohit");
empObj.setSalary(25000);

//retrieve values using Getters
System.out.println("Emp ID  "+empObj.getEmpId());
System.out.println("Name  "+empObj.getName());
System.out.println("Salary  "+empObj.getSalary());
```

class Example

```
public class Employee {

//Attributes
private int empId;
private String name;
private double salary;

//Accessors or Getters
public int getEmpId() {
return empId;
}

public String getName() {
return name;
}

public double getSalary() {
return salary;
}

//Mutators or Setters
public void setEmpId(int id) {
    empId = id;
}
public void setName(String name1) {
    name = name1;
}
public void setSalary(double sal) {
     salary = sal;
}
}
```

Create Object and Access Members

```
public class EmployeeMain
{
    public static void main(String a[])
    {
    //Create Object
        Employee empObj=new Employee();

        //assign meaningful state
        empObj.setEmpId(101);
        empObj.setName("Rohit");
        empObj.setSalary(25000);

        //retrieve values and print
        System.out.println("Emp ID  "+empObj.getEmpId());
        System.out.println("Name  "+empObj.getName());
        System.out.println("Salary  "+empObj.getSalary());
    }
}
```

EmployeeMain class depicts how to create an object for the Employee class. It also shows how to access the members of the class using that object.

Access Specifier

Information Hiding is implemented using Access Specifiers known as visibility modifiers.

They regulate access to classes, fields and methods.

They specify if a field or method in a class can be invoked from another class or sub-class.

In simple words, they can be used to restrict access.

Types Of Access Specifiers in Java

- public
- private
- protected
- default(no specifier)

Access Specifier

public

- Classes, methods, and fields declared as public can be accessed from any class in an application

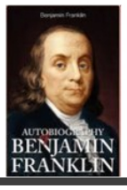
default

- If classes, variables, and methods have a default specifier we can access them only within the same package, but not from outside this package

private

- Private methods and fields can only be accessed within the same class to which the methods and fields belong

private



public

default

- When no access specifier is specified for an element, its accessibility level is default.
- There is no keyword default.

Access Specifier

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

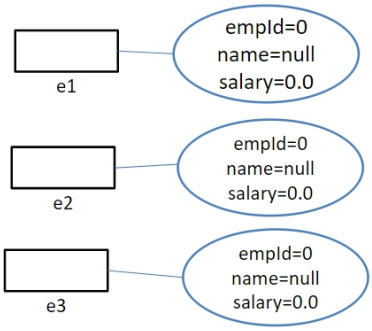
Let us discuss about protected, subclass and packages later

Constructor

TEKNOTURF

- Consider the following statements

```
Employee e1=new Employee();
Employee e2=new Employee();
Employee e3=new Employee();
```



Note :

- Objects are created.
- They are not in proper state.
- Instance variables are allotted the default value.

Constructor

TEKNOTURF

Constructor is a special method invoked implicitly when an object is created

It initializes the instance variables with proper value

Rules for writing a constructor

- Constructor name must be the same as the name of the class
- Constructors must not have a return type (not even void)

Types of Constructors

- Default Constructor
- Parameterized Constructor

Default Constructor

TEKNOTURF

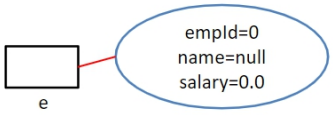
Default Constructor (no argument constructor)

A constructor with no arguments is known as a default constructor

Example

```
public class Employee  {
   private int empId;
   private String name;
   private float salary;

   public Employee()  {
       System.out.println("Default Constructor");
   }
   public static void main(String a[])
   {
         Employee e=new Employee();
   }
}
```

Default Constructor



Output : Default Constructor

Parameterized Constructor

TEKNOTURF

Parametrized Constructor

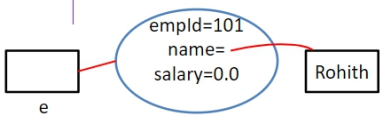
A constructor with an argument list is known as a parameterized constructor.

Example

```
public class Employee  {
   private int empId;
   private String name;
   private float salary;

   //Constructor
   public Employee(int id,String ename)  {
         System.out.println("In Parametrized Constructor");
        empId = id;
         name = ename;
   }
```

```
   public static void main(String a[])
   {
         Employee e=new Employee(101,"Rohith");
   }
}
```

Output : In Parametrized Constructor



Constructors

TEKNOTURF

- If a constructor is not written explicitly for a class, what happens when an object is created?
  - Java compiler builds a default constructor for that class and initiates the fields with the default value.
- When a constructor is explicitly written by the developer, the compiler will not provide the default constructor.
- If a constructor with parameters is declared and we want to create an object using no argument constructor, we must explicitly write a no argument constructor.

“this" keyword

```
class  Employee{
   int empId;
   String name;

   public Employee(int empId,String name)
   {
      empId = empId;
      name = name;
   }
   void display()
   {
       System.out.println(“ID  : “+empId+“ Name : "+name);
    }

    public static void main(String args[])
    {
        Employee  e1= new Employee(101,“Tom");
        e1.display();
    }
}
```

Output of this code :

ID : 0 Name : null

The reason for this output is: parameter and instance variables have the same name

The solution for this problem is to use the keyword “this”

“this" keyword

“this” is a reference variable that refers to the current object

Example : Usage of “this” in constructor for initializing the attributes

```
class  Employee{
   int empId;
   String name;

   public Employee(int empId,String name)
   {
      this.empId = empId;
      this.name = name;
   }
   void display()
   {
       System.out.println(“ID  : “+empId+“ Name : "+name);
    }
```

