



PACKAGES



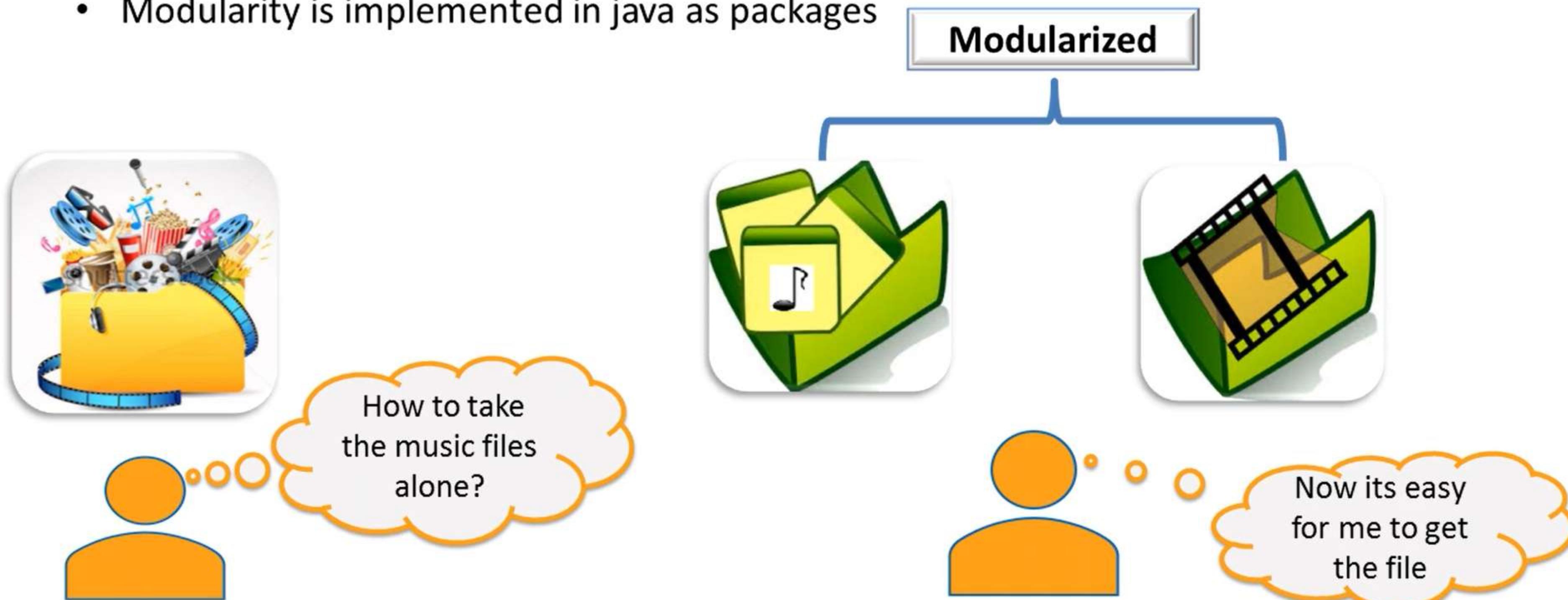
In this module you will learn -

- Package and its usage
- Creation of Package
- Usage of import statement
- Access restriction using packages
- Building class path
- Usage of static import



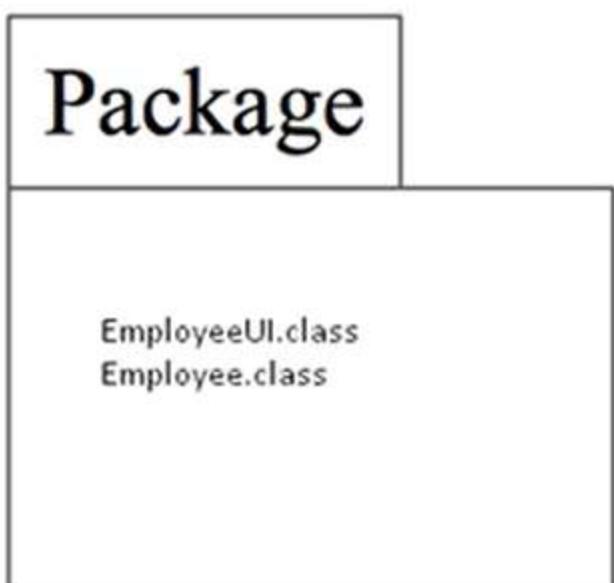
Modularity

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

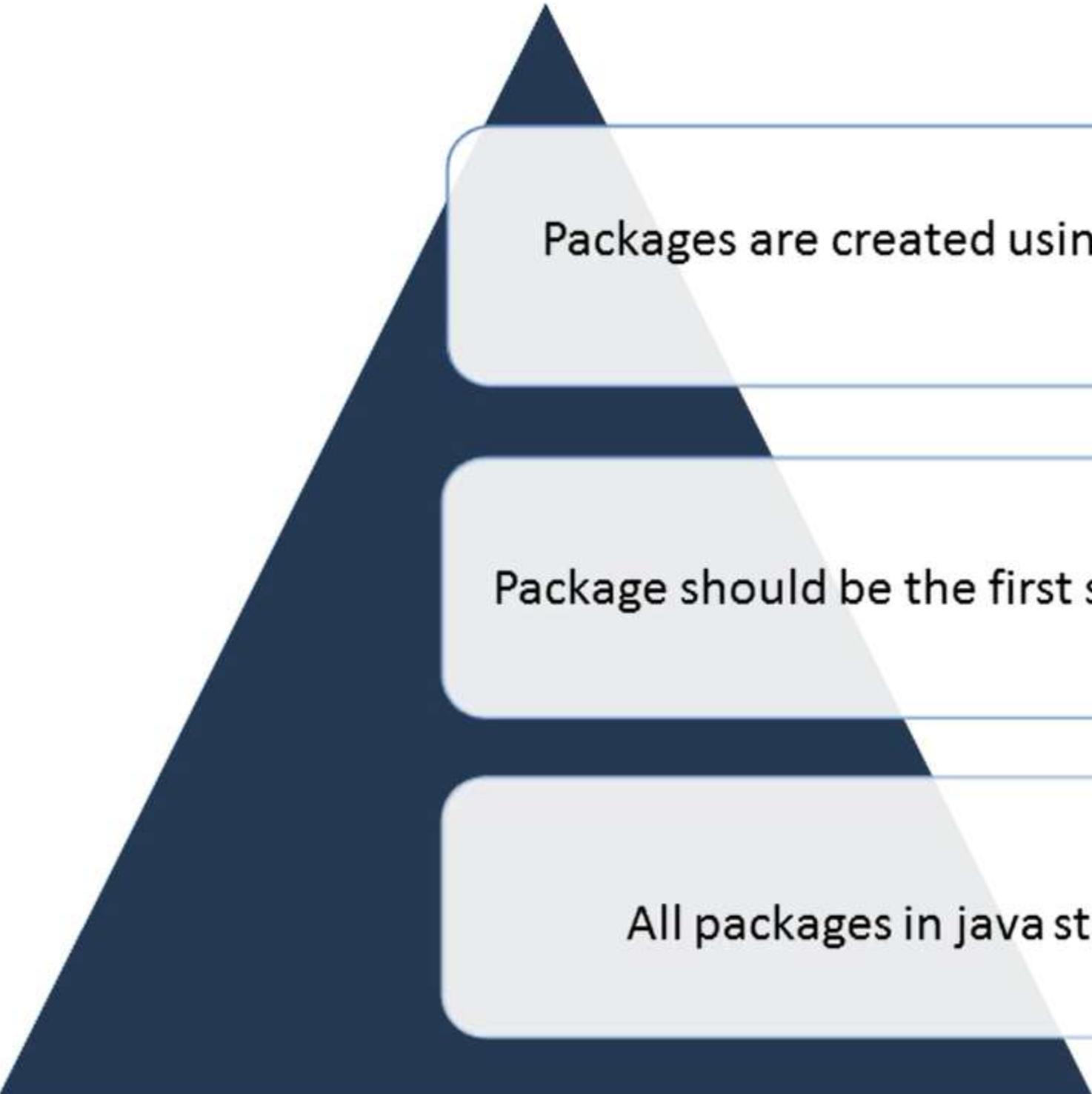


Packages

- 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



Packages are created using the keyword “package”

Package should be the first statement in the source file

All packages in java start with java or javax

Design Guide lines

Only related
classes should
be kept in the
same package

Package names
should be in
lowercase

Package names
are usually
named with
reverse internet
domain name
of the company

Example

- com.info.mymath
- com.info is the
domain name of
the company

mymath is the
package
created by the
programmer at
com.info

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

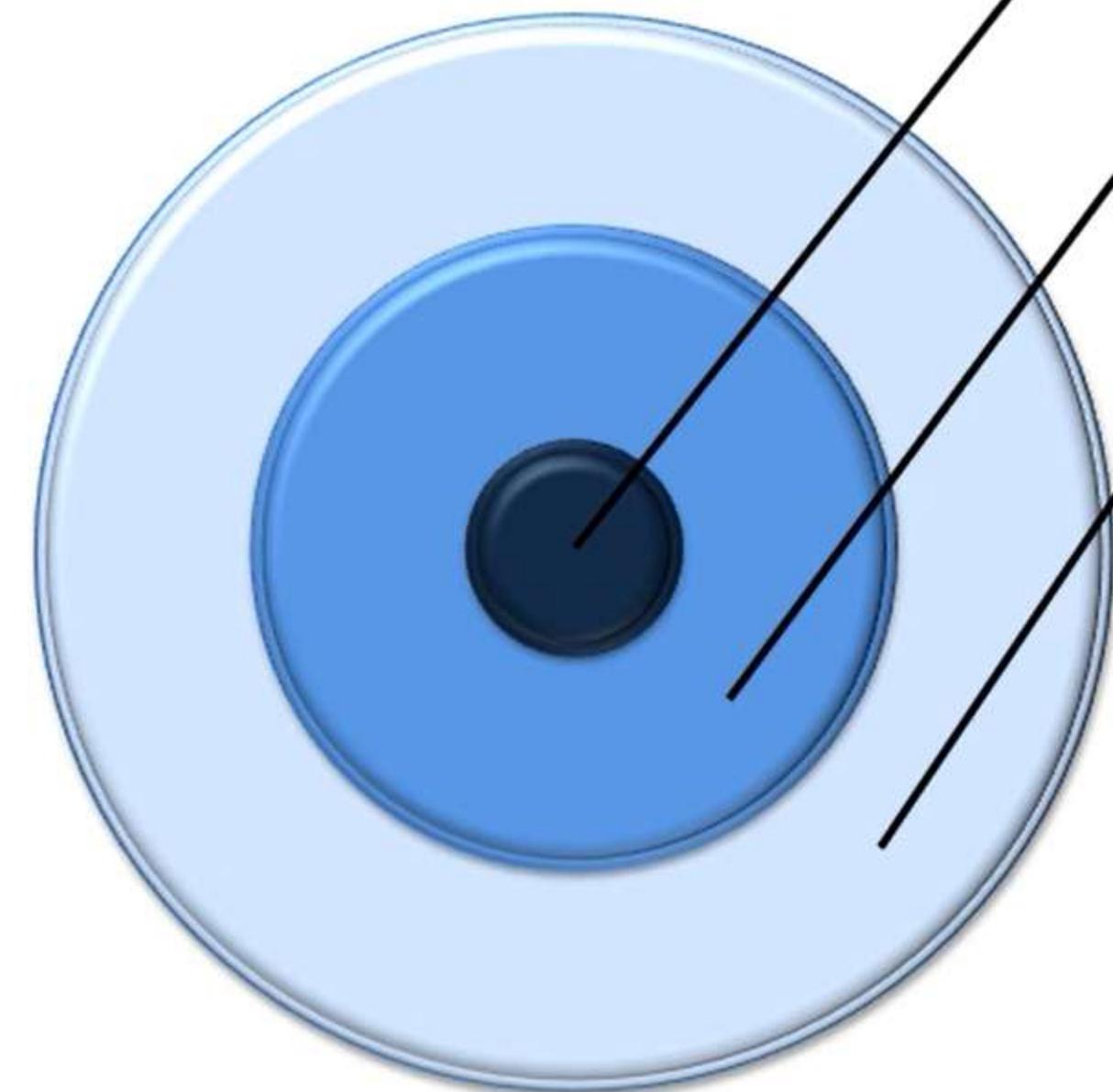
To compile

- `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());  
        }  
    }
```

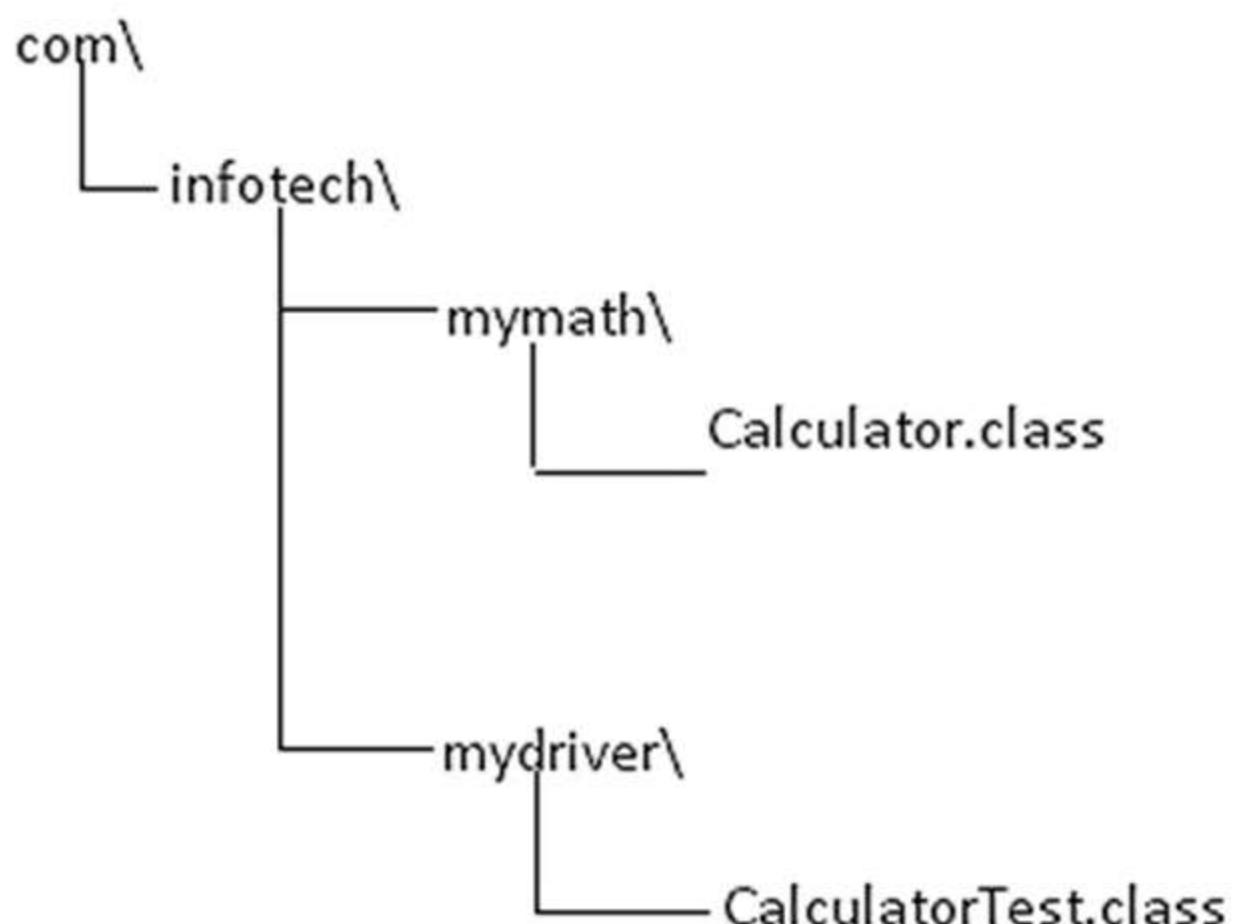
Example Contd

```
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

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

Summary

- Package and its usage
- Creation of Package
- Usage of import statement
- Access restriction using packages
- Building class path
- Usage of static import



DATE AND CALENDAR CLASS IN JAVA



IN THIS MODULE YOU WILL LEARN

- Create Date object
- Use SimpleDateFormat class
- Convert String to Date and vice versa
- Use Calendar class



DATE CLASS

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

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 `getDateFormatInstance`. There are many methods to get the `DateFormatInstance`. One of those methods is `getDateFormatInstance()`

```
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.

```
SimpleDateFormat sdf = new SimpleDateFormat(String pattern)
```

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

- Format the date using the above instance 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-yyyyy          "+formattedDate1);
        System.out.println("dd-M-yyyyy          "+formattedDate2);
        System.out.println("dd-MM-yyyyy hh:mm:ss a "+formattedDate3);
        System.out.println("G dd-MM-yyyyy HH:mm:ss "+formattedDate4);
    }
}
```

Output

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

INHERITANCE

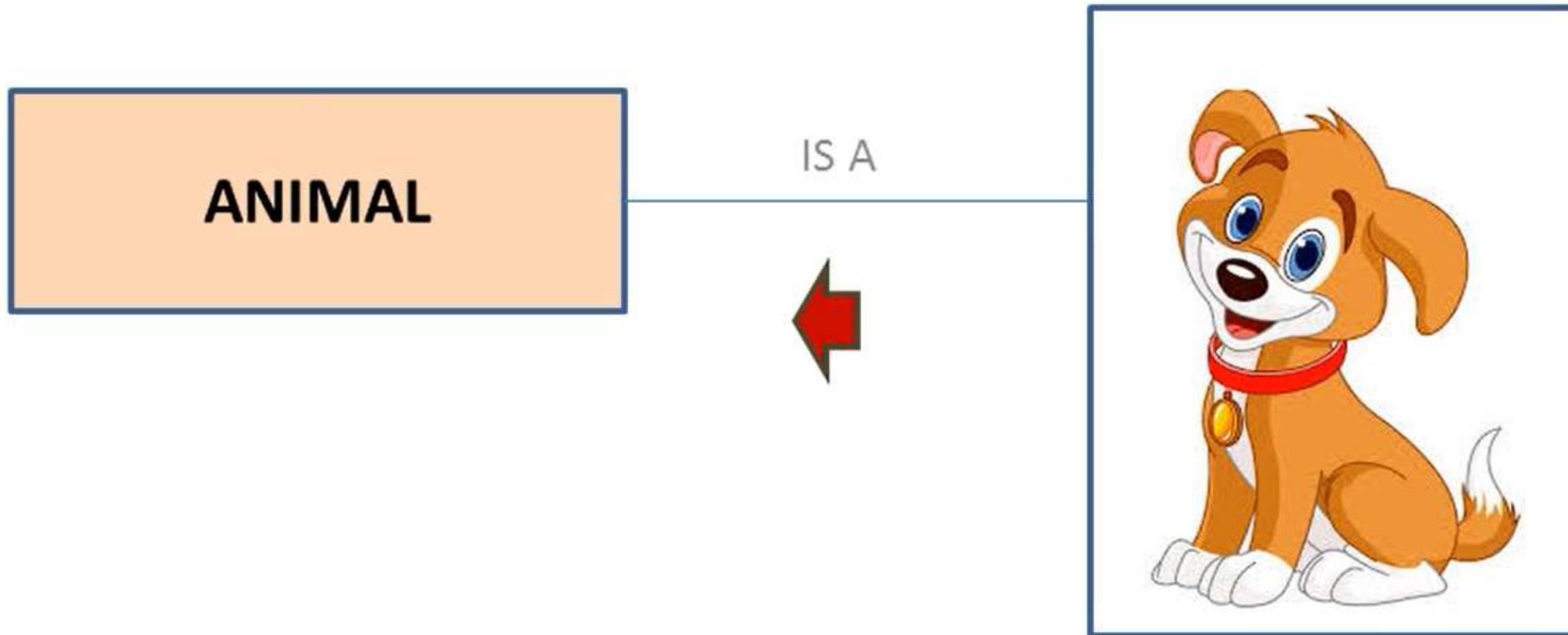


In this module you will learn

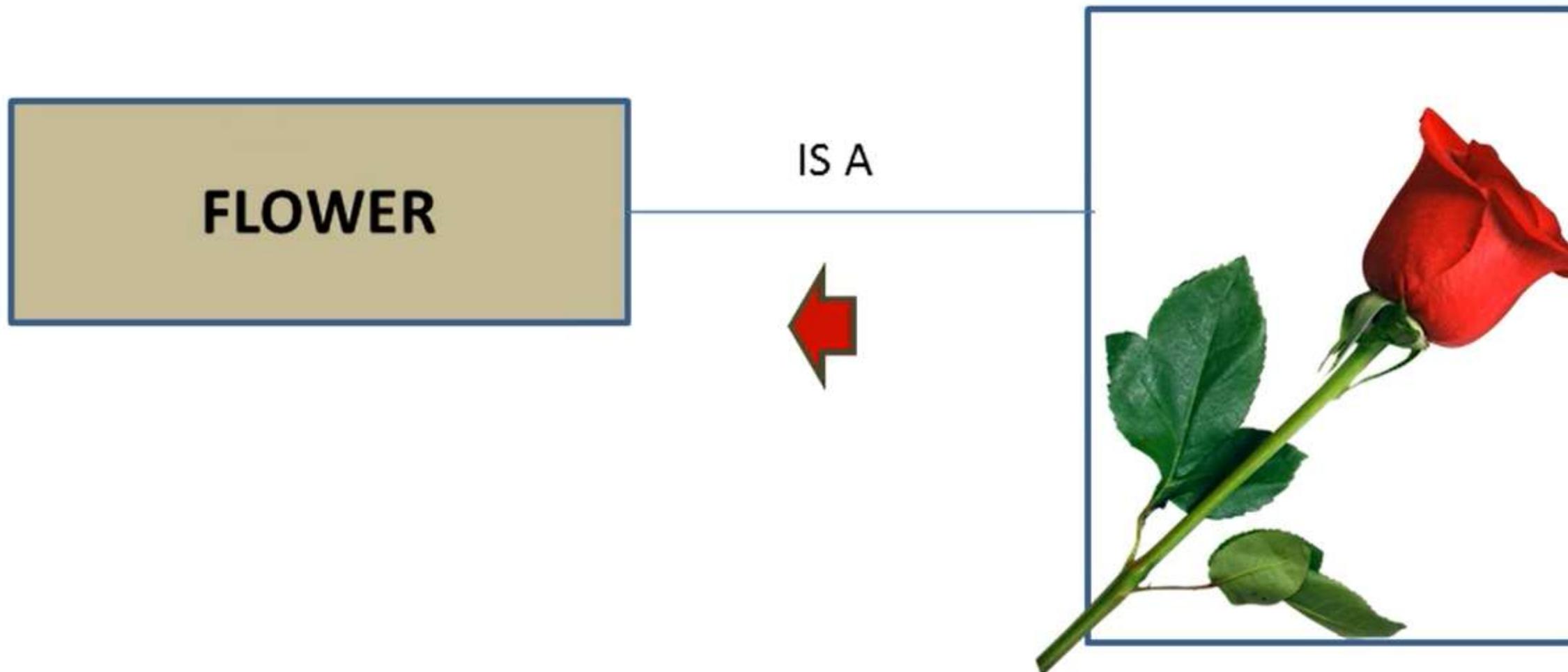
- Inheritance
- Types of Inheritance
- Method Overriding
- Usage of super keyword
- Usage of final keyword
- Cosmic class – Object
- Class Relationship – Aggregation and Composition



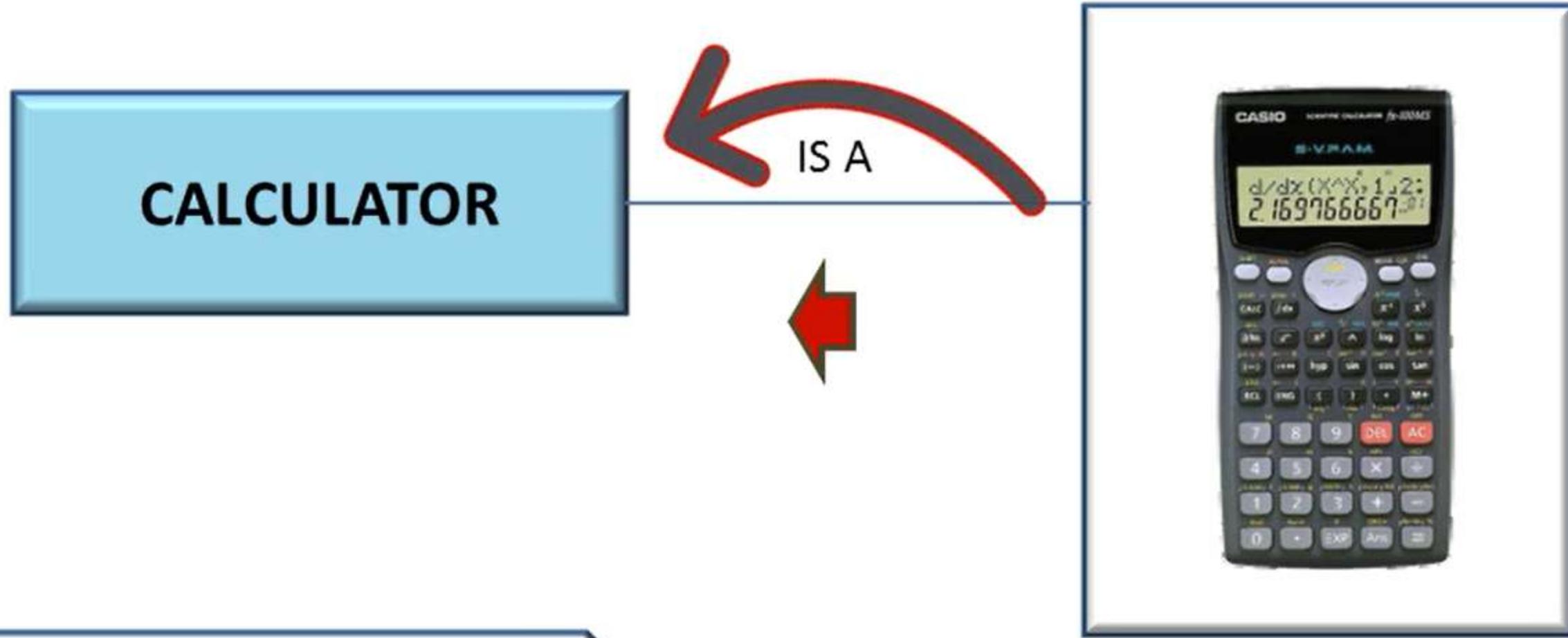
Inheritance - Overview



Inheritance - Overview



Inheritance - Overview



If the relationship between two classes is “**IS A**” or “**IS A TYPE OF**” then that relationship is termed as “**Inheritance**”

Inheritance



Inheritance is one of the major features of OO Concept

Inheritance

Inheritance is one of the major features of OO Concept.

Inheritance is the process in which one object can acquire the properties and behaviour of the parent object.

Using inheritance new classes can be built on already existing classes, so that the new class can reuse methods and fields present in the parent class.

Inheritance represents an **IS-A** relationship, also known as parent-child relationship.

Inheritance

- Different types of objects have certain features in common
- They also possess certain special features and behaviors that make them different

Example

Mountain Bike



Road Bike



Tandem Bike



Common Features : Speed, pedal cadence, gear

Special Features:

Additional
chain ring,
Thick tyre

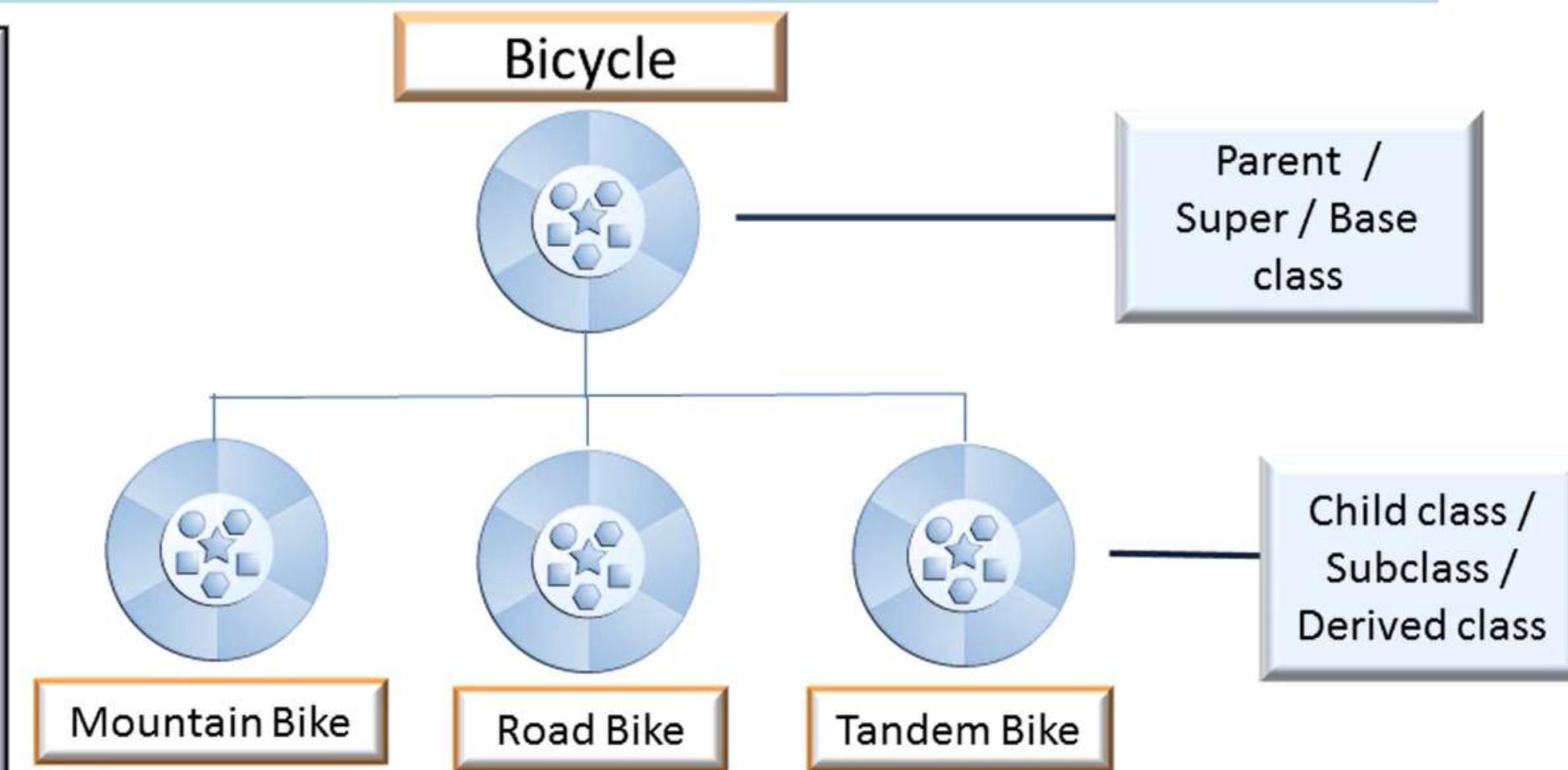
Drop handlebars,
Shock absorbers

Two seats and
two handlebars

Inheritance

Bicycle becomes the superclass of the classes Mountain Bike,
Road Bike and Tandem Bike.

By inheriting from an already existing class, the methods of the parent class can be reused and can also add new fields and methods for the child class



Inheritance

Benefits of Inheritance

- Code Reusability
- Easy maintenance

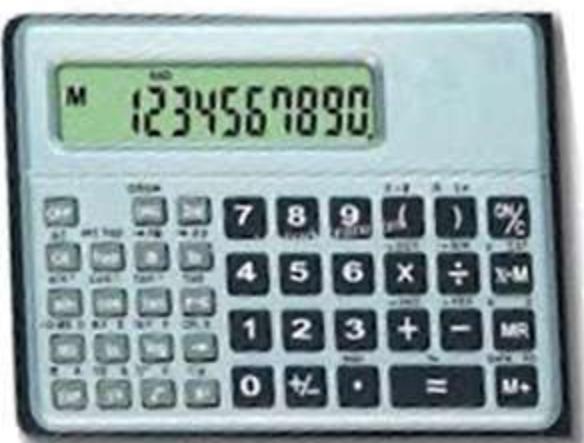
Attributes and methods defined in a super class are automatically inherited by all its sub classes and reused.

If any changes are to be incorporated, it can be done in the super class, which in turn gets reflected to all the sub classes, making maintenance easy.

SimpleCalculator
Add
Subtract
Multiply
Divide



ScientificCalculator
calculateSine
calculateCos
calculateLog
calculateSqrt



Inheritance

Example – Without inheritance

```
public class Employee {  
    private int empld;  
    private String name;  
  
    public Employee()  
    {  
        System.out.println("In Employee Constructor");  
    }  
    //Assume Getters and setters for all attributes  
}
```

```
public class PermanentEmployee  
{  
    private int empld;  
    private String name;  
    private int basicPay;  
  
    public PermanentEmployee()  
    {  
        System.out.println("In  
        PermanentEmployee constructor ");  
    }  
    // Assume Getters and setters for all  
    // attributes  
}
```

Inheritance

If there is an “IS A” relationship between two classes then an inheritance relationship can exist between them

Example - with Inheritance

- Permanent Employee is a type of Employee. So there is an inheritance relationship.

```
public class Employee {  
    private int empld;  
    private String name;  
    public Employee() //constructor  
    {  
        System.out.println("In Employee  
                Constructor");  
    }  
}
```

```
public class PermanentEmployee extends Employee  
{  
    private int basicPay;  
    public PermanentEmployee() //constructor  
    {  
        System.out.println("In PermanentEmployee  
                constructor");  
    }  
}
```

Inheritance

- Example (contd.)

```
public class EmployeeMain {  
    public static void main(String a[]) {  
        PermanentEmployee per_empObj=new PermanentEmployee();  
    }  
}
```

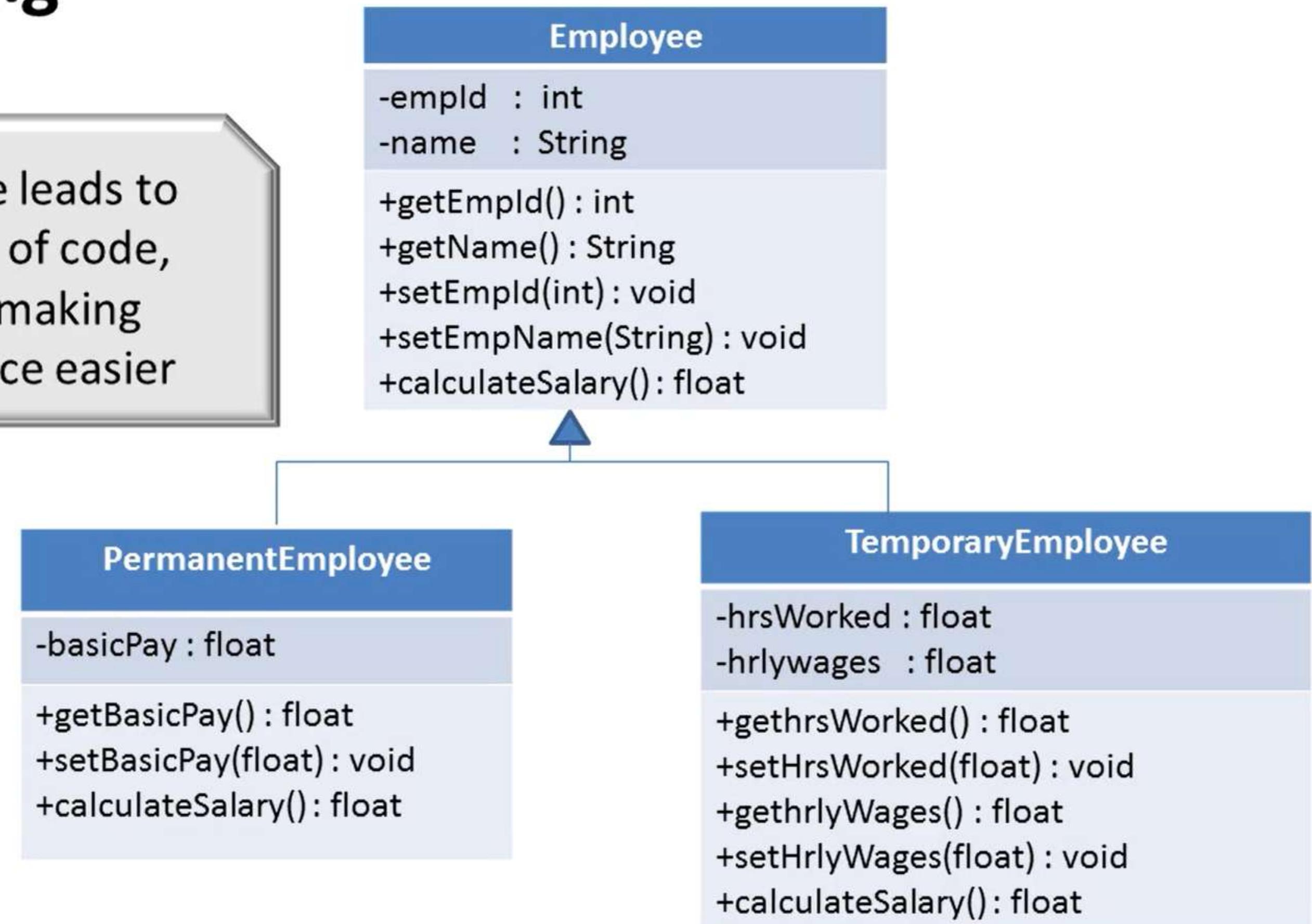
Output :

In Employee Constructor

In PermanentEmployee Constructor

Subclassing

Inheritance leads to reusability of code, thereby making maintenance easier



Inheritance

Private members of the superclass are not inherited by the subclass; instead they can only be indirectly accessed using public methods.

Members with default access specifier in superclass cannot be inherited by subclasses in other packages. They can be accessed directly using their names in subclasses within the same package as the superclass.

The protected modifier specifies that the member can only be accessed within its own package and, in addition, by a subclass of its class in another package.

Constructors are not members of a class. So they are not inherited by a subclass, but the constructor of the superclass can be invoked from the subclass.

Inheritance

Example : when members are not protected

```
public class Employee{  
    private int empld;  
    private String name;  
    public Employee() //constructor  
    {  
        System.out.println("In Employee  
                Constructor");  
    }  
    //Assume Getters and Setters  
}
```

```
public class PermanentEmployee extends Employee  
{    private int basicPay;  
    public PermanentEmployee() //constructor  
    {  
        System.out.println("In PermanentEmployee  
                constructor");  
    }  
    public void display()  
    {  
        System.out.println("ID "+getEmpID()+" Name "  
                            +getName()+" Bpay "+basicPay);  
        // As fields are private use public methods to access  
    }  
}
```

Inheritance

Example using protected

```
public class Employee{  
    protected int empld;  
    protected String name;  
    public Employee() //constructor  
    {  
        System.out.println("In Employee  
                Constructor");  
    }  
}
```

```
public class PermanentEmployee extends Employee  
{  
    private int basicPay;  
    public PermanentEmployee() //constructor  
    {  
        System.out.println("In PermanentEmployee  
                constructor");  
    }  
    public void display()  
    {  
        System.out.println("ID "+empld+" Name "  
                            +name+" Bpay "+basicPay);  
    }  
}
```

Access Control

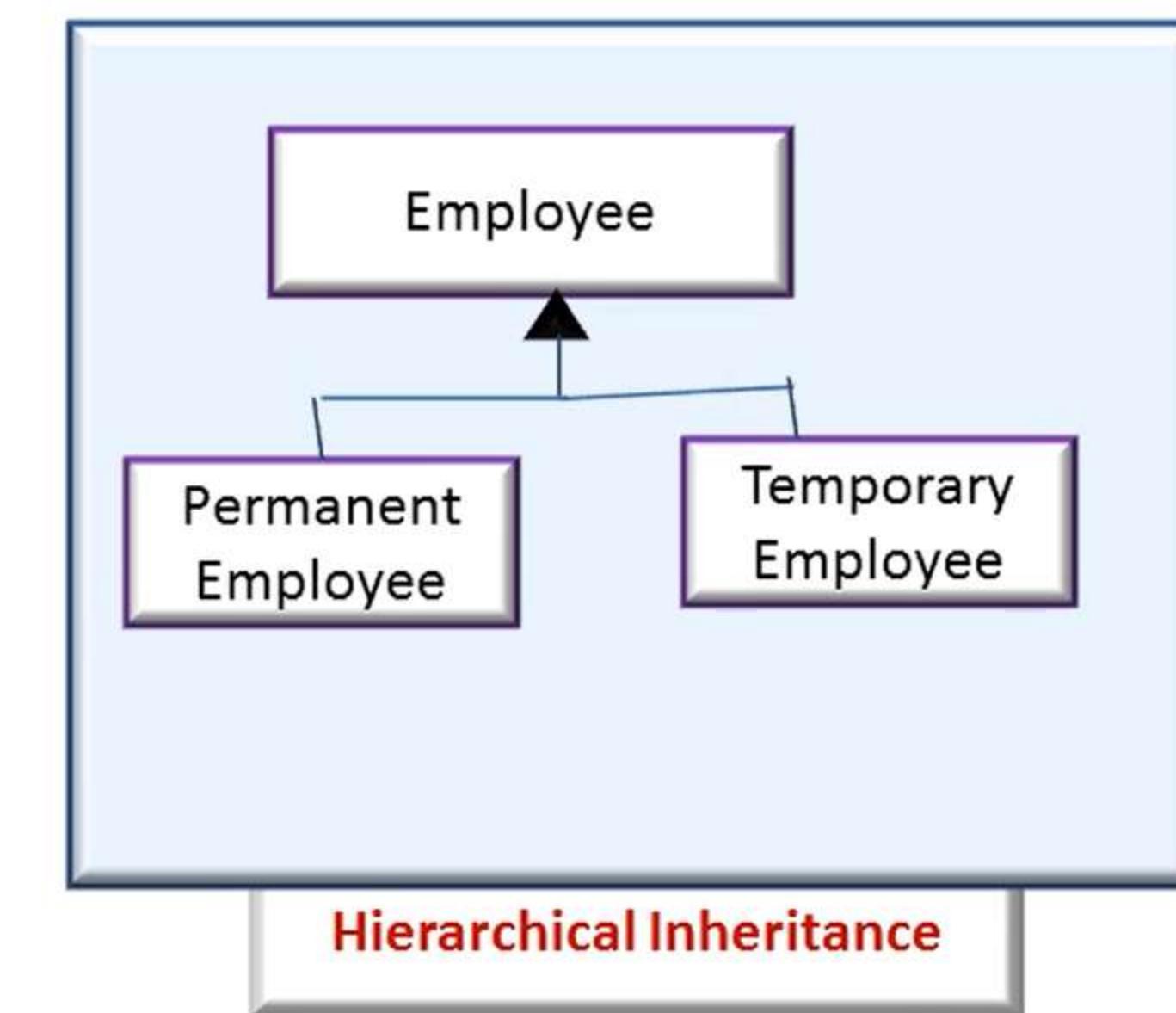
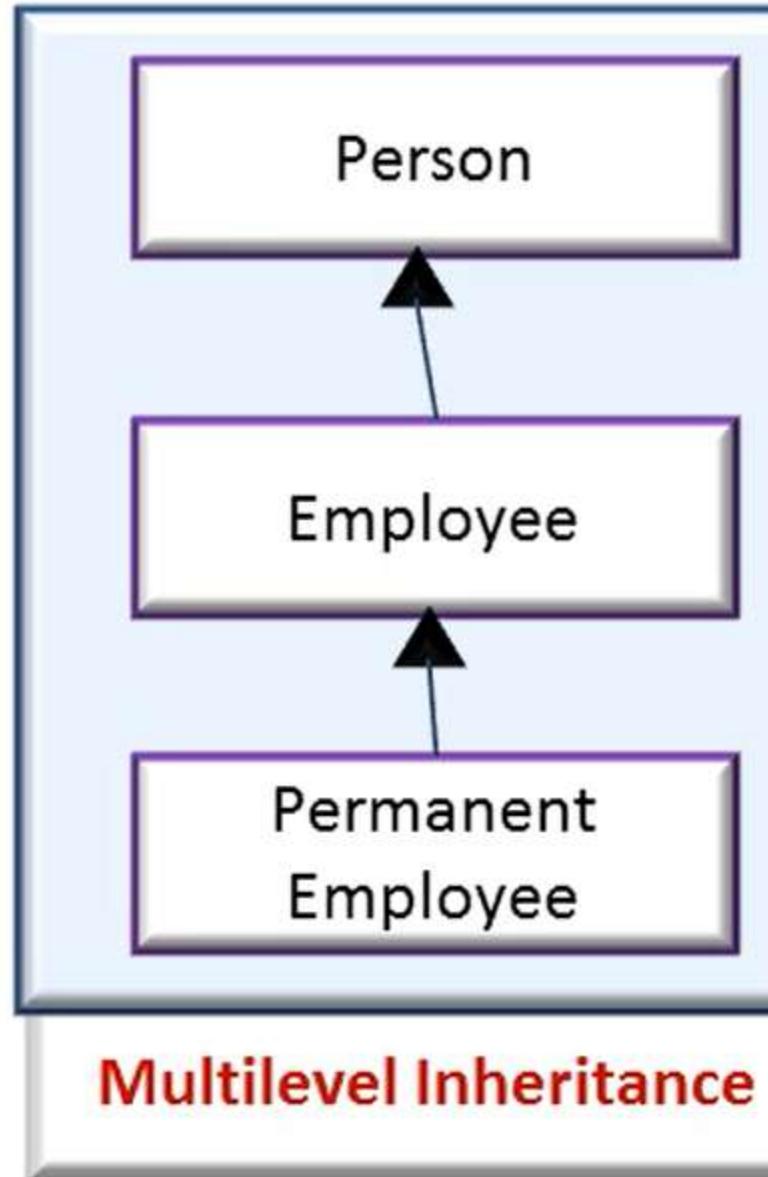
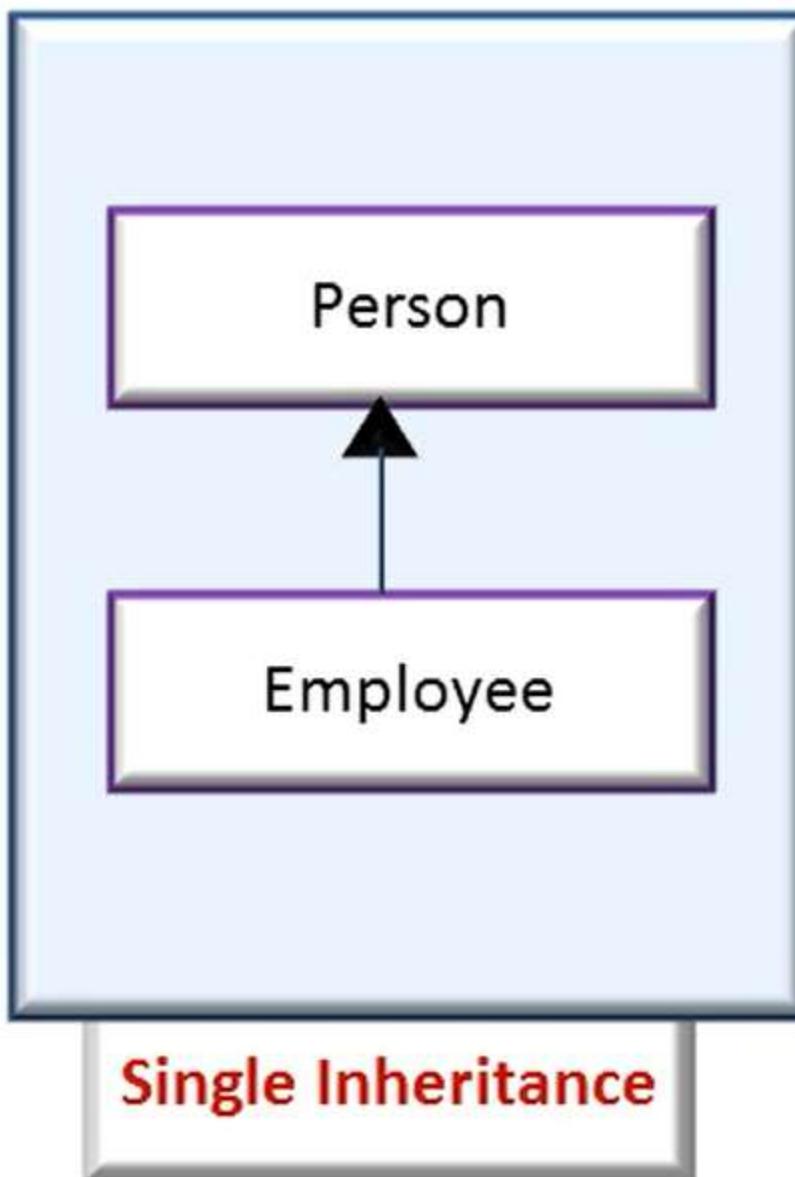
Access modifiers on class member declarations are listed here:

Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Types of Inheritance

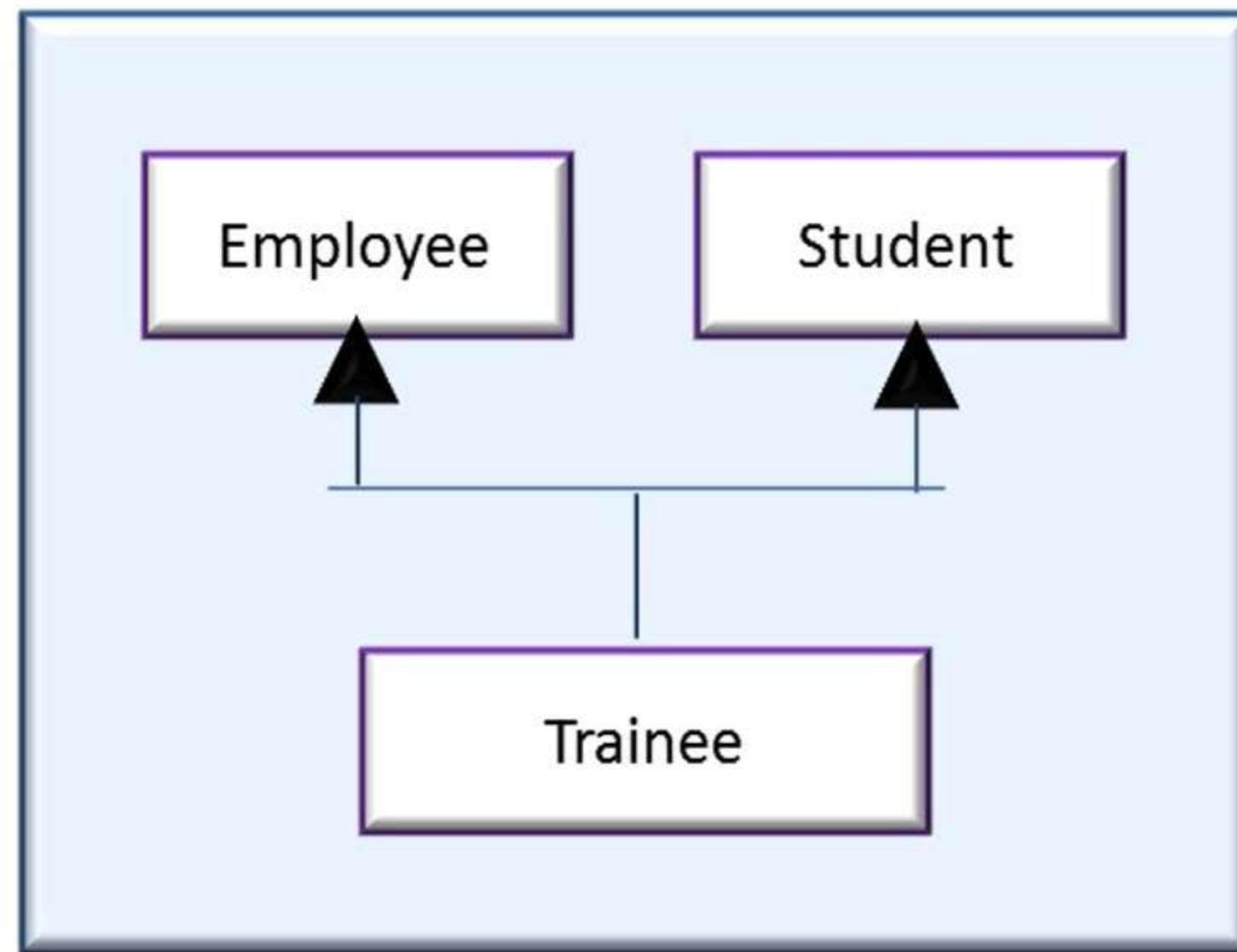
Types of Inheritance

- Single
- Multilevel
- Hierarchical



Types of Inheritance

- When a class extends more than one class we call that as multiple inheritance
- Multiple inheritance is not supported in java.



Method Overriding



Daughter inherits the properties
of Mom

Mom is the super class and
daughter is the subclass.

Method Overriding



Mom is the Super Class and daughter is the Sub Class.

Mom likes to listen to music. Daughter also likes to listen to music.

Mom likes melody, whereas daughter loves pop...

Here daughter overrides the behavior of mom



Overriding Methods

- A subclass can modify behavior inherited from a parent class
- A subclass can have a method with same name, argument list and return type (except co-variant type) as declared in the parent class, but with a different implementation. This concept is called Method overriding.
- Rules for Method overriding
 - Method name should be same as in the parent class
 - Methods argument list should be the same as in parent class
 - Same return type (or its sub type)
 - The subclass overridden method cannot have weaker access than super class method.
Example : If parent class method is declared as public, then the overridden method in sub class cannot be private or protected or default.
- When the overridden method is invoked using a sub class object, it will override the method in the parent class and execute the method in the child class. Hence the concept Method overriding

Method Overriding

- Rules on access specifier in method overriding
 - When overriding a method, the access level cannot be more restrictive than the overridden methods access level.*
 - If not, it results in compilation error*
 - The order of access level is private, default, protected and public (from more restrictive to less)*

Access Level in parent class	Access Level permissible in child class overridden method
Default	default , protected, public
Protected	protected, public
public	public

Method Overriding

Example :

```
public class Employee {  
    protected int empId;  
    protected String name;  
    public Employee() //constructor  
    {  
        System.out.println("In Employee Constructor");  
    }  
  
    //Assume you have written getters and setters for empId and name  
  
    public float calculateSalary()  
    {  
        System.out.println("In Super class calculateSalary Method");  
        return 1;  
    }  
}
```

Method Overriding

Example Contd.

```
public class PermanentEmployee extends Employee {  
    private int basicPay;  
    public PermanentEmployee() //constructor  
    {  
        System.out.println("In PermanentEmployee constructor");  
    }  
    // Assume you have written getters and setters for basicPay  
    public float calculateSalary() // Overriding the method in super class  
    {  
        float da = basicPay * 0.10f;  
        float pf = basicPay * 0.12f;  
        float salary = basicPay + da - pf;  
        return salary;  
    }  
}
```

Overridden Methods

Example Contd.

```
public class EmployeeUtility {  
    public static void main(String a[])  
    {  
        Employee empObj=new Employee();  
        empObj.setEmpld(101);  
        empObj.setName("Rohith");  
        empObj.calcuuateSalary();  
  
        PermanentEmployee perEmpObj=new  
            PermanentEmployee();  
        perEmpObj.setEmpld(102);  
        perEmpObj.setName("Tom");  
        perEmpObj.setBasicPay(35000);  
        System.out.println("Salary is "+  
            perEmpObj.calculateSalary());  
    }  
}
```

Output :

*In Employee Constructor
In Super class calculateSalary Method*

*In Employee Constructor
In PermanentEmployee constructor
Salary is 34300.0*

super

super keyword is a reference variable that refers to the immediate parent of a class

Creation of subclass object implicitly creates a parent class object, which can be referenced using the super keyword

Usage of super keyword

- super is used to invoke immediate parent class constructor
- super is used to invoke immediate parent class method

Use of super()

Usage of super to
invoke immediate
parent class
constructor

- The super() is used to invoke the base class's constructor
- Must be called from constructor of derived class
- Must be first statement within constructor
- Call must match the signature of a valid signature in the base class
- Implicitly called in the constructor, if omitted; so the base class must have a default constructor

super() and this() cannot be used within a same constructor

Usage of super

Example

```
class Employee {  
    public Employee()  
    {  
        System.out.println("In Employee  
                Constructor");  
    }  
}  
  
class PermanentEmployee extends  
        Employee{  
    public PermanentEmployee()  
    {  
        //Compiler adds super() implicitly  
        System.out.println("In  
                PermanentEmployee constructor");  
    }  
}
```

```
public class Main {  
    public static void main(String a[])  
    {  
        PermanentEmployee pObj=new  
            PermanentEmployee();  
    }  
}
```

Output :

*In Employee Constructor
In PermanentEmployee constructor*

Usage of super

Example

```
class Employee {  
    public Employee()  
    {  
        System.out.println("In Employee  
                Constructor");  
    }  
}  
  
class PermanentEmployee extends  
        Employee{  
    public PermanentEmployee()  
    {  
        super(); //can also be called explicitly  
        System.out.println("In  
PermanentEmployee constructor");  
    }  
}
```

```
public class Main {  
    public static void main(String a[])  
    {  
        PermanentEmployee pObj=new  
            PermanentEmployee();  
    }  
}
```

Output :

*In Employee Constructor
In PermanentEmployee constructor*

Usage of Super

- super must be the first statement in constructor
- Parametrized constructor of parent class can be invoked by passing the required parameters while calling super.

Example:

```
public class Employee{  
    protected int empld;  
    protected String name;  
  
    public Employee(int empld, String name)  
    {  
        this.empld = empld;  
        this.name = name;  
    }  
}
```

```
public void display()  
{  
    System.out.println("ID : "+id+  
                      " Name : "+name);  
}  
}
```

Usage of super

```
public class PermanentEmployee extends Employee{  
    private int basicPay;  
    public PermanentEmployee(  
        int emplId, String name, int basicPay)  
    {  
        super(emplId, name);  
        this.basicPay = basicPay;  
    }  
    public void display()  
    {  
        System.out.println("ID : "+id+"Name  
                           : "+name);  
        System.out.println("Basic Pay : "+basicPay);  
    }  
}
```

```
public class Main {  
    public static void main(String a[])  
    {  
        PermanentEmployee pObj=new  
        PermanentEmployee(102,"Tom",35000);  
        pObj.display(); //calls the  
                       //overridden method  
    }  
}
```

Output :

*ID: 102 Name : Tom
Basic Pay : 35000*

Usage of super

Usage of super keyword to invoke super class method

```
public class PermanentEmployee extends Employee{  
    private int basicPay;  
    public PermanentEmployee(  
        int emplId, String name, int basicPay)  
    {  
        super(emplId, name);  
        this.basicPay = basicPay;  
    }  
    public void display()  
    {  
        super.display(); //invokes the overridden method  
        System.out.println("Basic Pay : "+basicPay);  
    }  
}
```

```
public class Main {  
    public static void main(String a[]){  
        PermanentEmployee pObj=new  
        PermanentEmployee(102,"Tom",35000);  
        pObj.display(); //calls the  
                      //overridden method  
    }  
}
```

Both Employee and PermanentEmployee have display() method. Call to display() method of Employee class from PermanentEmployee class is possible using "super" keyword

Constructor chaining

In a derived class constructor, the base class constructor is invoked, either explicitly or implicitly, there will be a whole chain of constructors called, all the way back to the constructor of Object (parent of all classes). This is called constructor chaining.

```
class Person {
    public Person() {
        System.out.println("In Person constructor");
    }
}
class Employee extends Person {
    public Employee() {
        System.out.println("In Employee constructor");
    }
}

class PermanentEmployee extends Employee {
    public PermanentEmployee() {
        System.out.println("In PermanentEmployee constructor");
    }
}

public class Main{
    public static void main(String a[])
    {
        PermanentEmployee e=new PermanentEmployee();
    }
}
```

Output :

*In Person constructor
In Employee constructor
In PermanentEmployee constructor*

final

“final” keyword can be applied with

- variable - to stop value change
- method - to stop Method Overriding
- class - to stop inheriting

Using “final” with a variable

- An instance variable, class variable(static) or a local variable can be made **constant** by declaring it as final.
- A final variable can be assigned a value only once. It will remain unchanged.
- If a final variable holds reference to an object, the state of the object can change, but this variable will always refer to the same object.

final variable

Example:

```
public class Employee{  
    int empld;  
    String name;  
    final String panNo;  
    public Employee(int empld, String name, String panNo)  
    {  
        this.empld = empld;  
        this.name = name;  
        this.panNo = panNo;  
    }  
}
```

A variable declared as final and not initialized is called a **blank final variable**.

A blank final variable forces the constructors to initialize it.

final Method

Using “final” with method

- Methods declared as final cannot be overridden

Example:

```
class Employee{  
    final void display() {  
        System.out.println("In Employee display  
                           method");  
    }  
}  
  
class PermanentEmployee extends Employee{  
    void display() {  
        System.out.println("In PermanentEmployee  
                           display method");  
    }  
}
```

```
public class Main {  
    public static void main(String a[])  
    {  
        PermanentEmployee pObj=new  
                           PermanentEmployee();  
        pObj.display();  
    }  
}
```

Output :
Compile Time Error

final class

Using “final” with class

- A class declared as final cannot be inherited (extended)

Example:

```
final class Employee
{
}

class PermanentEmployee extends Employee
{
}

public class Main {
    public static void main(String a[]) {
        PermanentEmployee pObj=new
        PermanentEmployee();
    }
}
```

Output :
Compile Time Error

Inbuilt String class is final

Object class

Object class is the cosmic class in java. It is called so because it is the parent of all classes by default.

It is available in `java.lang` package

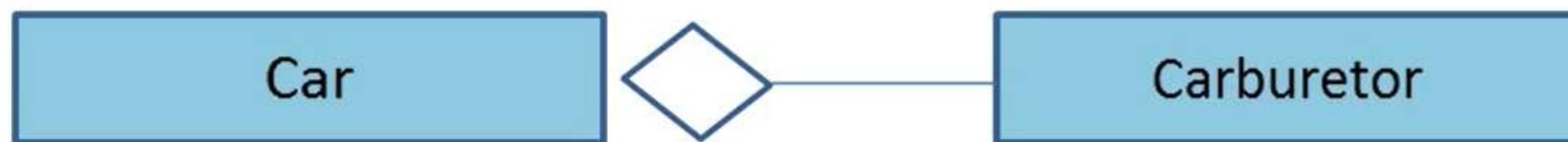
There is no super class for Object class

It has some methods common to all the classes. Few methods are:

- `getClass()`
- `equals()`
- `toString()`

“Has A” Relationship - Aggregation

- When there is a “Has A” relationship between two classes, we term that relationship as Aggregation
- Simple Example is Car “has a ” Carburetor. Or Carburetor “is a part of ” Car.



- Also termed as “whole – part “ relationship. The diamond notation points to the whole.
- In Aggregation, when the whole ceases to exist, the part can exist on its own

Aggregation Relationship

```
public class Carburetor {  
    //Attributes, methods  
}
```

```
public class Car {  
    private String regnNumber;  
    private String model;  
    private Carburetor carburetor;  
  
    public Car(String reg, String modl,  
              Carburetor c) {  
        carburetor = c;  
    }  
    //some code  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        Carburetor c = new Carburetor();  
        Car car = new Car("S101", "Ford", c);  
  
        }  
        //some code  
    }
```

Here even if car is destroyed, that is it is assigned a null value, the Carburetor object still exists.

Composition

- Composition is a stronger form of Aggregation.
- In composition, the part will live and die with the whole.
- Simple Example is Building “has” Floors.



Composition Relationship

```
public class Floor {  
    //Attributes, methods  
}
```

```
public class Building {  
    private String name;  
    private Floor floor;  
  
    public Building(String name) {  
        floor = new Floor();  
    }  
  
    //some code  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        Building b = new Building("MayFlower");  
  
        }  
        //some code  
    }
```

Here building is the owner for creating the floor. Hence if Building is destroyed, that is, it is assigned a null value, the Floor object also gets destroyed

Composition Relationship

```
class Customer
{
//Attributes and methods
}

class Account
{
String accNo;
double balance;
Customer customer;
//Other attributes and methods
}
```

```
class Bank {
List<Customer> custList = new
ArrayList<Customer>();
List<Account> accountList = new
ArrayList<Account>();

public void addCustomer(Customer c){
//some code
}
public void openAccount(String accNo,
double balance, Customer c) {
Account a = new Account(accNo,balance,c);
}
}
```

Summary

- Inheritance
- Types of Inheritance
- Method Overriding
- Usage of super keyword
- Usage of final keyword
- Cosmic class – Object
- Class Relationship – Aggregation and Composition



POLYMORPHISM



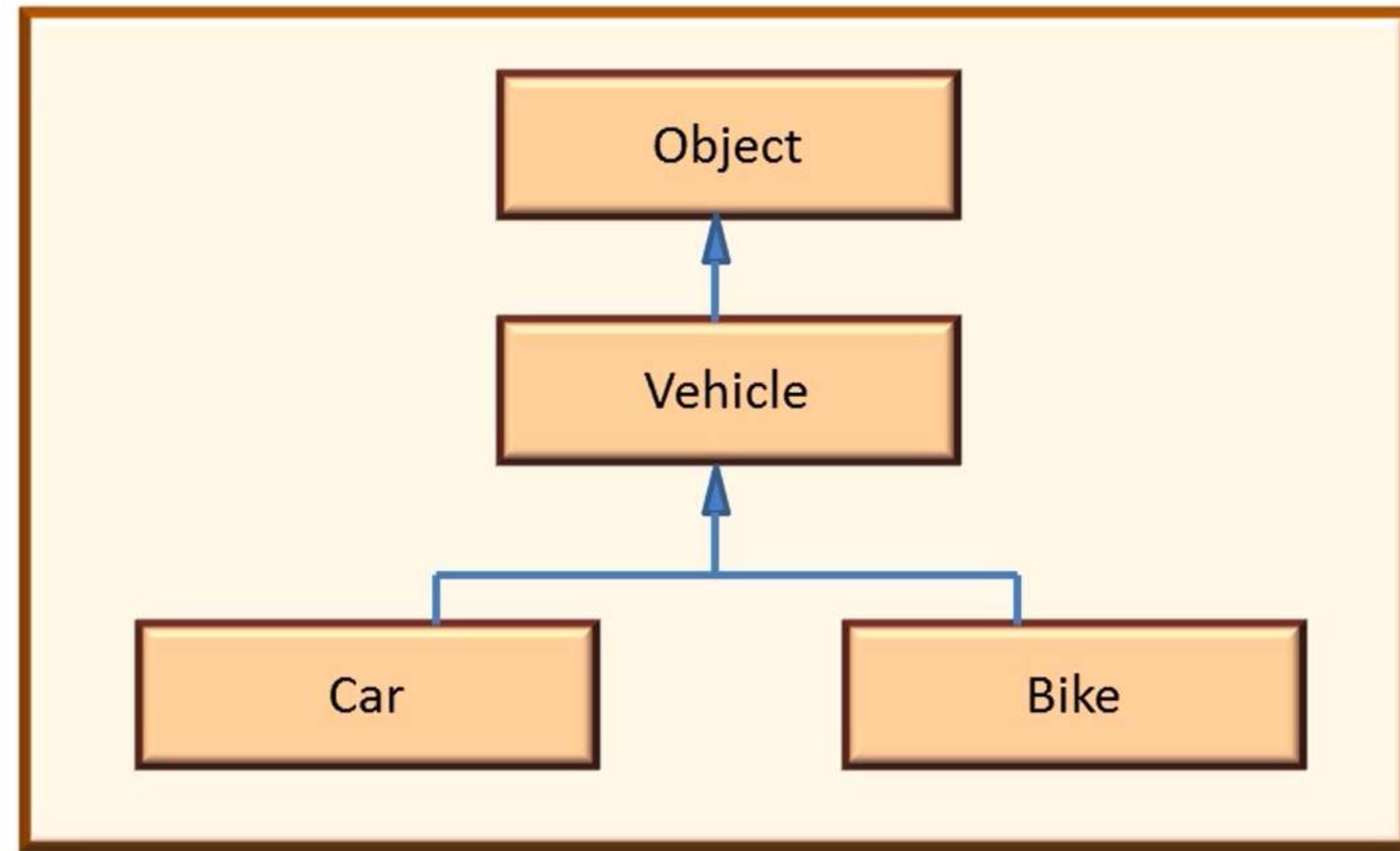
In this module you will learn

- Upcasting and downcasting
- Usage of instanceof operator
- Usage of equals method and toString method
- Polymorphism and its implementation in java



Upcasting and Downcasting

Consider a parent class and a child class. For eg., Vehicle is the parent class and Car and Bike are child classes.



Upcasting and Downcasting

When reference variable of Parent class refers to an object of Child class, it is known as Upcasting



Casting a reference of Parent class to one of its child class is called
Downcasting

Upcasting

Objects for these classes can be created as:

```
Vehicle v=new Vehicle(); // Vehicle reference pointing to Vehicle Object
```

```
Car c=new Car(); // Car reference pointing to Car Object
```

```
Vehicle v1=new Car(); // Vehicle reference pointing to Car Object
```

Note that a **parent class reference** can hold a **child class object**.

This we call as **Upcasting**

By casting, the object is not changed, but it is labeled differently.

Casting of an object of child class to its parent class is called Upcasting. It is done implicitly i.e., done automatically.

Upcasting can be done whenever there is an IS-A relationship.

Upcasting

```
public class Vehicle {  
    private int power;  
  
    public Vehicle(int power) {  
        this.power = power;  
    }  
    public int getPower() {  
        return power;  
    }  
    public void setPower(int power) {  
        this.power = power;  
    }  
    public void display() {  
        System.out.println("Vehicle Object");  
    }  
}
```

```
public class Car extends Vehicle {  
    private int noOfSeats=1;  
  
    public Car(int power, int noOfSeats) {  
        super(power);  
        this.noOfSeats = noOfSeats;  
    }  
  
    public int getNoOfSeats() {  
        return noOfSeats;  
    }  
    public void setNoOfSeats(int noOfSeats) {  
        this.noOfSeats = noOfSeats;  
    }  
    public void display() {  
        System.out.println("Car Object");  
    }  
}
```

Usage of instanceof operator

```
public class Truck extends Vehicle {  
    private float load;  
  
    public Truck(int power, float load) {  
        super(power);  
        this.load = load;  
    }  
    public float getLoad() {  
        return load;  
    }  
    public void setLoad(float load) {  
        this.load = load;  
    }  
    public void display() {  
        System.out.println("Truck Object");  
    }  
}
```

```
public class Road  
{  
    public void ride(Vehicle v)  
    {  
        if(v instanceof Car) {  
            System.out.println("Car Object in  
                ride");  
        }  
        else if(v instanceof Truck) {  
            System.out.println("Truck Object in  
                ride");  
        }  
        else {  
            System.out.println("Vehicle Object  
                in ride");  
        }  
    }  
}
```

Upcasting

```
public class MainVehicle {  
  
    public static void main(String a[]) {  
        Vehicle v1=new Vehicle(100);  
        Vehicle v2=new Car(100,5);  
        Vehicle v3=new Truck(200,1000);  
  
        v1.display(); //invokes Vehicle's display  
        v2.display(); //invokes Car's display  
        v3.display(); //invokes Truck's display  
  
        Road r=new Road();  
        r.ride(v1);  
        r.ride(v2);  
        r.ride(v3);  
    }  
}
```

Output

Vehicle Object

Car Object

Truck Object

Vehicle Object in ride

Car Object in ride

Truck Object in ride

Note :

In Road class, the argument passed for ride method is Vehicle reference.

It can accept objects of type Vehicle and also objects of class that inherits Vehicle. (Upcasting)

Upcasting

When an overridden method is called through a superclass reference, Java determines which version of the method to call, based upon the type of the object being referred to at the time the call occurs.

```
v1.display();  
v2.display();  
v3.display();
```

When a reference of Vehicle invokes the overridden method display, the method invoked depends on the object invoking that method and not on the reference.

Hence we call method overriding as **dynamic binding**.

Downcasting

To invoke a method that is specific to child class, we need

- a reference of that class or
- Cast the reference of the Parent to the Child class

Casting of parent class reference to that specific child class is known as Downcasting

```
Vehicle v=new Car(100,5); //Upcasting
```

To invoke the child class specific method, say getNoOfSeats(),

```
Car c= (Car) v;           //Downcasting  
c.getNoOfSeats();
```

Usage of instanceof operator

- Downcasting has to be performed manually
- Using instanceof in downcasting:

```
public class Road
{
    public void displayDetails(Vehicle v)
    {
        if(v instanceof Car) {
            Car c=(Car)v;
            System.out.println("No Of Seats" +c.getNoOfSeats());
        }
        else if(v instanceof Truck) {
            Truck t =(Truck)v;
            System.out.println(" Max Load " +t.getLoad());
        }
    }
}
```

OBJECT Class – equals method

- To compare if two objects are equal use the equals() method in Object class:

```
public boolean equals(Object o)
```

- As this method is used by all classes, it takes a reference of Object class as parameter.
- Parent class reference can hold any child class object.

Object class is the parent of all class. So the parent class reference can hold any child class object.

OBJECT Class – equals method

- Example

```
public class Employee {  
  
    protected int employeeId;  
    protected String name;  
  
    public Employee(int employeeId, String name) {  
        this.employeeId = employeeId;  
        this.name = name;  
    }  
  
    public static void main(String a[]) {  
        Employee empObj1 = new Employee(101, "Peter");  
        Employee empObj2 = new Employee(102, "Tom");  
        Employee empObj3 = new Employee(101, "Peter");  
        Employee empObj4 = empObj1;  
        System.out.println(empObj1.equals(empObj2)); //false  
        System.out.println(empObj1.equals(empObj3)); //false  
        System.out.println(empObj1.equals(empObj4)); //true  
    }  
}
```

empObj1 and empObj2 hold different objects.

So that comparison returns false.

empObj1 and empObj3 hold different objects that have same values.

But as they hold different address this comparison returns false.

empObj1 and empObj4 both hold the same object. So that comparison returns true.

OBJECT Class – equals method

- When using equals method, if the comparison is to be done for the values held in object and not reference, then override the equals method.
- Override the equals method in the Employee class
- Invoke the equals method for the employee objects.

OBJECT Class – equals method

- Example

```
public class Employee {  
    protected int employeeId;  
    protected String name;  
  
    public Employee(int employeeId, String name) {  
        this.employeeId = employeeId;  
        this.name = name;  
    }  
  
    public boolean equals(Object o)  
    {  
        Employee e = (Employee) o;  
        if(e.employeeId == this.employeeId &&  
            e.name.equals(this.name))  
            return true;  
        else  
            return false;  
    }  
  
    public static void main(String a[]){  
        Employee empObj1 = new Employee(101,"Peter");  
        Employee empObj2 = new Employee(102,"Tom");  
        Employee empObj3 = new Employee(101,"Peter");  
        Employee empObj4 = empObj1;  
        System.out.println(empObj1.equals(empObj2)); //false  
        System.out.println(empObj1.equals(empObj3));//true  
        System.out.println(empObj1.equals(empObj4));//true  
    }  
}
```

empObj1.equals(empObj3) invokes the overridden equals method in Employee class.

empObj1 is the current object, 'this' and empObj3 is passed as o.

To compare this.employeeId with parameter o.employeeId,
employeeId is specific to child Employee.

So, here it needs to do downcasting.

Then perform the necessary comparison and return true or false accordingly.

OBJECT Class – `toString` method

- Create an object for Employee and print the reference:

```
Employee empObj1 = new Employee(101, "Peter");  
System.out.println(empObj1);
```

- This print statement invokes the `toString` method in Object class.
The `toString` method invokes the `hashCode` method in Object class, which returns an int.
This `toString` method converts the `hashCode` to hexa decimal and prints that value.
- To make this print the employee details, override the `toString` method
What needs to be printed needs to be returned as String.

OBJECT Class – `toString` method

- Example

```
public class Employee {  
    protected int employeeId;  
    protected String name;  
  
    public Employee(int employeeId, String name) {  
        this.employeeId = employeeId;  
        this.name = name;  
    }  
  
    public String toString() {  
        return "ID "+employeeId+" Name "+name;  
    }  
  
    public static void main(String a[]) {  
        Employee empObj1 = new Employee(101,"Peter");  
        System.out.println(empObj1);  
    }  
}
```

Output :
ID 101 Name Peter

Polymorphism Overview



Mobile behaves differently in different places. It behaves as Camera, Music Player, Phone, Send Message, Chat, Play games etc.,

One object takes many forms. This we call as polymorphism

In OOP, Polymorphism is the concept in which a variable or a function can take many forms.

Polymorphism

Polymorphism is a Latin word which means many (poly) forms (morph)

Polymorphism is the capability of a method to do different things based on the object that it is acting upon



Move()

Trotting



Swimming



Flying

Polymorphism

Two types of Polymorphism

Compile Time Polymorphism / Static Binding

Run Time Polymorphism / Dynamic Binding

Static Binding is achieved through Method Overloading

Method overloading

- In a class, there is more than one method with the same name, but different parameters.
- Depending on the parameters passed, the corresponding method is invoked. This is decided at Compile time. Hence we call method overloading as compile time polymorphism or static binding.

Polymorphism

Dynamic Binding Is achieved through Method Overriding

Method overriding

- When a method in a sub class has the same name and signature as a method in the super class, then the method in sub class overrides the method in the super class.
- When an overridden method is called through a superclass reference, Java determines which version of the method to call, based upon the type of the object being referred to at the time the call occurs.
- As method overriding depends on the object invoking it, we call this concept as run time polymorphism or dynamic binding.

Summary

- Upcasting and downcasting
- Usage of instanceof operator
- Usage of equals method and toString method
- Polymorphism and its implementation in java



ABSTRACT CLASS



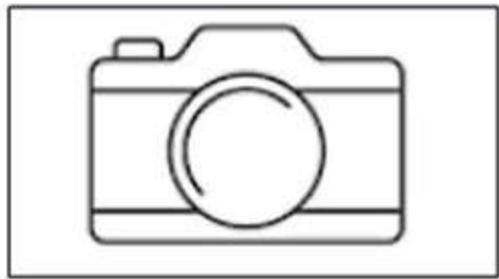
In this module you will learn

- Abstract methods
- Abstract class
- Inherit an abstract class
- Abstraction achieved through Abstract class



Abstract class - Overview

Camera



Functions expected in a
camera



Take photo
Save photo
Take video
Delete photo
Delete video

All functions of camera are implemented in these
objects in their own way.

Abstract class - Overview

Likewise, we can write abstract methods leading to abstract classes in java. Let's discuss abstract class in detail...

Abstract class

A class declared as abstract is an abstract class

If a class contains any abstract method it has to be declared as abstract

Syntax : **abstract class class_name**

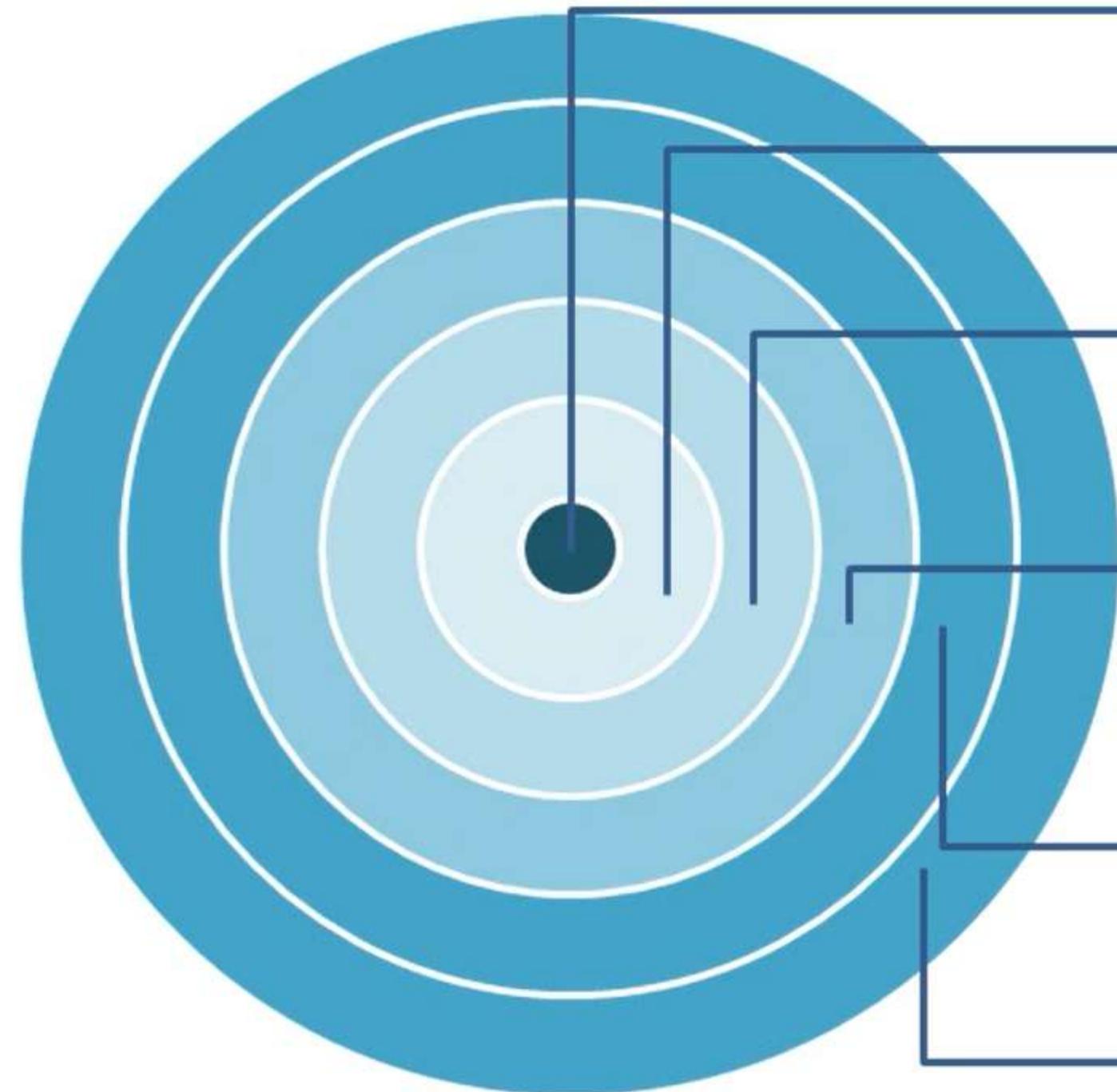
What is an Abstract Method?

- A method in a class that is just declared without any implementation
- It just has the method signature

Syntax : **abstract return_type function_name(<parameterlist>);**

Abstract class can also have attributes and normal methods with implementations along with abstract methods.

Abstract class



- Abstract class cannot be instantiated.
- Abstract class can be subclassed
- When a class inherits an abstract class with abstract methods, it has to provide implementation for all the abstract methods to make it a concrete class
- If the subclass does not implement all the abstract methods, then the subclass also needs to be declared abstract
- Abstract class can contain non abstract methods also
- If an object need not be created for a class, it can be declared as abstract though it does not have any abstract methods.

Abstract classes contains zero or more abstract methods, which are later implemented by concrete classes.

A class cannot be both abstract and final.

Abstract Class

Example

```
abstract public class Employee {  
    protected int empld;  
    protected String name;  
  
    public Employee(int empld, String name) {  
        this.empld = empld;  
        this.name = name;  
    }  
    public int getEmpld() {  
        return empld;  
    }  
    public void setEmpld(int empld) {  
        this.empld = empld;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    abstract public float calculateSalary();  
}
```

Abstract Class

Example Contd.

```
public class PermanentEmployee extends Employee {  
    private int basicPay;  
    public PermanentEmployee(int emplId,  
                            String name, int basicPay) {  
        super(emplId, name);  
        this.basicPay = basicPay;  
    }  
    public int getBasicPay() {  
        return basicPay;  
    }  
    public void setBasicPay(int basicPay) {  
        this.basicPay = basicPay;  
    }  
}
```

```
// Implementation of abstract method  
public float calculateSalary(){  
    float da = basicPay * 0.10f;  
    float pf = basicPay * 0.12f;  
    float salary = basicPay+da-pf;  
    return salary;  
}  
}
```

Abstract Class

Example Contd.

```
public class TemporaryEmployee extends Employee {  
    private int noOfHrs;  
    private int hrlywages;  
    public TemporaryEmployee(int empld,  
        String name, int noOfHrs, int hrlywages) {  
        super(empld, name);  
        this.noOfHrs = noOfHrs;  
        this.hrlywages = hrlywages;  
    }  
    public int getNoOfHrs() {  
        return noOfHrs;  
    }  
    public void setNoOfHrs(int noOfHrs) {  
        this.noOfHrs = noOfHrs;  
    }
```

```
    public int getHrlywages() {  
        return hrlywages;  
    }  
    public void setHrlywages(int hrlywages) {  
        this.hrlywages = hrlywages;  
    }  
    //Implementation of abstract method  
    public float calculateSalary(){  
        float salary = noOfHrs * hrlywages;  
        return salary;  
    }  
}
```

Abstract Class

Example Contd.

```
public class EmployeeUtility{  
    public static void main(String a[]) {  
        Employee permanentEmployeeObj=new PermanentEmployee(101,"Rohith",20000);  
        System.out.println("Salary of "+permanentEmployeeObj.getName()+" is "+  
                           permanentEmployeeObj.calculateSalary());  
  
        Employee tempEmpObj=new TemporaryEmployee(102,"Tom",60,200);  
        System.out.println("Salary of "+tempEmpObj.getName()+" is "+  
                           tempEmpObj.calculateSalary());  
    }  
}
```

Output :

Salary of Rohith is 19600.0
Salary of Tom is 12000.0

Using the parent reference, we can invoke all methods in the parent and the overridden methods in the child. When overridden methods are invoked, which method is invoked depends on the object it holds leading to run time polymorphism.