```
    public static void main(String args[])
    {
        Employee  e1= new Employee(101,“Tom");
        e1.display();
    }
}
```

Output of this code :

ID : 101 Name : Tom

“this” keyword

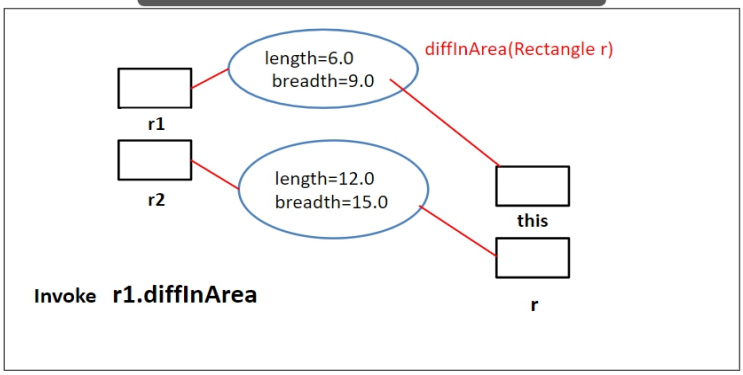
Example : Usage of “this” in method

```
public class Rectangle {
   private float length;
   private float breadth;

   //Constructor
   public Rectangle(float length, float breadth)
   {
       this.length = length;
       this.breadth = breadth;
   }
   //Function to calculate Area
   public float calcArea()
   {
       return length*breadth;
   }
```

```
//difference between the area of 2 rectangle
public float diffInArea(Rectangle r)
{
       float area_difference =
              Math.abs(this.calcArea() -
                     r.calcArea());
       return area_difference;
}
public static void main(String a[])
{
   Rectangle r1=new Rectangle(6,9);
   Rectangle r2=new Rectangle(12,15);
   float diffInArea=r1.diffInArea(r2);
   System.out.println("Difference in Area : “
                     +diffInArea);
}
}
```

"this" keyword

Example : Usage of "this" in method



Constructor Overloading

A class can have any number of constructors but they should differ in the parameter list.

This technique is called constructor overloading.

The parameter list should vary in any of the following:

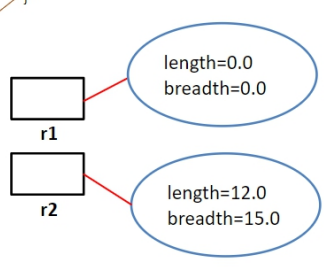
- Number of parameters
- Type of parameter
- Order of parameter

The compiler identifies which constructor is to be invoked based on the number of parameters in the list and their type.

Constructor Overloading

```
public class Rectangle {
private float length;
private float breadth;

//Constructor Overloading

public Rectangle()
{
    length=0;
    breadth=0;
}
public Rectangle(float length, float breadth)
{
  this.length = length;
  this.breadth = breadth;
}
```

```
public static void main(String a[])
{
    Rectangle r1=new Rectangle();
    Rectangle r2=new Rectangle(12,15);
}
}
```



Invoking a Constructor

- Constructor is invoked implicitly when an object is created.
- It cannot be invoked explicitly.
- Within a class, one constructor can call another constructor using "this" keyword.
- Example:

```
public class Employee {

private int empId;
private String name;
private double salary;
private String panNo;

public Employee(int empId, String name, double salary)
{
    this.empId = empId;
    this.name = name;
     this.salary = salary;
}
```

```
public Employee(int empId, String name,
          double  salary, String panNo)
{
   this(empId,name,salary);
   this.panNo = panNo;
}
}
```

Note : this() should be the first statement in the constructor

Method Overloading

- In a class, if there is more than one method with the same name, but differing in parameters, those methods are called *overloaded methods*. This concept is called *Method Overloading*.

```
sing_a_Song()
{
    smooth and clear
}
```

```
sing_a_Song(illness)
{
  lots of disturbance
}
```

Method Overloading

Rules for Method Overloading

- Methods should have same name but different arguments list
- Argument list should differ in
  - *Number of parameters*
  - *Type of parameter*
  - *Order of parameter*
- Return Type of the method is not to be considered

Method Overloading

Example

Number of Arguments are the same, but vary in the type

```
public int add(int num1,int num2)
{
   return num1+num2;
}
public int add(int num1,int num2,int num3)
{
   return num1+num2+num3;
}
public String add(String s1,String s2)
{
   return s1+s2;
}
```

Vary in Number of Arguments

Method Overloading

Valid overloaded methods

1. public void calculateArea(int a,int b) { }
   public void calculateArea(float a,float b) { }
   public void calculateArea(int a) { }
   public void calculateArea(int a,float b) { }

2. public void calculateArea(int a,int b) { }
   public void calculateArea(float a,float b) { }

3. public void calculateArea(int a,float b) { }
   public void calculateArea(float a,int b) { }

In-valid overloaded methods

1. public void calculateArea(int a) { }
   public int calculateArea(int a) { }

Not valid because the number of arguments is same and the change is only in the return type.

2. public void calculateArea(int a) { }
   public void calculateArea(int x) { }

Not valid because the signature is same

Object of a class as Attribute

```
public class Customer
{
    private int customerId;
    private String name;
    private String mobileNumber;

    //Constructor, Getters and Setters

}
```

```
public  class Account
{
    private int accoutNumber;
    private double balance;
    private Customer customerObj;
    public Account(int accoutNumber, double balance, Customer customerObj) {
        this.accoutNumber = accoutNumber;
        this.balance = balance;
        this.customerObj = customerObj;
    }
    public Customer getCustomerObj() {
        return customerObj;
    }
    public void setCustomerObj(Customer customerObj) {
        this.customerObj = customerObj;
    }
      //Other Getters and Setters
    public void display(){
        System.out.println("Account Details");
        System.out.println("Account Number "+accoutNumber);
        System.out.printf("Balance : %.2f",balance);
        System.out.println("Customer Name "+customerObj.getName());
        System.out.println("Mobile number "+customerObj.getMobileNumber());
    }
}
```

Object of a class as Attribute