Abstract Class

- Consider the below method in an abstract class

```
abstract void calculateInsuranceBonus();
```

VALID OVERRIDDEN METHODS

- void calculateInsuranceBonus() { }
- protected void calculateInsuranceBonus() { }
- public void calculateInsuranceBonus() { }

INVALID OVERRIDDEN METHODS

-  private void calculateInsuranceBonus() { }

```
abstract protected void calculateInsuranceBonus();
```

VALID OVERRIDDEN METHODS

- protected void calculateInsuranceBonus() { }
- public void calculateInsuranceBonus() { }

INVALID OVERRIDDEN METHODS

-  private void calculateInsuranceBonus() { }
-  void calculateInsuranceBonus() { }

Abstract Class

- Consider the below abstract class

```
abstract public class Account {  
    int accountNumber;  
    double balance;  
  
    public double getBalance() {  
        return balance;  
    }  
    abstract public double calculateInterest();  
    abstract public void displayInterest(float interest);  
}
```

VALID OVERRIDDEN METHODS

class SavingsAccout extends Account {

 public double calculateInterest() {
 return balance * 0.02 ;
 }
 public void displayInterest(float interest) {
 }
}

INVALID OVERRIDDEN METHODS

class SavingsAccout extends Account {
 public double calculateInterest() {
 return balance * 0.02 ;
 }
}

class SavingsAccout extends Account {

 double calculateInterest() {
 return balance * 0.02 ;
 }
 void displayInterest(float interest) {
 }
}

OOP Concept - Abstraction In Java

In Java, abstraction is achieved by Abstract class and Interface

An abstract class can have zero or more abstract methods

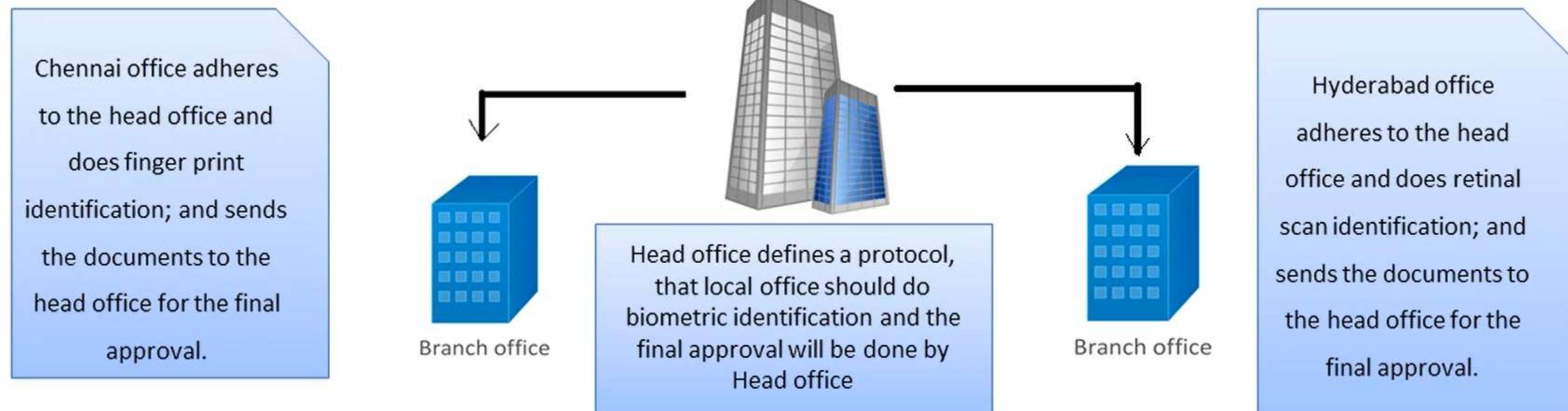
An instance cannot be created for an abstract class.

An abstract class can have both concrete methods and abstract methods.

Hence 0 – 100% abstraction is achieved through abstract class concept.

Abstraction- Real world scenario

Let us consider the Aadhaar card issuing process. The head office for the final approval is present in Bangalore and we have two local offices one at Chennai and the other at Hyderabad. Aadhaar card issuing involves 2 steps. Biometric identification and the other is final approval and issuing the acknowledgement. Chennai does the biometric identification through the finger print and Hyderabad does it through retinal scan. So the Bangalore head office has decided to delegate the biometric identification to both the local offices and take care of the final approval and issuing the acknowledgement



OOP Concept - Abstraction In Java

There will be information on what a method will do instead of how it does

```
abstract class BangaloreHeadOffice {  
    abstract public void performBiometricIdentification();  
  
    public void approveAadharCard() {  
        //some code  
    }  
}  
  
class ChennaiOffice extends BangaloreHeadOffice {  
    public void performBiometricIdentification() {  
        System.out.println("Use Finger print");  
    }  
}
```

```
class HyderabadOffice extends BangaloreHeadOffice {  
    public void performBiometricIdentification()  
    {  
        System.out.println("Use  
        retinal scan");  
    }  
}
```

Here the Bangalore head office knows what is to be done - performBiometricIdentification. But how it is done is not known. Hence here it is abstract.

OOP Concept - Abstraction In Java

Another Example

```
abstract public class Employee {  
    //Atributes, Constructor, Getter, Setter  
    abstract public float calculateSalary();  
  
    public void display() {  
        //some code  
    }  
}
```

```
public class PermanentEmployee  
    extends Employee {  
    private float salary;  
    private float incentive;  
  
    public float calculateSalary() {  
        //sum of salary and incentive  
    }  
}
```

Here the Employee class has both abstract and concrete methods.

This class knows what the calculateSalary method should do – return the total salary. But how it is done is unknown.

Summary

- Abstract methods
- Abstract class
- Inherit an abstract class
- Abstraction achieved through Abstract class



INTERFACES



In this module you will learn

- Interface and its usage
- Relate a class to interfaces
- Relate interface to interface
- default methods in Interface
- static methods in Interface
- 100% Abstraction achieved through interface



Interfaces

In software engineering, there are scenarios where disparate groups of developers agree to a contract that describes what is expected from the product.

Each group writes their code to meet the contract not bothering how other groups code.

Interfaces are such contracts.

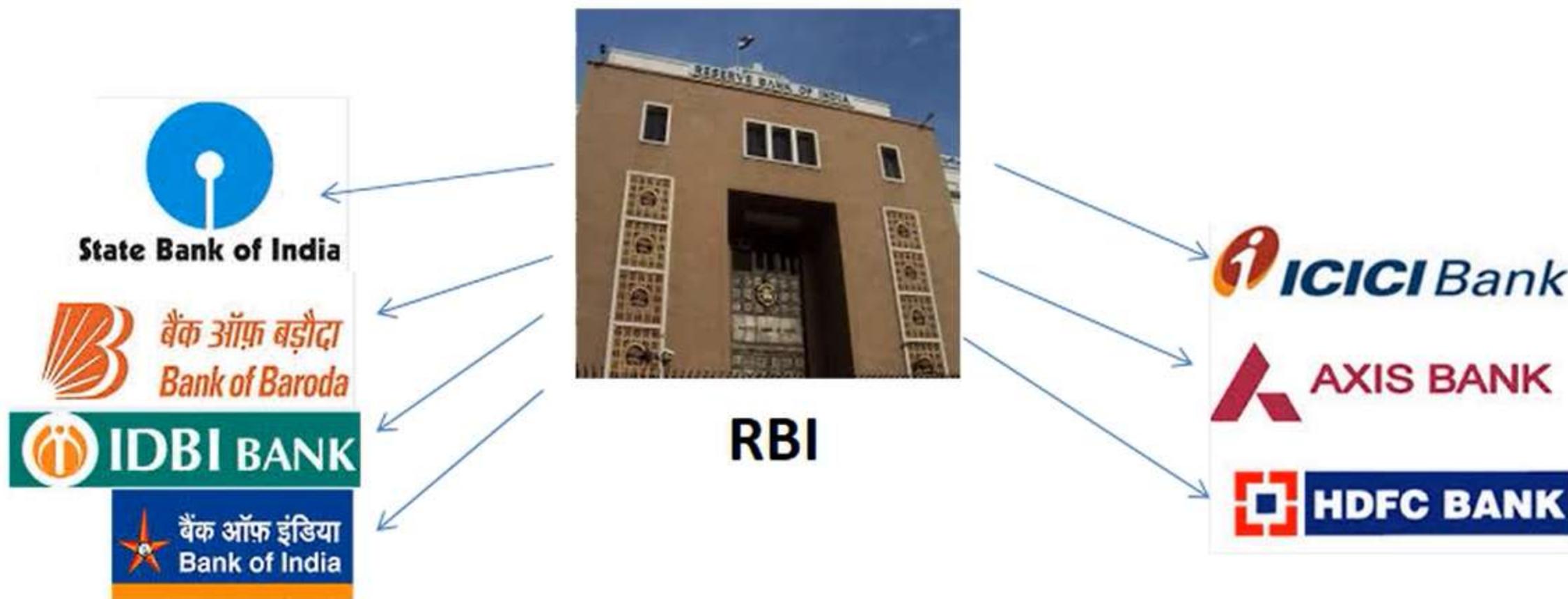
Interfaces

RBI fixes the base interest for Home Loan.

All banks calculate the interest for home loan based on some specific factors of theirs.

Here RBI acts as an interface with the method calculateHomeLoanInterest as abstract.

All banks implement this interface and give implementation for this abstract method.



An Example

Assume a futuristic society people, where rely on robotic cars for transport .



The engineers in Automobile Industry develop software that operates the car to perform operations like stop, start, accelerate, turn left, etc.,



Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS position data and wireless transmission of traffic conditions and use that information to drive the car.



The automobile manufacturer issues a contract that describes the methods to be implemented to make the car move.



First the industrial group is not bothered about *how* the other group's software is implemented. Each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

An interface is a “contract” which an implementing class must adhere to

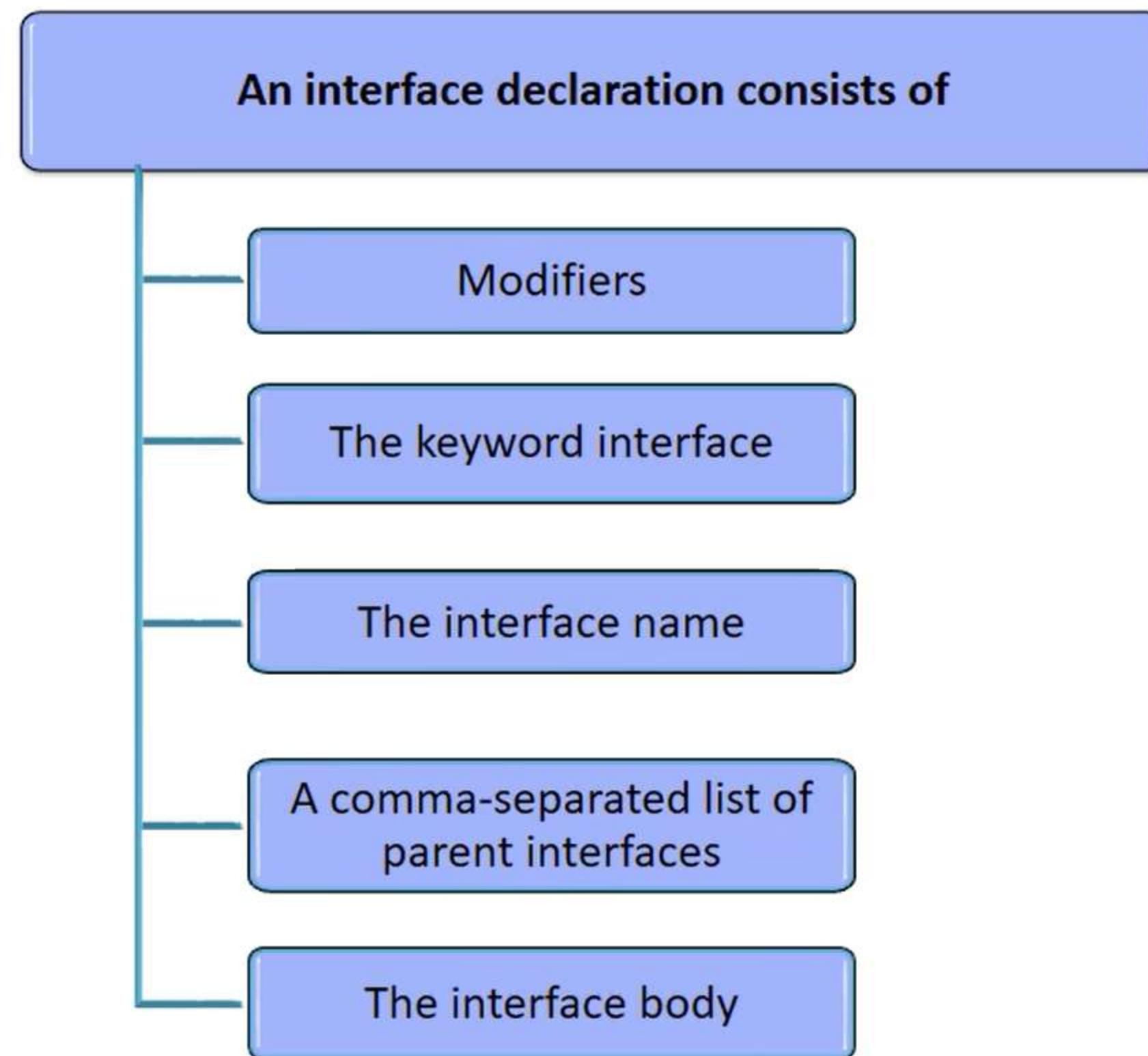
Interface is a 100% abstract class

Interface can contain variables and methods.

- All methods that are just declared in an interface are by default public abstract
- All variables in an interface are implicitly public static final

Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces

Defining an Interface



Defining an Interface

```
public interface GroupedInterface extends Interface1, Interface2, Interface3{  
  
    // Constant Declarations  
    //base of natural logarithms  
    double E = 2.718282;          //implicitly public static final  
  
    //Method signatures - implicitly all methods that are just declared are public //and abstract  
    void operation1(int x,double d);  
    int operation2(String s);  
}
```

Note :

One interface can extend
many interfaces.

Implementing the Interface

To declare a class that implements an interface, include an implements clause in the class declaration.

Class Implementing the interface need not have an “is-a “ relationship.

A class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

By convention, the implements clause follows the extends clause, if there is one.

Create an Interface

Use the keyword **interface** instead of class

```
public interface Shape {  
  
    //Methods just declared without implementation,  
    //are by default public abstract  
    float calculateArea();  
    float calculatePerimeter();  
}
```

Implement the Interface

An interface will be implemented by a class using the keyword “implements”

This class should implement all the methods in the interface. Otherwise, that class becomes abstract

Implementing class can have its own methods

Implementing class can also extend only one super class or abstract class

Implement the Interface

Example:

```
public class Square implements Shape {  
    private float side;  
    public Square(float side) {  
        this.side = side;  
    }  
    public float calculateArea() {  
        float area=side * side;  
        return area;  
    }  
    public float calculatePerimeter() {  
        float perimeter=4*side;  
        return perimeter;  
    }  
}
```

```
public class Rectangle implements Shape {  
    private float length;  
    private float breadth;  
    public Rectangle(float length, float breadth) {  
        this.length = length;  
        this.breadth = breadth; }  
    public float calculateArea() {  
        float area=length * breadth;  
        return area; }  
    public float calculatePerimeter() {  
        float perimeter=2*(length+breadth);  
        return perimeter;  
    }  
}
```

Using an Interface as a Type

When a new interface is defined, a new reference data type is defined

Interface cannot be instantiated, but a reference of interface can be created

If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

```
public class ShapeUtility {  
    public static void main(String a[]) {  
        Shape s=new Square(10);  
        System.out.println("Area of square "+s.calculateArea());  
        System.out.println("Perimeter of square "+s.calculatePerimeter());  
        s=new Rectangle(2.5f,12);  
        System.out.println("Area of Rectangle "+s.calculateArea());  
        System.out.println("Perimeter of Rectangle "+s.calculatePerimeter());  
    } } 
```

Implement the Interface

- One class can implement multiple interfaces.
- If so, class should implement abstract methods in all the interface it implements.
- Example

```
interface BlueTooth {  
    void sendFile();  
    void receiveFile();  
}  
  
interface Phone {  
    void makeCall();  
    void attendCall();  
}  
  
interface WhatsApp {  
    void sendMessage();  
    void receiveMessage();  
}
```

```
class SmartPhone implements BlueTooth,  
                           Phone, WhatsApp {  
    //Implement all methods in the interface  
}
```

```
class iPhone extends SmartPhone  
    implements BlueTooth, WhatsApp {  
}
```

Extending the Interface

Interfaces can be extended to include additional contracts

```
public interface Loan{  
  
    void operation1(int x,double d);  
    int operation2(String s);  
}
```

```
public interface HomeLoan extends Loan{  
  
    void operation3();  
    float operation4(float f);  
}
```

Extending the Interface

One interface can extend multiple interfaces

```
interface BlueTooth {
    void sendFile();
    void receiveFile();
}

interface WhatsApp {
    void sendMessage();
    void receiveMessage();
}

interface Mobile extends BlueTooth, WhatsApp {
    void makeCall();
    void attendCall();
}
```

Any class that implements Mobile should implement the methods in Mobile, BlueTooth and WhatsApp

Default Methods In Interface

Interfaces can have default method - Java 8 Feature

Default Methods are

- Methods defined in an interface with the default keyword
- Must be implemented

Classes that implement that interface can

- either override this default method
- or use the method as such

This leads to backward compatibility

Default Methods In Interface

```
public interface Insurance
{
    void applyInsurance();
    void approveInsurance();
    double claimInsurance();

    public default void display()
    {
        System.out.println("In default
                           method display");
    }
}
```

```
public class Car implements Insurance
{
    public void applyInsurance() { }
    public void approveInsurance() { }
    public double claimInsurance() { }

    public static void main(String args[])
    {
        Car c=new Car();
        c.display(); //similar to invoking
                    // Insurance.display()
    }
}
```

Default Methods In Interface



Default Methods In Interface

```
public class Car implements Vehicle, Insurance //DOES NOT COMPILE
{
    public void applyInsurance() { }
    public void approveInsurance() { }
    public double claimInsurance() { }
    public void getColor() { }
    public void drive() { }

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

Default Methods In Interface

```
public class Car implements Vehicle, Insurance // COMPILES
{
    public void applyInsurance() { }
    public void approveInsurance() { }
    public double claimInsurance() { }
    public void getColor() { }
    public void drive() { }

    //Overrides
    public void display() {
        System.out.println("In Car display method");
    }
    public static void main(String args[]) {
        Car c=new Car();
        c.display();
    }
}
```

Default Methods In Interface

Rules for default methods in an interface

A default method can be declared only within an interface.

A default method must be marked with the default keyword and should have a method body.

A default method is not assumed to be static, final, or abstract.

It may be used or overridden by a class that implements the interface.

A default method is public by default.

Code will not compile if it is marked as private or protected.

Static Methods In Interface

Interface can have static methods

These static methods should be implemented

```
public interface StudentService{  
    public Student getStudent(int studentId);  
  
    static public int getStudentCount()  
    {  
        //logic to find the count  
        return 0;  
    }  
}
```

Static Methods In Interface

The class implementing an interface can have the same static method as in the interface with different implementation. It is not overriding

```
public class Hosteller implements StudentService{
    public void getStudentInfo(int studentId) {
        //search and return the student info
    }
    public static int getStudentCount() {
        // return count of student
        return 0;
    }
}
```

Static Methods In Interface

A static method in an interface can be invoked only by using the interface name and not by using the reference of the interface

```
public class StudentTest {  
    public static void main(String[] args) {  
        // Legal - calls the static method in the CustomerService interface  
        int custCount = StudentService.getStudentCount();  
  
        // Legal - calls the static method in the MidwestCustomerService interface  
        int custCount2 = Hosteller.getStudentCount();  
        StudentService ss = new Hosteller();  
  
        // Cannot access a static method defined in an interface using instance variable  
  
        ss.getStudentCount(); //Compilation error.  
        System.out.println(StudentService.getStudentCount());//Compiles  
        System.out.println(Hosteller.getStudentCount());//Compiles  
    }  
}
```

OOP Concept - Abstraction In Java

In Java, 100% abstraction is achieved through Interface

Unlike abstract class, no attribute or constructor or instance methods are there in an interface.

All methods declared in it are abstract and public.

An instance cannot be created for an interface.

Hence 100% abstraction is achieved through interface in java.

OOP Concept - Abstraction In Java

Example

```
public interface Loan
{
    public double issueLoan();
    public void repayLoan(double amt);
}
```

```
public class Employee implements Loan {
    //Attributes, constructor, methods

    public double issueLoan() {
        //some logic
    }

    public void repayLoan(double amt) {
        //some logic
    }
}
```

- Loan interface knows what the methods issueLoan and repayLoan should do. But how it is done is implemented by the class that implements the interface.
- As all methods are abstract, 100% abstraction is achieved by interface.

Summary

- Interface and its usage
- Relate a class to interfaces
- Relate interface to interface
- default methods in Interface
- static methods in Interface
- 100% Abstraction achieved through interface



LAMBDA EXPRESSION



In this module you will learn

- Functional Interface
- Lambda Expression and its usage
- Implement Lambda Expressions
- Built-in Functional Interface
- Map Built-in Functional Interface to Lambda



Functional Interfaces

Functional interfaces are interfaces which have only one abstract method in it.

This is a Java 8 feature.

These interfaces are also called Single Abstract Method interfaces (SAM Interfaces)

These interfaces can be annotated as `@FunctionalInterface`, which is optional.

`@FunctionalInterface` can be used for compiler level errors.

A Functional interface can have only one abstract method but it can have default method and static method

Functional Interfaces

In a functional interface, the single abstract method gets matched up to the lambda expression if it has a compatible method signature.

For parameters and return type, generics can also be used.

Few Functional interface before Java 8

```
public interface Comparator <T> {  
    public int compare(T o1, T o2);  
}  
  
public interface Runnable {  
    public void run();  
}
```

Functional Interfaces - Example

```
@FunctionalInterface
interface MaxFinder {
    public int maximum(int num1,int num1);
}
public class MaxFinderImpl implements MaxFinder {
    public int maximum(int num1, int num1) {
        return num1 > num2 ? num1 : num2;
    }
    public static void main(String a[])  {
        MaxFinder max= new MaxFinderImpl();
        int res = max.maximum(20,30);
    }
}
```

Functional Interfaces –Example

Same Example Using Lambda Expression

```
@FunctionalInterface  
interface MaxFinder  
{  
    public int maximum(int num1,int num2);  
}  
public class MaxFinderImpl  
{  
    public static void main(String a[]) {  
        MaxFinder max= (num1,num2) -> num1 > num2 ? num1 : num2;  
        int res = max.maximum(20,30);  
    }  
}
```

Lambda Expression

Why Lambdas?

Enables functional programming. Lambda expressions are used to pass Code as Data.

Avoid boilerplate code - **boilerplate code** or **boilerplate** refers to sections of **code** that have to be included in many places with little or no alteration

Enables support for parallel processing

Lambda expression avoids unwanted anonymous inner class

Syntax

(argument-list) -> {body}

argument-list: It can be empty or non-empty as well.

arrow-token: It is used to link arguments-list and body of expression.

body: It contains expressions and statements for lambda expression.

Lambda Expression

- Using lambda expression we can assign a block of code to a variable and we can pass the variable around the application where the code is needed.
- For example

＊＊＊

```
aBlockOfCode = public void greet() {  
    System.out.println("Hello World");  
}
```

Lambda Expression

- ❖ After removing the unwanted things the code is left with

```
aBlockOfCode = ()-> {  
    System.out.println("Hello World"); }
```

- ❖ '{}' is optional if the block of code is just a line.

```
aBlockOfCode = ()-> System.out.println("Hello World");
```

- ❖ Note : What is the type of aBlockOfCode?

Lambda Expression

```
interface MyLambda
{
    void display();
}

MyLambda aBlockOfCode = () ->
    System.out.println("Hello World");

aBlockOfCode.display(); // display 'Hello World'
```

Lambda Expression

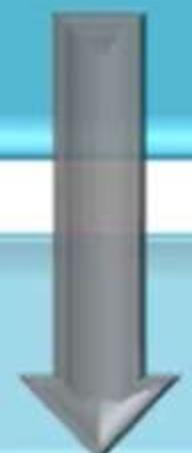
- The variable assigned with Lambda expression can be passed around the application in order reuse the code. Or, we use inline lambda expression.
- Ways of calling a function ***firstLambda***
 - `firstLambda(aBlockofCode);`
 - `firstLambda(()-> System.out.println("Hello World")); //using inline lambda expression.`
- Signature of '***firstLambda***' method.

```
return_type firstLambda(MyLambda variableName)
{
}
```

Example – Lambda expression

```
squareNumberFunction = (int a) -> return a*a;
```

```
squareNumberFunction = (int a) -> a*a;
```



```
interface Mylambda2
{
    int squareFunction(int a);
}

• Mylambda2 squareNumberFunction = (a) -> a*a;
```

Example – Lambda expression

```
Interface MyLambda3 // define functional interface
{
    int divideNumber(int a, int b);
}

//defining lambda expression
MyLambda3 numberDivideFunction = ( x, y ) -> {
    if(y==0) return 0;
    return x/y;
}

//calling method using lambda variable
numberDivideFunction.divideNumber(20,5);
```

Lambda Expression

Before Lambda

```
interface MathOperation{  
    public int add(int a,int b);  
}  
  
public class Interfacedemo {  
    public static void main(String a[]) {  
        MathOperation math=new MathOperation() {  
            @Override  
            public int add(int a, int b) {  
                return a+b;  
            }  
        };  
        System.out.println("The result is "+math.add(5, 10));  
    }  
}
```

After Lambda

```
interface MathOperation {  
    public int add(int a,int b);  
}  
  
public class Interfacedemo {  
    public static void main(String arg[]){  
        MathOperation sum = (a,b)->a+b;  
        System.out.println(sum.add(20, 15));  
    }  
}
```

Functional Interface and Lambda

Functional Method	Lambda Expression
int fun(int a)	(int a) -> a+10;
int fun(int a,int b)	(int a,int b) -> a+b;
int fun(int a,int b)	(int a,int b) -> { int min = a>b ? a : b; return min; }
String fun()	() -> "Hello World"
void fun()	() -> System.out.println("Hello World");
int fun(String str)	(String s) -> s.length() Or S -> s.length()

Built-in Functional Interfaces



Interface Name	Method Name	Arguments	Returns	Example
Predicate<T>	test	T	boolean	Has this album been released yet ?
Consumer<T>	accept	T	void	Printing a value
Function<T,R>	apply	T	R	Get the value of an attribute from an Object
Supplier<T>	get	None	T	A factory method
UnaryOperator<T>	apply	T	T	Logical not (!)
BinaryOperator<T>	apply	T , T	T	Multiply two numbers

Built – in functional interfaces

PREDICATE - boolean test(T t)

Predicate is a Functional Interface that can be used anywhere you need to evaluate a boolean condition. The test method evaluates a predicate on the given argument that results in true or false..

```
Predicate<String> panCardValid = str ->
    str.matches("[A-Z]{5}[0- 9]{4}[A-Z]{1}");
String panCardNo = "ANCZE1234Q";

boolean result = panCardValid.test(panCardNo);
System.out.println(result);

if("Coasta Ri9955".matches("[A-Za-z ]+")){valid = true;}
System.out.println(valid);
```

FUNCTION – R apply(T t)

Function is a Functional Interface defined with generic types T and R. The method apply that takes an argument of type T and returns a result of type R.

```
Function<Integer,Integer> function = (x) -> x*10;

System.out.println(function.apply(15)); //150
System.out.println(function.apply(20)); //200
```

Built – in functional interfaces

CONSUMER : void accept(T t)

Consumer is a Functional Interface that has a method accept, that takes a single input as argument and returns no result. It performs some operation on the passed argument.

```
Consumer consumer = str -> System.out.println(str);
consumer.accept("Welcome");

Student studentObj = new Student(101, "Peter");
consumer.accept(studentObj);
```

SUPPLIER : T get()

Supplier is a Functional Interface that does the opposite of the Consumer. It takes no arguments but returns some value of type T.

```
Supplier<String> supplier = ()-> {
    Scanner sc=new Scanner(System.in);
    String name=sc.next();
    return name;
} ;

System.out.println(supplier.get());
```

Supplier functional interfaces

```
public class Student {  
  
    private int id;  
    private String name;  
    private String gender;  
    private int age;  
  
    public Student(int id, String name, String gender, int age) {  
        this.id = id;  
        this.name = name;  
        this.gender = gender;  
        this.age = age;  
    }  
    //Assume 4 argument Constructor and Getters and Setters  
    //are written  
  
    @Override  
    public String toString() {  
        return "Student ID "+id+", Name "+name+", Gender "+  
               gender+", Age "+age;  
    }  
}
```

```
public class FunctionalInterfaceDemo {  
  
    public static void main(String[] args) {  
  
        Supplier<Student> supplier = () -> {  
            Scanner sc = new Scanner(System.in);  
            int id=sc.nextInt();  
            String name=sc.next();  
            String gender=sc.next();  
            int age=sc.nextInt();  
  
            return new Student(id,name,gender,age);  
        };  
  
        Student[] studentList = new Student[3];  
        for(int i=0; i<3; i++){  
            System.out.println("Enter Student "+(i+1)+" Details");  
  
            studentList[i] = supplier.get();  
        }  
  
        for(Student student : studentList)  
            System.out.println(student);  
    }  
}
```

Supplier functional interfaces

```
public class FunctionalInterfaceDemo {  
  
    public static void main(String[] args) {  
  
        Supplier<Student> supplier = () -> {  
            Scanner sc = new Scanner(System.in);  
            int id=sc.nextInt();  
            String name=sc.next();  
            String gender=sc.next();  
            int age=sc.nextInt();  
  
            return new Student(id,name,gender,age);  
        };  
  
        Student[] studentList = new Student[3];  
        for(int i=0; i<3; i++){  
            System.out.println("Enter Student "+(i+1)+" Details");  
  
            studentList[i] = supplier.get();  
        }  
  
        for(Student student : studentList)  
            System.out.println(student);  
    }  
}
```

OUTPUT

Enter Student 1 Details

101

Peter

Male

25

Enter Student 2 Details

102

Pinky

Female

18

Enter Student 3 Details

103

Raghu

Male

19

Student ID 101, Name Peter , Gender Male, Age 25

Student ID 102, Name Pinky , Gender Female, Age 18

Student ID 103, Name Raghu , Gender Male, Age 19

Consumer functional interfaces

```
public class FunctionalInterfaceDemo {  
  
    public static void main(String[] args) {  
  
        //Continuation of previous main method  
  
        System.out.println("Student Details using Consumer");  
  
        //Using Consumer to print the Student object  
        Consumer consumer = (value) -> System.out.println(value);  
  
        for(Student studentObj : studentList) {  
  
            consumer.accept(studentObj);  
        }  
    }  
}
```

OUTPUT

```
Enter Student 1 Details  
101  
Peter  
Male  
25  
Enter Student 2 Details  
102  
Pinky  
Female  
18  
Enter Student 3 Details  
103  
Raghu  
Male  
19  
Student Details using Consumer  
Student ID 101, Name Peter , Gender Male, Age 25  
Student ID 102, Name Pinky , Gender Female, Age 18  
Student ID 103, Name Raghu , Gender Male, Age 19
```

Predicate functional interfaces

```
public class FunctionalInterfaceDemo {  
  
    public static void main(String[] args) {  
  
        //Continuation of previous main method  
        System.out.println("Student of age below 21");  
  
        Predicate<Student> predicate = (student)->student.getAge()<=20;  
  
        for(Student studentObj : studentList){  
  
            if(predicate.test(studentObj))  
                System.out.println(studentObj);  
        }  
    }  
}
```

OUTPUT

Enter Student 1 Details

101

Peter

Male

25

Enter Student 2 Details

102

Pinky

Female

18

Enter Student 3 Details

103

Raghu

Male

19

Student of age below 21

Student ID 102, Name Pinky , Gender Female, Age 18

Student ID 103, Name Raghu , Gender Male, Age 19

Function Functional Interface

```
public class FunctionalInterfaceDemo {  
  
    public static void main(String[] args) {  
  
        //Continuation of previous main method  
  
        System.out.println("Name of students whose age is less than "  
                           + "or equal to 20");  
        //Using Function  
  
        Function<Student, String> function = (student)->  
                                         student.getName();  
  
        for(Student student : studentList) {  
            System.out.println(function.apply(student));  
        }  
    }  
}
```

OUTPUT

```
Enter Student 1 Details  
101  
Peter  
Male  
25  
Enter Student 2 Details  
102  
Pinky  
Female  
18  
Enter Student 3 Details  
103  
Raghu  
Male  
19  
  
Name of students whose age is less than or equal to 20  
Pinky  
Raghu
```

Example - LAMBDA

```
interface Calculator
{
    public float calculate(float a, float b);
}
```

```
public class Main
{
    public static Calculator performCalculation(char c){
        Calculator calc=null;
        switch(c)
        {
            case '+': calc = (a,b) -> a+b;
            case '-': calc = (a,b) -> a-b;
            case '*': calc = (a,b) -> a*b;
            case '/': calc = (a,b) -> a/b;
        }
        return calc;
    }
}
```

```
//Main class
public static void main(String args[]){
    Calculator c = (a,b) -> a+b; // Directly

    //By calling the function performCalculation

    Calculator add = performCalculation('+');
    Calculator diff = performCalculation('-');
    Calculator pdt = performCalculation('*');
    float f1 = c.calculate(5,6);

    float x=4.5f; float y=2.5f;
    float f2 = add.calculate(x,y);
    float f3 = c.calculate(x,y);
    float f4 = c.calculate(x,y);
    System.out.println(f1+ " "+f2+ " "+f3+ " "+f4);
    //11.0 1.8 7.0 7.0
}
```

Summary

- Functional Interface
- Lambda Expression and its usage
- Implement Lambda Expressions
- Built-in Functional Interfaces
- Map Built-in Functional Interface to Lambda



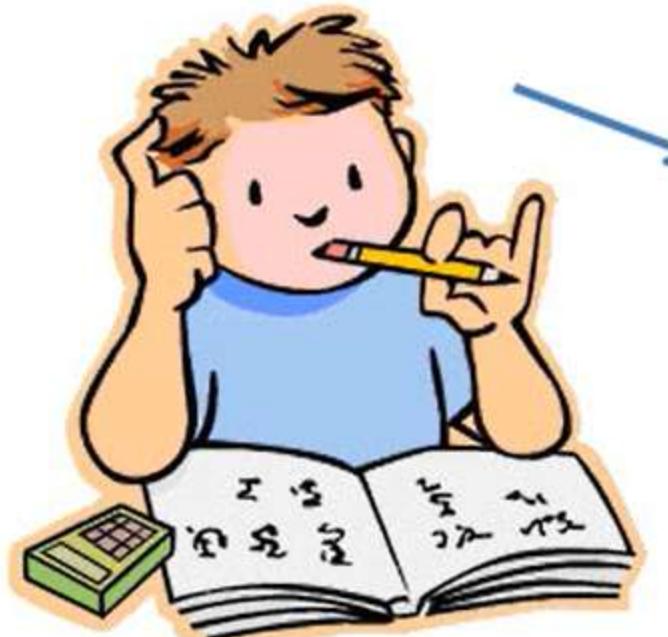
Exception Handling

Exception Handling

Part 1



Error Handling



Exception occurs during execution

Student Progress Report

Student Name:	LEGEND:	O - Outstanding	S - Satisfactory	N - Needs Improvement	Communicated to caregiver via:
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> Conference
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> Mail
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> Telephone
Academics					
Math	Conduct				
Reading	Courteous				
Language	Respects others				
Sporting	Respects others' property				
Social Studies	Obedience				
Science/Health	Refains from excessive talking				
Handwriting	Refains from disrupting others				
Art/Music	Comments				
Physical Edu.					
Computer					
Teacher signature _____ Date _____ Parent signature (Please SIGN & RETURN) _____					

Exception object containing information about it is created and thrown by the run time system



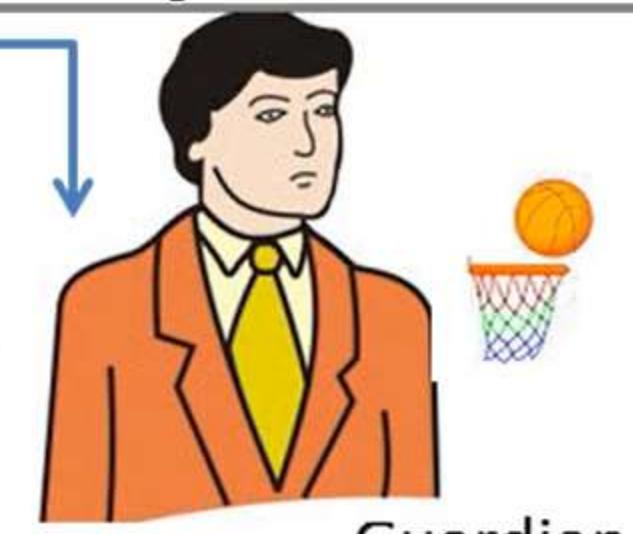
Runtime checks if principal can handle the exception. If not, the exception is thrown to the guardian.



Parents

In Java, Handlers handle the exception

Exception is thrown to the next method in the invocation chain until it is handled



Guardian

Runtime checks if guardian can handle the exception. If not, the exception is thrown to the parent

In this module you will learn

- Exception
- Exception Types
- Exception Hierarchy
- Try-catch-finally
- Try with Multi catch



Error Handling

Applications will encounter errors while executing

Reliable applications should handle errors as gracefully as possible

Errors

- Should be the “exception” and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
 - Databases become unreachable
 - Hard drive fails

Exceptions

What is an Exception ?

- Exceptional event - typically an abnormality or error that occurs during runtime
- Cause the normal flow of a program to be disrupted

Examples

- Divide by zero errors
- Accessing the elements of an array beyond its range
- Invalid input
- Opening a non-existent file

Advantages of Exceptions

Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program

Propagating Errors Up the Call Stack

If the `readFile` method is the fourth method in a series of nested method calls made by the main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`

Grouping and Differentiating Error Types

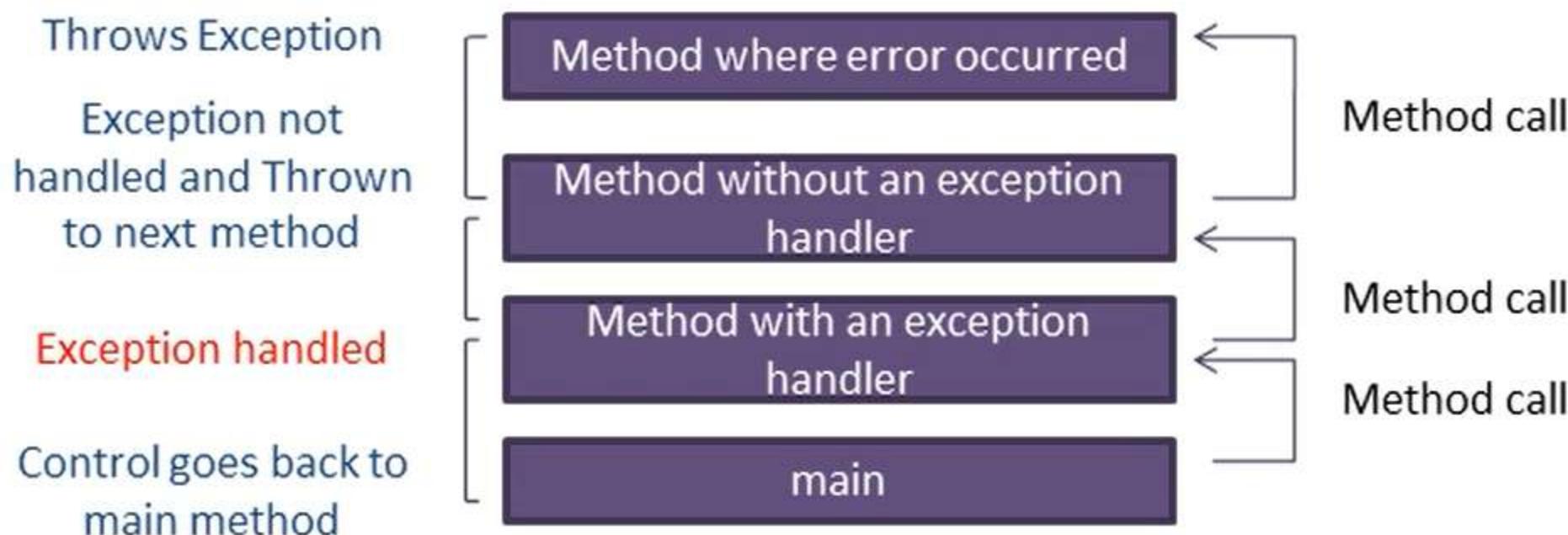
An example of a group of related exception classes in the Java platform are those defined in `java.io` — `IOException` and its descendants

Exception

When an exception occurs within a method, the method typically creates an exception object and hands it off to the runtime system

- Creating an exception object and handing it to the runtime system is called “throwing an exception”
- Exception object has information about the error, its type and state of the program when the error occurred

The runtime system searches the call stack for the method that has the block of code that handles the exception



Exception

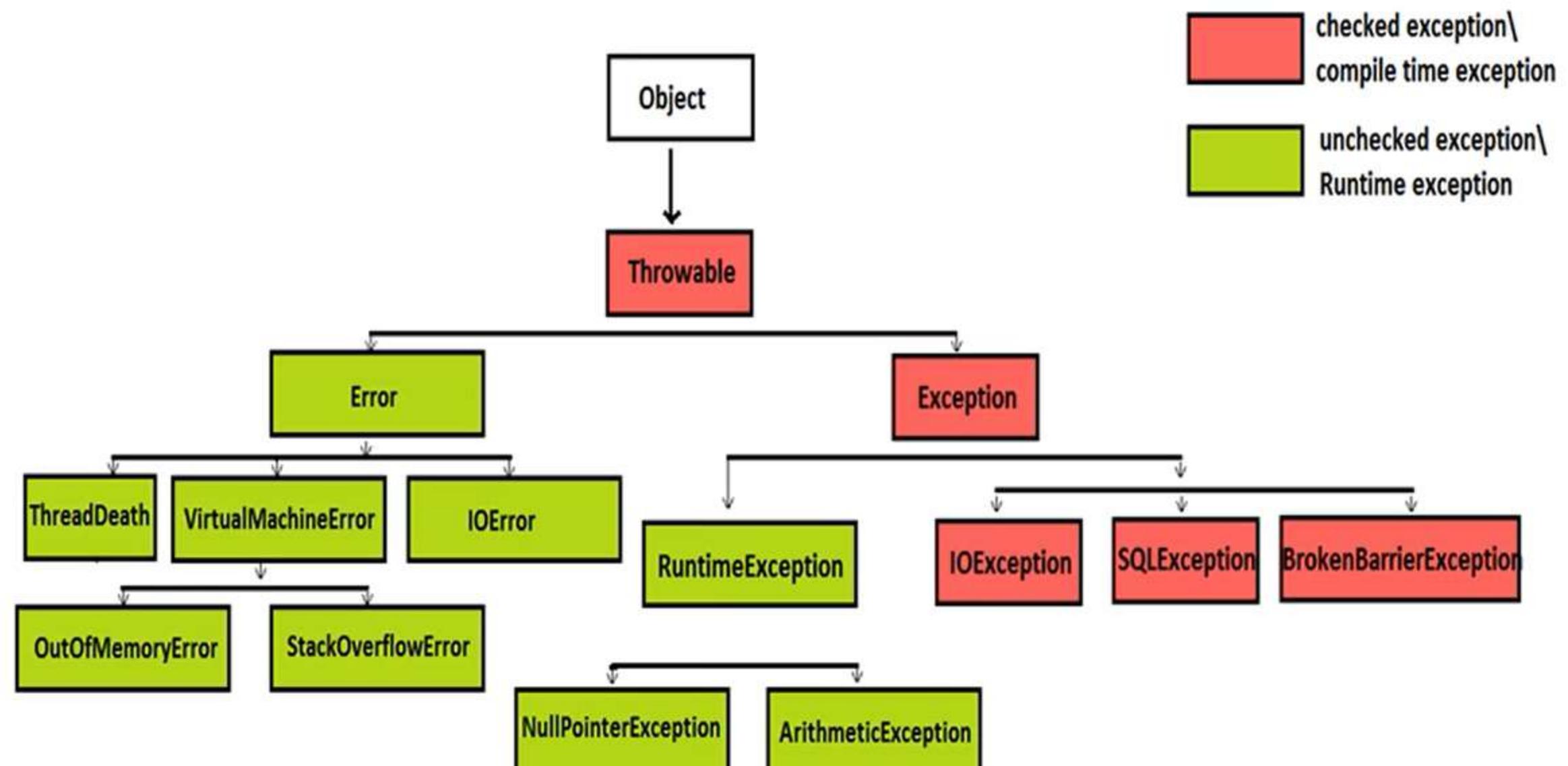
When an appropriate handler is found, the runtime system passes the exception to the handler

An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler

The exception handler chosen is said to catch the exception.

If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates and uses the default exception handler.

Exception Hierarchy



The Error class

Errors

- Used by the Java run-time system to handle errors occurring in the run-time environment
- Generally beyond the control of user programs
- Examples
 - Out of memory errors
 - Hard disk crash

Checked and Unchecked Exception

Checked exception

- Java compiler checks if the program either catches or lists the occurring checked exception
- Checked exceptions should be explicitly handled or properly propagated
- If not, compiler error will occur

Unchecked exceptions

- Not subject to compile-time checking for exception handling
- Built-in unchecked exception classes
 - Error
 - RuntimeException and their subclasses
- Handling all these exceptions may make the program cluttered and may become difficult to manage

Exception Example