```
public class Main
{
    public static void main(String args[])
    {
        Customer customerObj = new Customer(1,"Pinky","9998887771");
        Account accountObj = new Account(1001,20000,customerObj);
        accountObj.display();
        accountObj.getCustomerObj().setMobileNumber("9128345671");
    }
}
```

Association Relationship

Association are relationship between classes and are represented using solid line

A Faculty is assigned to a Student as mentor.

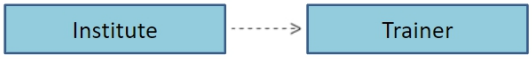

```
public class Faculty {

   //Attributes, methods

}
```

```
public class Student {
    private int studentId;
    private String studentName;
    private Faculty facultyObj;
}
```

Dependency Relationship

- If two classes are related such that change in structure or behavior of a class affects the other related class then the relationship between those classes are dependency
- If a class uses an object of another class either as a parameter to a method or as a local variable in a method, without having that object as an attribute, then there exists a “Uses” relationship (dependency) between them.

An University is dependent on Professor



Dependency Relationship

TEKNOTURF

```
public class  Trainer {

   //Attributes, methods

}
```

```
public class  Institute  {
   private String name;

   public void allocateTrainer(){
        Trainer t = new Trainer();
   }

   public void  perfAppraisal( Trainer t ) {

   }
}
```

TEKNOTURF

STATIC IN JAVA

Static

TEKNOTURF

Certain things are available for each individual apartment, whereas certain things are common for all apartments.

Likewise, in a class, certain attributes will be applicable for each instance of that class and some attributes will be common to all instances in a class. These common attributes are termed 'static'.

static

TEKNOTURF

When an object is created for a class, it will have its own copy of variables, called instance variables.

Certain fields are common to all objects in a class. We call these variables as class variables.

These variables are also called static variables because they are declared with the keyword static.

Static keyword is used as a modifier on variables, methods, block and nested classes.

A static modifier associates an attribute or method to the class as a whole, rather than as any particular instance of that class.

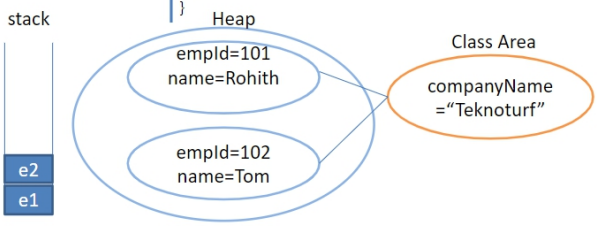
To access an instance member, an object is required. Whereas, static members can be accessed directly using the class name itself as className.staticmember

"this" cannot be referenced from a static context.

static variable

```
class Employee
{
  private int empId;
  private String name;
  private static String companyName="Teknoturf"

  //Constructor
  public Employee(int empId,String name)  {
       this.empId=empId;
       this.name=name;
}
```

```
public static void main(String a[])
 {
     Employee e1=new Employee(101,"Rohith");
     Employee e2=new Employee(102,"Tom")
     System.out.println("Company Name is "+
                        Employee.companyName);
     //This will also work
     //System.out.println("Company Name is "+
                          e1.companyName);
 }
}
```

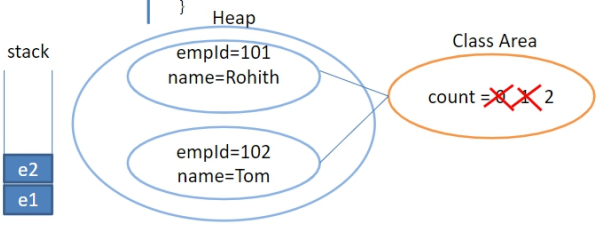


static variable

Usage of static for count of objects created

```
class Employee
{
  private int empId;
  private String name;
  private static int count;

  //Constructor
  public Employee(int empId,String name)  {
       this.empId=empId;
       this.name=name;
       count++;
}
```

```
public static void main(String a[])
 {
     Employee e1=new Employee(101,"Rohith");
     Employee e2=new Employee(102,"Tom")
     System.out.println("Total Objects Created :  "+
                        Employee.count);      //2
     //This will also work
     //System.out.println("Total Objects created : "+
          e1.count + " "+e2.count);   //2  2
 }
}
```



static method

```
public class Employee {

  private int empId;
  private String name;
  private static int count;

  //Constructor
  public Employee(int empId,String name)
  {
       this.empId=empId;
       this.name=name;
       count++;
  }
  public void setEmpId(int empId)
  {
        this.empId = empId;
  }
```

```
public void setName(String name)
{
    this.name = name;
}

public int getEmpId()
{
   return empId;
}
public String getName()
{
   return name;
}
public static int displayCount()
{
   return count;
}
}
```

static method can access only static members

static method