```
public class NumberFormatExceptionDemo{  
    public static void main(String[] args) {  
        int sum=0;  
        for(String a : args){  
            sum +=Integer.parseInt(a); }  
        System.out.println("Sum is "+sum);  
    }  
}
```

Output :

```
java NumberFormatExceptionDemo 8 12 1 4 2  
Sum is 27
```

```
java NumberFormatExceptionDemo 8 12 four 3.0 2  
Exception in thread "main" java.lang.NumberFormatException: For input string: "four"  
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
at java.lang.Integer.parseInt(Integer.java:580)  
at java.lang.Integer.parseInt(Integer.java:615)  
at modelNumberFormatExceptionDemo.main(NumberFormatExceptionDemo.java:7)
```

Catching and Handling Exceptions

Exception can be handled using Exception Handler

Exception Handler is the block of code that can process the exception object.

Exception can be handled using

- try - catch - finally
- throws

try Block

try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block

```
try {  
    code  
}  
catch and finally blocks . . .
```

catch Block

Associate exception handlers with a try block by providing one or more catch blocks directly after the try block

Each catch block is an exception handler and handles the type of exception indicated by its argument

The argument type, declares the type of exception that the handler can handle and must be a class that inherits from the Throwable class

This catch block

- can contain the code to print the exception and also code to recover from the exception
- gets executed only if an exception object is thrown from its corresponding try block

No code can be between the try and the catch blocks

```
try {  
    } catch (ExceptionType name) {  
    } catch (ExceptionType name) {  
    }
```

The try-catch Example

```
public class NumberFormatExceptionDemo{  
    public static void main(String[] args) {  
        int sum=0;  
        for(String a : args) {  
            try{  
                sum +=Integer.parseInt(a);  
            }  
            catch(NumberFormatException e) {  
                //a not an int  
                System.err.println(a+" is not an integer ");  
            }  
        }  
        System.out.println("Sum is "+sum);  
    }  
}
```

```
java NumberFormatExceptionDemo 8 12 four 3.0 2
```

Output :

four is not an integer

3.0 is not an integer

Sum is 22

finally Block

The finally block always executes whether or not an exception occurs

It allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

Putting cleanup code in finally block is always a good practice

This block is optional

A try block can have multiple catch blocks , but only one finally block

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

finally Block

```
public class DivideByZeroException {  
  
    public static void main(String[] args) {  
        int result = divide(100,0); // Line 2  
        System.out.println("result :" + result);  
    }  
  
    public static int divide(int totalSum, int totalNumber)  
    {  
        int quotient = -1;  
        System.out.println("Computing Division.");  
        try{  
            quotient = totalSum/totalNumber;  
        }  
        catch(ArithmaticException e) {  
            System.out.println("Exception :" +  
                               e.getMessage());  
        }  
    }  
}
```

```
    finally {  
        if(quotient != -1) {  
            System.out.println("Finally Block Executes");  
            System.out.println("Result :" + quotient);  
        }  
        else {  
            System.out.println("Finally Block Executes.  
Exception Occurred");  
        }  
    }  
    return quotient;  
}  
}
```

Output :
Computing Division.
Exception : / by zero
Finally Block Executes. Exception Occurred
result : -1

try with Multiple catch block

If a try block can throw multiple exceptions it can be handled using multiple catch blocks

If a try has multiple catch blocks, it should be ordered from subclass to super class

```
try {  
    // code that might throw one or more exceptions  
  
} catch (MyException e1) {  
    // code to execute if a MyException exception is thrown  
  
} catch (MyOtherException e2) {  
    // code to execute if a MyOtherException exception is thrown  
  
} catch (Exception e3) {  
    // code to execute if any other exception is thrown  
}
```

try with multiple catch Example

```
public class MultipleCatchExample {  
    public static void main (String args[]) {  
        int array[]={20,10,30};  
        int num1=15,num2=0;  
        int result=0;  
        try {  
            result = num1/num2;  
            System.out.println("The result is" +result);  
            for(int index=0;index<3;index++) {  
                System.out.println("The value of array are" +array[index]);  
            }  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Error. Array is out of Bounds");  
        }  
        catch (ArithmaticException e) {  
            System.out.println("Can't be divided by Zero");  
        }  
    }  
}
```

Output:
Can't be divided by Zero

In the example,
the first catch
block is skipped
and the second
catch block
handles the
error

Summary

- Exception
- Exception Types
- Exception Hierarchy
- Try-catch-finally
- Try with Multi catch



Exception Propagation and User Defined Exceptions



In this module you will learn

- Exception Propagation
- User defined Exceptions
- Try with resources



Nested try

We can have nested try and catch blocks(a try block inside another try block)

If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers

This continues until one of the catch statements succeeds, or until all of the nested try statements are done in

If none of the catch statements match, then the Java runtime system will handle the exception.

Nested try

```
import java.io.*;  
public class NestedTry{  
    public static void main (String args[])throws IOException {  
        int num1=2,num2=0,res=0;  
        try{  
            FileInputStream fis=null;  
            fis = new FileInputStream (new File (args[0]));  
            try{  
                res=num1/num2;  
                System.out.println("The result is"+res);  
            }  
            catch(ArithmetricException e){  
                System.out.println("divided by Zero");  
            }  
        }  
    }  
}
```

```
        catch (FileNotFoundException e){  
            System.out.println("File not found!");  
        }  
        catch(ArrayIndexOutOfBoundsException e){  
            System.out.println("Array index is Out  
                            of bound! Argument required");  
        }  
        catch(Exception e){  
            System.out.println("Error.." +e);  
        }  
    }  
}
```

Nested try

If the above program is executed without giving any file name, the output will be “Array index is Out of bound! Argument required ” (available in outer try)

If the above program is executed by giving a file name that does not exist, the output will be “File not found! ” (available in outer try)

If the above program is executed by giving a file name that exists, the output will be “divided by Zero ” (available in inner try).

Call Stack Mechanism

If an exception is not handled in the current try-catch block, it is thrown to the caller of that method

If the exception gets back to the main method and is not handled there, the program is terminated abnormally

A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred

The following code shows how to call the getStackTrace method on the exception object

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

Rules for try-catch-finally

For each try block there can be zero or more catch blocks but only one finally block

The catch blocks and finally block must always appear in conjunction with a try block

A try block must be followed by either at least one catch block or one finally block

The order of the exception handlers in the catch block must be from the most specific exception

throw and throws usage

A program can explicitly throw an exception using the **throw** statement besides the implicit exception thrown. The general format of the **throw** statement is

throw <exception reference>;

The Exception reference must be of type `Throwable` class or one of its subclasses.

throw new ArithmeticException("Division attempt by 0");

A **throws** clause can be used in the method prototype as

```
Method() throws <ExceptionType_1>,..., <ExceptionType_n> {  
    //statements  
}
```

Each `<ExceptionType_i>` can be a checked or unchecked or sometimes even be a custom Exception. The exception type specified in the `throws` clause in the method prototype can be a super class type of the actual exceptions thrown.

throws and throw - Example

```
public class DivideByZeroException {  
  
    public static void main(String[] args) {  
        try{  
            int result = divide(100,10);  
            result = divide(100,0);  
            System.out.println("result :" +result);  
        }  
        catch(ArithmeticException e){  
            System.out.println("Exception :" +  
                e.getMessage());  
        }  
    }  
  
    public static int divide(int totalSum, int totalNumber)  
        throws ArithmeticException  
    {  
        int quotient = -1;  
        System.out.println("Computing Division.");  
    }  
}
```

```
try{  
    if(totalNumber == 0){  
        throw new ArithmeticException("Division attempt  
            by 0");  
    }  
    quotient = totalSum/totalNumber;  
}  
finally{  
    if(quotient != -1){  
        System.out.println("Finally Block Executes");  
        System.out.println("Result :" + quotient);  
    }else{  
        System.out.println("Finally Block Executes. Exception  
            Occurred");  
    }  
}  
return quotient;  
}
```

Handle or Declare Rule

Use the handle or declare rule as follows

Handle the exception by using the try-catch-finally block

Declare that the code causes an exception by using the throws clause

- void trouble() throws IOException { ... }
- void trouble() throws IOException, MyException { ... }

Other Principles

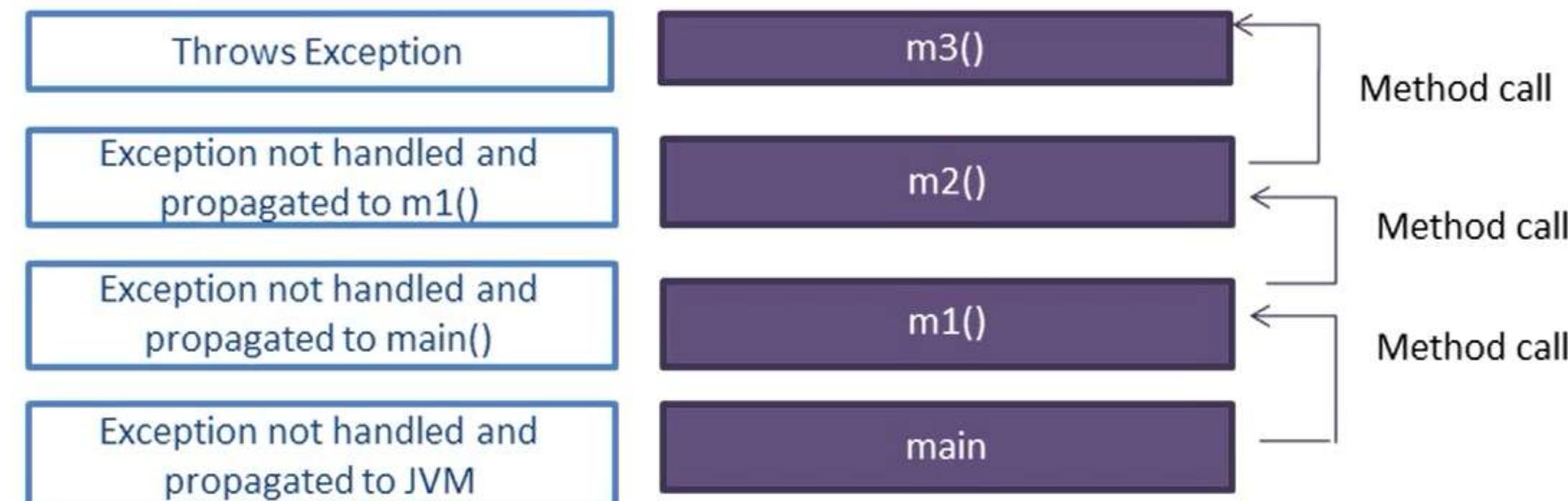
You do not need to declare runtime exceptions or errors

You can choose to handle runtime exceptions

Exception Propagation

Exception propagation is a way of propagating exception from a method to the previous method in the call stack until it is caught.

If uncaught thrown to JVM and program gets terminated



Exception Propagation

By default, unchecked exceptions are automatically propagated

```
public class ExceptionPropogationDemo
{
    public static void main(String a[]) {
        m1(); //Line 4
    }
    public static void m1() {
        m2(); //Line 7
    }
    public static void m2() {
        m3(); //Line 10
    }
    public static void m3() {
        throw new ArithmeticException(); //Line13
    }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmetricException
at
ExceptionPropogationDemo.m3(ExceptionPropogationDemo.java:13)
at
ExceptionPropogationDemo.m2(ExceptionPropogationDemo.java:10)
at ExceptionPropogationDemo.m1(ExceptionPropogationDemo.java:7)
at
ExceptionPropogationDemo.main(ExceptionPropogationDemo.java:4)
```

Exception Propagation

For propagating checked exceptions method must throw exception by using throws keyword.

```
public class ExceptionPropogationDemo
{
    public static void main(String a[]) throws IOException {
        m1();
    }
    public static void m1() throws IOException {
        m2();
    }
    public static void m2() throws IOException {
        m3();
    }
    public static void m3() throws IOException {
        throw new IOException();
    }
}
```

```
Exception in thread "main" java.io.IOException
at ExceptionPropogationDemo.m3(ExceptionPropogationDemo.java:15)
at ExceptionPropogationDemo.m2(ExceptionPropogationDemo.java:12)
at ExceptionPropogationDemo.m1(ExceptionPropogationDemo.java:9)
at ExceptionPropogationDemo.main(ExceptionPropogationDemo.java:6)
```

Method Overriding and Exceptions

The overriding method can throw

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overriding method cannot throw

- Additional exceptions not thrown by the overridden method
- Super classes of the exceptions thrown by the overridden method

Method Overriding and Exceptions

Example 1 :

```
class Parent{  
    public void method1() throws Exception  
    {  
        System.out.println("Parent");  
    }  
}  
  
public class Child extends Parent {  
    public void method1() throws Exception  
    {  
        System.out.println("Child");  
    }  
    public static void main(String a[]) throws  
        Exception  
    {  
        Parent p=new Child();  
        p.method1();  
    }  
}
```

Example 2 :

```
class Parent {  
    public void method1() throws Exception  
    {  
        System.out.println("Parent");  
    }  
}  
  
public class Child extends Parent {  
    public void method1() throws ArithmeticException  
    {  
        System.out.println("Child");  
    }  
    public static void main(String a[]) throws Exception  
    {  
        Parent p=new Child();  
        p.method1();  
    }  
}
```

Method Overriding and Exceptions

Example 1 :

```
class Parent {  
    public void method1() throws Exception  
    {  
        System.out.println("Parent");  
    }  
}  
  
public class Child extends Parent {  
    public void method1()  
    {  
        System.out.println("Child");  
    }  
}  
  
public static void main(String a[]) throws Exception  
{  
    Parent p=new Child();  
    p.method1();  
}
```

Example 2 :

```
class Parent {  
    public void method1() throws ArithmeticException  
    {  
        System.out.println("Parent");  
    }  
}  
  
public class Child extends Parent {  
    public void method1() throws Exception  
    {  
        System.out.println("Child");  
    }  
}  
  
public static void main(String a[]) throws Exception  
{  
    Parent p=new Child();  
    p.method1();  
}
```

Compile Time Error

User Defined Exception

Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors.

While defining a user defined exception, we need to extend the Exception class.

User Defined Exceptions - Example

```
class InvalidAgeException extends Exception {  
    public InvalidAgeException(String message)  
    {  
        super(message);  
    }  
  
    public class CustomException {  
        public static void validateAge(int age) throws  
            InvalidAgeException {  
            if(age < 18)  
            {  
                throw new InvalidAgeException("Not a  
                    valid Age to vote");  
            }  
  
            else {  
                System.out.println("Eligible to vote");  
            }  
        }  
  
        public static void main(String arg[]) {  
            try {  
                validateAge(15);  
            }  
            catch(InvalidAgeException e) {  
                System.out.println(e.getMessage());  
            }  
        }  
    }  
}
```

Try with multiple catch

Java 7, introduced the ability to catch multiple exceptions in the same catch block, also known as *multi-catch*.

```
public static void main(String[] args) {
    try {
        Path path = Paths.get("dolphinsBorn.txt");
        String text = new String(Files.readAllBytes(path));
        LocalDate date = LocalDate.parse(text);
        System.out.println(date);
    } catch (DateTimeParseException | IOException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}
```

//Compile Time Error
catch(Exception1 e | Exception2 e | Exception3 e)

//Works correctly
catch(Exception1 | Exception2 | Exception3 e)

Try with multi-catch

Java intends multi-catch to be used for exceptions that aren't related, and it prevents from specifying redundant types in a multi-catch.

```
try {  
    throw new IOException();  
}  
catch (FileNotFoundException | IOException e) {}  
// DOES NOT COMPILE
```

FileNotFoundException is a subclass of IOException.
Specifying it in the multi-catch is redundant, and the compiler gives a message as :
The exception FileNotFoundException is already caught by the alternative IOException

Try with resources

When we work with other resources, say file or database, its essential to close the resources once their usage is over

Assume, operation of reading data from one file and write to another file

The code will be as follows

```
public void writeData(Path path1, Path path2)
    throws IOException {
    BufferedReader in = null;
    BufferedWriter out = null;
    try {
        in = Files.newBufferedReader(path1);
        out = Files.newBufferedWriter(path2);
        out.write(in.readLine());
    } finally {
        if (in != null) in.close();  if (out != null)
                                    out.close();
    }
}
```

Resources used

Resources closed

Try with resources

```
public void writeData(Path path1, Path path2) throws IOException {  
    try (BufferedReader in = Files.newBufferedReader(path1);  
         BufferedWriter out = Files.newBufferedWriter(path2)) {  
        out.write(in.readLine());  
    }  
}
```

There is no longer code just to close resources.

The new *try-with-resources* statement automatically closes all resources opened in the try clause.

This feature is also known as *automatic resource management*, because Java automatically takes care of the closing.

There is no finally block in the try-with-resources code. It exists implicitly.

Try with resources

Any resources that should be automatically closed

```
try (BufferedReader r = Files.newBufferedReader(path1);
     BufferedWriter w = Files.newBufferedWriter(path2));
{
    out.write(in.readLine());
}
```

Resources are closed at this point

Try with resources statement can have a catch or finally block which will be run in addition to the implicit one. The implicit finally block gets executed before any programmer coded lines are executed. Resources are closed in the reverse order from which they were created.

Try with resources

```
try (Scanner s = new  
Scanner(System.in)) {  
    s.nextLine();  
} catch(Exception e) {  
    s.nextInt(); // DOES NOT COMPILE  
} finally{  
    s.nextInt(); // DOES NOT COMPILE  
}
```

The problem here, is that the Scanner has gone out of scope at the end of the try clause.

```
public class Sample {  
    public static void main(String  
        a[]) {  
        try (Sample s=new Sample();) {  
            System.out.println(s);  
        }  
    }  
}
```

Java doesn't allow this, because it doesn't know how to close Sample. It throws an error
"The resource type Sample does not implement java.lang.AutoCloseable"

To use a class in try with resources, Java requires it to implement an interface, AutoCloseable.

Try with resources

```
public class Sample implements AutoCloseable {  
    public void close() {  
        System.out.println("Close gate");  
    }  
    public static void main(String[] args) {  
        try (Sample s = new Sample() ) {  
            System.out.println("Hello");  
        }  
    }  
}
```

Auto Closeable is an interface with an abstract method
public void close() throws Exception;

Summary

- Exception Propagation
- User defined Exceptions
- Try with resources



Collections & Generics

GENERICS IN COLLECTION



In this Module You will learn

- Generics in Collection
- Working with Generics
- Difference between Comparable and Comparator
- Apply Polymorphism in Generics
-



GENERICS

Introduced from Java 5

Allows the programmer to specify the datatype to be stored in the collection

Provides compile-time type safety

Eliminates the need for casts

Provides the ability to create compiler-checked homogeneous collections

Using non-generic collections :

```
ArrayList list = new  
ArrayList();  
  
list.add(0,new Integer(42));
```

Using generic collections :

```
ArrayList<Integer> list = new ArrayList<Integer>();  
  
list.add(0,new Integer(42));  
  
int total = list.get(0);
```

Generic classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.

These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Generic Class contd.

Example-

```
public class Box<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();
        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));
        System.out.println("Integer Value : " + integerBox.get());
        System.out.println("String Value : "+stringBox.get());
    }
}
```

Output :

Integer Value : 10

String Value : Hello World

Generic class using Type Parameters

Classes showing how to use type parameters

Category	Non Generic Class	Generic Class
Class declaration	<code>public class ArrayList extends AbstractList implements List</code>	<code>public class ArrayList<E> extends AbstractList<E> implements List <E></code>
Constructor declaration	<code>public ArrayList (int capacity);</code>	<code>public ArrayList<E> (int capacity);</code>
Method declaration	<code>public void add((Object o) public Object get(int index)</code>	<code>public void add(E o) public E get(int index)</code>
Variable declaration examples	<code>ArrayList list1; ArrayList list2;</code>	<code>ArrayList <String> list1; ArrayList <Date> list2;</code>
Instance declaration examples	<code>list1 = new ArrayList(10); list2 = new ArrayList(10);</code>	<code>list1= new ArrayList<String> (10); list2= new ArrayList<Date> (10);</code>

Generics with Collection

```
import java.util.*;  
public class SetExample {  
    public static void main(String a[])  
    {  
        Set<String> set = new HashSet<String>();  
        set.add("First");  
        set.add("2nd");  
        set.add("third");  
        //This line generates a compilation error  
        // set.add(new Float(4.0f));  
        //This line generates a compilation error  
        // set.add(5);  
        set.add("third");          //duplicate, not added  
        System.out.println(set);  
    }  
}
```

Generics with Collection

```
public class PhoneBook {  
    HashMap<String, String> phoneBook;  
  
    public PhoneBook(){  
        phoneBook= new  
HashMap<String, String>();  
    }  
  
    public void put(String name, String phno) {  
        phoneBook.put(name,phno);  
    }  
  
    public String get(String name){  
        return phoneBook.get(name);  
    }  
}
```

```
public class ListExample {  
    public static void main(String a[]){  
        PhoneBook ph=new PhoneBook();  
        String name="Peter";  
        ph.put("John","9994441231");  
        ph.put("Peter","9994441232");  
        ph.put("James","9994441233");  
        System.out.println("Phone Number  
of  
"+name+" is "+ph.get("Peter"));  
    }  
}
```

Output :
Phone Number of Peter is 9994441232

ArrayList for User defined class

```
public class Employee {  
    private int id;  
    private String name;  
  
    //Write the Getters, Setters and Constructor  
    @Override  
    public String toString() {  
        return "ID : "+id+" Name :  
        "+name;  
    }  
}
```

Output :

```
ID : 101 Name : Tina  
ID : 109 Name : Pooja  
ID : 122 Name : George  
ID : 101 Name : Peter
```

```
import java.util.*;  
  
public class ArrayListDemo {  
  
    public static void main(String[] args) {  
        Employee e1 = new Employee(101,"Tina");  
        Employee e2 = new Employee(109,"Pooja");  
        Employee e3 = new  
Employee(122,"George");  
        Employee e4 = new Employee(101,"Peter");  
        List<Employee> empList=new  
ArrayList<Employee>();  
        empList.add(e1);  
        empList.add(e2);  
        empList.add(e3);  
        empList.add(e4);  
        for(Employee e : empList)  
            System.out.println(e);  
    }  
}
```

equals() and hashCode()

- `public boolean equals(Object obj)` - A method of Object class and checks if the object passed as an argument is equal to the current instance.
- `public int hashCode()` - A method of Object class, that returns an integer value represented by a hashing function.
By default the hashCode() returns a random integer that is unique for each instance or object.
- For HashSet or HashMap, overriding both hashCode() and equals() methods are required.
Two objects having the same hash code does not imply that they are equal.
But two objects that are equal will have the same hashCode.

```
@Override  
public int hashCode()  
{  
    //some code  
}
```

```
@Override  
public boolean equals(Object obj)  
{  
    //some code  
}
```

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{  
    private int id;  
    private String name;  
  
    //Assume you have written getters, setters and constructor  
  
    @Override  
    public int hashCode() {  
        return id%7;  
    }  
    @Override  
    public boolean equals(Object o) {  
        Employee e=(Employee)o;  
        if(this.id == e.id)  
            return true;  
        else  
            return false;  
    }  
  
    @Override  
    public String toString() {  
        return "Id = " + id + ", Name = " + name;  
    }  
}
```

Assume there is a class Employee with id, name and age as attributes. Also assume the constructor, getters and setters are written

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{  
  
    private int id;  
    private String name;  
  
    //Assume you have written getters, setters and constructor  
  
    @Override  
    public int hashCode() {  
        return id%7;  
    }  
    @Override  
    public boolean equals(Object o) {  
        Employee e=(Employee)o;  
        if(this.id == e.id)  
            return true;  
        else  
            return false;  
    }  
  
    @Override  
    public String toString() {  
        return "Id = " + id + ", Name = " + name;  
    }  
}
```

For the user defined object, it is the responsibility of the programmer to override the hashCode() and equals() method to avoid duplicates.

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{  
    private int id;  
    private String name;  
  
    //Assume you have written getters, setters and constructor  
  
    @Override  
    public int hashCode() {  
        return id%7;  
    }  
    @Override  
    public boolean equals(Object o) {  
        Employee e=(Employee)o;  
        if(this.id == e.id)  
            return true;  
        else  
            return false;  
    }  
  
    @Override  
    public String toString() {  
        return "Id = " + id + ", Name = " + name;  
    }  
}
```

When we add elements to HashSet, it generates the hashCode for each object.

And if any object exist with the same code, it invokes the equals method.

If the equals method returns true, the object will not be added, else it will be added to the HashSet.

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");    //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja");   //hash code is 4
        Employee employeeObj3 = new Employee(122,"George");  // hash code is 3,
                                                               //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter");   //hash code is 3,
                                                               //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList) {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }
    }
}
```

For the employeeObj1, the hash code is computed as $101 \% 7$ which is 3; Tina is added as it is the first object and Does not need to invoke equals() method to check duplicates

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");    //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja");   //hash code is 4
        Employee employeeObj3 = new Employee(122,"George");  // hash code is 3,
                                                               //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter");   //hash code is 3,
                                                               //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList) {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }
    }
}
```

For the employeeObj2, the hash code is computed as $109 \% 7$ which is 4; Pooja is added as the hash code is different and does not need to invoke equals() method to check duplicates

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");    //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja");   //hash code is 4
        Employee employeeObj3 = new Employee(122,"George");  // hash code is 3,
                                                               //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter");   //hash code is 3,
                                                               //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList) {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }
    }
}
```

For the employeeObj3, the hash code is computed as $122 \% 7$ which is 3; As 3 is already added, the HashSet also invokes equals() method to check duplicates on id. But since 122 and 101 are different, George is added

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");    //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja");   //hash code is 4
        Employee employeeObj3 = new Employee(122,"George");  // hash code is 3,
                                                               //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter");   //hash code is 3,
                                                               //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList) {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }
    }
}
```

For the employeeObj4, the hash code is computed as $101 \% 7$ which is 3; As hash code 3 already exists, the HashSet also invokes equals() method to check duplicates on id. But 101 id is duplicate, and so Peter is not added

Comparable and Comparator

Comparable and Comparator are used for sorting collection of objects.

To apply sorting methods of Arrays or Collections on any user defined class, that class should implement the Comparable interface.

Comparable interface has **compareTo(Tobj)** method. This method should be overridden by the class implementing Comparable to sort the objects in the collection.

Comparator interface has *compare(Object o1, Object o2)* method which needs to be implemented by the class to sort the collection of objects.

Comparable Vs Comparator

Comparable

We can sort the collection of objects on the basis of a single element. Say, by id or by name or by price.

Comparable affects the original class.

Comparable provides compareTo() method to sort elements.

Comparator

We can sort the collection on the basis of multiple elements such as id, name, price etc.

Comparator doesn't affect the original class.

Comparator provides compare() method to sort elements.

Comparable Example

```
public class Employee implements Comparable<Employee> {  
    private int id;  
    private String name;  
    private int age;  
    //Assume you have written Getters, Setters and Constructor  
  
    public int compareTo(Employee obj) {  
        if(this.id==obj.id)  
            return 0;  
        else if(this.id > obj.id)  
            return 1;  
        else  
            return -1;  
    }  
  
    public String toString() {  
        return "ID : "+id+" Name : "+name+" Age"+age;  
    }  
}
```

For the user defined object, it's the responsibility of the programmer to implement the Comparable interface and override the compareTo() method to store the elements in the sorted manner.

If the method returns 1 or -1 the elements will be sorted either in the ascending or the descending order.

TreeSet Example

```
import java.util.TreeSet;

public class TreeSetDemo {

    public static void main(String[] args) {

        TreeSet<Employee> empList = new TreeSet<Employee>();

        empList.add(new Employee(107, "George"));
        empList.add(new Employee(105, "John"));
        empList.add(new Employee(102, "Tom"));
        empList.add(new Employee(105, "Peter")); //not added

        for(Employee empObj : empList){

            System.out.println(empObj);
        }
    }
}
```

In the example, when we add the Employee objects into the TreeSet, the compareTo method will be invoked.

If the employee id is same it will return 0. In that case that object will not be added to the TreeSet thus restricting duplicates.

If the method returns 1 or -1 the elements will be sorted either in the ascending or the descending order.

Comparator

The Comparator interface is used when there is a need to sort objects in an order other than their natural ordering

The Comparator is also used for user defined objects like Employee, Customer or Student which does not have a natural ordering or does not implement Comparable

Like the Comparable interface, the Comparator interface also has a single method.

```
public interface Comparator<T> {  
    int compare(T object1, T object2);  
}
```

Comparator Example

To compare age - Ascending

```
public class Employee {  
    private int id;  
    private String name;  
    private int age;  
  
    //Assume you have written Getters, Setters and Constructor  
    public String toString() { return "ID : "+id+ " Name : "+name+ " Age"+age; }  
}
```

```
import java.util.Comparator;  
  
public class AgeAscendingComparator implements Comparator<Employee>  
{  
    public int compare(Employee emplObj1, Employee emplObj2)  
    {  
        return emplObj1.getAge() - emplObj2.getAge();  
    }  
}
```

To compare name

```
import java.util.Comparator;  
  
public class NameComparator implements Comparator<Employee>  
{  
    public int compare(Employee emplObj1, Employee emplObj2)  
    {  
        return emplObj1.getName().compareTo(emplObj2.getName());  
    }  
}
```

To compare age - Descending

```
import java.util.Comparator;  
  
public class AgeDescendingComparator implements Comparator<Employee>  
{  
    public int compare(Employee emplObj1, Employee emplObj2)  
    {  
        return emplObj2.getAge() - emplObj1.getAge();  
    }  
}
```

Comparator Example

```
import java.util.*;  
  
public class EmployeeListDemo{  
  
    public static void main(String[] args) {  
  
        List<Employee> empList = new ArrayList<Employee>();  
        Employee employeeObj1=new Employee(101, "John", 28);  
        Employee employeeObj2=new Employee(109, "Peter", 25);  
        Employee employeeObj3=new Employee(102, "Alan", 35);  
        empList.add(employeeObj1);  
        empList.add(employeeObj2);  
        empList.add(employeeObj3);  
  
        Collections.sort(empList, new NameComparator());  
        System.out.println(empList);  
  
        Collections.sort(empList, new AgeAscendingComparator());  
        System.out.println(empList);  
  
        Collections.sort(empList, new AgeDescendingComparator());  
        System.out.println(empList);  
  
    }  
}
```

Output

Name Comparator

[Id : 102 Name : Alan Age : 35, Id : 101 Name : John Age : 28, Id : 109 Name : Peter Age : 25]

Age Ascending

[Id : 109 Name : Peter Age : 25, Id : 101, Name : John Age : 28, Id : 102 Name : Alan Age : 35]

Age Descending

[Id : 102 Name : Alan Age : 35, Id : 101 Name : John Age : 28, Id : 109 Name : Peter Age : 25]

Comparator Example

```
public class Employee {  
  
    private int id;  
    private String name;  
    private double salary;  
  
    //Assume you have written Getters, Setters and Constructor  
    public String toString() {  
        return "ID : "+id+" Name : "+name+" Salary: "+salary;  
    }  
}
```

In the example, we have written two comparator classes, one to compare based on the name and another to compare based on the salary.

To compare name

```
import java.util.Comparator;  
  
public class NameComparator implements Comparator<Employee>  
{  
    public int compare(Employee emplObj1, Employee emplObj2)  
    {  
        return emplObj1.getName().compareTo(emplObj2.getName());  
    }  
}
```

To compare salary

```
import java.util.Comparator;  
  
public class SalaryComparator implements Comparator<Employee>  
{  
    public int compare(Employee emplObj1, Employee emplObj2)  
    {  
        return Double.valueOf(emplObj1.getSalary()).compareTo  
              (emplObj2.getSalary());  
    }  
}
```

Comparator Example

```
import java.util.*;  
  
public class EmployeeListDemo{  
  
    public static void main(String[] args) {  
  
        List<Employee> empList = new ArrayList<Employee>();  
        Employee employeeObj1=new Employee(107, "John", 20000);  
        Employee employeeObj2=new Employee(105, "John", 30000);  
        Employee employeeObj3=new Employee(109, "Tom", 40000);  
        Employee employeeObj4=new Employee(108, "John", 25000);  
        empList.add(employeeObj1);  
        empList.add(employeeObj2);  
        empList.add(employeeObj3);  
        empList.add(employeeObj4);  
  
        TreeSet<Employee> empListSortedByName = new TreeSet<>(new NameComparator());  
        empListSortedByName.addAll(empList);  
        for(Employee empObj : empListSortedByName)  
            System.out.println(empListSortedByName);  
  
        System.out.println("*****");  
        TreeSet<Employee> empListSortedBySalary = new TreeSet<>(new SalaryComparator());  
        empListSortedBySalary.addAll(empList);  
        for(Employee empObj : empListSortedBySalary)  
            System.out.println(empListSortedBySalary);  
  
    }  
}
```

OUTPUT:
ID: 107 Name : John Salary:20000.0
ID: 109 Name : Tom Salary:40000.0

ID: 107 Name : John Salary:20000.0
ID: 108 Name : John Salary:25000.0
ID: 105 Name : John Salary:30000.0
ID: 109 Name : Tom Salary:40000.0

Map Example

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

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

        Map<String, String> map=new LinkedHashMap<>();

        map.put("Rahul", "9878787655");
        map.put("Raghu", "8971231233");
        map.put("Pooja", "7891234567");

        Set<String> keys = map.keySet();

        for(String name : keys) {
            System.out.println("Key is "+name);
            System.out.println("Value is "+map.get(name));
        }

        for(Map.Entry<String, String> entrySet : map.entrySet()) {
            System.out.println("Key is "+entrySet.getKey());
            System.out.println("Value is "+entrySet.getValue());
        }

        if(map.containsKey("Rahul")) {
            map.put("Rahul", map.get("Rahul")+",7893455675");
        }
    }
}
```

Generics AND POLYMORPHISM

Generic collections gives the benefits of type safety

Assume the Student class is an abstract class.

Class Hosteller and DayScholar classes are derived from Student.

```
public class Student {  
    private int id;  
    private String name;  
    private double tuitionFees;  
  
    public Student(int id, String name, double tuitionFees) {  
        this.id = id;  
        this.name = name;  
        this.tuitionFees = tuitionFees;  
    }  
  
    //Assume Getters Setters are written  
  
    public String toString(){  
        return "ID "+id+" Name "+name+" Tution Fees "+tuitionFees;  
    }  
}
```

```
public class DayScholar extends Student {  
  
    private double transportFees;  
  
    public DayScholar(int id, String name, double tuitionFees, int transportFees) {  
        super(id, name, tuitionFees);  
        this.transportFees = transportFees;  
    }  
  
    public double calculateTotalFees(){  
        return getTuitionFees() + transportFees;  
    }  
  
    public String toString()  
    {  
        return super.toString() + " Transport Fees " + transportFees;  
    }  
}
```

Generics AND POLYMORPHISM

List<Student> studList = new ArrayList<Student>(); can accommodate any Student object getting added which can be a Student or a Hosteller or DayScholar object

```
public class Hosteller extends Student {  
  
    private double hostelFees;  
    private double messFees;  
  
    public Hosteller(int id, String name, double tuitionFees,  
                    double hostelFees, double messFees) {  
        super(id, name, tuitionFees);  
        this.hostelFees = hostelFees;  
        this.messFees = messFees;  
    }  
  
    public double calculateTotalFees(){  
        return getTuitionFees() + hostelFees + messFees;  
    }  
  
    public String toString()  
    {  
        return super.toString() + " Hostel Fees " +  
               hostelFees + " Mess Fees " + messFees;  
    }  
}
```

```
import java.util.*;  
  
public class StudentDAO {  
  
    List<Student> studentList = new ArrayList<Student>();  
  
    public void addStudent(Student studentObj){  
        studentList.add(studentObj);  
    }  
  
    public List<Student> getStudentList() {  
        return studentList;  
    }  
  
    public void setStudentList(List<Student> studentList) {  
        this.studentList = studentList;  
    }  
}
```

Generics AND POLYMORPHISM

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Student studentObj1=new Student(101,"Raghu",75000);  
  
        Hosteller hostellerObj1=new Hosteller(102,"Dravid",  
                                              80000,40000,20000);  
  
        DayScholar dayScholarObj1=new DayScholar(103,"Raghu",  
                                                 90000,40000);  
        StudentDAO dao = new StudentDAO();  
  
        dao.addStudent(studentObj1);  
        dao.addStudent(dayScholarObj1);  
        dao.addStudent(hostellerObj1);  
  
        System.out.println(dao.getStudentList());  
    }  
}
```

Output:

```
[ID 101 Name Raghu Tution Fees 75000.0,  
 ID 103 Name Raghu Tution Fees 90000.0 Transport  
          Fees 40000.0,  
 ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees  
          40000.0 Mess Fees 20000.0]
```

Generics AND POLYMORPHISM

Observe the below code

```
import java.util.*;  
  
public class StudentDAO {  
  
    List<Student> studentList = new ArrayList<Student>();  
  
    public void addStudentList(List<Student> studentList){  
        studentList.addAll(studentList);  
    }  
    public List<Student> getStudentList() {  
        return studentList;  
    }  
    public void setStudentList(List<Student> studentList) {  
        this.studentList = studentList;  
    }  
}
```

Generics AND POLYMORPHISM

```
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        Hosteller hostellerObj1=new Hosteller(102,"Dravid",
                                             80000,40000,20000);
        Hosteller hostellerObj2=new Hosteller(103,"Dravid",
                                             80000,40000,20000);
        Hosteller hostellerObj3=new Hosteller(104,"Dravid",
                                             80000,40000,20000);

        List<Hosteller> hostellerList = new ArrayList<Hosteller>();
        hostellerList.add(hostellerObj1);
        hostellerList.add(hostellerObj2);
        hostellerList.add(hostellerObj3);

        StudentDAO dao = new StudentDAO();

        The method addStudentList(List<Student>) in the type StudentDAO is not applicable for the arguments (List<Hosteller>)
        dao.addStudentList(hostellerList);

        System.out.println(dao.getStudentList());
    }
}
```

When hostellerList is added to studentList in StudentDAO class using addStudentList method, it results in compilation error,

This is because List<Student> studList= new ArrayList<Hosteller>(); is invalid in generics. Both the types must be the same.

Generics AND POLYMORPHISM

To overcome the error, change the parameter to addStudentList as
List<? extends Student> instead of List<Student>

Updated StudentDAO class

```
import java.util.*;  
  
public class StudentDAO {  
  
    List<Student> studentList = new ArrayList<Student>();  
  
    public void addStudentList( List<? extends Student> studentList ){  
        this.studentList.addAll(studentList);  
    }  
    public List<Student> getStudentList() {  
        return studentList;  
    }  
    public void setStudentList(List<Student> studentList) {  
        this.studentList = studentList;  
    }  
}
```

Output :

```
[ID 102 Name Dravid Tution Fees 80000.0  
Hostel Fees 40000.0 Mess Fees 20000.0,  
ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees 40000.0 Mess  
Fees 20000.0, ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees  
40000.0 Mess Fees 20000.0]
```

Collections class

Collections class in java.util package extends directly from Object class

It consists of static methods that operate on or return collection

Few methods in Collections class are

Method	Description
Collections.sort(List<T extends Comparable<? super T>> list)	Sorts the given list in ascending order according to the natural ordering
Collections.binarySearch(List<T extends Comparable<? super T>> list, T key)	Searches the element T in the given list. It returns the index of the searched element if found; else (-) insertion point.
Collections.shuffle(List<?> list)	Randomly shuffles the elements in the list. This will return different results in different calls.
Collections.reverse(List<?> list)	Reverses the order of the elements in the specified list.

Collections example

- In the example, we have used various collections method to do a particular task..
- For the sort method, we have passed the arraylist and the elements will be sorted in the natural order.
- The binarysearch method, takes the arraylist and the elements as argument and it will return the index position of that element.
- The shuffle method, takes the arraylist as argument and shuffle the elements

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {

    public static void main(String[] args) {
        List<String> nameList = new ArrayList<String>();
        nameList.add("Peter");
        nameList.add("Albert");
        nameList.add("Jim");
        nameList.add("Pinky");
        nameList.add("James");
        nameList.add("John");

        // Print elements of nameList
        System.out.println("The collection is "+nameList);

        // Sorting List in ascending order according to the natural ordering
        Collections.sort(nameList);
        System.out.println("Sorted List: "+nameList);

        // Searching a name from list
        int searchIndex = Collections.binarySearch(nameList, "Pinky");
        if(searchIndex >=0)
        {
            System.out.println("Name found at index "+searchIndex);
        }
        else
        {
            System.out.println("Name not found.");
        }

        //Shuffles the list
        Collections.shuffle(nameList);
        System.out.println("Shuffled List: "+nameList);
    }
}
```

Arrays class

Present in `java.util` package

ARRAYS CLASS HAS A COLLECTION OF STATIC METHODS TO WORK WITH ARRAYS LIKE SORTING AND SEARCHING

FEW METHODS IN ARRAYS CLASS ARE:

`equals`

`copyOf`

`sort`

Arrays - example

Example for equals method

```
import java.util.Arrays;  
  
public class ArraysDemo {  
  
    public static void main(String[] args) {  
        int a[]={2,4,6,8,10};  
        int b[]={2,4,6,8,10};  
        System.out.println(Arrays.equals(a,b)); //returns true  
  
        int c[]={2,4,6,8,10};  
        int d[]={10,8,4,2,6};  
        System.out.println(Arrays.equals(c,d)); //returns false  
  
    }  
}
```

The equals method accepts two arguments, both the arguments needs to be an array.

Both arrays should be of the same data type and one dimensional

Return true if both arrays contain same elements in the same order

Arrays - example

Example for copyOf method

```
import java.util.Arrays;  
  
public class ArraysDemo {  
  
    public static void main(String[] args) {  
        int num1[]={2,4,6,8,10};  
        int num2[]={};  
        num2=Arrays.copyOf(num1,3);  
  
        for(int num : num2)  
            System.out.println(num); // prints 2 4 6  
  
        int num3[]={1,2,3};  
  
        int num4[]={};  
        num4=Arrays.copyOf(num3,5);  
        for(int num : num4)  
            System.out.println(num); // prints 1 2 3 0 0  
  
    }  
}
```

The copyOf method accepts two arguments, the first should be an array and the second should be the number of elements that should be copied.

This method returns an array which is a copy of the originalArray.

It copies elements from 0 index position.

If the number of elements exceeds the length of originalArray, it pads those positions with default value.

Arrays - example

Example for sort method

```
import java.util.Arrays;  
  
public class ArraysDemo {  
  
    public static void main(String[] args) {  
        int num1[]={12,41,32,16,10};  
        Arrays.sort(num1);  
  
        for(int num : num1)  
            System.out.println(num); // prints 10 12 16 32 41  
  
        int num2[]={82,41,32,16,10,46,72,89};  
        Arrays.sort(num2,1,4); //sorts the numbers from 1 (inclusive) to 4 (exclusive)  
  
        for(int num : num2)  
            System.out.println(num); //prints 82 16 32 41 10 46 72 89  
  
    }  
}
```

The sort method sorts the array in ascending order.

We can also sort the elements in a specific range.

Arrays - example

Example to sort user defined object

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

class Employee{
    int id; String name;
    double salary;
    public Employee(int id, String name, double salary) {
        super();
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    public String getName()
    {
        return name;
    }
    public String toString()
    {
        return "ID : "+id+ " Name : "+name+ " Salary : "+salary;
    }
}

class EmpNameComp implements Comparator
{
    public int compare(Object obj1, Object obj2) {
        Employee employeeObj1=(Employee)obj1;
        Employee employeeObj2=(Employee)obj2;

        return employeeObj1.getName().compareTo(employeeObj2.getName());
    }
}

```

```

public class ArraysExample {

    public static void main(String[] args) {

        List<Employee> emplist = new ArrayList<Employee>();
        emplist.add(new Employee(107,"John",20000));
        emplist.add(new Employee(105,"John",30000));
        emplist.add(new Employee(102,"Tom",40000));
        emplist.add(new Employee(108,"John",25000));

        Object employeeObj[]=emplist.toArray(); //To convert the collection to an Array

        Arrays.sort(employeeObj,new EmpNameComp());

        for(Object obj : employeeObj){
            Employee empObj=(Employee)obj;
            System.out.println(empObj);
        }
    }
}

```

In this example, we have used `toArray()` function to convert the collection into an array.

We have passed the array and comparator as an argument to the sort method. The sort method will sort the object based on the logic written in the comparator class. Here we are sorting the objects based on name.

Summary

- Generics in Collection
- Working with Generics
- Difference between Comparable and Comparator
- Apply Polymorphism in Generics
-





COLLECTION



In this Module You will learn

- Collection Framework
- List and Set
- ArrayList
- HashSet and TreeSet
- Map – HashMap and TreeMap
- Nature of each collection

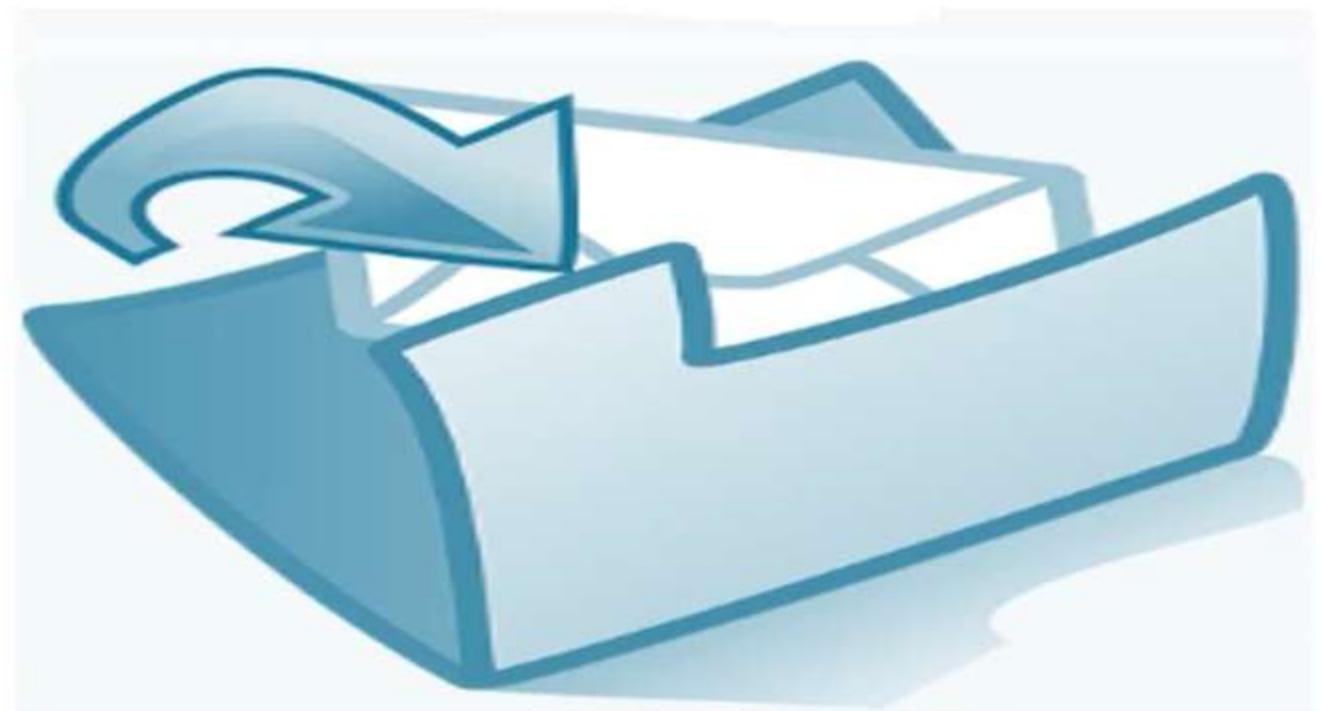


Collection

Collection is a container that groups multiple elements into a single unit where each element is an object



Collection is used to store, retrieve and manipulate aggregate data



Collection Framework

All classes and interfaces related to Collection are present in `java.util` package

Collection framework

- Is a set of utility class and interfaces.
- Designed for working with collection of objects.
- Will hold objects

Collection framework contains

- Set of interfaces
- Classes that implement those interfaces
- Utility classes for performing operations on collections

Types of Collection

Simple Collection

- Sets have no duplicate elements.
- Lists are ordered collections that can have duplicate elements.

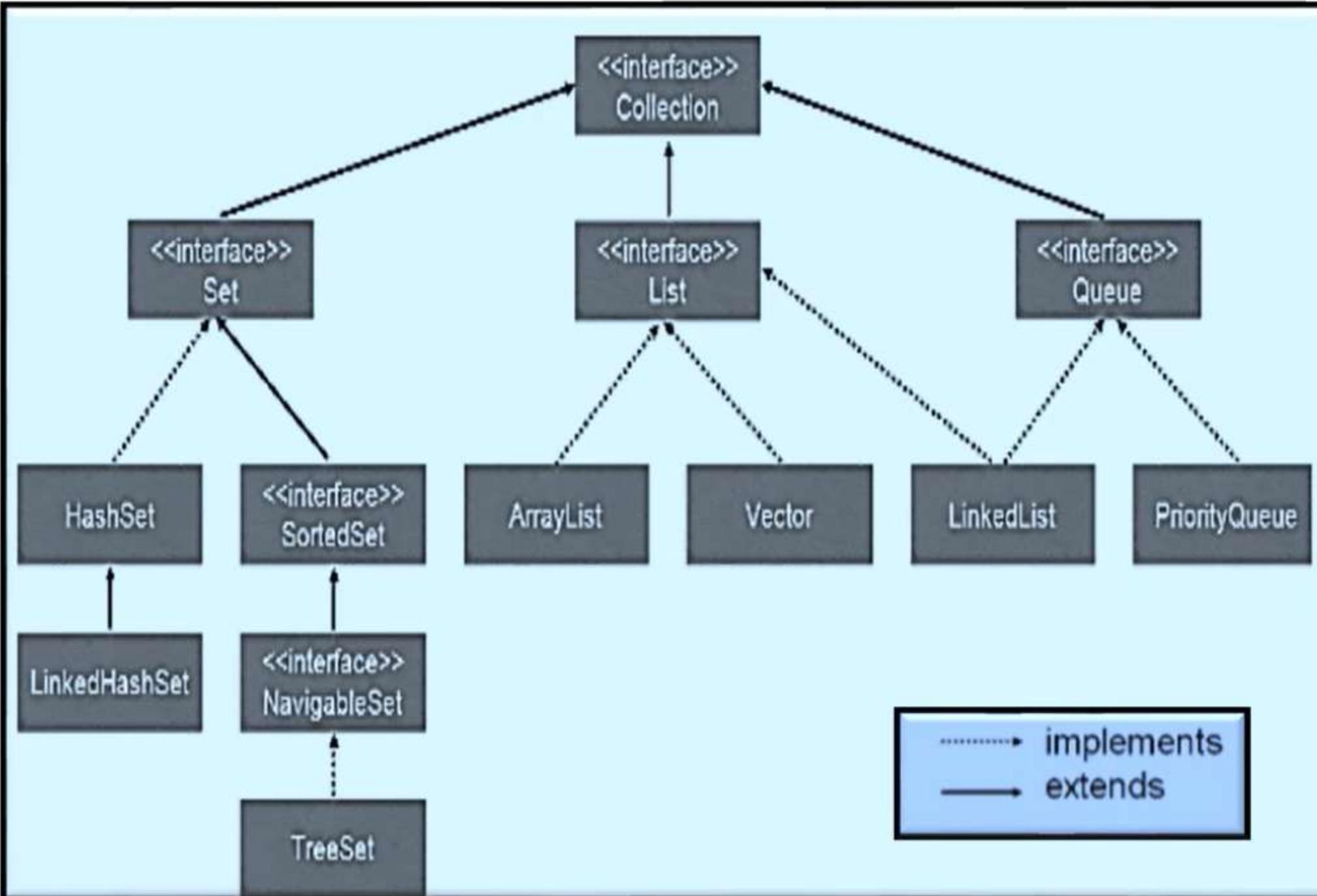
Map

- Map uses key/value pairs to associate an object (the value) with a key.

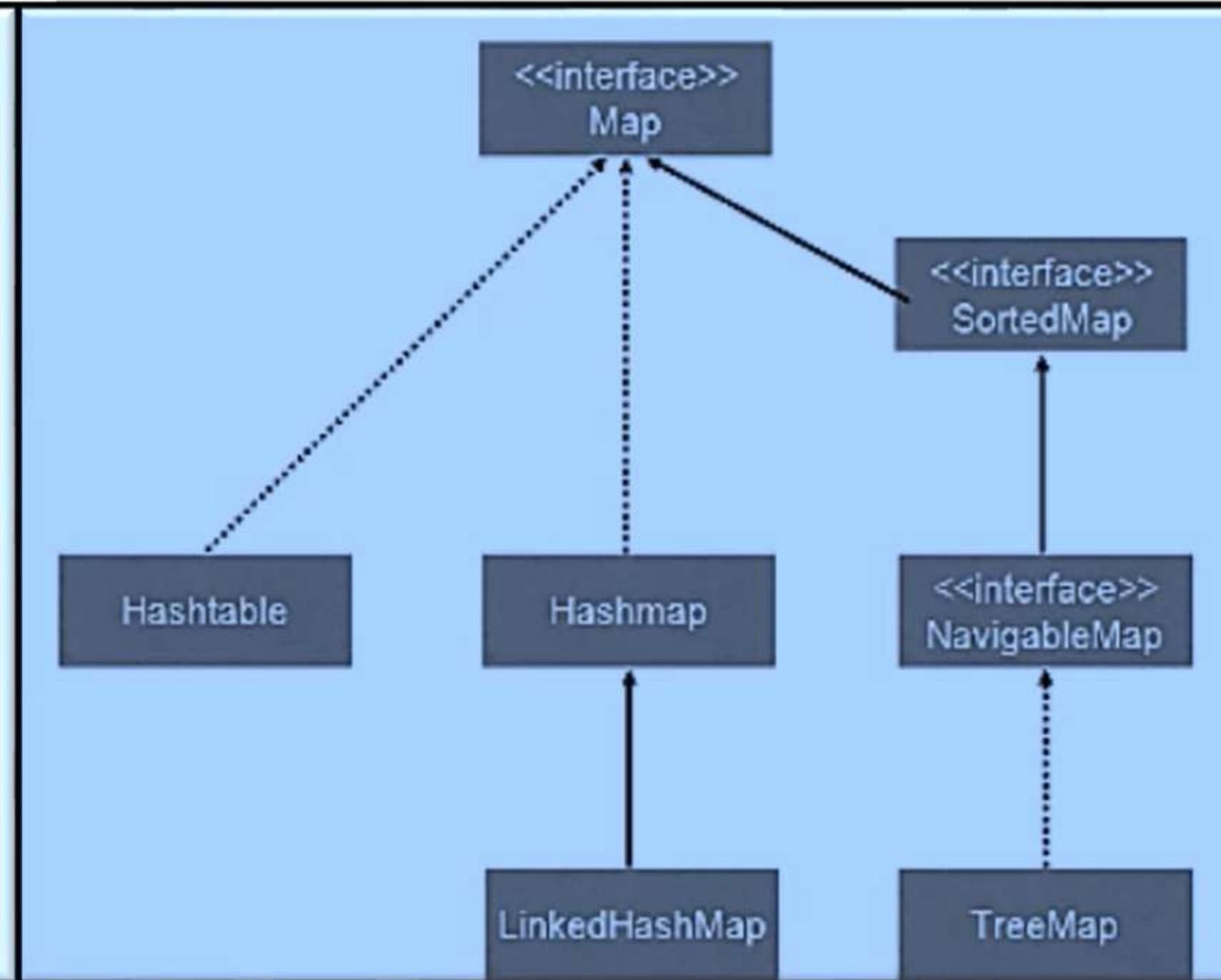
Both set and map can be sorted or unsorted.

Collection Hierarchy

Collection Hierarchy



Map Hierarchy



Collection – Implementing Classes

There are several general purpose implementations of the core interfaces
(Set, List, Deque and Map)

	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

List Interface

Used for storing elements in an ordered way

Allows index based additions and retrieval of items

Allows duplicate elements

List Implementations

- Vector
- ArrayList
- LinkedList

ArrayList

Implements
List interface

Holds objects

Supports duplicate
values to be
inserted into the
list

Allows random
retrieval and
insertion of
elements

Initially gets created
with an initial capacity,
when this gets filled, the
list automatically grows

ArrayList Methods

Method	Description
void add(int index, Object element)	Inserts the specified element at the specified position in this list.
boolean add(Object o)	Appends the specified element to the end of this list.
boolean addAll(Collection c)	Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean addAll(int index, Collection c)	Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void clear()	Removes all of the elements from this list.
boolean contains(Object elem)	Returns true if this list contains the specified element.
Object get(int index)	Returns the element at the specified position in this list.

More methods refer to:

<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

List Example

```
import java.util.*;  
public class ListExample {  
    public static void main(String a[])  
    {  
  
        List list = new ArrayList();  
        list.add("First");  
        list.add("2nd");  
        list.add("third");  
        list.add(new Float(4.0f));  
        list.add(new Integer(6));  
        list.add(5);  
        list.add("third");          //duplicate added  
        list.add(5);              //duplicate added  
        System.out.println(list);  
    }  
}
```

Output : [First, 2nd, third, 4.0, 6, 5, third, 5]

Set Interface

A Set is a collection with no duplicate elements.

A HashSet stores elements in a hash table.

A TreeSet stores elements in a balanced binary tree.

A LinkedHashSet is an ordered hash table.

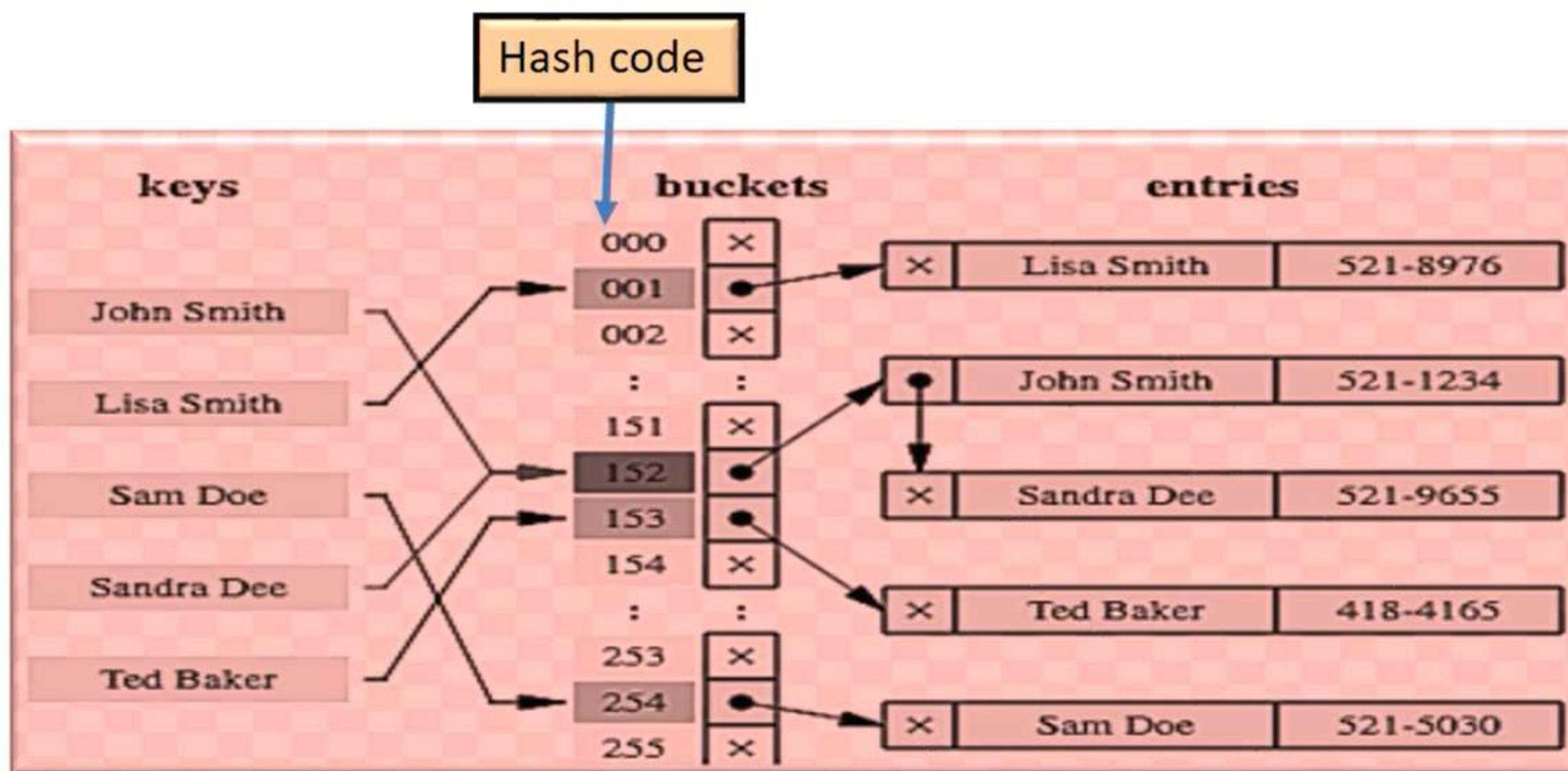


HashSet

HashSet is a powerful data structure that can be used to retrieve objects quickly in a set.

HashSet is an Unordered collection

A hash code is used to organize the objects in a HashSet



HashSet Methods

Method	Description
<code>boolean add(Object element)</code>	Adds the specified element to this set if it is not already present.
<code>void clear()</code>	Removes all of the elements from this set.
<code>boolean contains(Object o)</code>	Returns true if this set contains the specified element.
<code>boolean isEmpty()</code>	Returns true if this set contains no elements.
<code>boolean remove(Object o)</code>	Removes the specified element from this set if it is present.
<code>int size()</code>	Returns the number of elements in this set.

- More methods refer to: <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>

TreeSet

Tree Set provides the properties of a set in a sorted collection

Objects can be inserted in any order but are retrieved in a sorted order

TreeSet uses Comparable interface to compare objects for sorting

It is up to the programmer to implement the compareTo() method for user-defined objects

A Set Example

```
import java.util.*;
public class SetExample {
    public static void main(String a[])
    {
        Set set = new HashSet();
        set.add("First");
        set.add("2nd");
        set.add("third");
        set.add(new Float(4.0f));
        set.add(new Integer(6));
        set.add(5);
        set.add("third");           //duplicate, not added
        set.add(5);               //duplicate, not added
        System.out.println(set);
    }
}
```

- The output generated from this program is: [4.0, third, 5, 6, 2nd, First]

Map Interface

Map are sometimes called associative arrays

Example : Dictionary

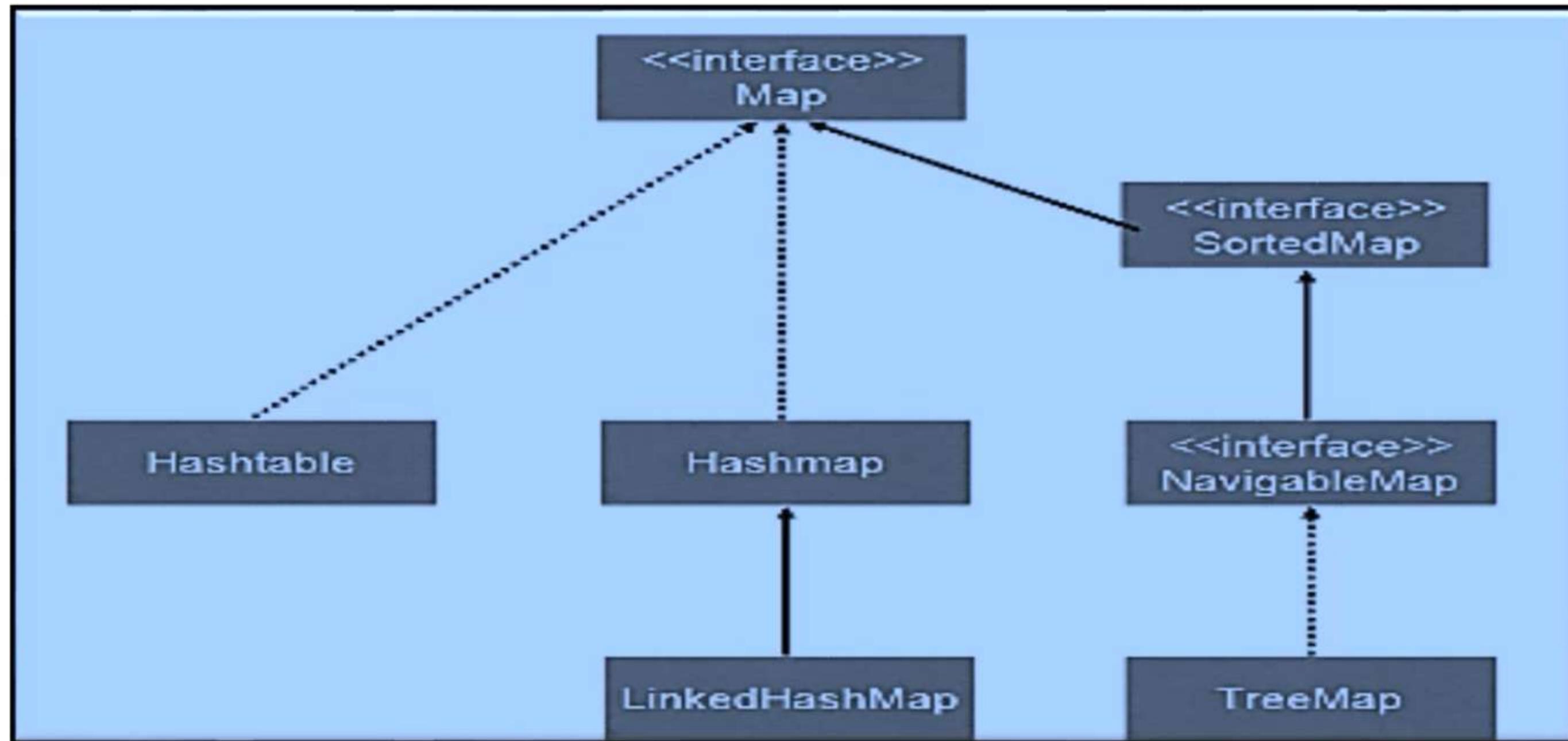
A Map object describes mappings from keys to values:

- Duplicate keys are not allowed
- One-to-many mappings from keys to values are not permitted

The contents of the Map interface can be viewed and manipulated as collections

- `entrySet` – Returns a Set of all the key-value pairs.
- `keySet` – Returns a Set of all the keys in the map.
- `values` – Returns a Collection of all values in the map.

Map implementation



MAP INTERFACE METHODS

Methods	Description
void clear()	Removes all mappings from this map (optional operation).
boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
Set entrySet()	Returns a set view of the mappings contained in this map.
Object get(Object key)	Returns the value to which this map maps the specified key.
Set keySet()	Returns a set with all the keys contained in this map.
Object put(Object key, Object value)	Associates the specified value with the specified key in this map.
void putAll(Map t)	Copies all of the mappings from the specified map to this map
Object remove(Object key)	Removes the mapping for this key from this map if it is present
int size()	Returns the number of key-value mappings in this map.
Collection values()	Returns a collection with all the values contained in this map.

A Map Example

```
import java.util.*;  
public class MapExample {  
    public static void main(String a[]) {  
        Map map=new HashMap();  
        map.put("First","1st");  
        map.put("2nd",new Float(2.0f));  
        map.put("third","3rd");  
        //Duplicate - Overrides the previous assignment  
        map.put("third","III");  
  
        //To view the map  
        //Returns the set view of keys  
        Set set=map.keySet();  
        //Returns collection view of values  
        Collection collection = map.values();  
        // Returns set view of key value mappings  
        Set mapset=map.entrySet();  
        System.out.println(set+"\n"+collection+"\n"+mapset);  
    }  
}
```

Output :

[2nd, First, third]

[2.0, 1st, III]

**[2nd=2.0, First=1st,
third=III]**

Sorted Map

SortedMap is similar to **SortedSet**, but it holds the elements as key-value pair

The key will be in a sorted manner

Implementation classes: **TreeMap**

TreeMap: Key Objects can be inserted in any order but are retrieved in a sorted order

TreeMap internally uses `compareTo()` methods to compare the key objects for sorting.

Summary

- Collection Framework
- List and Set
- ArrayList
- HashSet and TreeSet
- Map – HashMap and TreeMap
- Generics
- List, Set and Map using Generics
- Collections class



ITERATE A COLLECTION



In this Module You will learn

- **Various ways to Iterate a Collection**
- **Iterate a collection using for loop**
- **Iterate a collection using enhanced for loop**
- **Usage of Iterator**
- **Iterate a Map**



Iterating Collections

Iteration is the process of retrieving every element in a collection

The basic iterator interface allows to scan forward through any collection

For iterating a list, we can use the `ListIterator` which allows to scan a list both forward and backward and insert or modify elements

Iterating Collections

Following are the ways to iterate the collection and fetch elements one by one

for loop

- Can be used only with List
- Reads the elements by specifying the index using the get() method

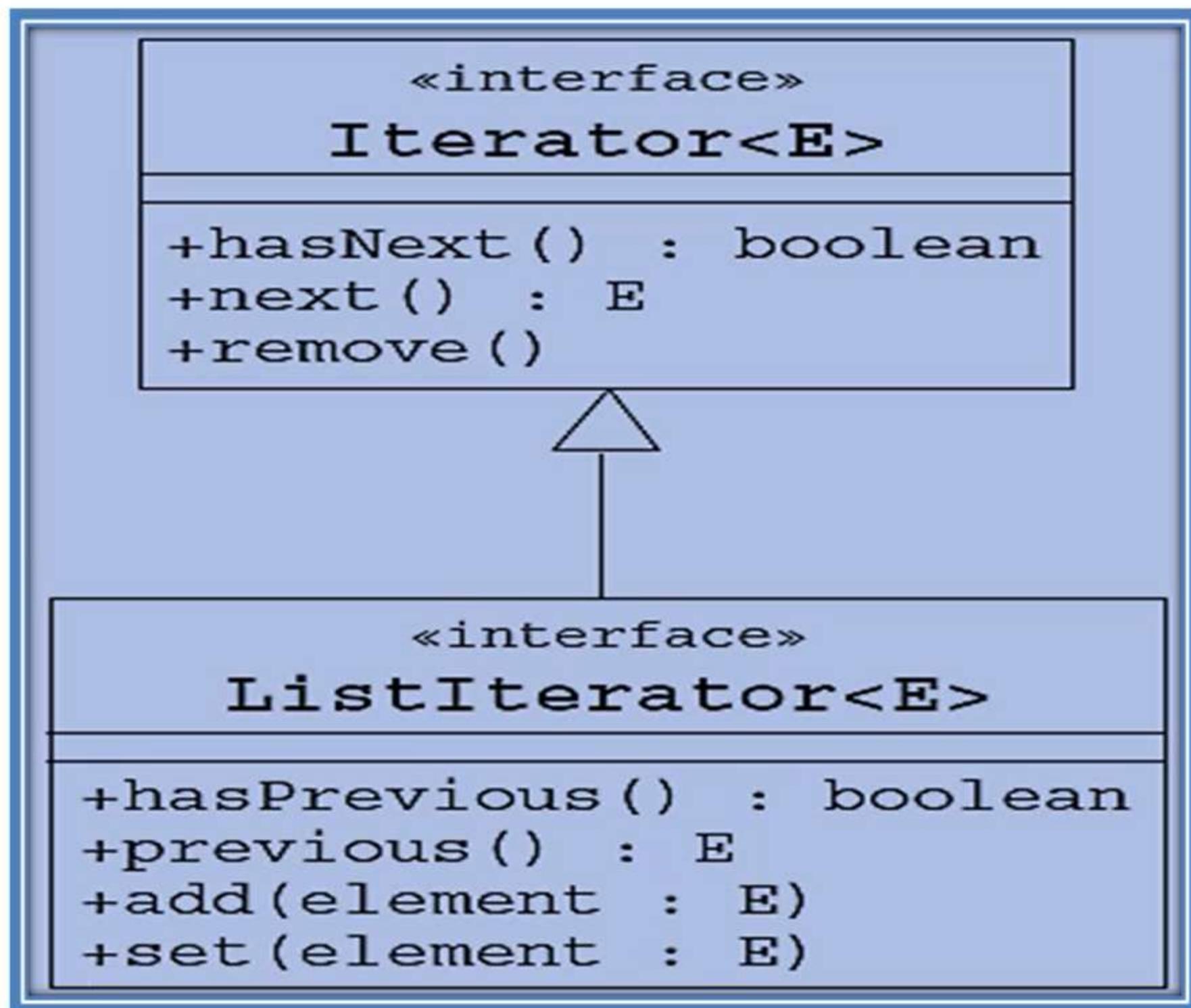
for each loop

- Can be used with both List and Set
- Iterates and fetches the element one by one until all the elements in the list are covered

Iterator

- Can be used with both List and Set
- Provides methods to iterate over the collection

Iterator Interface



Using for, Foreach and Iterator

```
List<Student> studentList = new  
        ArrayList<Student>();  
//add some elements  
for(int i=0;i<studentList.size();i++) {  
    System.out.println(studentList.get(i));  
}
```

```
List<Student> studentList = new  
        ArrayList<Student>();  
//add some elements  
for(Student studentObj : studentList) {  
    System.out.println(studentObj);  
}
```

```
List<Student> studentList = new ArrayList<Student>();  
//add some elements  
Iterator<Student> elements = studentList.iterator();  
while(elements.hasNext()) {  
    Student studentObj=elements.next();  
    System.out.println(studentObj);  
}
```

Iteration Using For loop

```
import java.util.ArrayList;
import java.util.List;
public class Test {

    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("First");
        list.add("2nd");
        list.add("third");
        list.add(new Float(4.0f));
        list.add(new Integer(5));
        list.add("third");
        list.add(5);
        for(int i=0;i<list.size();i++)
        {
            System.out.println(list.get(i));
        }
    }
}
```

- The list.size() will return the value as 7
- The get method takes the index position and fetches the object in that location

Iteration Using For loop(with Typecasting)

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.List;
public class Test {

    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        for(int i=0;i<list.size();i++)
        {
            String name=list.get(i); Type mismatch: cannot convert from Object to String
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

Why is there an error in the line

```
String name=list.get(i)
```

The get method will return an Object, it's the responsibility of the programmer to type cast to the corresponding object.

We need to typecast as

```
String name=(String)list.get(i);
```

Iteration Using Enhanced For Loop

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.List;
public class Test {

    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        for(Object obj:list)
        {
            String name=(String)obj;
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

Enhanced for loop, is used for fetching each object. It is then type casted and displayed if the length is 5

Iteration Using Iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {

    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        Iterator iobj=list.iterator();
        while(iobj.hasNext())
        {

            String name=(String)iobj.next();
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

The iterator() method returns an iterator over the elements in the ArrayList.

Iteration Using Iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        Iterator iobj=list.iterator();
        while(iobj.hasNext())
        {
            String name=(String)iobj.next();
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

- The iterator() method returns an iterator over the elements in the ArrayList.
- In the while loop, we check whether the iterator has more elements by invoking the hasNext() method

Iteration Using Iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        Iterator iobj=list.iterator();
        while(iobj.hasNext())
        {
            String name=(String)iobj.next();
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

- The iterator() method returns an iterator over the elements in the ArrayList.
- In the while loop, we check whether the iterator has more elements by invoking the hasNext() method
- Using next() we fetch the object from the iterator, this method returns an object , it is then type casted and displayed if the length is 5

Iteration Using Iterator

To iterate the elements in a HashMap

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Directory{
    public static void main(String[] args) {
        Map map = new HashMap();
        map.put("Alan", "9843012345"); //Key, Value pair
        map.put("Scott", "7780521155");
        map.put("Thom", "8002598765");

        //Iterating using Iterator
        Iterator iter = map.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            System.out.println(entry.getKey() + " - " + entry.getValue());
        }
    }
}
```

- In Map, Iteration is done using `entrySet().iterator()` method; It helps in iterating the elements of the collection
- Created a HashMap of Names and Phone numbers
- Names are the Keys and Phone Numbers are the values

Output:

```
Thom - 8002598765
Alan - 9843012345
Scott - 7780521155
```

Foreach Method

This method was introduced in java 8

```
import java.util.ArrayList;
import java.util.List;

public class Test {

    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add(10.6);
        list.add("mysql");
        list.add(123);

        list.forEach(s->System.out.println(s));
    }
}
```

-> is a lambda expression

- forEach() method is defined in Iterable and Stream interface.
- It is a default method in the Iterable interface.
- Collection classes which extends Iterable interface can use forEach loop to iterate elements.
- Method : void forEach(Consumer<super T>action) -
Performs the given action for each element of the Iterable until all the elements have been processed

For more methods refer to the link:: <https://docs.oracle.com/en/java/javase>

Foreach Method

Example to iterate over map

```
import java.util.HashMap;
import java.util.Map;
class Customer
{
    int id;
    String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Customer(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
}
```

```
public class StreamDemo {
    public static void main(String[] args) {
        Map<Integer, Customer> map = new HashMap<Integer, Customer>();
        map.put(1, new Customer(12, "peter"));
        map.put(2, new Customer(34, "john"));
        map.put(3, new Customer(34, "tom"));
        map.forEach((k, v) -> System.out.println("Key : " + k + " Value : " + v.getId()));
    }
}
```

Summary

- **Various ways to Iterate a Collection**
- **Iterate a collection using for loop**
- **Iterate a collection using enhanced for loop**
- **Usage of Iterator**
- **Iterate a Map**



STREAM API



In this Module You will learn

- Usage of stream API
- Few intermediate methods in Stream API
- Few terminal operation on stream
- Collectors



Stream API

Stream represents a sequence of objects from a source, which supports aggregate operations

- The source here refers to a Collection, IO Operation or Arrays that provide data to a Stream
- Stream performs automatic iteration, whereas in collection explicit iteration is needed
- Stream keeps the order of the data as it is in the source
- The operations can be executed in series or in parallel
- Like functional programming languages, Streams support Aggregate Operations
- Common aggregate operations are filter, map, reduce, find, match, sort

Creation of stream object

Collection interface has two methods to generate a stream

`stream()`

- Returns a sequential stream considering collection as its source

`parallelStream()`

- Returns a parallel Stream considering collection as its source

Methods in Stream API

Various intermediate methods used in stream are:

- distinct
- limit
- sorted
- filter
- min
- max

The terminal operations, is invoked at the end of the pipeline. It will close the stream in some meaningful way. Example : foreach method

distinct - Example

In this example, we have used the `asList()` method of arrays class to convert the set of values into a collection.

We have created the stream and invoked the `distinct()` method, this method will return a stream consisting of the distinct elements.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
        Stream stream = numbers.stream(); //Creating a stream

        Stream stream1 = stream.distinct(); //Fetch the distinct elements and return a new stream
    }
}
```

collect - Example

The collect method is used to get list, map or set from a stream.

The stream will not affect the original collection.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
        Stream stream = numbers.stream(); //Creating a stream

        Stream stream1 = stream.distinct(); //Fetch the distinct elements
    }
}
```

```
List<Integer> distinctList = (List<Integer>)
    stream1.collect(Collectors.toList());
System.out.println(distinctList);

}
```

Output:
[3, 2, 7, 5]

limit - Example

We have created the stream and invoked the limit() method.
This method will return a stream of first 3 elements.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
        Stream stream = numbers.stream(); //Creating a stream

        Stream stream1 = stream.limit(3); //Returns a new stream
                                    //of first three elements.
    }
}
```

```
List<Integer> distinctList = (List<Integer>)
    stream1.collect(Collectors.toList());

System.out.println(distinctList);

}
```

Output:
[3, 2, 2]

filter - Example

The filter method is used to eliminate elements based on a criteria.

```
import java.util.Arrays;
import java.util.List;

public class StreamDemo {

    public static void main(String[] args) {
        List<String> strings = Arrays.asList("apple","","orange","pineapple","","lime");
        long count=strings.stream().filter(string -> string.isEmpty()).count();
        System.out.println(count);

    }
}
```

Output:
[2]

forEach - Example

An easy way to loop over the stream elements is usage of forEach loop.

```
import java.util.Arrays;
import java.util.List;

public class StreamDemo {

    public static void main(String[] args) {
        List<String> strings = Arrays.asList("apple","","orange","pineapple","","lime");
        strings.stream().filter(string -> !(string.isEmpty())).forEach(System.out::println);
    }
}
```

Output:

apple
orange
pineapple
lime

filter - Example

The 'filter' method is used to eliminate elements based on a criteria

```
import java.util.*;  
  
public class StreamDemo {  
  
    public static void main(String[] args) {  
        List<Employee> employees = new ArrayList<Employee>();  
        employees.add(new Employee(1,"john",20000));  
        employees.add(new Employee(2,"tom",30000));  
        employees.add(new Employee(3,"tom",10000));  
        employees.stream().filter(e -> e.getSalary() > 15000).forEach(e->System.out.println(e));  
    }  
}
```

toArray - Example

To convert the stream to an array

```
import java.util.*;  
  
public class StreamDemo {  
  
    public static void main(String[] args) {  
        List<Employee> employees = new ArrayList<Employee>();  
        employees.add(new Employee(1,"john",20000));  
        employees.add(new Employee(2,"tom",30000));  
        employees.add(new Employee(3,"tom",10000));  
        Employee e[]=(Employee[])employees.stream().toArray();  
    }  
}
```

Collectors

Collectors are used to summarize the results based on the criteria.