```
public class EmployeeMain
{
    public static void main(String a[])
    {
        Employee e1=new Employee(101,"Rohith");
        Employee e2=new Employee(102,"Tom");
        System.out.println("Count of objects : "+Employee.displayCount()); //Count of objects : 2
    }
}
```

Static method accessed directly using class name

static

A static method can access only other static members of a class.

A non-static method can access all members i.e., both static and non static members of a class.

The main() method in java is public static.

public access specifier denotes anyone can access/invoke it, such as JVM.

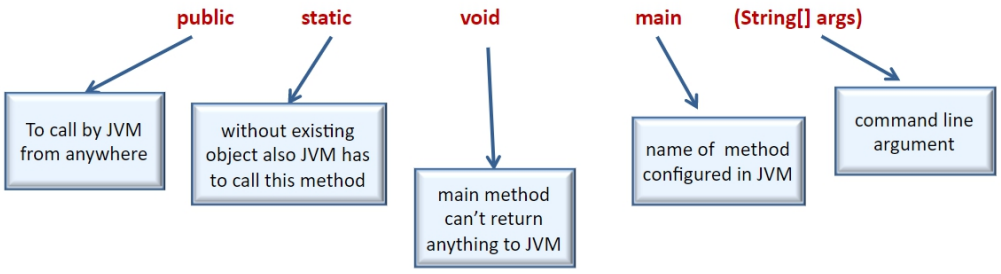
static keyword allows main() to be called before an object of the class has been created.

main() is called by the JVM before any objects are made. As it is static, it is directly invoked from class.

static

The main method is declared as static

public static void main(String a[])



static block

TEKNOTURF

- Static variables can be initialized using the static block
- Static block gets executed when the class gets loaded in the memory
- There can exist multiple static blocks. They get executed in the same order in which they are written in the code

```
class StaticBlockDemo{
    static String language;
    static {
            language="Java";
    }
    public static void main(String a[]) {
            System.out.println("Language "+language);
    }
}
```

Initialization block

TEKNOTURF

- Instance Initialization Blocks are used to initialize instance variables
- Initialization blocks are executed whenever an instance is created for that class and before constructors are invoked.

```
public class Employee {
        {
                System.out.println("In Initialization block");
        }
        public Employee() {
                System.out.println("In constructor");
        }
        public static void main(String arg[]) {
                Employee e=new Employee();
        }
}
```

Output

```
In Initialization block
In constructor
```

Wrapper Class

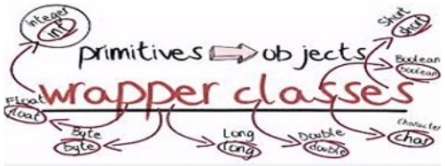
TEKNOTURF

Java has eight primitive data types - byte, short, int, long, char, float, double, boolean

Object representation of primitive data types are called wrapper classes.

The java.lang package contains wrapper classes that corresponds to each primitive type

Certain concepts in java, like Collection work on objects.
Primitives can be used in those concepts using wrapper classes.



Wrapper Class

TEKNOTURF

Wrapper class wraps around a data type and gives it an object appearance

Use this object wherever primitive is required as object

Wrapper class has methods to unwrap the object and give the data

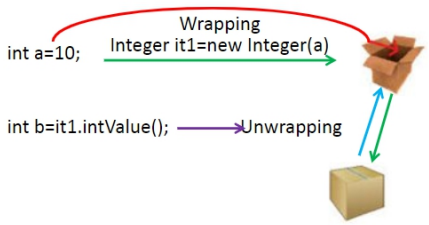
Primitive	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Instantiating Wrapper Class

TEKNOTURF

Instantiating wrapper class

- All Wrapper classes can be instantiated using their primitive type as the argument
  - Integer iobj=new Integer(10);
  - Double num=new Double(8.2);
  - Character cobj=new Character('e');
  - Boolean isdone=new Boolean(false);



Wrapping - int datatype a is converted to object it1

Use it1 when a is required as object

Unwrapping - Data is unwrapped using the method intValue() in Integer class

Type casting

TEKNOTURF

All wrapper classes have many methods that help convert from the associated type to another type

```
Integer num = new Integer(4);
float flt = num.floatValue(); //stores 4.0 in flt

Double dbl = new Double(8.2)
int val = dbl.intValue(); //stores 8 in val
```

Each numeric class contains a static method to convert a representation in a String to the associated primitive

```
int num1 = Integer.parseInt("3");
double num2 = Double.parseDouble("4.7");
```

Wrapper Class – Autoboxing & Unboxing

TEKNOTURF

Autoboxing and Unboxing features are introduced from Java 5

Autoboxing

- Converting a primitive value into an object of the corresponding wrapper class

Example : Integer i=10;

UnBoxing

- Converting an object of a wrapper type to its corresponding primitive value

Example :

```
Integer i=10;   // boxing
int y = i;      // unboxing
```

Wrapper Class – autoboxing & unboxing

Prior to java 5

```
Integer y = new Integer(567);      // wrap a data type
int x = y.intValue();               // unwrap it
x++;                                // use it
y = new Integer(x);                 // re-wrap it
```

From java 5 this is done automatically by the compiler itself by the concept of AutoBoxing and Unboxing

```
Integer y = new Integer(567);       // make it
y++;                                // unwrap it, increment it, rewrap it
System.out.println("y = " + y);     // print it
```

Scanner Class

System.out is used to display the output text on the console on executing a java program

System.in is used to get the input from the user, but not directly

Prior to Java 1.5, getting input from a resource like keyboard or file was a complex task

Java 1.5 introduced the class Scanner reducing the complexity

Scanner class is defined in java.util package

Scanner

import java.util package to get input using Scanner

Scanner class parse the input into tokens based on the delimiter

Default delimiter is whitespace

The delimiter is recognized by Character.isWhitespace(char)

Delimiters other than whitespace can also be used

The resulting tokens are then converted into values of different types using various next methods

Scanner

To get input from the keyboard

- Create a Scanner object

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

- Methods to read tokens from a Scanner class
  - nextByte()
  - nextInt()
  - nextShort()
  - nextLong()
  - nextFloat()
  - nextDouble()
  - nextBoolean()
  - next()
  - nextLine()

Scanner

To read an input of type byte

```
byte b = sc.nextByte();
```

To read an input of type int

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

To read an input of type String without space

```
String city = sc.next();
```

To read an input of type String with space in between

```
String sentence = sc.nextLine();  // reads till the end of line
```

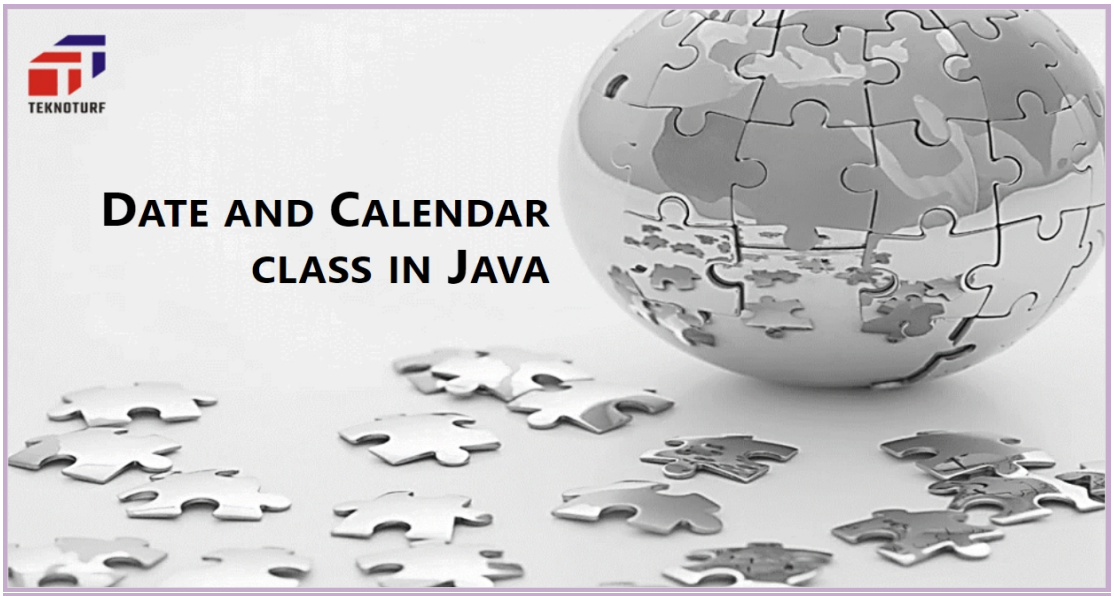
Scanner

- Sample Program

```
import java.util.Scanner;

public class ScannerDemo  {
    public static void main(String a[])  {
        Scanner sc=new Scanner(System.in);
        byte b = sc.nextByte();
        short s = sc.nextShort();
        int x = sc.nextInt();
        long l = sc.nextLong();
        float f = sc.nextFloat();
        double d = sc.nextDouble();
        boolean status=sc.nextBoolean();
        String city = sc.next();
        String name = sc.nextLine();
        char gender = sc.next().charAt(0); //String method charAt(0) is used
    }
}
```

TEKNOTURF

Date and Calendar class in Java



Date class

TEKNOTURF

Date class represents date and time in java

Available in java.util package and java.sql package.

This session focuses on Date class in java.util package

Date class encapsulates the **current date and time**

Has two constructors

Date() - Creates a Date object with current date and time

Date(long milliseconds) - Creates a date object for the given milliseconds from

January 1, 1970, 00:00:00 GMT.

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

public class Test
{
    public static void main(String args[])
    {
        Date dateObj1 = new Date();
        //To get the milliseconds for the current time
        long millisec = System.currentTimeMillis();
        Date dateObj2 = new Date(millisec);
        System.out.println(dateObj1);
        System.out.println(dateObj2);
    }
}
```

Output

```
Fri Sep 13 15:05:11 IST 2019
Fri Sep 13 15:05:11 IST 2019
```

Methods in Date class

TEKNOTURF

Method	Description
boolean after(Date when)	Tests if this date is after the specified date
boolean before(Date when)	Tests if this date is before the specified date
Int compareTo(Date when)	compares this date with the given date. Returns 0 if equal, <0 if the argument date is after this date, >0 otherwise.
long getTime()	returns the time represented by this date object.
void setTime(long milliseconds)	Changes the date and time of the date object to the given time.
String toString()	Converts the Date object into a String.

```
import java.util.Date;

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

        Date dateObj1 = new Date(2019,12,31);
        Date dateObj2 = new Date(2019,10,25);

        int status = dateObj1.compareTo(dateObj2);

        if(status==0){
            System.out.println("Both dates are equal");
        }
        else if(status>0){
            System.out.println("dateObj1 is after dateObj2");
        }
        else{
            System.out.println("dateObj1 is before dateObj2");
        }
    }
}
```

Output

```
dateObj1 is after dateObj2
```

METHODS IN DATE CLASS

Example

```
import java.util.Date;

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

        Date dateObj1 = new Date(2019,12,31);
        Date dateObj2 = new Date(2019,10,25);

        int status = dateObj1.compareTo(dateObj2);

        if(dateObj1.after(dateObj2)){
            System.out.println("dateObj1 is after dateObj2");
        }
        else if(dateObj1.before(dateObj2)){
            System.out.println("dateObj1 is before dateObj2");
        }
        else
            System.out.println("Both dates are equal");

    }
}
```

Here deprecated constructor is used. Avoid using this. It is replaced with Calendar API

Output

dateObj1 is after dateObj2

DATE FORMATTING

WHAT IS DATE FORMATTING ?

Date in Java is given as : Fri Sep 13 15:05:11 IST 2019

Being users, one may provide the date as 13/09/2019, while another may provide it as 09/13/2019 etc.

Converting unformatted date to a format as required by the end user is termed as Date formatting.