The various aggregate operations performed are:

- Sum
- Average
- Min
- Max

Collectors – Sum Example

To perform the sum operation use `Collectors.summingXXX()` with `Stream.collect()`

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

    public static void main(String ap[])
    {
        List<Employee> employees = new ArrayList<Employee>();
        employees.add(new Employee(1,"john",20000));
        employees.add(new Employee(2,"tom",30000));
        employees.add(new Employee(3,"ram",10000));
        double total =
employees.stream().collect(Collectors.summingDouble(Employee::getSalary));
        System.out.println("Total Employees Salary : " + total);
    }
}
```

Output:

Total Employees Salary : 60000.0

Collectors – Average Example

To perform the average operation use Collectors. averagingXXX() with Stream.collect()

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

    public static void main(String ap[])
    {
        List<Employee> employees = new ArrayList<Employee>();
        employees.add(new Employee(1,"john",20000));
        employees.add(new Employee(2,"tom",30000));
        employees.add(new Employee(3,"ram",10000));
        double average =
employees.stream().collect(Collectors.averagingDouble(Employee::getSalary));
        System.out.println("Avg Salary : " + average);    }
}
```

Output:
Avg Salary : 20000.0

Collectors – max Example

To perform the max operation use Collectors. maxBy() with Stream.collect()

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

    public static void main(String ap[])
    {
        List<Employee> employees = new ArrayList<Employee>();
        employees.add(new Employee(1,"john",20000));
        employees.add(new Employee(2,"tom",30000));
        employees.add(new Employee(3,"ram",10000));
        System.out.println("Employee name
:"+employees.stream().collect(Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary))).get().get
EmpName());
    }
}
```

Output:
Employee name : tom

Stream API - Example



```
List<Employee> employees = new ArrayList<Employee>();
    employees.add(new Employee(1,"john",20000));
    employees.add(new Employee(2,"tom",30000));
    employees.add(new Employee(3,"tim",10000));
```

```
List<Employee> list=employees.stream().filter(e->e.getEmpName().startsWith("t")).collect(Collectors.toList());
for(Employee e:list)
{
    System.out.println(e.getEmpName());
}
```

Stream API - Example



```
List<Employee> employees = new ArrayList<Employee>();  
    employees.add(new Employee(1,"john",20000));  
    employees.add(new Employee(2,"tom",30000));  
    employees.add(new Employee(3,"tim",10000));
```

```
List<Integer> list=employees.stream().map(Employee::getEmpName).map(String::length).collect(Collectors.toList());  
for(Integer i:list)  
{  
    System.out.println(i);  
}
```

Summary

- Usage of stream API
- Few intermediate methods in Stream API
- Few terminal operation on stream
- Collectors



JAVA DATABASE CONNECTIVITY



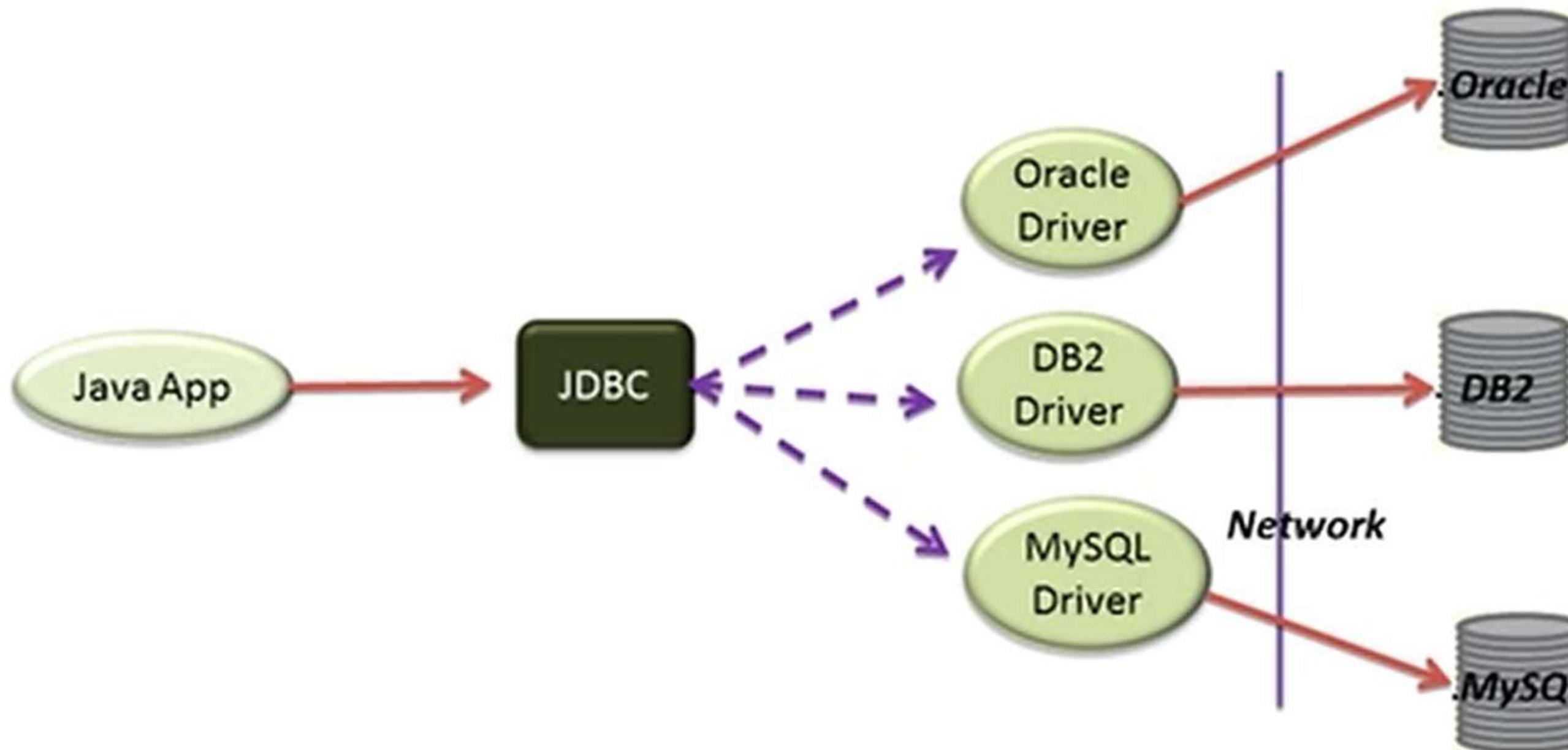
In this module You will learn

- Java Database Connectivity
- Database Connectivity Architecture
- Different Drivers
- JDBC APIs
- Database Access Steps
- Using Transaction
- Calling database procedures



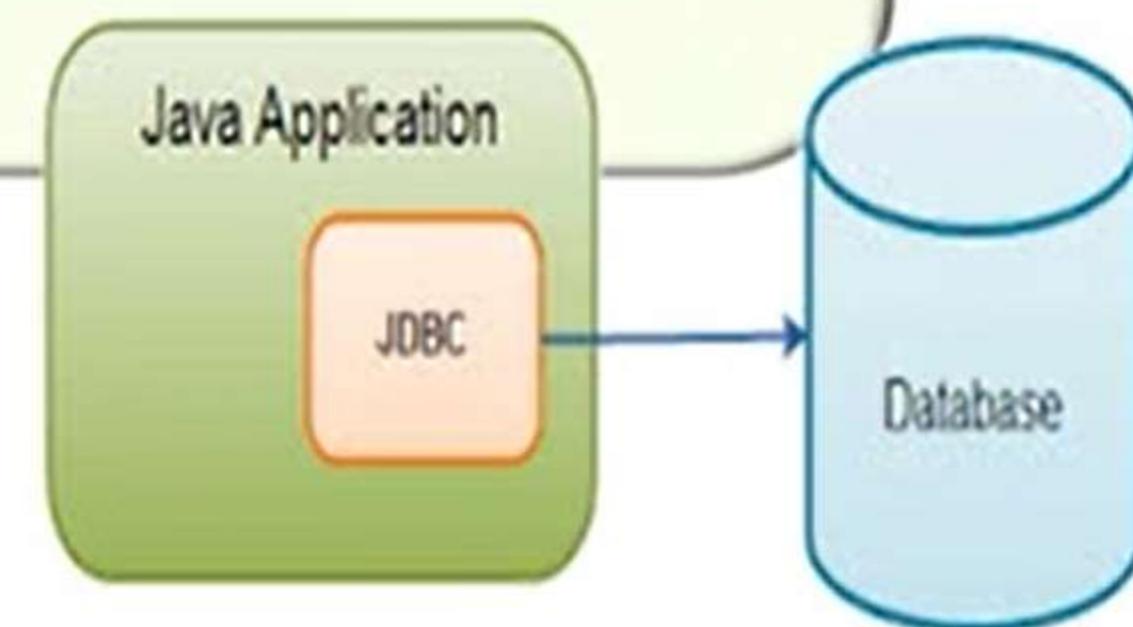
Java Database Connectivity

Using JDBC, we can persist data in the database from Java application.



JDBC

- JDBC provides the facility of connecting to any database and to manipulate data in the database through Java Programs.
- The JDBC API consists of a set of interfaces and classes written in the Java programming language.
- Interfaces and classes are present in `java.sql` package
- Java code calls JDBC library, which loads the JDBC Driver that talks to a specific database



Database Connectivity Architecture

Driver

Driver types are used to categorize the technology used to connect to the database.

JDBC driver vendor uses these types to describe how their product operates.

The different types of jdbc drivers are:

- Type 1: JDBC-ODBC Bridge driver (Bridge)
- Type 2: Native-API/partly Java driver (Native)
- Type 3: All Java/Net-protocol driver (Middleware)
- Type 4: All Java/Native-protocol driver (Pure)

Type I Driver

- JDBC-ODBC driver translates JDBC calls into ODBC calls and sends them to ODBC driver for passing to the database.
- ODBC native code and native database client code must be loaded on each client machine

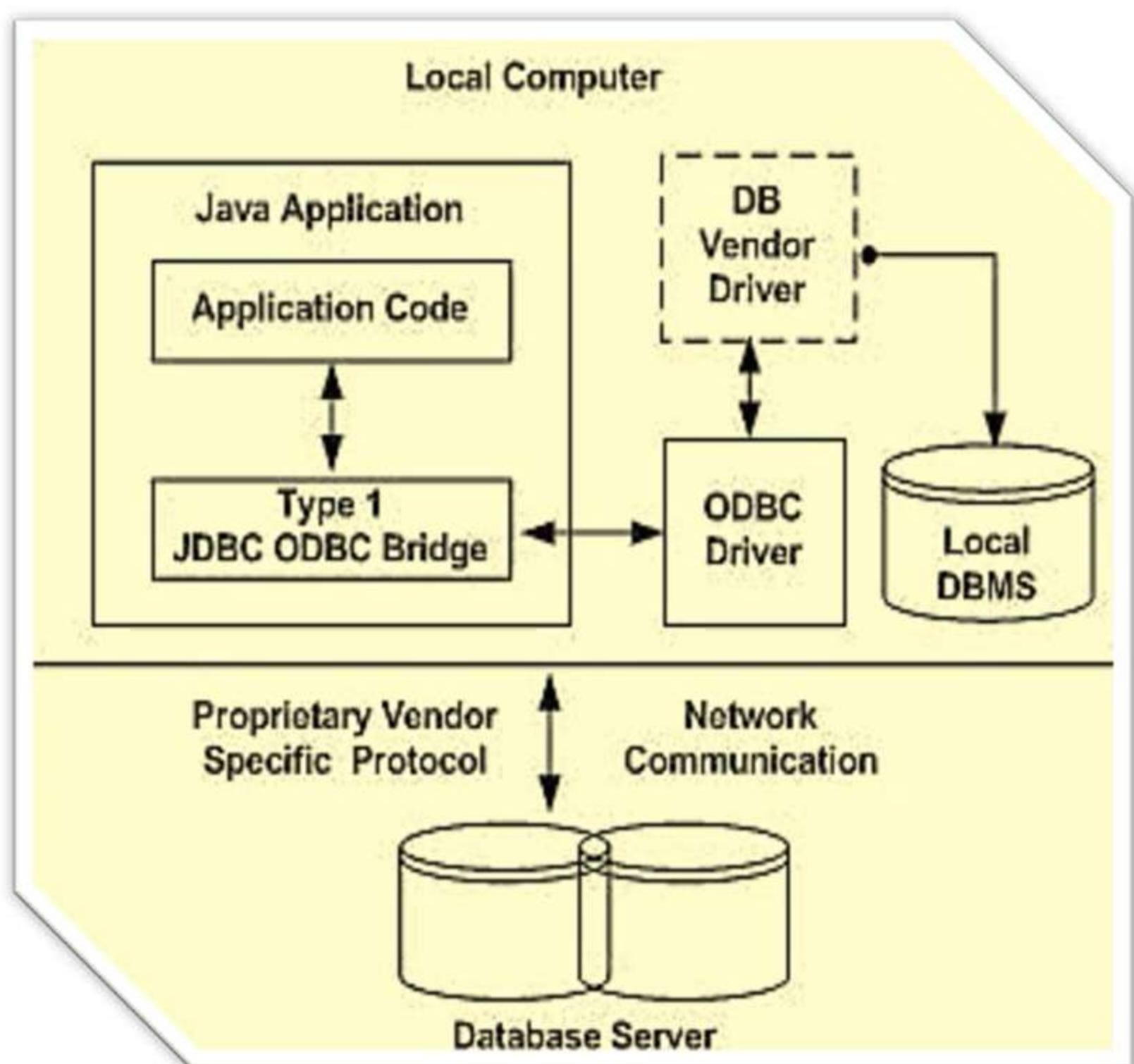


- Almost any database for which ODBC driver is installed, can be accessed.



- Performance overhead, since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native database connectivity interface.
- The ODBC driver needs to be installed on the client machine.

Type I Architecture



Type II Driver

- Native-API, partly Java technology-enabled driver
- Converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS.
- Requires that some binary code be loaded on each client machine.

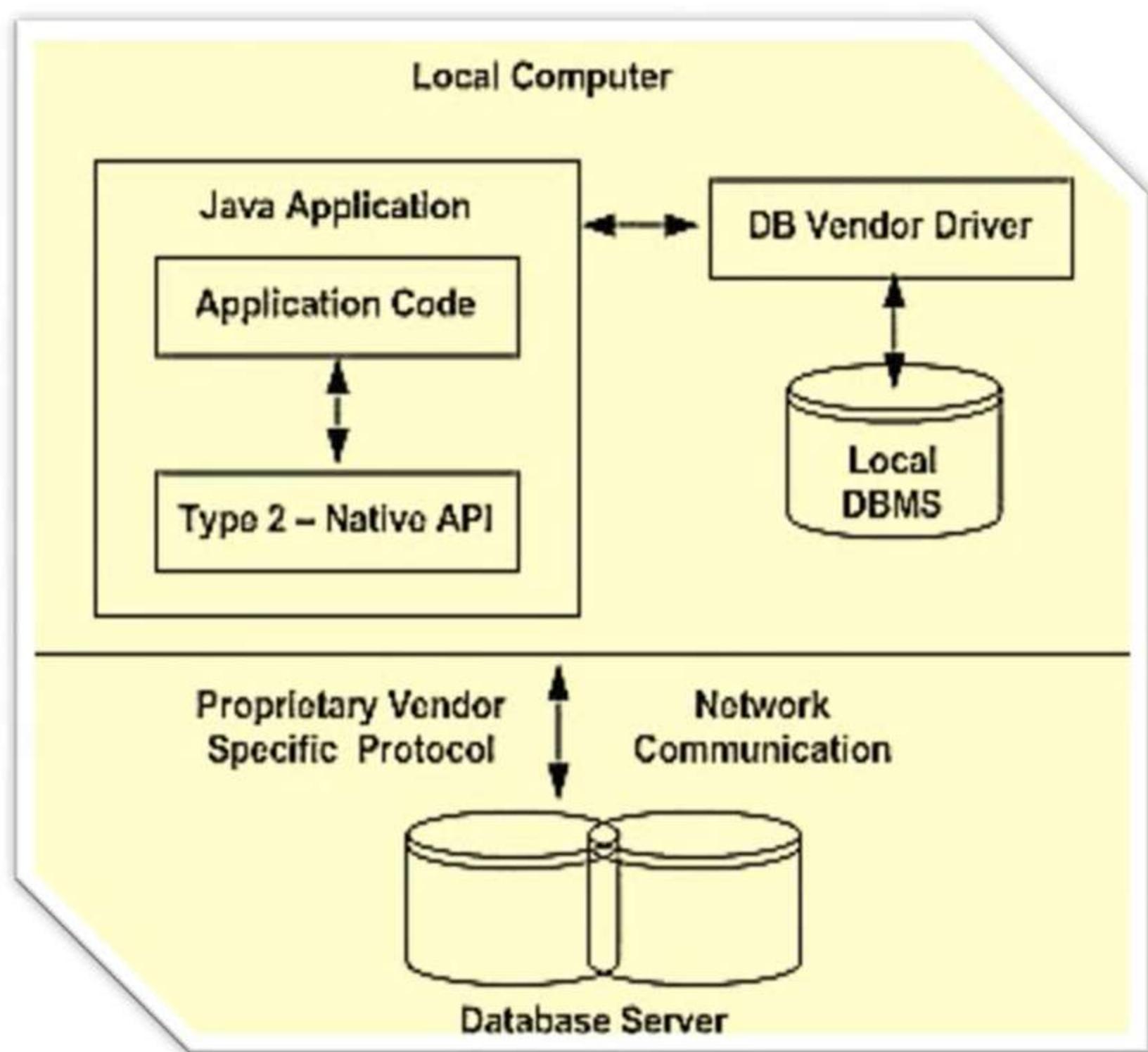


- Better performance than Type 1, since no jdbc to odbc translation is needed.



- The vendor client library needs to be installed on the client machine.
- Not all databases provide the client side library.

Type II Driver



Type III Driver

- Net-protocol, a complete Java technology-enabled driver
- Translates JDBC API calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server
- The server middleware is able to connect all of its Java technology-based clients to many different databases
- The specific protocol used depends on the vendor.
- The most flexible JDBC API alternative

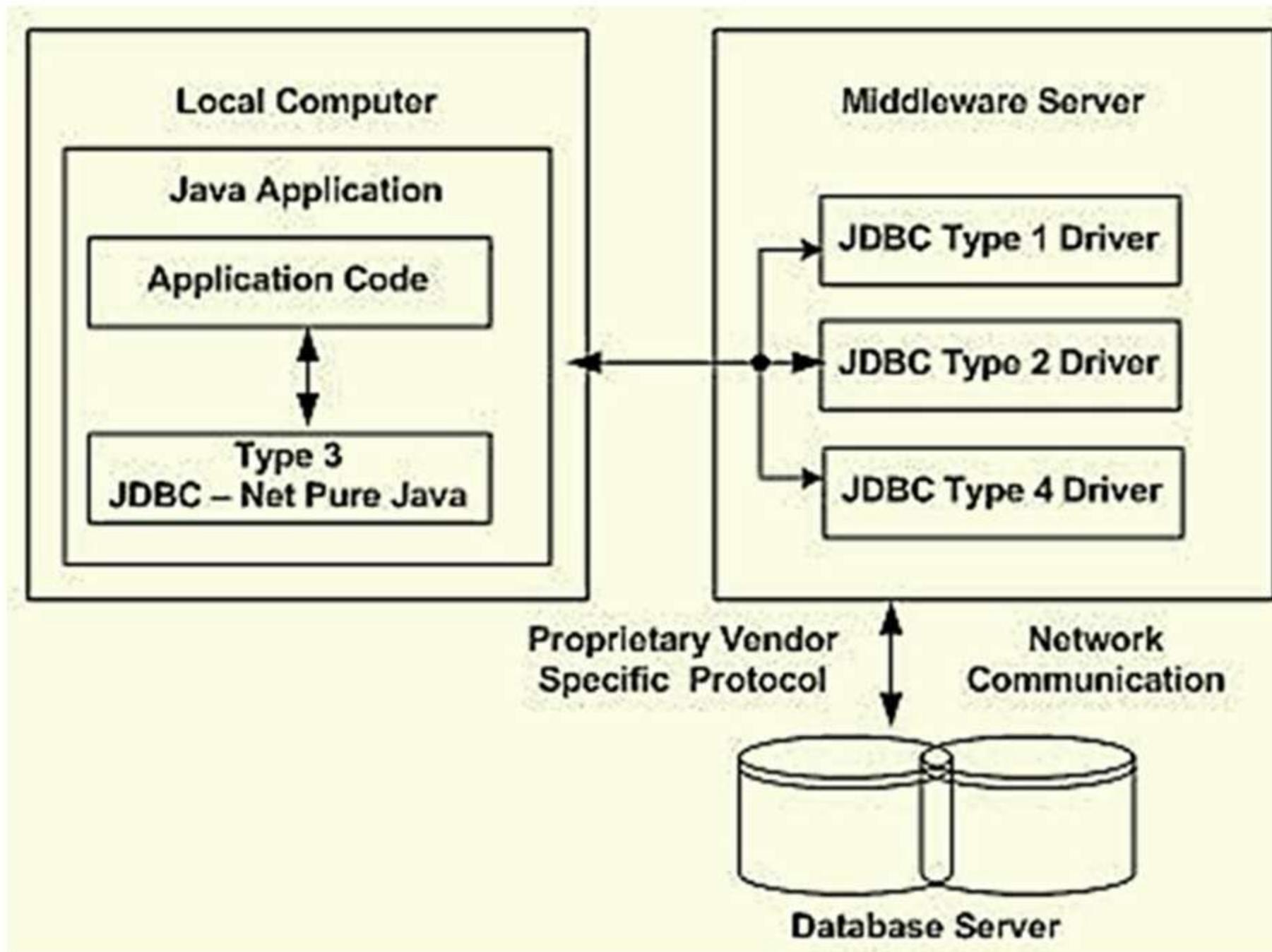


- Since the communication between the client and the middleware server is database independent, there is no need for the vendor db library on the client machine.
- At client side a single driver can handle any database.



- Requires database-specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck.

Type III Architecture

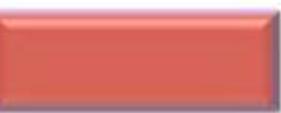


Type IV Driver

- Native-protocol, a complete Java technology-enabled driver
- Converts JDBC technology calls into the network protocol used by DBMS directly
- Allows a direct call from the client machine to the DBMS
- Several database vendors use Type IV Driver

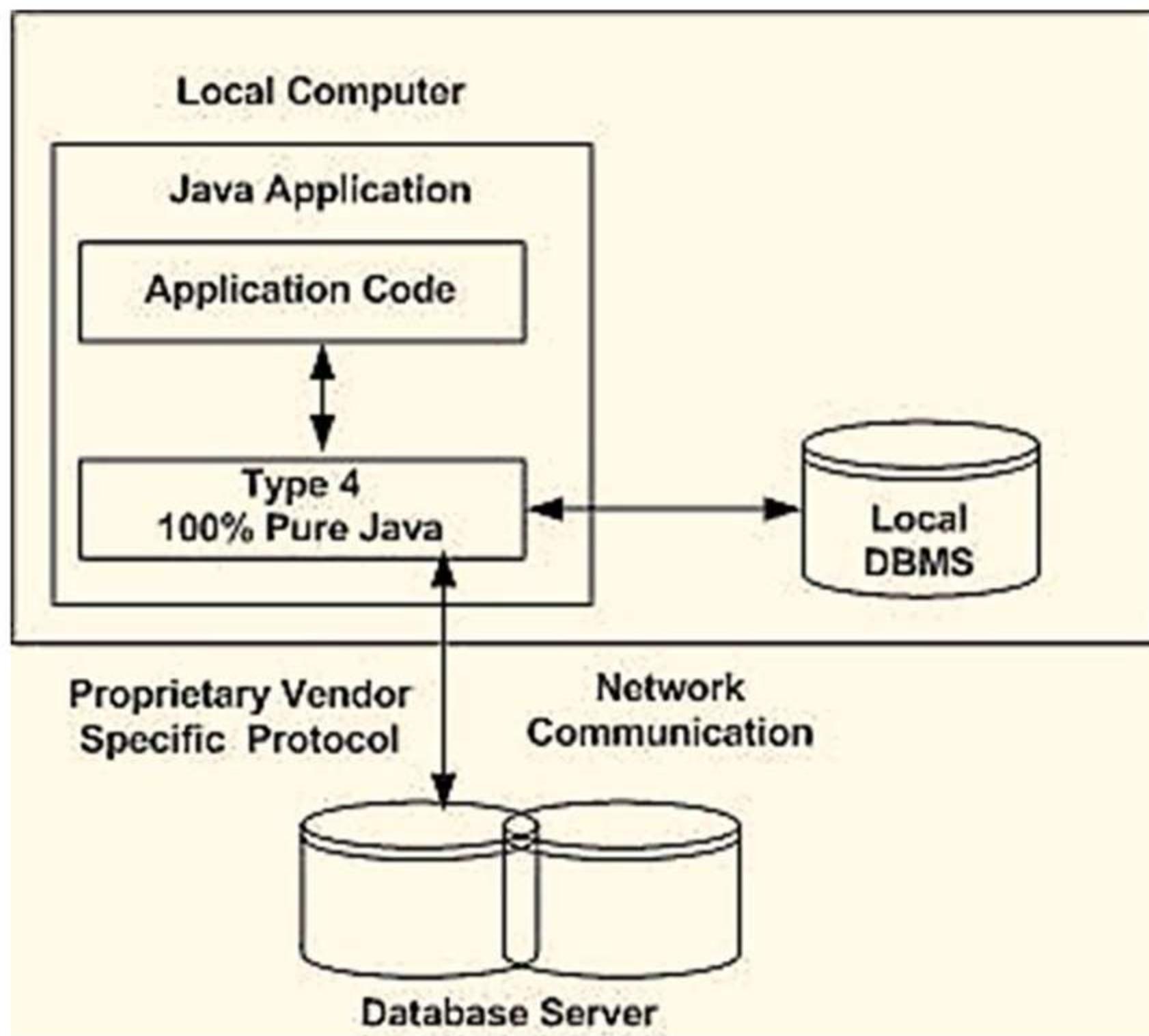


These drivers don't translate the requests to db to ODBC or pass it to the client API for the db, nor do they need a middleware layer for request indirection.



At client side, a separate driver is needed for each database

Type IV Architecture



JDBC API

DriverManager

- This class manages a list of database drivers.

Driver

- This interface handles the communications with the database server.

Connection

- Interface with the methods for contacting a database.

Statement

- Used to submit the SQL statements to the database.

ResultSet

- These objects hold data retrieved from a database after you execute an SQL query using Statement objects

SQLException

- This class handles any errors that occur in a database application.

Database Access Steps

Load the Driver

Define the Connection URL

Establish the Connection(Set
the auto commit status)

Get the statement Object

Execute the Queries

Process the Results

Close the Connection

Loading the Driver

The Driver of the Database has to be loaded before acquiring the connection.

Using Class.forName

- `Class.forName(String classname)`
- Can be used to load any java class and can also be used to load the database driver class.
- `Class.forName("oracle.jdbc.driver.OracleDriver");`

By Creating the instance of the Driver class

- The instance of the Driver class can be created as creating the instance of any class.

Making a Connection

- The connection can be acquired by using getConnection method of Driver Manager
- The overloaded methods
 - `getConnection(String url)`
 - `getConnection(String url, Properties prop)`
 - `getConnection(String url, String user, String password)`
- Example To Connect to Oracle database
 - `String url = "jdbc:oracle:thin:@oracleserver:1521:sampledatabase";`
 - `Connection connection= DriverManager.getConnection(url, "scott", "tiger");`

Making a Connection

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

JDBC Example

```
public class Employee {  
    private int id;  
    private String name;  
    private float salary;  
  
    public Employee(int id, String name, float salary) {  
        this.id = id;  
        this.name = name;  
        this.salary = salary;  
    }  
    public String toString() {  
        return "id :" + id + " name :" + name + " salary :" + salary;  
    }  
}
```

JDBC Example

```
public class EmployeeDB {  
    public static void main(String[] args) throws ClassNotFoundException,  
                                         SQLException {  
        Employee e = null;  
          
        Class.forName("oracle.jdbc.driver.OracleDriver");  
        Connection con = DriverManager.getConnection(  
            "jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");  
    }  
}
```

Creating Statement Object

- Used to create SQL statement
- Object of the statement can be acquired with the help of Connection Object
- Useful when static SQL statements are used at runtime.

```
Statement st=connection.createStatement();
```

Methods in Statement

- `int executeUpdate(String SQL)`
 - Returns the number of rows affected on executing the SQL statement.
 - Used for DML operation – insert, update, delete
- `ResultSet executeQuery(String SQL)`
 - Returns a ResultSet object.
 - Used for executing select statement
- `boolean execute(String SQL)`
 - Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
 - Can be used for any DB operation

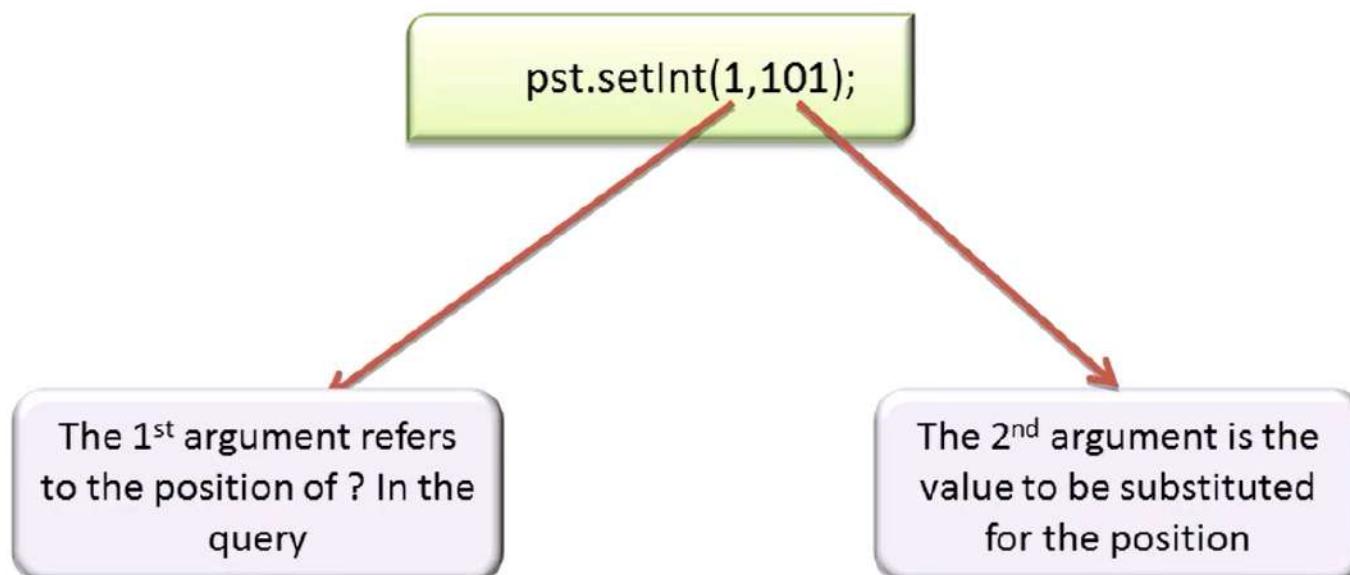
Prepared Statement

- The *PreparedStatement* interface extends the Statement interface which comes with added functionality.
- The Prepared Statement accepts parameters for which values can be assigned at runtime.
- All parameters in JDBC are represented by the “?” symbol, which is known as the parameter marker.

```
PreparedStatement pst = connection.prepareStatement("select * from  
employee where id = ?");
```

Prepared Statement

- The **setXXX()** method binds the user value to the parameters, where **XXX** represents the Java data type of the user value.



JDBC Example

gets PreparedStatement

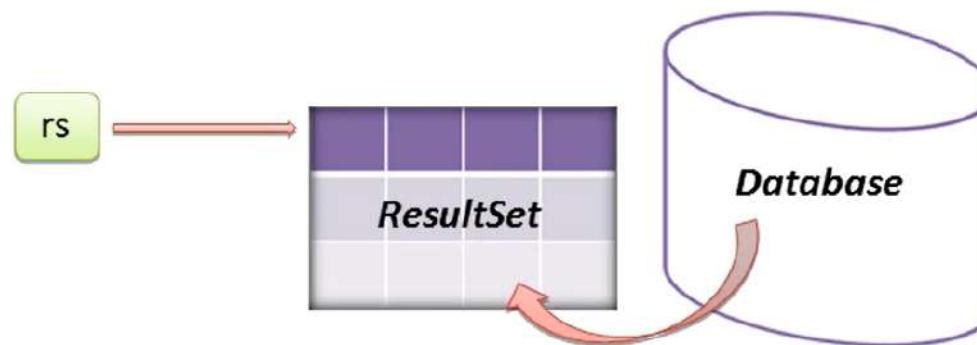
```
PreparedStatement ps=con.prepareStatement("insert  
          into employee values(?, ?, ?);  
  
ps.setInt(1, 1);  
ps.setString(2, "Mini");  
ps.setFloat(3, 30000);  
ps.executeUpdate();
```

execute the
statement

Working with ResultSet

- Resultset interface represents the result set of a database query
- It is available in java.sql package
- A ResultSet object contains a cursor, which points to the current row in the result set.

```
Statement st=connection.createStatement();
ResultSet rs= st.executeQuery("select * from employee");
```



Methods in ResultSet

- public boolean next()
 - Moves the cursor to the next row.
 - This method returns false if there are no more rows in the result set

```
Statement st=connection.createStatement();
ResultSet rs= st.executeQuery("select * from employee ");
while(rs.next())
{
    ....
    ....
}
```

While loop will get executed
until all the employee details
have been read

Methods to retrieve result Data

- `public XXX getXXX(String ColumnName)`
 - Returns the value from the given column
- `public XXX getXXX(int columnIndex)`
 - Returns the value from the given column index
 - Column index is the order of columns in the select query
 - Column index starts from 1

Present for all Java data types
except for char

JDBC Example

```
Statement s = con.createStatement();
ResultSet rs = s.executeQuery("select * from employee1 where id =1");

if(rs.next()){
    int id = rs.getInt(1);
    String name = rs.getString(2);
    float salary = rs.getFloat(3);
    String desg = rs.getString(4);
    e=new Employee(id, name, salary, desg);
}

System.out.println(e);
```

get Statement

Process the
ResultSet

Close the connection

- The close() method of Connection interface is used to close the connection.
- By closing connection object, statement and ResultSet will be closed automatically.

```
connection.close();
```

It is **strongly recommended** that an application explicitly commits or rolls back an active transaction prior to calling the close method.

JDBC – Complete Example

Assume a class Employee with attributes id, name and salary with constructor, getter, setter and toString method

```
public class DBHandler {  
    Connection con=null;  
    public Connection getConnection(){  
        try {  
            Class.forName("oracle.jdbc.driver.OracleDriver");  
  
            con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",  
                                         "system","oracle");  
        }  
        catch(ClassNotFoundException | SQLException e) {  
            System.out.println("Check with the administrator");  
        }  
        return con;  
    }  
    public void closeConnection() {  
        try {      con.close(); }  
        catch (SQLException e) {System.out.println("Check with the administrator");}  
    }  
}
```

JDBC – Complete Example

//To add an Employee to database

```
public static void addEmployee(Employee empObj){  
    DBHandler db=new DBHandler();  
    Connection con=db.getConnection();  
    try {  
        Statement stmt=con.createStatement();  
        String query="insert into employee (id,name,salary)  
                    values("+empObj.getId()+","+empObj.getName()+",'"+  
                    empObj.getSalary()+"');  
        int status = stmt.executeUpdate(query);  
        if(status>0)  
            System.out.println("Added successfully");  
    }  
    catch(SQLException e){  
        System.out.println("Not Added.. Contact Administrator");  
    }  
    db.closeConnection();  
}
```

JDBC – Complete Example

//To update an employee

```
public static void updateEmployee(int id, float salary){  
    DBHandler db=new DBHandler();  
    Connection con=db.getConnection();  
    String query="update employee set salary=? where id=?";  
    try{  
        PreparedStatement preparedStmt=con.prepareStatement(query);  
        preparedStmt.setFloat(1,salary);  
        preparedStmt.setInt(2,id);  
        int status = preparedStmt.executeUpdate();  
        if(status>0)  
            System.out.println("Updated successfully");  
        else  
            System.out.println("Not updated");  
    }  
    catch(SQLException e){System.out.println("Not updated.. Contact Administrator");}  
    db.closeConnection();  
}
```

JDBC – Complete Example

//To retrieve Employee using ResultSet

```
ArrayList<Employee> empList=new ArrayList<Employee>();
String query="select id,name,salary from employee";
try{
    PreparedStatement preparedStmt = con.prepareStatement(query);
    ResultSet rs=preparedStmt.executeQuery();
    while(rs.next()) {
        int id=rs.getInt(1); //retrieval using column index
        String name=rs.getString("name"); //retrieval using column name
        float salary=rs.getFloat(3);
        Employee emp=new Employee(id, name, salary);
        empList.add(emp);
    }
}
catch(SQLException e) {
    System.out.println("Not updated.. Contact Administrator");
}
db.closeConnection();
```



Properties class

- The properties object holds the set of properties that can be saved to the persistent storage or loaded from the persistent storage
- Properties class is available in `java.util` package
- Each of the property is represented as key-value pair .
 - Key and value is of type `String`.
- As a good programming practice, all the configuration details (ex: driver class name, url,username etc) can be placed separately in the properties file and fetched inside the application using the Properties class.

Properties file

- Holds the data as key – value pair
- properties file has .properties as extension
- Used to store the details that has frequent changes/updations.
- Advantage
 - No Recompilation
 - If any configuration is changed later, then the java file need not be recompiled
- Example:db.properties

#holds the configuration details

```
db.driver=oracle.jdbc.driver.OracleDriver  
db.url=jdbc:oracle:thin:@localhost:oracle  
db.username=john  
db.password=xxx
```

Steps to read the data from the properties file

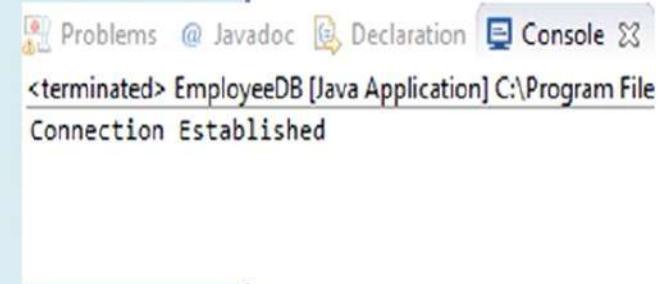
- Create the property file with .properties extension
- Store the data as key-value pair
- Load the properties file in the java program
 - Create the object for Properties class
 - `java.util.Properties prop = new java.util.Properties();`
 - Load the properties file
 - `FileInputStream fis=new FileInputStream("path of the .properties file");`
 - Example:
 - »`FileInputStream fis=new FileInputStream("db.properties");`
 - `prop.load(fis);`

Steps to read data from the properties file

- To get the property value
 - `prop.getProperty("key");`
 - Example:
 - If the db.properties file contains a key called db.url then to access it in the java program
 - `prop.getProperty("db.url");`

JDBC Example with Properties class

```
public class EmployeeDB{  
    public static void main(String[] args) throws ClassNotFoundException,SQLException,IOException{  
        Employee e = null;  
  
        FileInputStream input=new FileInputStream("db.properties");  
        Properties prop=new Properties();  
        prop.load(input);  
        String driver=(String)prop.get("db.driver");  
        String url=(String)prop.get("db.url");  
        String userName=(String)prop.get("db.username");  
        String password=(String)prop.get("db.password");  
        Class.forName(driver);  
        Connection con= DriverManager.getConnection(url,userName,password);  
        System.out.println("Connection Established");  
    }  
}
```



Transaction Management

To commit a transaction

- `connection.commit();`

To rollback a transaction

- `connection.rollback();`

JDBC Connection by default follows auto commit mode.

- `connection.setAutoCommit(false);`

Using Transactions

- Transactions should be managed
 - To increase performance,
 - to maintain the integrity of the stored procedures and
 - to use distributed transactions
- Transaction control the changes applied to a database.
 - Treats an SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.
- To commit the changes applied, invoke the `commit()` method on connection object as - **conn.commit();**
- To roll back updates to the database made invoke the `rollback()` method on the connection object as - **conn.rollback();**

ResultSet MetaData

- The meta information about ResultSet can be obtained using ResultSetMetaData
- ResultSetMetaData is used to get the column name, total number of columns, column type, table Name etc.

```
Statement st=connection.createStatement();
ResultSet rs= st.executeQuery("select * from employee");
ResultSetMetaData rsm= rs.getMetaData();
```

ResultSet MetaData - Methods

Method	Description
public int getColumnCount()	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)	it returns the column name of the specified column index.
public String getColumnTypeName(int index)	it returns the column type name for the specified index.
public String getTableName(int index)	it returns the table name for the specified column index.

```
System.out.println("Total columns: "+rsm.getColumnCount());
System.out.println("Column Name of 1st column: "+rsm.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsm.getColumnTypeName(1));
```

CallableStatement

- CallableStatement interface is used to call the **stored procedures and functions**.
- CallableStatement instance is created using prepareCall() method in Connection interface.