JAVA PROVIDES TWO CLASSES FOR FORMATTING A DATE - DATEFORMAT AND SIMPLEDATEFORMAT

Date Format class is an abstract class and is the parent of SimpleDateFormat class

These classes are available in java.text package

Converting a Date to String is called formatting and converting a String to Date is called parsing

Fri Sep 13 15:05:11 IST 2019 → 13/09/2018 - Formatting

13/09/2018 → Fri Sep 13 15:05:11 IST 2019 - Parsing

FORMAT DATE USING DATEFORMAT

STEPS TO FORMAT DATE USING DATEFORMAT

- Create a formatter using the method getDateInstance. There are many methods to get the DateFormatInstance. One of those methods is getDateInstance()

  DateFormat df = DateFormat.getDateInstance(Style,Locale);

  Style - style with which Date needs to be formatted. It can be SHORT, MEDIUM, LONG OR FULL. If style is not provided, it takes DEFAULT.

  Locale is the locale with which date needs to be formatted.

- Invoke the format method using the above DateFormat instance

  String dateFormatted = df.format(Date dateObj);

  Returns string containing the formatted date.

FORMAT DATE USING DATEFORMAT

Various Styles available (output with respect to Locale English)

Style	Formatted output
DEFAULT	Sep 13, 2019
SHORT	9/13/19
MEDIUM	Sep 13, 2019
LONG	September 13, 2019
FULL	Friday, September 13, 2019

FORMAT DATE USING DATEFORMAT

Example

```
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class Test
{
    public static void main(String args[])
    {
        Date dateObj1 = new Date();

        DateFormat df = DateFormat.getDateInstance();
        String str1 = df.format(dateObj1);

        df = DateFormat.getDateInstance(DateFormat.DEFAULT,Locale.UK);
        String str2=df.format(dateObj1);

        df = DateFormat.getDateInstance(DateFormat.SHORT,Locale.UK);
        String str3=df.format(dateObj1);

        df = DateFormat.getDateInstance(DateFormat.MEDIUM,Locale.UK);
        String str4=df.format(dateObj1);

        df = DateFormat.getDateInstance(DateFormat.LONG,Locale.UK);
        String str5=df.format(dateObj1);

        df = DateFormat.getDateInstance(DateFormat.FULL,Locale.UK);
        String str6=df.format(dateObj1);

        System.out.println("Date with getDateInstance              "+str1);
        System.out.println("Date with getDateInstance(Default,UK)  "+str2);
        System.out.println("Date with getDateInstance(Short,UK)    "+str3);
        System.out.println("Date with getDateInstance(medium,UK)   "+str4);
        System.out.println("Date with getDateInstance(long,UK)     "+str5);
        System.out.println("Date with getDateInstance(full,UK)     "+str6);
    }
}
```

Output

```
Date with getDateInstance              13 Sep, 2019
Date with getDateInstance(default,UK)  13-Sep-2019
Date with getDateInstance(short,UK)    13/09/19
Date with getDateInstance(medium, UK)  13-Sep-2019
Date with getDateInstance(long, UK)    13 September 2019
Date with getDateInstance(full, UK)    Friday, 13 September
2019
```

FORMAT DATE USING SIMPLEDATEFORMAT – DATE TO STRING

- SimpleDateFormat class has methods to format and parse date. Format of the date can be some user defined format.

  Example of formats :

  dd/MM/yyyy
  dd-M-yyyy hh:mm:ss

- Format date using SimpleDateFormat

  Create an instance of SimpleDateFormat, by passing the required pattern.

  SimpleDateFornat sdf = new SimpleDateFormat(String pattern)

  pattern refers the required format like “dd-MM-yyyy”.

- Format the date using the above instancec of SimpleDateFormat

  String requiredDate = sdf.format(Date dateObj)

  Example

  String requiredDate = sdf.format(new Date());

  This statement formats the current date to the format specified in sdf.

PATTERNS IN SIMPLEDATEFORMAT

Few patterns used with SimpleDateFormat

Pattern	Meaning	Example
G	era designator	AD
y	year	2019
M	month in year	September and 09
d	day in month	13
h	hour in am/pm (1-12)	2
D	day in year	
E	day in week	
F	day of week in month	
a	am/pm marker	PM

FORMAT DATE USING SIMPLEDATEFORMAT - DATE TO STRING

Example

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class Test
{
    public static void main(String args[])
    {
        Date dateObj = new Date();

        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
        String formattedDate1 = sdf.format(dateObj);

        sdf = new SimpleDateFormat("dd-M-yyyy");
        String formattedDate2 = sdf.format(dateObj);

        sdf = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss a");
        String formattedDate3 = sdf.format(dateObj);

        sdf = new SimpleDateFormat("G dd-MM-yyyy HH:mm:ss");
        String formattedDate4= sdf.format(dateObj);

        System.out.println("dd-M-yyyy               "+formattedDate1);
        System.out.println("dd-M-yyyy               "+formattedDate2);
        System.out.println("dd-MM-yyyy hh:mm:ss a   "+formattedDate3);
        System.out.println("G dd-MM-yyyy HH:mm:ss   "+formattedDate4);
    }
}
```

Output

```
dd-M-yyyy                13-09-2019
dd-M-yyyy                13-9-2019
dd-MM-yyyy hh:mm:ss a    13-09-2019 11:24:43 AM
G dd-MM-yyyy HH:mm:ss    AD 13-09-2019 11:24:43
```

FORMAT DATE USING SIMPLEDATEFORMAT – STRING TO DATE

Parse String to date using the parse method.

In this case, the argument passed in SimpleDateFormat specifies the date format of the String containing the date.