```
CallableStatement stmt=conn.prepareCall("{call employee_insert(?,?)}");
```

Here employee_insert is the name of the function / procedure and ? Indicates the arguments. This procedure takes two arguments.

CallableStatement - Example

```
import java.sql.*;
public class Sample {
public static void main(String[] args) {
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe", "system","oracle");
        CallableStatement stmt=con.prepareCall("{call emp_insert(?,?)}");
        stmt.setInt(1,1011);
        stmt.setString(2,"Adith");
        stmt.execute();
        System.out.println("Added successfully");
    }
    catch(ClassNotFoundException e) {
        System.out.println("Exception "+e.getMessage());
    }
    catch(SQLException e){
        System.out.println("Exception "+e.getMessage());
    }
}
```

CallableStatement

Procedure

```
create or replace procedure emp_insert
(id IN NUMBER, name IN VARCHAR2)
is
begin
  insert into employee (id,name) values(id,name);
end;
/
```

Successfully added in the database

Summary

- Java Database Connectivity
- Database Connectivity Architecture
- Different Drivers
- JDBC APIs
- Database Access Steps
- Using Transaction
- Calling database procedures



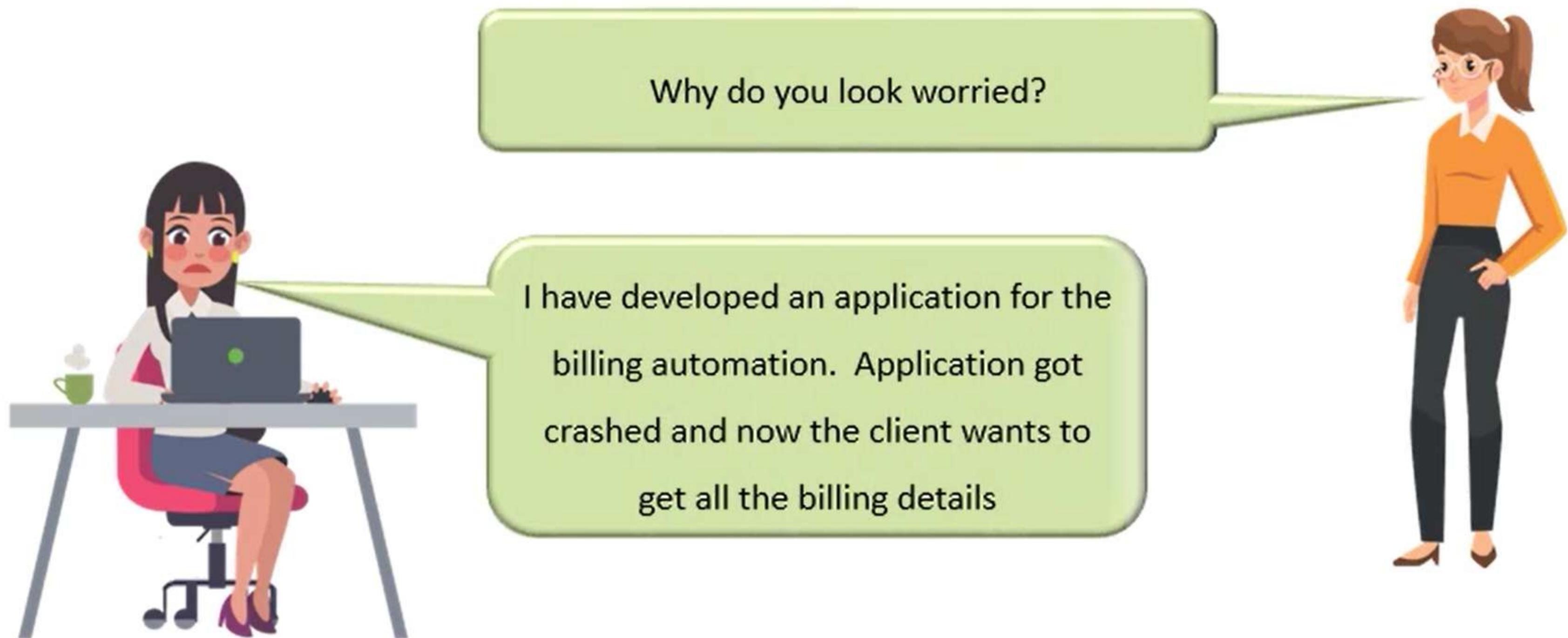
I/O



In this module you will learn about

- What are Files
- What are Streams
- Understand different types of Streams
- Understand Byte oriented Streams
- Understand the types of Byte oriented Streams





Let us see the conversation between the Developer and his friend



I have developed an application for the
billing automation. Application got
crashed and now the client wants to
get all the billing details

Data's can be made to persist in
the hard disk even after the
application has crashed using
File Handling mechanism in java



Let us see the conversation between the Developer and his friend

The File class

Files in java are used to store the data.

Java Program can read and write data from and to files, respectively.

Files are classes from `java.io` package which helps in

- Creating file objects
- Manipulating file objects



Creating Files in Java

File(String pathname)

```
File f = new File("sample.txt");
f.createNewFile();
```

File(String parent, String child)

```
File f1 = new File("FileDemo", "sample1.txt");
f1.createNewFile();
```

File(File parent, String child)

```
File f2 = new File("filedemo1");
f2.mkdir();
File f3 = new File(f2,"sample2.txt");
f3.createNewFile();
```

Creates an Java File Object

Creates file in the current folder

} Creates file in the given folder name
which is existing.

} Creates folder in the current directory

Creates file in the folder referred by the
file reference

File Methods

File information

`String getName()`

`String getPath()`

`String getAbsolutePath()`

`String getParent()`

`long lastModified()`

`long length()`

File modification

`boolean renameTo(File newName)`

`boolean delete()`

Directory utilities

`boolean mkdir()`

`String[] list()`

File tests

`boolean exists()`

`boolean canWrite()`

`boolean canRead()`

`boolean isFile()`

`boolean isDirectory()`

`boolean isAbsolute();`

`boolean isHidden();`

File Operation

```
// Create a directory  
  
System.out.println("Creating directory...");  
  
String fileName = "myDir";  
  
File dir = new File(fileName);  
  
dir.mkdir();  
  
System.out.println(fileName + (dir.exists()? " exists": " does not exist"));  
  
System.out.println(fileName + " is a " + (dir.isFile()? "file." :"directory."));  
  
System.out.println("-----");
```

Creates a directory

Checks whether given directory / file exists in the current location

Checks whether given file reference is a file or directory

File Operation

```
// creating file inside directory  
  
fileName = "employee.txt";  
  
File fn = new File(dir,fileName);  
  
fn.createNewFile();  
  
System.out.println(fn.getName() +" is created");  
  
if (dir.isDirectory()) {  
  
    String content[] = dir.list();  
  
    System.out.println("The content of this directory:");  
  
    for (int i = 0; i < content.length; i++) {  
  
        System.out.println(content[i]);  
  
    }  
    System.out.println("-----");  
}
```

Gets the name of file / dir

Checks whether the given File reference is a directory

Lists all the file / directory inside the current directory

File Operation

```
if (!dir.canRead()) {  
    System.out.println(  
        dir.getName() +  
        " is not readable.");  
}
```

true if the file specified by this pathname exists and can be read by the application else it is false

```
System.out.println(dir.getName() + " is " + dir.length() + " bytes long.");
```

Length of the file in terms of bytes

```
System.out.println(dir.getName() + " was modified " + dir.lastModified() + " milliseconds back.");
```

```
//deleting directory
```

```
dir.delete();
```

Deletes the file / directory

```
System.out.println(dir.getName() + " is deleted");
```

Returns time that the file was last modified.

File Operation

```
<terminated> FileDemo [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_21\bin\javaw.exe (May 15, 2015 12:30:02 PM)
Creating directory...
myDir exists
myDir is a directory.
-----
employee.txt is created
The content of this directory:
employee.txt
-----
myDir is 0 bytes long.
myDir was modified 1431669752742 milliseconds back.
myDir is deleted
```

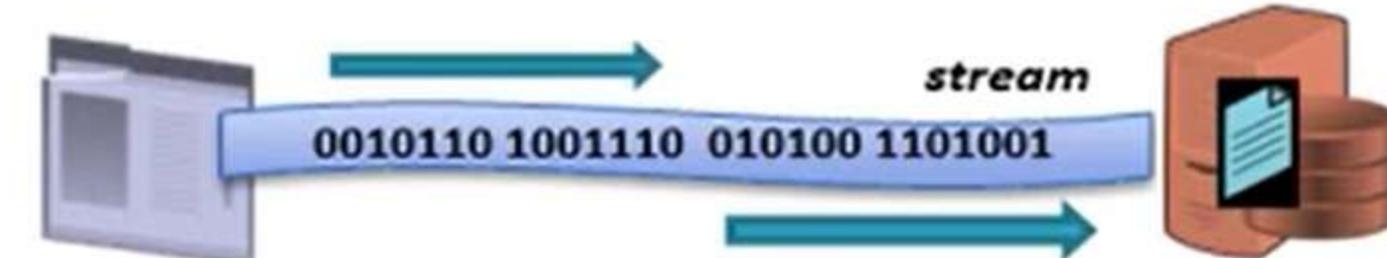
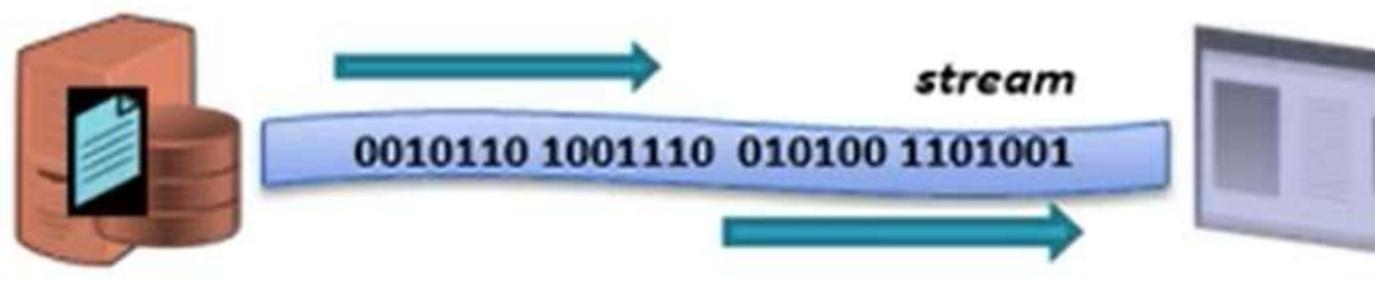
Streams

Streams are ordered sequence of data that have source or destination

Streams are used to read data from a source location or write data to any destination

Source can be keyboard, file or disk

Destination can be console, file or disk



Types of Streams

Two type of Streams are

Input streams

Input Streams are used to read data from any external source in to the java application.

Byte Stream – Reads data in byte format

Character Stream – Reads data in character format

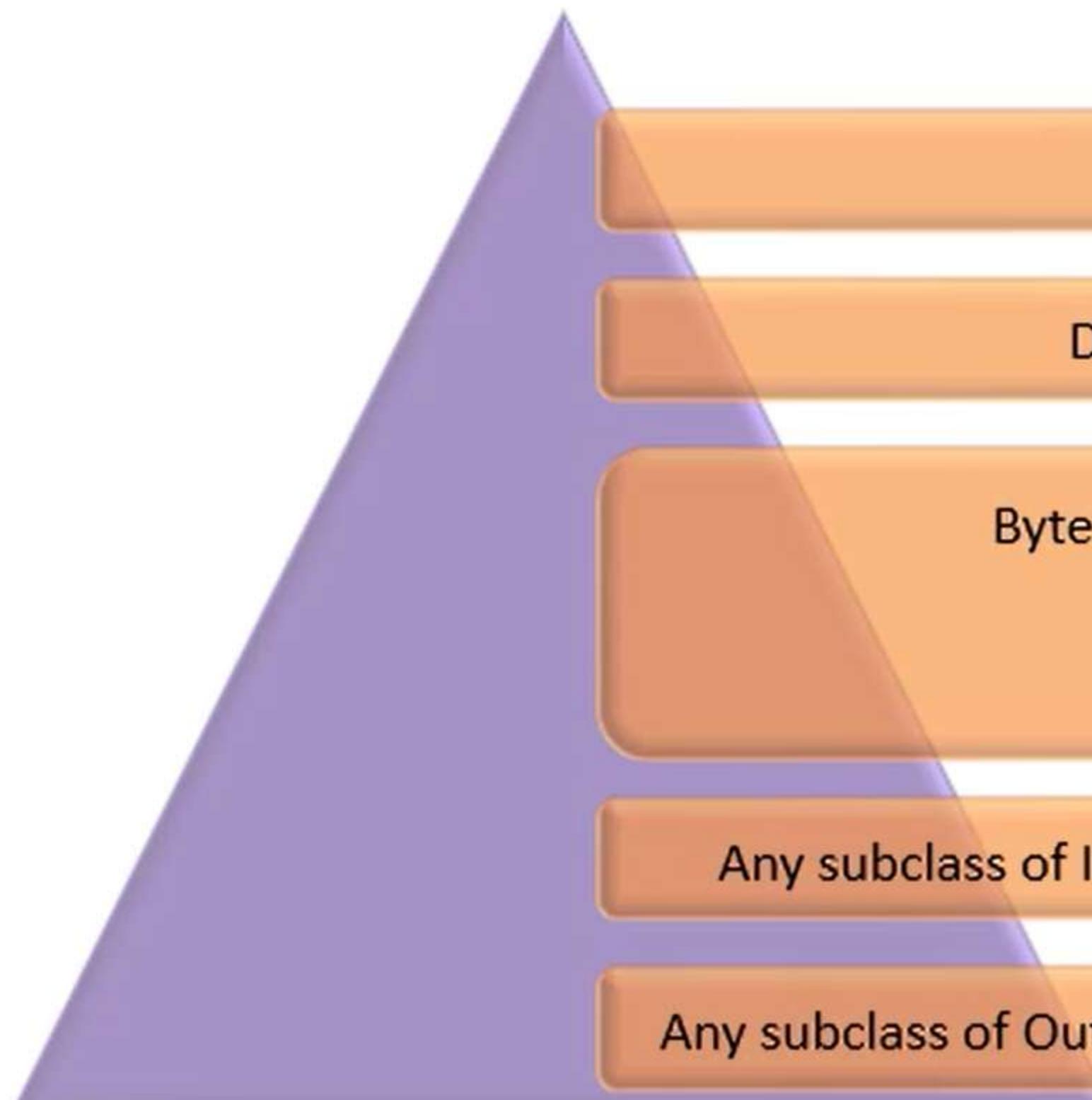
Output streams

Output Streams are used to write data from java application to the external destination.

Byte Stream – Writes data in byte format

Character Stream – Writes data in character format

Byte Oriented Streams



It can work only with bytes.

Does not support unicode character set

Byte Oriented Stream has two abstract classes

- InputStream
- OutputStream

Any subclass of InputStream is used to read the bytes from any source

Any subclass of OutputStream is used to write the bytes to any destination.

InputStream class

InputStream is an abstract class because of its abstract method `read()`

`read()` throws `IOException`

The child classes of `InputStream` are supposed to override the `read()` method and provide its own implementation.

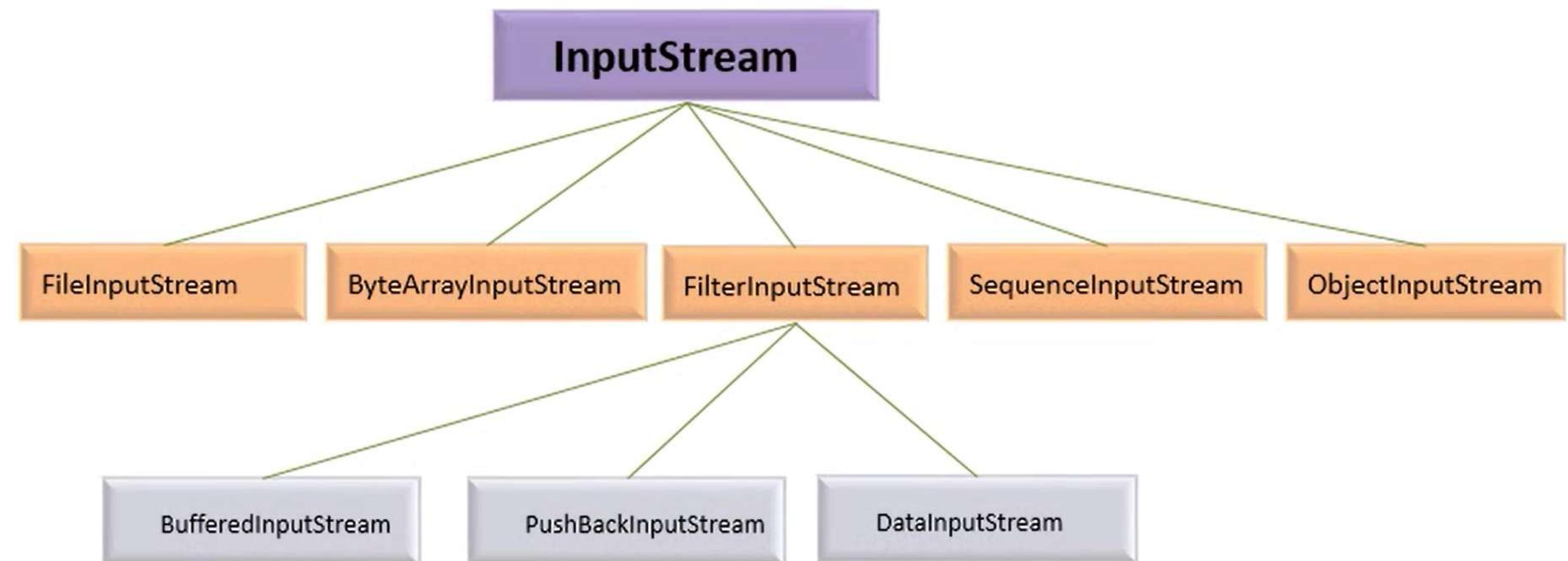
Few of the subclasses of `InputStream` class are:

- `FileInputStream`
- `BufferedInputStream`
- `ObjectInputStream`

Few methods of InputStream

Method Name	Description
int read()	Reads the next byte of data from the input stream. The value byte is returned as an int in the range of 0 to 255. If the end of stream is reached and no more bytes to read then -1 is returned.
int read(byte[] arr)	Reads “arr.length” number of bytes from the input stream and stores them in to the byte array arr.
void close()	Closes the input stream and releases all the resources associated with this stream
int available()	Returns the available number of bytes that can be read from this input stream
long skip(long n)	Skips over and discards n bytes of data from this input Stream
void reset()	Repositions this stream to the mark position, marked by the mark() method
void mark(int limit)	Marks the current position in this input stream. The limit indicates the number of bytes to be read before the reset is called

InputStream Hierarchy



Note:

In this session we will focus on `FileInputStream` and `BufferedInputStream` classes. Kindly refer to <https://www.oracle.com/> for other stream API

FileInputStream

FileInputStream is used to read the contents of the file as a stream of bytes

read() of FileInputStream is used to read the contents of file byte by byte

Two constructors of FileInputStream are

- `FileInputStream(String fileName)`
- `FileInputStream(File fileRef)`
 - Both these constructors throws an exception called `FileNotFoundException` which is derived from `IOException`.
 - Exception is thrown when it is unable to find the file
- Since `FileNotFoundException` is a checked exception, it has to be handled with in try –catch –finally block or using `throws`

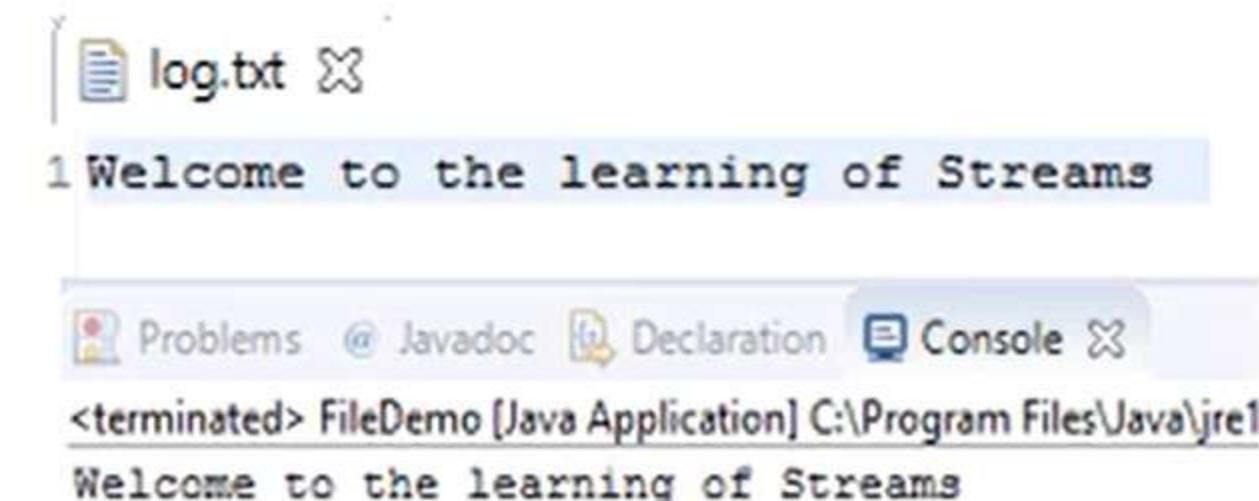
FileInputStream Example

```
import java.io.FileInputStream;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        FileInputStream fis=null;
        try {
            fis=new FileInputStream("\\\\IODemo\\\\log.txt");
            while((ch=fis.read()) != -1) {
                System.out.print((char)ch);
            }
        } catch(FileNotFoundException e) {
            //log the error
        } catch (IOException e) {
            //log the error
        }
        finally {
            try {
                fis.close();
            } catch (IOException e) {
                //log the error
            }
        }
    }
}
```

Creates a FileInputStream object with the fileName as the string

read() reads a single byte at a time. When no bytes to read it returns -1

Always close the stream in finally block. Close also throws an exception.so enclose within try-catch



BufferedInputStream

It creates an internal byte array called “buffer”.

The bytes that are read from the input stream are stored in the buffer.

Since it uses buffer the performance is much faster than other streams

BufferedInputStream buffers the data from other input streams like FileInputStream, DataInputStream etc.

Data will be loaded into the buffer based on the size specified when creating the buffer.

Now the data will be read from the buffer and hence the performance is much faster.

Default buffer size is 2048 bytes.

BufferedInputStream

BufferedInputStream also has a constructor that allows to specify the buffer size

Important methods of
BufferedInputStream

`mark()` → It creates a marker in the input stream

`reset()` → Repositions this stream to the position at the time the mark method was last called on this input stream.

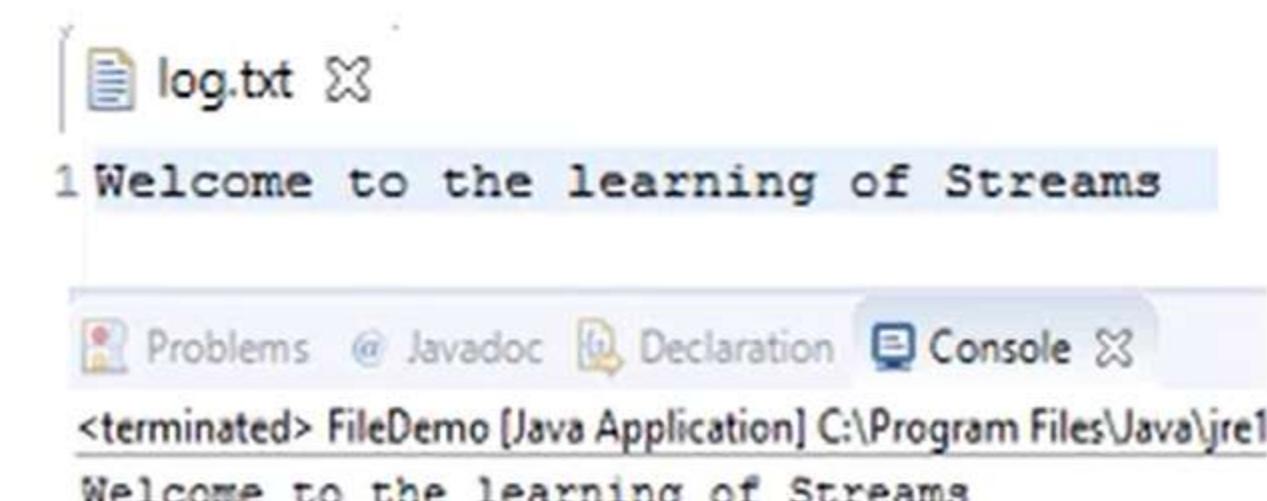
BufferedInputStream Example

```
import java.io.BufferedReader;  
public class FileDemo {  
    public static void main(String[] args) {  
        int ch=0;  
        BufferedReader bis=null;  
        FileInputStream fis=null;  
        byte[] buffer=new byte[1024];  
        String str=new String();  
        try{  
            bis = new BufferedReader(new FileInputStream("\\\\IODemo\\\\log.txt"));  
            while((ch=bis.read(buffer))!=-1) {  
                str=new String(buffer,0,ch);  
                System.out.print(str);  
            }  
        }catch(FileNotFoundException e) {  
            //log the error  
        } catch (IOException e) {  
            //log the error  
        }  
        finally  
        {  
            try {  
                bis.close();  
            } catch (IOException e) {  
                //log the error  
            }  
        }  
    }  
}
```

Creates a buffer of size 1024

FileInputStream is wrapped
inside the
BufferedInputStream

Iterates through the
stream and displays the
data in the console



OutputStream

OutputStream represents writing the output stream of bytes to an external destination like network, socket, console, file

OutputStream is an abstract class in java.io package

OutputStream is abstract due to abstract method write()

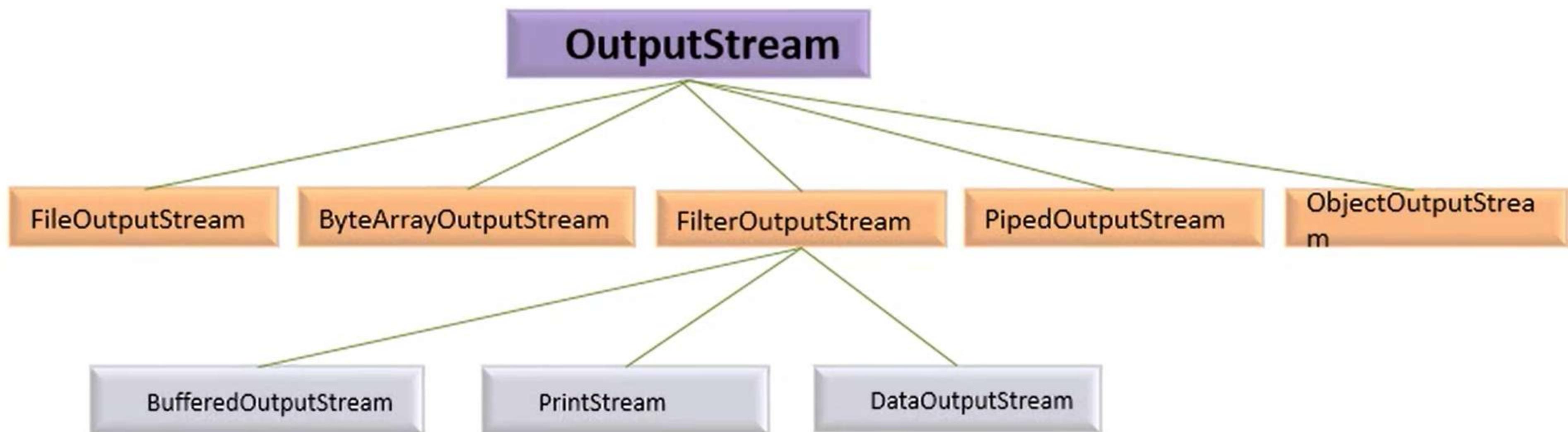
Examples for OutputStream:

- FileOutputStream, BufferedOutputStream, ObjectOutputStream

OutputStream methods

Method	Description
void write(int b)	Writes the specified byte to this output stream.
void write(byte[] barr)	Writes barr.length bytes from the specified byte array to this output stream
void write(byte[] barr, int offset,int length)	Writes length bytes from the specified byte array starting at offset off to this output stream
void flush()	Flushes this output stream and forces any buffered output bytes to be written out
void close()	Closes the stream and releases the resources

OutputStream Hierarchy



Note:

This session focuses on FileOutputStream and ObjectOutputStream. Kindly refer to <https://www.oracle.com/> for other stream API

FileOutputStream Example

```
import java.io.*;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        FileOutputStream fos=null;
        String str="Welcome to Streams";
        try{
            fos = new FileOutputStream("\\\\IODemo\\\\log.txt");
            byte barr[]={str.getBytes()};
            fos.write(barr);
            fos.flush();
        }catch(FileNotFoundException e) {
            //log the error
            e.printStackTrace();
        } catch (IOException e) {
            //log the error
            e.printStackTrace();
        }
        finally
        {
            try {
                fos.close();
            } catch (IOException e) {
                //log the error
            }}}
```

FileOutputStream object is created with the file name as string. By default the contents are overwritten, if the contents needs to be appended in the file then the second argument true should be given

String is converted to byte array and the contents are written into the file

BufferedOutputStream

Uses the buffer to hold the data to be written in to the file

BufferedOutputStream writes the data in to the destination only when the buffer is utilized fully

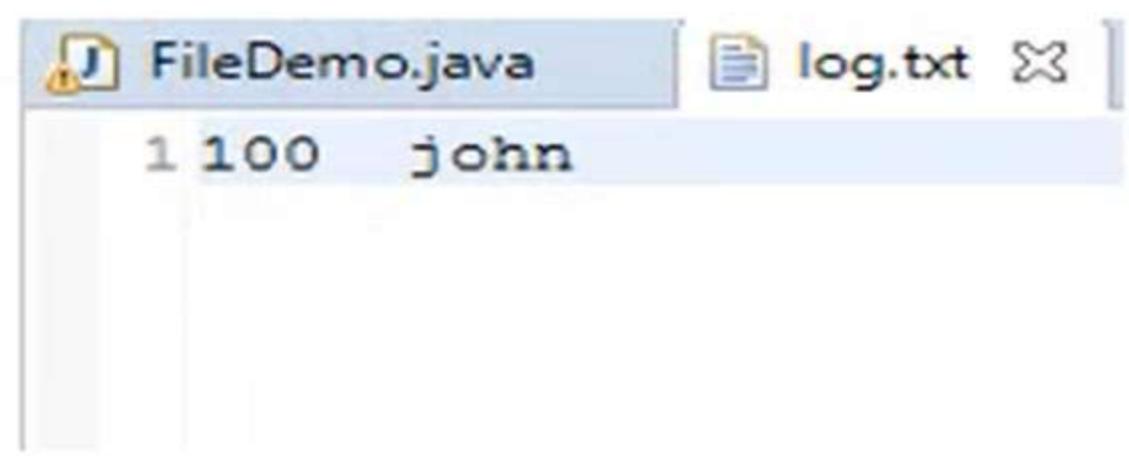
Improves the performance by reducing the number of times the system writes data in the actual location

```
BufferedOutputStream bos=new BufferedOutputStream(new FileOutputStream("c:\log.txt"),  
2048);
```

- The above line creates an object for BufferedOutputStream with a buffer size of 2 KB to write data to the specified file

BufferedOutputStream Example

```
import java.io.*;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        BufferedOutputStream bos=null;
        FileOutputStream fos=null;
        try{
            fos = new FileOutputStream("\\\\IODemo\\\\log.txt");
            bos=new BufferedOutputStream(fos);
            Scanner s=new Scanner(System.in);
            System.out.println("Enter the customer id");
            int custId=s.nextInt();
            System.out.println("Enter the customer name");
            String custName=s.next();
            String record= custId+" "+custName;
            bos.write(record.getBytes());
        }catch(FileNotFoundException e) {
            //log the error
        } catch (IOException e) {
            //log the error
        }
        finally
        {
            try {
                bos.close();
            } catch (IOException e) {
                //log the error
            }}}}}
```



FileDemo.java

log.txt

1 100 john

Summary

- What are Files
- What are Streams
- Understand different types of Streams
- Understand Byte oriented Streams
- Understand the types of Byte oriented Streams





CHARACTER STREAMS



In this module you will learn

- What is Character Stream
- The types of Character Stream
- How to read and write data using the Readers and Writers



Character Stream

Character Stream is used to read and write character data (UNICODE CHARACTER).

Character Stream has two abstract classes

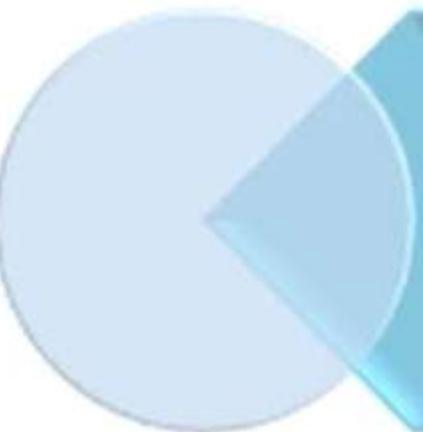
Reader:

Abstract class that has the read(), reads the character from any source.

Writer:

Abstract class that has the write(), writes character to any destination

Reader class



Reader class reads character(16 bits) from any source



Reader class is available in java.io package

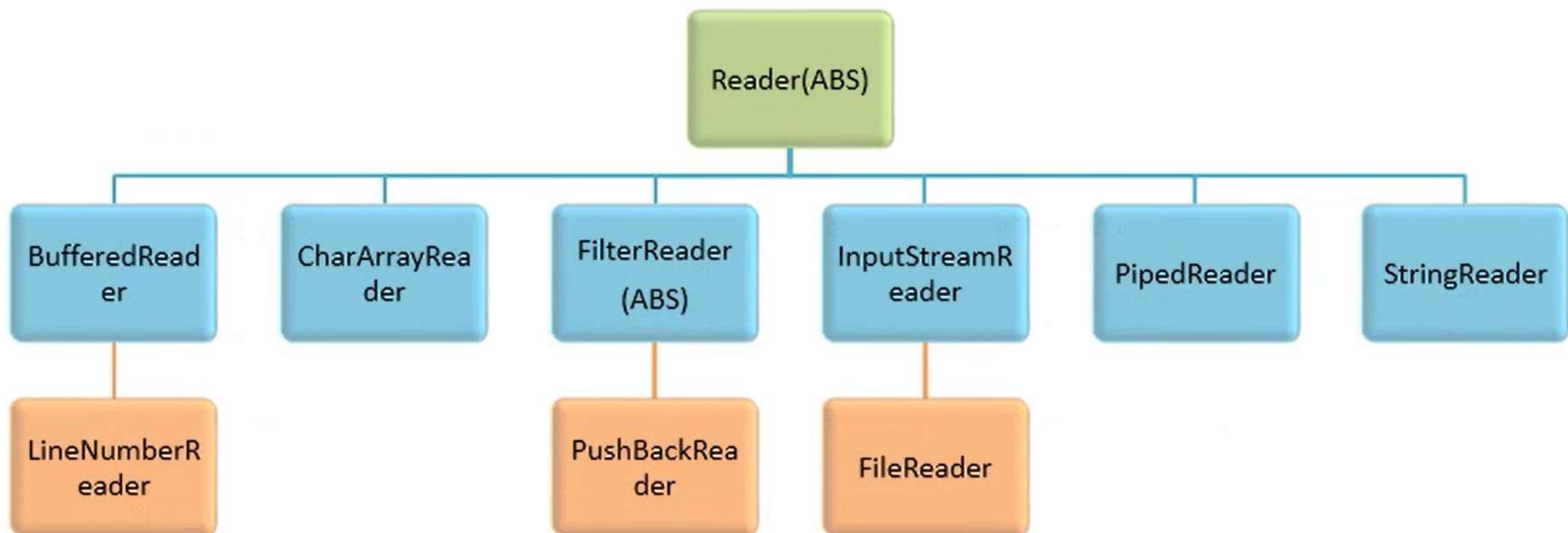


All inherited classes of Reader provide its own implementation
to read the character from various sources.

Methods of Reader class

Method	Description
int read()	Reads a single character
int read(char[])	Reads character into an array
int read(char[] buff,int offset,int length)	Reads character into a portion of an array
boolean ready()	Tells whether this stream is ready to be read
void close()	Closes this stream and releases the resources
int skip(long)	Skip the characters

Reader class Hierarchy



Note: This session focuses on FileReader and BufferedReader. Kindly go through www.oracle.com for more Reader classes.

FileReader

FileReader is used to read the contents of the file as characters

read() of FileReader is used to read the contents of file character by character.

Two constructors of FileReader are

FileReader(String
fileName)

FileReader(File
fileRef)

Both these constructors throw an exception called FileNotFoundException which is derived from IOException.

Exception is thrown when it is unable to locate the file.

Since FileNotFoundException is a checked exception, it has to be handled within try –catch –finally block or using throws

FileReader Example

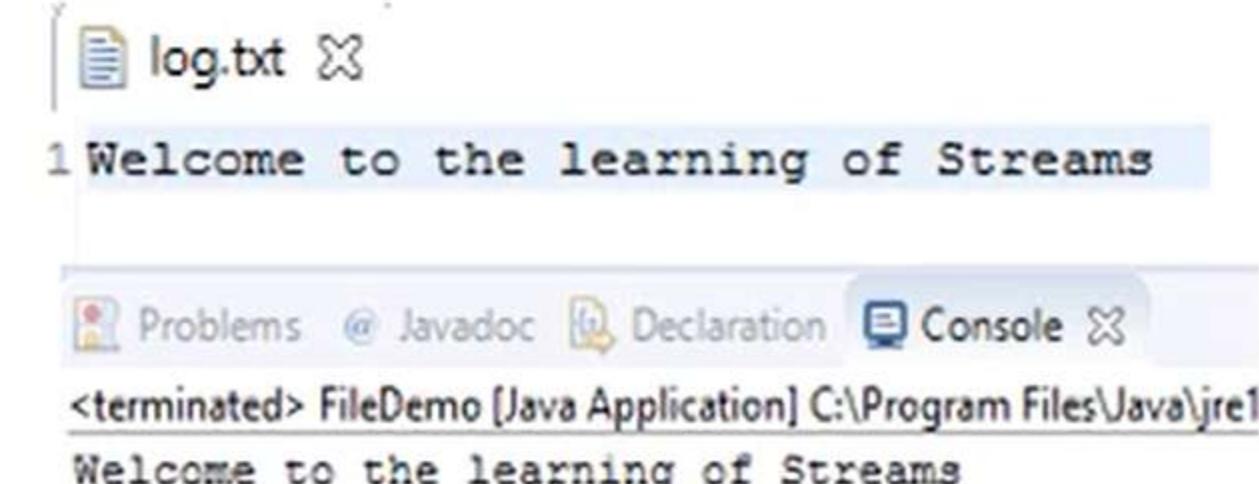
```

import java.io.*;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        FileReader fis=null;
        try {
            fis=new FileReader("\\\\IODemo\\\\log.txt");
            while((ch=fis.read())!=-1)
            {
                System.out.print((char)ch);
            }
        } catch(FileNotFoundException e)
        {
            //log the error
        } catch(IOException e)
        {
            //log the error
        } finally
        {
            try {
                fis.close();
            } catch(IOException e)
            {
                //log the error
            }
        }
    }
}
  
```

Creates a FileReader object with the fileName as the string

read() reads a single character at a time. When there is no character to read it returns -1

Always closes the stream in finally block. Close also throws an exception.so enclose within try-catch



The screenshot shows a Java application running in an IDE. The code in the editor is identical to the one above. In the bottom right corner, the terminal window displays the output: "Welcome to the learning of Streams". Above the terminal, a file named "log.txt" is shown with the same content.

BufferedReader

- Reads text from the character input stream.
- Buffers the characters and hence has better performance than the FileReader

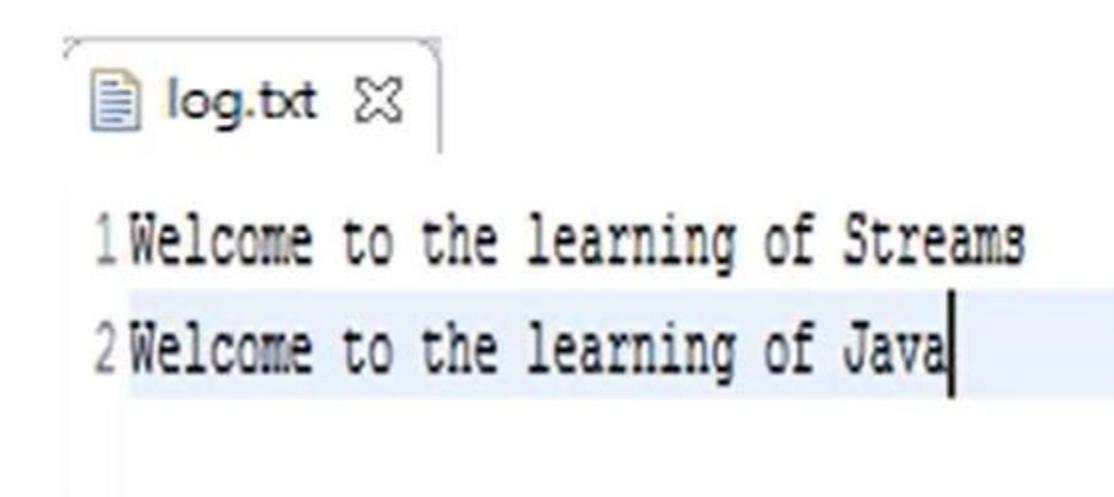
Method	Description
<code>int read()</code>	Reads a single character from the character input stream
<code>int read(char[])</code>	Reads characters into an array
<code>String readLine()</code>	Reads a line of text
<code>void close()</code>	Closes this stream and releases the resources

BufferedReader Example

```
import java.io.*;  
public class FileDemo {  
    public static void main(String[] args) {  
        int ch=0;  
        BufferedReader br=null;  
        FileReader fis=null;  
        try {  
            String str=null;  
            fis=new FileReader("\\\\IODemo\\\\log.txt");  
            br=new BufferedReader(fis);  
            while((str=br.readLine())!=null)  
            {  
                System.out.println(str);  
            }  
        catch(FileNotFoundException e)  
        { //log the error  
        }  
        catch(IOException e)  
        { //log the error  
        }  
        finally  
        {  
            try {  
                br.close();  
            }  
            catch(IOException e)  
            {  
                //log the error  
            }  
        }  
    }  
}
```

Creates a BufferedReader for the FileReader

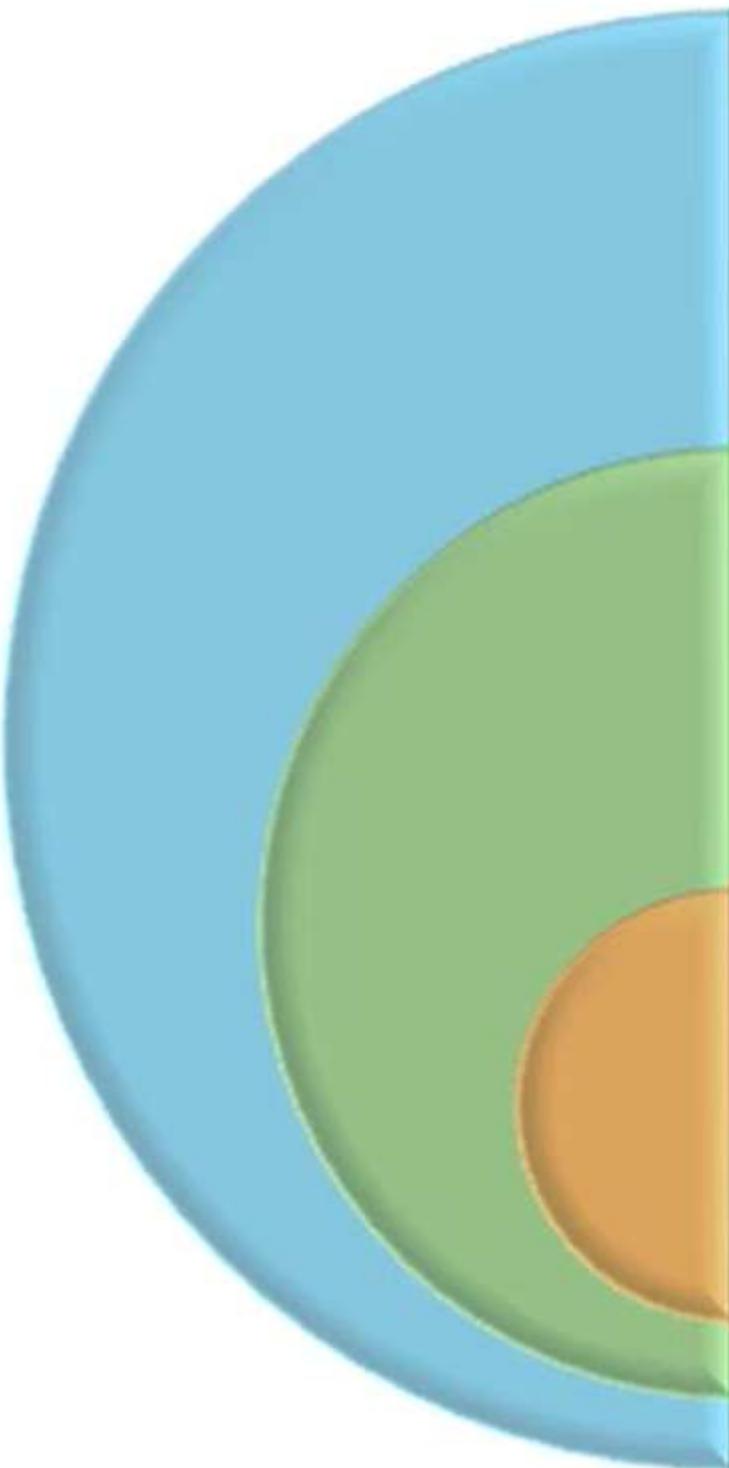
readLine () reads the entire line as String. When no more line to be read it returns null



log.txt

1 Welcome to the learning of Streams
2 Welcome to the learning of Java

Writer Class



Writer class writes character(16 bits) to any destination

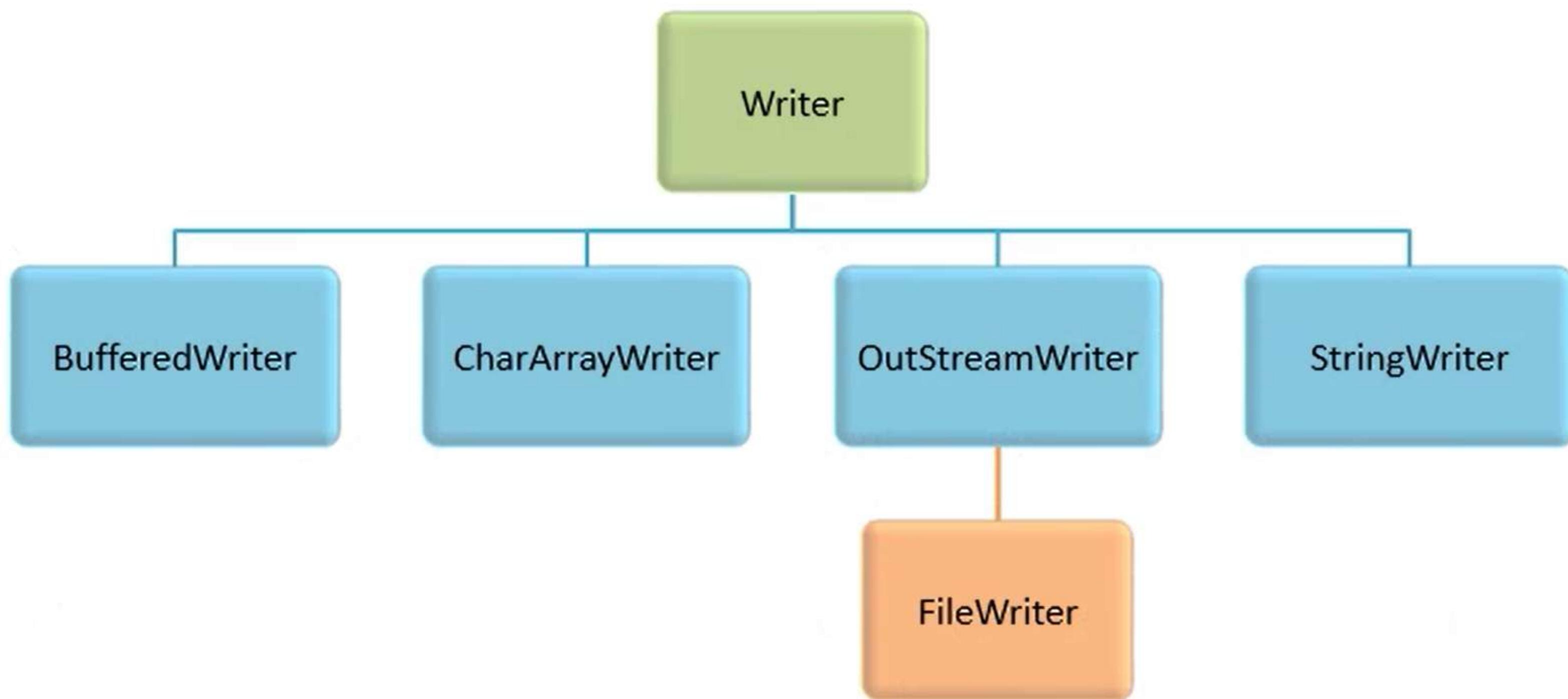
Writer class is available in java.io package

All inherited classes of Writer provide its own implementation to write the character to any destination(console, network socket).

Methods in Writer class

Methods	Description
void write(int)	Writes a single character
void write(char[])	Writes an array of characters
void write(char[],int offset,int length)	Writes a portion of an array of characters
void close()	Closes this stream and releases the resources.

Writer class Hierarchy



Note:

This session focuses on `FileWriter` and `BufferedWriter`. Kindly refer to <https://www.oracle.com/> for other stream API

FileWriter class

FileWriter is used for storing the contents to the file

write() method of the FileWriter writes the contents of the file

Example:

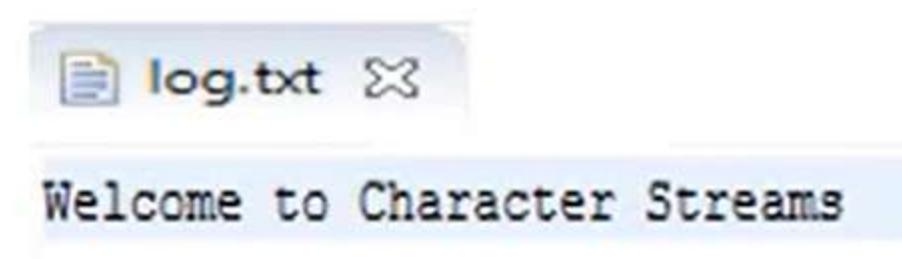
```
FileWriter fw=new  
FileWriter(\IO\log.txt);  
  