```
Date dateObj = sdf.parse(String date);
```

setLenient(boolean) method in DateFormat class specifies whether the interpretation of the date and time of the DateFormat object is to be lenient or not.

Set lenient as false using **sdf. setLenient(false);** so that when invalid date is converted parse method throws ParseException.

Example

```
String str = "25/09/2019";
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy"); //This pattern represents that of str
Date dateObj = sdf.parse(str);
```

This case ParseException needs to be handled

FORMAT DATE USING SIMPLEDATEFORMAT - STRING TO DATE

Example

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Test {
    public static void main(String args[]) throws ParseException {
        String str = "22-09-2019";

        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
        sdf.setLenient(false);
        Date dateObj1=sdf.parse(str);

        str="22/12/18";
        sdf = new SimpleDateFormat("dd/MM/yy");
        sdf.setLenient(false);
        Date dateObj2=sdf.parse(str);

        str="15/09/19 15:25:16";
        sdf = new SimpleDateFormat("dd/MM/yy HH:mm:ss");
        sdf.setLenient(false);
        Date dateObj3=sdf.parse(str);

        System.out.println("dd/MM/yyyy          "+dateObj1);
        System.out.println("dd/MM/yy            "+dateObj2);
        System.out.println("dd/MM/yy HH:mm:ss   "+dateObj3);
    }

}
```

Output

```
dd-MM-yyyy          Sun Sep 22 00:00:00 IST 2019
dd/MM/yy            Sat Dec 22 00:00:00 IST 2018
dd/MM/yy hh:mm:ss   Sun Sep 15 15:25:16 IST 2019
```

CALENDAR CLASS

Calendar is an abstract class for processing Date, like adding few days to a date or adding months to a date etc.

Few fields in the Calendar class are DATE, MONTH, YEAR, HOUR etc.

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class Test
{
    public static void main(String args[]) throws ParseException
    {
        Calendar calendar = Calendar.getInstance();

        System.out.println("The current date is : " + calendar.getTime());
        calendar.add(Calendar.DATE, -20);
        System.out.println("20 days ago: " + calendar.getTime());
        calendar.add(Calendar.MONTH, 3);
        System.out.println("3 months later: " + calendar.getTime());
        calendar.add(Calendar.YEAR, 5);
        System.out.println("5 years later: " + calendar.getTime());
    }
}
```

Output

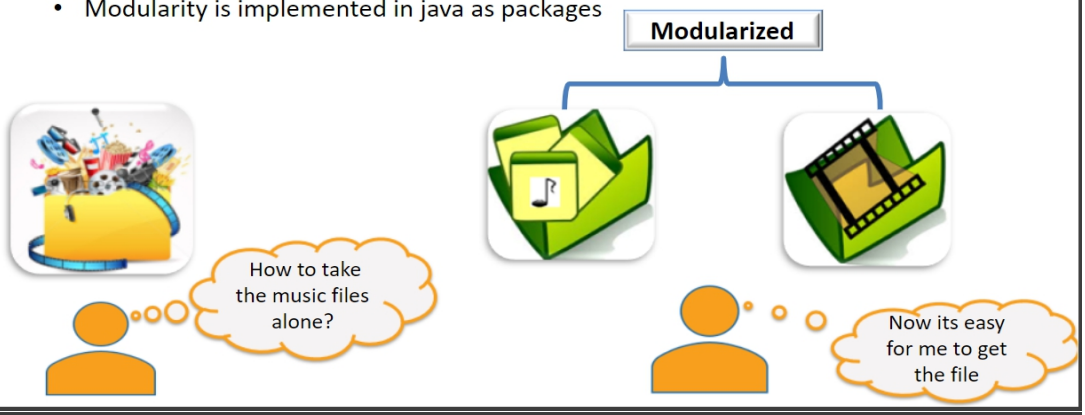
```
The current date is : Fri Sep 13 00:00:00 IST 2019
20 days ago: Sat Aug 24 00:00:00 IST 2019
3 months later: Sun Nov 24 00:00:00 IST 2019
5 years later: Sun Nov 24 00:00:00 IST 2024
```

calendar.getTime () method returns a Date object

Modularity

TEKNOTURF

- Modularity is a way to break a program into smaller units
- Modularity is implemented in java as packages

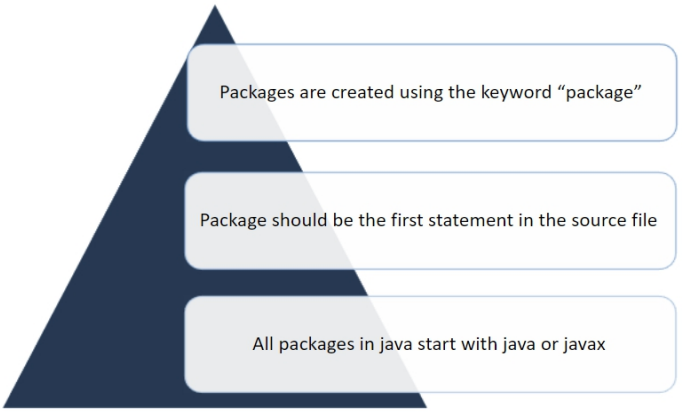


Packages

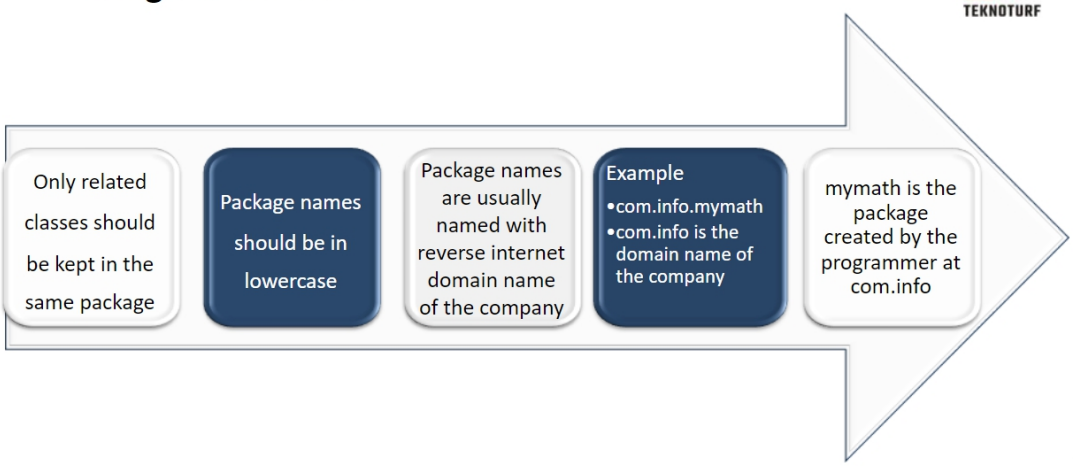
TEKNOTURF

- Used to group semantically related classes
- Help to manage large software systems
- Can contain classes, interfaces and sub-packages
- Are created using the keyword package
- Help in removing name collisions and providing access restrictions

Package Creation



Design Guide lines



Example

```
package <packagename>
class <class name>{
      //logic
}
```

Calculator.java

```
package mymath;
 public class Calculator{
      private int num1=10;
      private int num2=20;
      public int addTwoNum(){
                return num1+num2;
       }
}
```

Compile and Execute

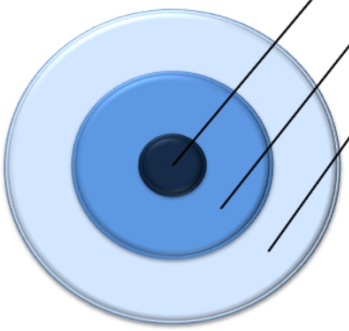
To compile

- javac –d <path for the package> sourcefile
- -d is used to inform the compiler to create a package structure
- <path for the package> specifies the location where the package needs to be created.
- .(dot) specifies the current directory
- javac –d . Calculator.java

To execute

- java <fully qualified class name>
- Fully qualified class name is package name.class name
- java mymath.Calculator

Sub packages



Packages are usually defined using the hierarchical naming pattern

To create a sub package each level is separated using periods (.)

Example

com.info.mymath

import

When a class in one package needs to use the class in a different package, the import keyword is used

Basic syntax of the import statement:

- import <pkg_name>[.<sub_pkg_name>].<class_name>;
  OR
- import <pkg_name>.[<sub_pkg_name>.]*;

The import statement does the following:

- Precedes all class declarations
- Tells the Compiler where to find the classes

Three ways to do import

- import <packagename.classname>
  imports the class of that package in the current package
- import <packagename.*>
  * Is a wild card character that imports all the classes in the current package
- Fully qualified classname
  Use fully qualified class name without import to make the class available in the current package

Example for import

```
package  com.infotech.mymath;
      public class Calculator{
            private int num1;
            private int num2;
            public Calculator()
              {             num1=10;
                            num2=20;
              }
            public int addTwoNum(){
                        return num1+num2;
            }
      }

package com.infotech.mydriver;
import com.infotech.mymath.Calculator;  // 1st way
            public class CalculatorTest{
            public static void main(String[] args)
            { Calculator cobj=new Calculator();
                        System.out.println(cobj.addTwoNum());
            }
}
```

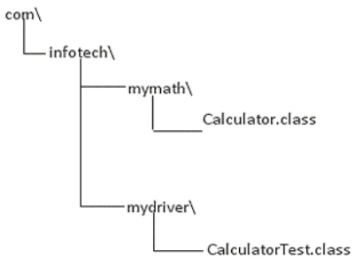
Example Contd

```
package com.infotech.mydriver;
import com.infotech.mymath.*;  //2nd way
            public class CalculatorTest{
            public static void main(String[] args)
            { Calculator cobj=new Calculator();
                        System.out.println(cobj.addTwoNum());
            }
}

package com.infotech.mydriver;
            public class CalculatorTest{
            public static void main(String[] args)
            {
            com.infotech.mymath.Calculator cobj=new
            com.infotech.mymath.Calculator();//3rd way
                        System.out.println(cobj.addTwoNum());
            }
}
```

Directory Layout and Packages

Packages are stored in the directory tree containing the package name.

Package structure created for the example :



CLASSPATH

Classpath contains the list of directories or the jar files location

Compiler searches these directories to look for the needed .class files

CLASSPATH is also used by the JVM for loading the classes

CLASSPATH is set one level above the package.

CLASSPATH can be set in two ways

- As an environmental variable
- Using the command line – classpath option

  Example : To set in windows

  set classpath=%classpath%;<location of the .class file>

Static import

From J2SE 5.0, static fields and methods can be imported using static import

Static import allows you to refer to the members of another class without writing that class's name.

For example

- Import all the static fields and methods of the Math class

```
import static java.lang.Math.*;
double val= PI;
```

- Import a specific field or method

```
import static java.lang.Math.abs;
double  d= abs(-10.4);
```

Core Java Packages

java.lang

- Provides classes that are fundamental to the design of the java programming language
- Imported implicitly for any java program
- Example: Wrapper classes, String, StringBuffer, Object

java.util

- Contains Collection Framework, Date and time , Internationalization Support and utility classes

java.io

- Contains classes for Input/Output Operations

Core Java Packages

TEKNOTURF

java.math

- Provides classes for performing arbitrary precision integer arithmetic(Big Integer) and arbitrary precision decimal arithmetic(Big Decimal)

java.sql

- Provides the API for accessing and processing data stored in the data source(usually a relational database)

java.text

- Provides classes and interfaces for handling text, dates, number and messages in a manner independent of the natural languages