• Creates the FileWriter  
object
```

```
Fw.write("Welcome to  
Character Stream");  
  
• Writes the content into  
the file (log.txt)
```

FileWriter Example

```
import java.io.*;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        FileWriter fw=null;

        try {
            fw=new FileWriter("\\\\IODemo\\\\log.txt");
            fw.write("Welcome to Character Streams");
            fw.flush();
        }
        catch(FileNotFoundException e)
        {
            //log the error
        }
        catch(IOException e)
        //{
        //log the error
        //}
        finally
        {
            try {
                fw.close();
            }
            catch(IOException e)
            {
                //log the error
            }
        }
    }
}
```



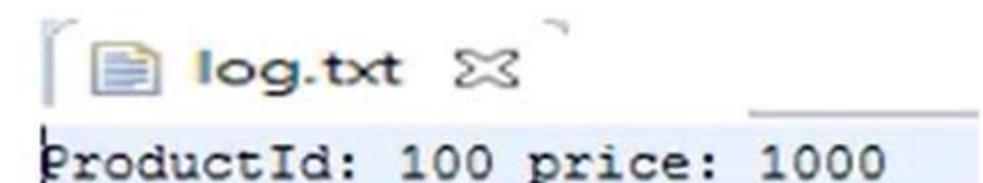
BufferedWriter

- Uses internal buffer to write the character into the destination
- BufferedWriter takes another Writer as an argument
- Has better performance than other Writers

Method	Description
<code>write(int ch)</code>	Writes a single character
<code>write(String str,int offset,int length)</code>	Writes a portion of the String
<code>void flush()</code>	Flushes the stream
<code>void close()</code>	Closes the stream, flushing it first

BufferedWriter Example

```
import java.io.*;
public class FileDemo {
    public static void main(String[] args) {
        int ch=0;
        BufferedWriter bw=null;
        FileWriter fw=null;
        try {
            fw=new FileWriter("C:\\\\Users\\\\Sripriya\\\\eclipse-workspace\\\\IODemo\\\\log.txt");
            bw=new BufferedWriter(fw);
            Scanner sc=new Scanner(System.in);
            System.out.println("Enter the product id");
            String id=sc.next();
            System.out.println("Enter the product price");
            int price=sc.nextInt();
            String record="ProductId: "+id+" price: "+price;
            bw.write(record);
        }
        catch(FileNotFoundException e)
        {
            //log the error
        }
        catch(IOException e)
        //{
        //log the error
        }
        finally
        {
            try {
                bw.close();
            }
            catch(IOException e)
            {
                //log the error
            }
        }
    }
}
```



Summary

- What is Character Stream
- The types of Character Stream
- How to read and write data using the Readers and Writers



SERIALIZATION



In this module you will learn

- What is Object Stream
- Facts on Serialization
- Understand ObjectOutputStream and ObjectInputStream
- Steps for Serialization and Deserialization
- Use of transient keyword



Object Stream

Object Streams are used to save the state to an object.

- This is achieved by converting it into a stream and storing it into a file or database. This is known as **Serialization**.
- This can be used at a later point of time for retrieving the stored values and restoring them into objects which is **Deserialization**.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

Few facts about Serialization

The class has to implement Serializable interface

Serializable interface is a marker interface.

If a class is serializable then all of its sub classes are also serializable

Marker interface is an interface with no methods in it.

Since Serializable is a marker interface it has no methods. It is simply used to indicate that the class may be serialized.

ObjectOutputStream and ObjectInputStream

ObjectOutputStream

Used for Serializing the object

Writes the object in to the persistant storage

ObjectOutputStream has a method called writeObject(ObjectToBePersisted).

writeObject(object) is used for writing the object in to the harddisk

ObjectInputStream

Used for de-serializing the object

ObjectInputStream has a method called readObject().

readObject() reads the serialized object and loads it into the memory.

Steps for Serialization

Write a class that implements Serializable interface so that its objects can be serialized



Create a FileOutputStream object to store the serialized object



Create a ObjectOutputStream object



Wrap the ObjectOutputStream with the FileOutputStream.



Use the writeObject(object) on the ObjectOutputStream which in turn uses its FileOutputStream object and writes the serialized object into the file.

transient keyword

Used in Serialization.

When transient is applied to the data member, it indicates that data member will not be serialized

transient modifier is applied only to instance variables

For Example: Assume Employee object needs to be Serialized. Employee has attributes empld,empName and salary. If employee object is serialized then all these information(empld,empName and salary) will be serialized. If salary need not be saved/serialized then salary attribute has to be declared transient.

```
public class Employee{  
    private transient float  
    salary;  
    ..  
}
```

Steps for De-Serialization

Create a Object for FileInputStream.



Create a Object for ObjectInputStream to read the object from the file



Wrap ObjectInputStream with FileInputStream



Use the readObject() on ObjectInputStream to read the object back in to the memory.

Example

Employee is the class whose objects need to be serialized and deserialized.

```
public class Employee implements Serializable
{
    public String name;
    public String address;
    public int SSN;
}
```

Serialization

```
import java.io.*;
public class Serialization {
    public static void main(String [] args) {
        Employee e = new Employee();
        e.name = "John";
        e.address = "No.4, 7th lane, New York";
        e.SSN = 11122333;
        try {
            FileOutputStream fileOut = new FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.printf("Serialized data is saved in /tmp/employee.ser");
        }
        catch(IOException i) {
            i.printStackTrace();
        } } }
```

Deserialization

```
import java.io.*;
public class Deserialization {
public static void main(String [] args) {
Employee e = null;
try {
FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
e = (Employee) in.readObject();
in.close();
fileIn.close();
}
catch(IOException i) {
i.printStackTrace();
return;
}
catch(ClassNotFoundException c) {
System.out.println("Employee class not found");
c.printStackTrace();
return;
}
System.out.println("Name: " + e.name);
System.out.println("Address: " + e.address);
System.out.println("Number: " + e.number);  }}
```

Output

Name: John

Address: "No.4, 7th lane, New
York";

Number:101

Summary

- What is Object Stream
- Facts on Serialization
- Understand ObjectOutputStream and ObjectInputStream
- Steps for Serialization and Deserialization
- Use of transient keyword





THREADS



Objective

- Process versus Threads
- Multi Tasking versus Multi threading
- Ways to Create Threads
- Thread Attributes
- Synchronization
- Inter thread Communication



Process and Threads



Process is an instance of a program

Threads are independent, concurrent paths of execution within a program

Threads are referred to as light-weight process

- Fewer resources are used to create a thread than creating a new process.

Processes have their own execution environment and memory space

Threads which is spawned from the same process share the same memory space

A process remains alive until all of its non-daemon threads complete the execution.

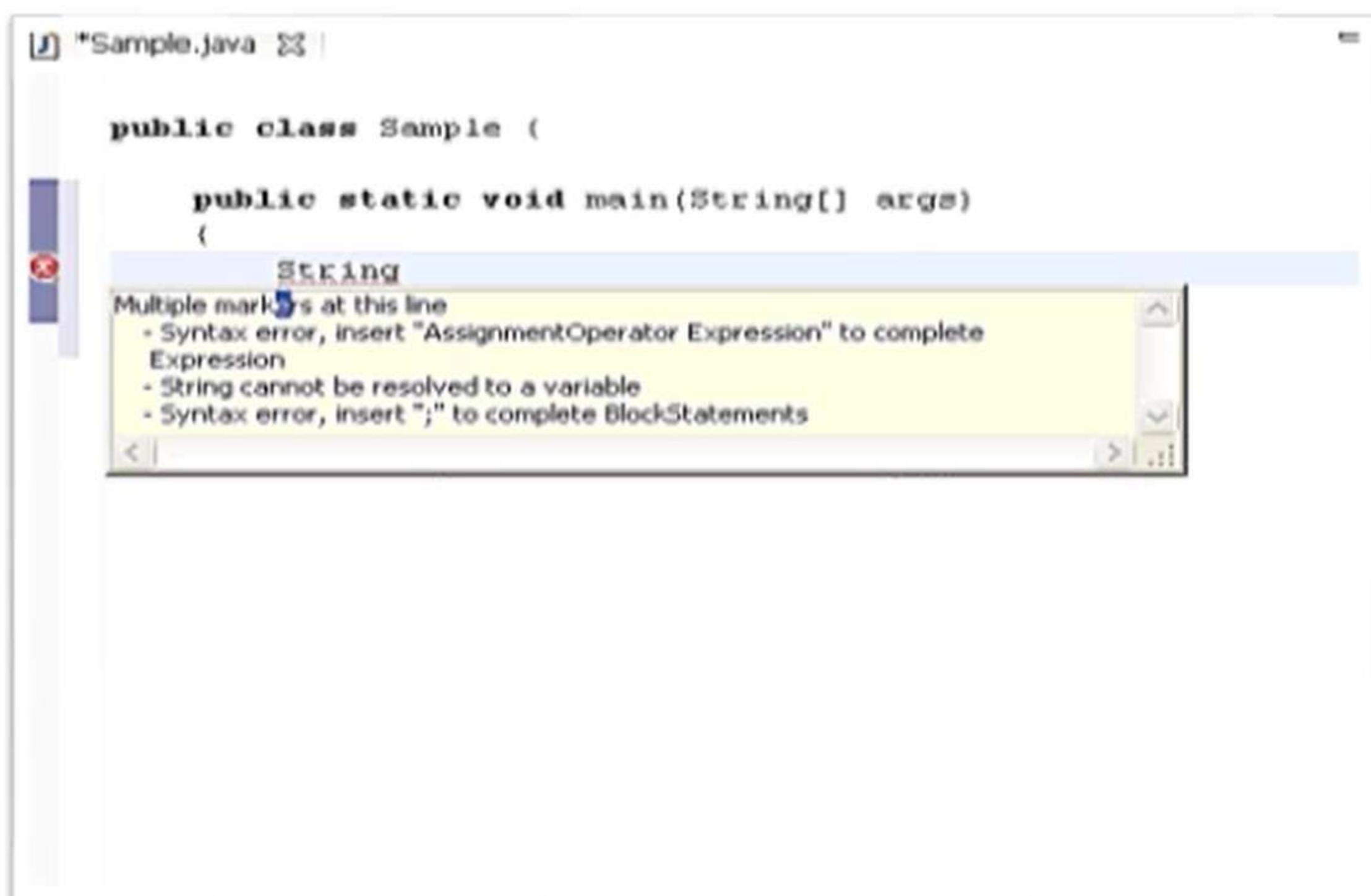
Process Vs Thread

- Let us consider eclipse.exe.
- What happens when we start eclipse?



Eclipse.exe process gets started

Thread



The screenshot shows an Eclipse IDE interface with a Java file named "Sample.java". The code contains a syntax error in the main method. A tooltip window is open over the error, displaying the following message:

String
Multiple markers at this line
- Syntax error, insert "AssignmentOperator Expression" to complete Expression
- String cannot be resolved to a variable
- Syntax error, insert ";" to complete BlockStatements

Semantic Checker is the thread
that runs inside the eclipse.exe
process

Thread

- A **thread** is an independent path of execution within a program.
- Threading allows multiple activities to coexist within a single program.



One process one thread



One process multiple threads

Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit only if Daemon threads are executing
- Daemon threads die when the Java VM exits
- JVM creates a main thread to execute any java program which is a daemon thread

Multitasking and Multithreading

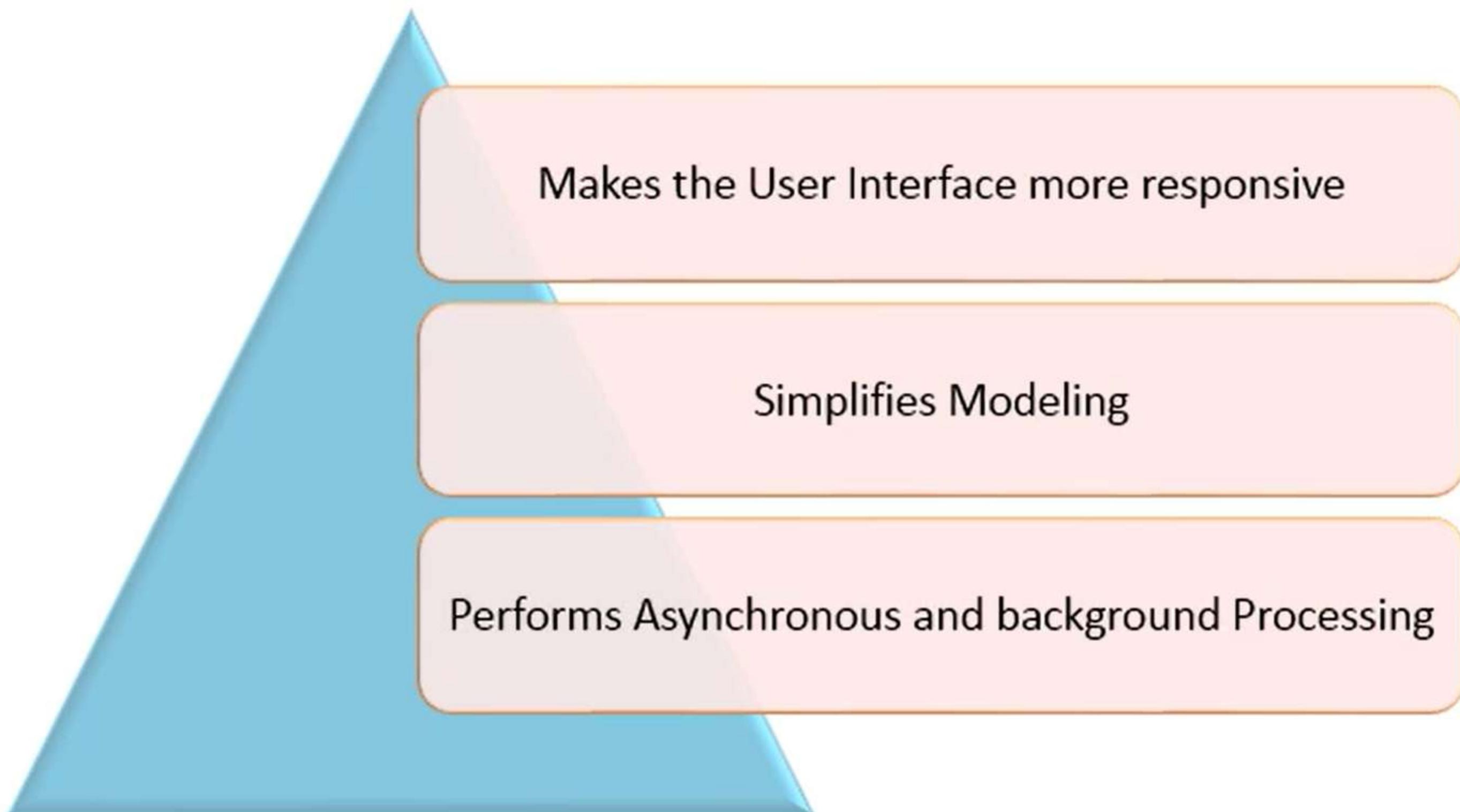
- In Multitasking the operating system can switch between tasks by resuming the task from the suspended state.

Multitasking is the ability to run multiple application programs concurrently.

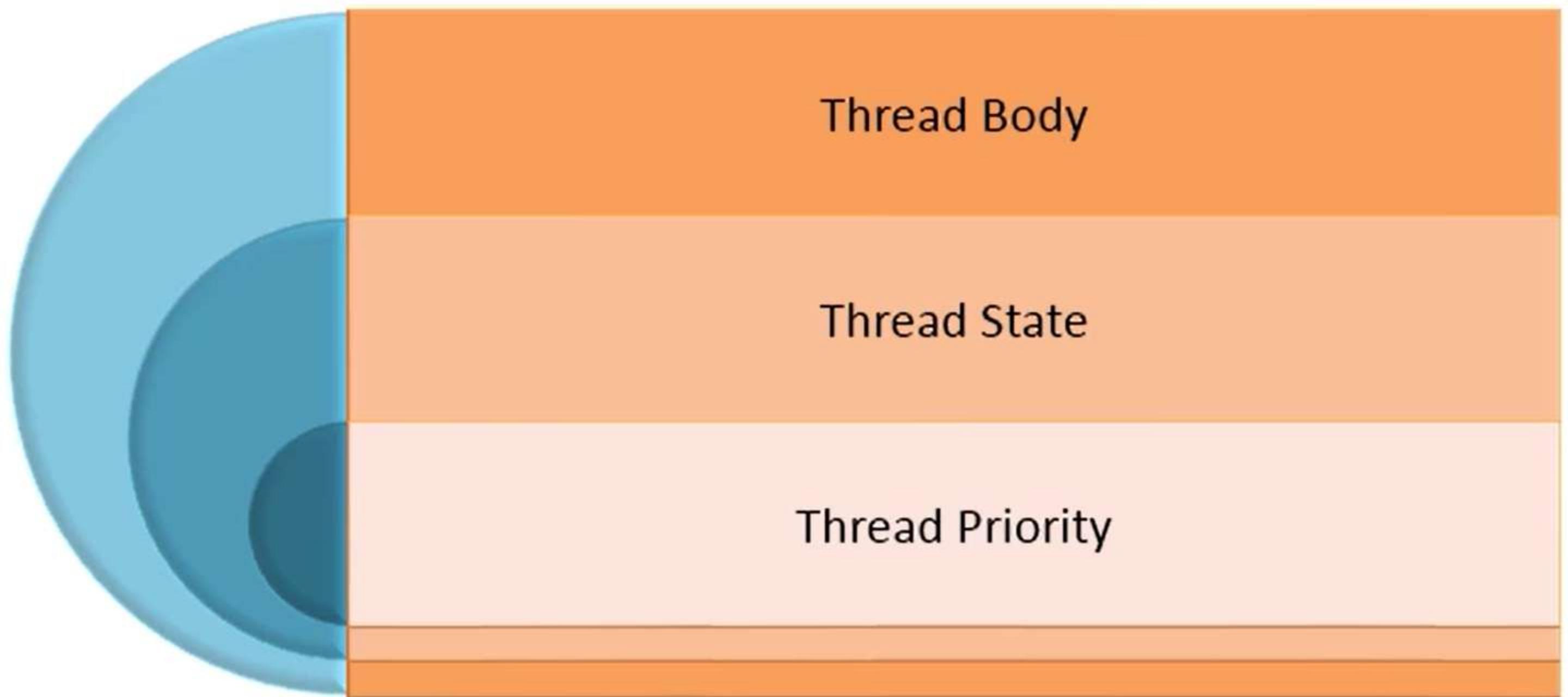
- Multithreading allows servicing more than one request to the same program by having one instance of the program in memory but spawning multiple threads to service each request.

Multithreading is the ability to run multiple parts within the same program concurrently.

Benefits of Using a Thread



Thread Attributes



Thread Body

The core part of the thread is the Thread body that is defined via the run() method

All actions the thread is expected to do has to be provided in the run() method

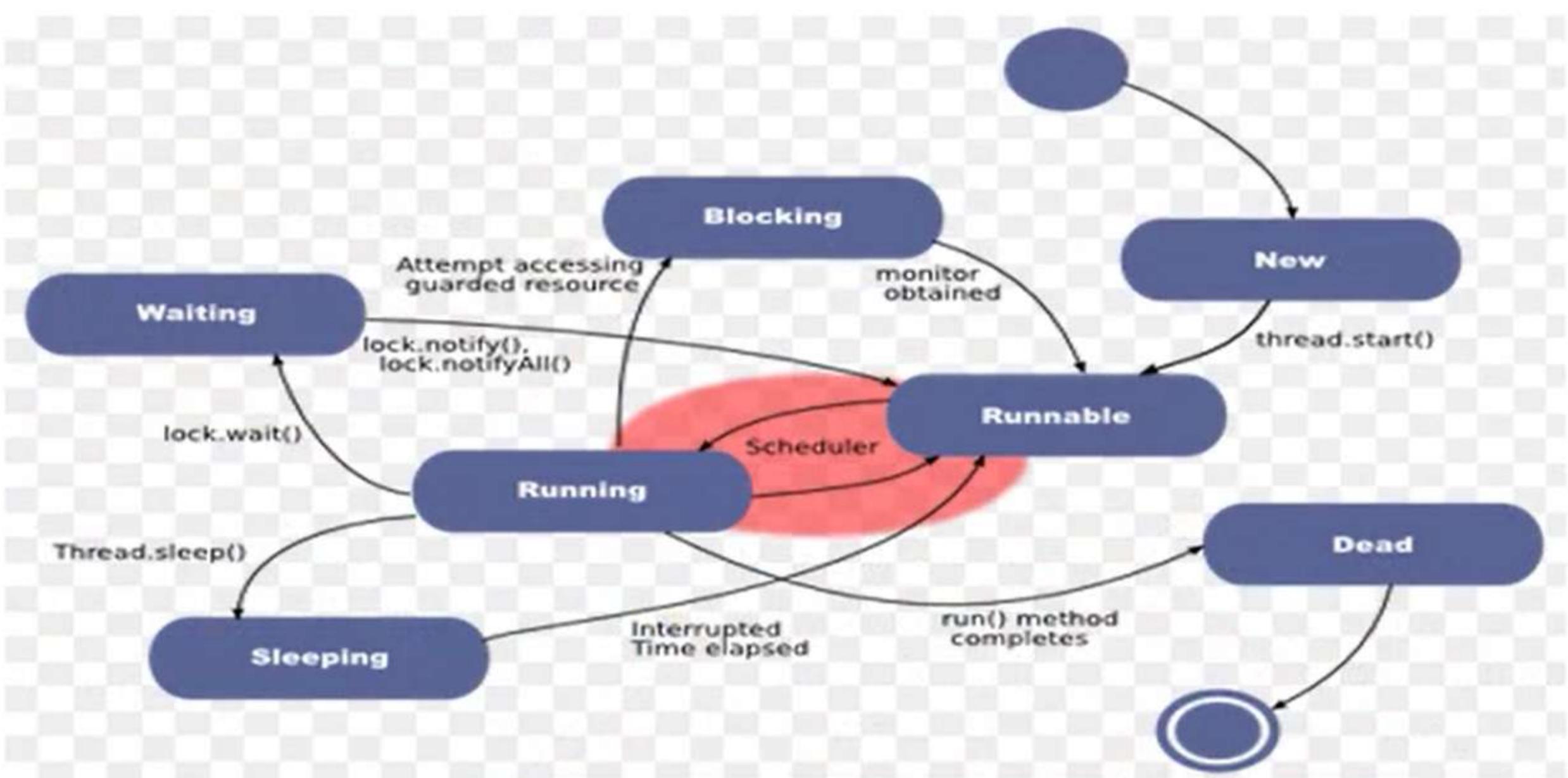
The thread body can be defined in 2 ways

Extending Thread class and override the Thread's run() method

Implementing the Runnable interface and implementing the run() method

Thread State

A java thread can be possibly in any one of the following states during its life time



Thread Priority

Every thread has a priority. By default the priority is 5

When a thread is created, it inherits the priority of the thread that created it

The priority values range from 1 to 10, in increasing order of priority

The priority of the Thread can be modified using the setPriority() method

The priority of a thread may be obtained using getPriority()

Priority constants are defined:

MIN_PRIORITY=1

MAX_PRIORITY=10

NORM_PRIORITY =5

Thread Priority

A thread runs until

- A higher priority thread becomes
Runnable

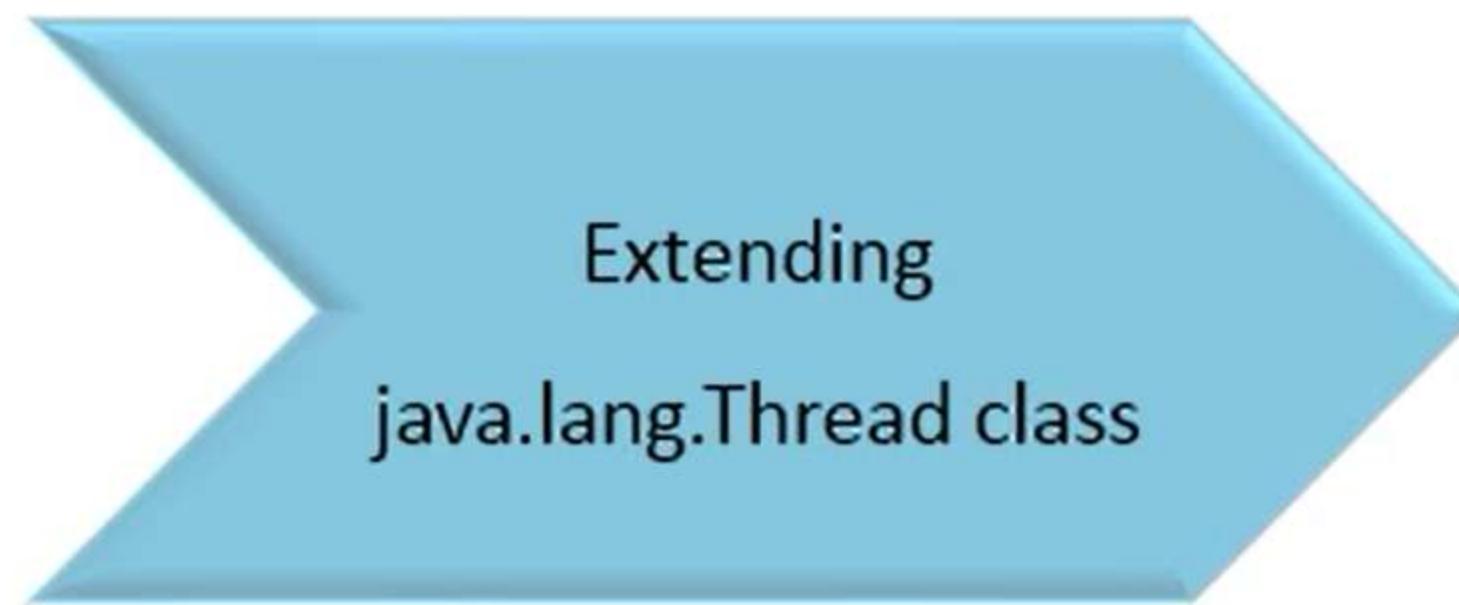
- Running thread issues a yield() or its
run() is complete

- On time-slicing systems, the time
allotted for that thread expires

The thread scheduler
in certain situations
can choose lower
priority thread over a
higher priority thread
for execution.

Program logic should
not rely on thread
priority.

Different ways of creating a Thread



Extending
`java.lang.Thread` class



Implementing
`java.lang.Runnable`
interface

Methods in Thread Class

start()

Causes this thread to begin execution

Java Virtual Machine calls the run method of this thread.

Thread.sleep

Puts the **current thread** to sleep for the specified amount of time. The time specified will be in milliseconds

A thread can be interrupted from its sleep.

The thread does not lose ownership of any monitors.

interrupt

interrupts the corresponding thread if it is sleeping or waiting.

If interrupt is invoked on the thread that is blocked by the invocation of wait() method of the Object class or join(),sleep() methods of the Thread class, then its interrupt status will be cleared and it will receive an **InterruptedException**

setPriority

To set the priority of a Thread

Methods in Thread Class

getPriority

To get the priority of a Thread

join

One thread can wait for another to complete using the join() method

Thread.yield

Causes the currently running thread to go back to runnable to allow other threads of same priority to get their turn

setName

To set the name for a thread

getName

To retrieve the name of a thread

Creating Threads - Example

```

public class Transaction extends Thread {
    Account accobj;

    public Transaction(Account accobj) {
        this.accobj = accobj;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
        float balance=0;
        balance=accobj.deposit(1000);
        System.out.println("Amount after deposit "+balance);
        balance=accobj.withdraw(500);
        System.out.println("Amount after withdraw "+balance);
    }

    public static void main(String[] args) {
        Account accobj = new Account(32456745,2000);
        Transaction t1 = new Transaction(accobj);
        t1.setName("John");
        Transaction t2 = new Transaction(accobj);
        t2.setName("Mary");
        t1.start();
        t2.start();
    }
}

```

Order of output is non-deterministic in a thread program

John
Mary
Amount after deposit 3000.0
Amount after deposit 4000.0
Amount after withdraw 3500.0
Amount after withdraw 3000.0

Creating Threads - Example

```
public class Account {  
    private long accNo;  
    private float balance;  
  
    public Account(long accNo, float balance) {  
        this.accNo = accNo;  
        this.balance = balance;  
    }  
  
    public float withdraw(float bal) {  
        balance=balance-bal;  
        return balance;  
    }  
  
    public float deposit(float bal) {  
        balance=balance+bal;  
        return balance;  
    }  
}
```

Creating a Thread-Runnable

```
public class Transaction implements Runnable {
    Account accobj;

    public Transaction(Account accobj) {
        this.accobj = accobj;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName());
        float balance=0;
        balance=accobj.deposit(1000);
        System.out.println("Amount after deposit "+balance);
        balance=accobj.withdraw(500);
        System.out.println("Amount after withdraw "+balance);
    }

    public static void main(String[] args) {
        Account accobj = new Account(32456745,2000);
        Transaction t1 = new Transaction(accobj); //Runnable instance
        Thread thread1=new Thread(t1);
        thread1.setName("John");
        Transaction t2 = new Transaction(accobj); //Runnable instance
        Thread thread2=new Thread(t2);
        thread2.setName("Mary");
        thread1.start();
        thread2.start();
    }
}
```

Java Synchronization

Synchronizing threads means access to shared data or method, execution logic in a multithreaded application is controlled in such a way that only one thread can access the shared data at a time.

Java uses the concept of monitors

Java uses the concept that every object has a lock

Synchronized Method

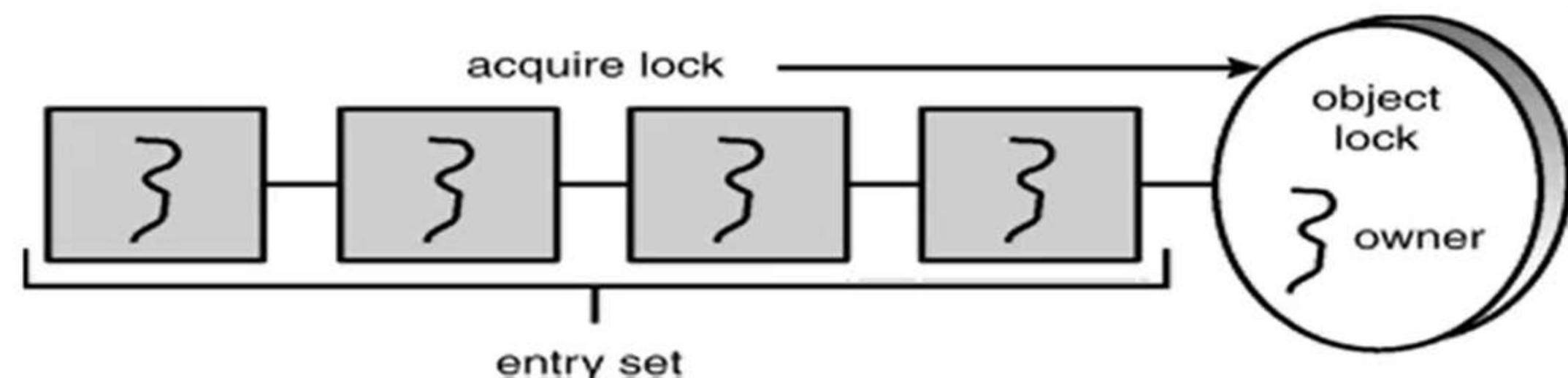
Every object has a lock associated with it.

Calling a synchronized method attempts to possess the lock

- If no one owns the lock, then this thread owns the lock.

If a calling thread does not own the lock, the calling thread is placed in the entry set for the object's lock.

The lock is released when a thread exits the synchronized method.



Synchronized Method -Example

```
public class Account {  
    private long accNo;  
    private float balance;  
  
    public Account(long accNo, float balance) {  
        this.accNo = accNo;  
        this.balance = balance;  
    }  
  
    public synchronized float withdraw(float bal) {  
        balance=balance-bal;  
        return balance;  
    }  
  
    public synchronized float deposit(float bal) {  
        balance=balance+bal;  
        return balance;  
    }  
}
```

Account Object's withdraw and deposit methods are synchronized so as to avoid inconsistency between withdraw and deposits transactions done by two or more joint account holders on the same account object

Synchronized block

Synchronized methods

Implicitly lock the current object

Synchronized *blocks*

lock on an arbitrary, specified object

Alternative technique of synchronized methods for getting the consistent result.

Example for Synchronized Block

```
public class Account {  
    private long accNo;  
    private float balance;  
  
    public Account(long accNo, float balance) {  
        this.accNo = accNo;  
        this.balance = balance;  
    }  
  
    public float withdraw(float bal) {  
  
        synchronized(this)  
        {  
            balance=balance-bal;  
            return balance;  
        }  
    }  
  
    public float deposit(float bal) {  
  
        synchronized(this)  
        {  
            balance=balance+bal;  
            return balance;  
        }  
    }  
}
```

Inter Thread Communication

Java threads can share data between each other

Consider the classic queuing problem, where one thread is producing some data and another is consuming it.

To avoid polling, Java includes an elegant inter thread communication mechanism via the following methods.

All three methods can be called only from within a **synchronized** context.

`wait()`

- current thread gives up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.

`notify()`:

- wakes up the first thread that called `wait()` on the same object.

`notifyAll()`:

- This method wakes up all the threads that called `wait()` on the same object .
- The highest priority thread will run first.

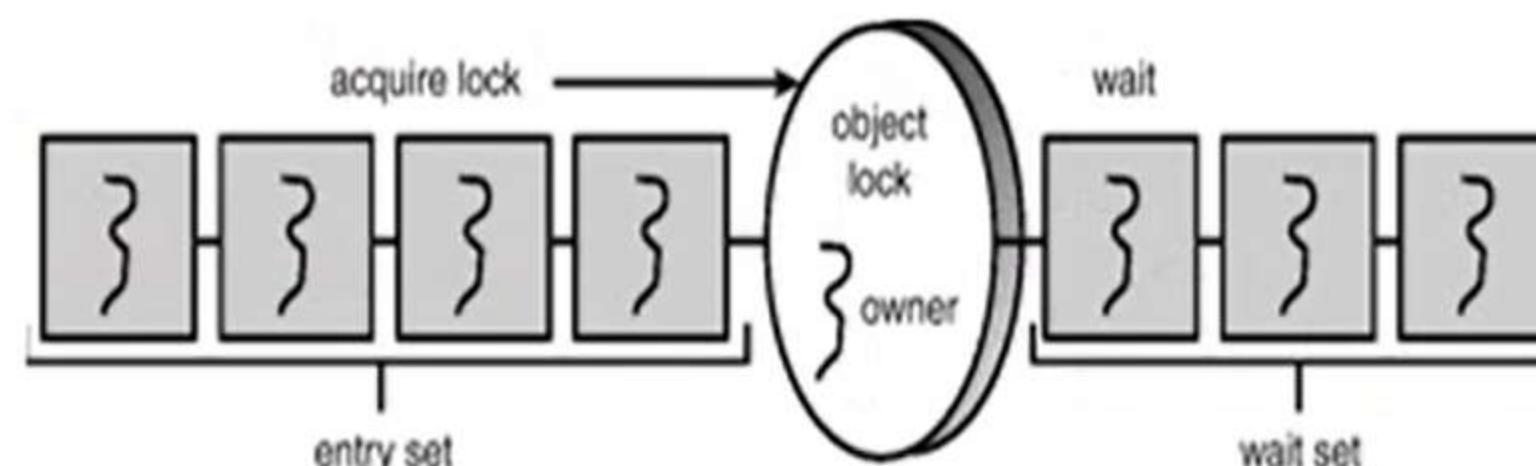
wait() method

When a thread calls wait(), the following occurs:

Thread releases the object lock.

Thread state is set to blocked.

Thread is placed in the wait set.



notify() method

When a thread calls notify(),
the following occurs:

T can now compete for
the object's lock again.

selects an arbitrary thread
 T from the wait set.

moves T to the entry set.

sets T to Runnable.

notifyAll() method

notify() selects an arbitrary thread from the wait set.

Java does not allow you to specify the thread to be selected.

notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.

notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.

Inter Thread Communication-Example

```
class MessageBox {
    String msg;
    boolean valueSet = false;
    synchronized String get() {
        if(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Consumer Reads: " + msg);
        valueSet = false;
        notify();
        return msg;
    }

    synchronized void put(String msg) {
        if(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.msg = msg;
        valueSet = true;
        System.out.println("Producer Writes: " + msg);
        notify();
    }
}
```

Inter Thread Communication-Example

```
class Producer implements Runnable {
    MessageBox box;
    Producer(MessageBox q) {
        this.box = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        String data[] = {"Delhi", "Mumbai", "Chennai"};
        for(String msg: data) {
            box.put(msg);
        }
    }
}

class Consumer implements Runnable {
    MessageBox box;
    Consumer(MessageBox q) {
        this.box = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            box.get();
        }
    }
}
```

Inter Thread Communication-Example

```
public class ThreadInteraction_Demo{  
  
    public static void main(String args[]) {  
        MessageBox box = new MessageBox();  
        new Producer(box);  
        new Consumer(box);  
    }  
}
```

Summary

- Process versus Threads
- Multi Tasking versus Multi threading
- Ways to Create Threads
- Thread Attributes
- Synchronization
- Inter thread Communication



JUNIT



In this Module You will learn

- Why Testing
- Junit Framework
- Why Junit
- Installing and Running Junit
- Annotation and Assertion in Junit
- Testing with JUnit
- Test Suite And Test Runner



Testing

Software testing is a process of verifying and validating a software application or program

- Meets the business and technical requirements that guided its design and development
- Works as expected



A clever person
solves a problem.
A wise person
avoids it.”

• *Albert Einstein*



Bug Free Software

Why Testing?

Testing is important because software bugs might be

- more expensive and
- also dangerous leading to human loss.

- European Space Agency's Ariane 5 rocket failure
 - In June 1996, the first flight of the European Space Agency's Ariane 5 rocket failed shortly after launching.
 - This resulted in an uninsured loss of \$500,000,000.
 - The disaster was traced to the lack of exception handling for a floating-point error when a 64-bit integer was converted to a 16-bit signed integer.



Why Testing?

- Loss of NASA Mars Climate Orbiter
 - In October 1999 the \$125 million NASA Mars Climate Orbiter, an interplanetary weather satellite was lost in space due to a data conversion error.
 - Investigators discovered that software on the spacecraft performed certain calculations in English units (yards) when it should have used metric units (meters)



This image of Mars on September 7, 1999 is the only image acquired by the Orbiter.

Unit Testing

Unit Testing is used to check whether a particular module is implementing its specification

Who Performs

- normally performed by software developers themselves or their peers
- In rare cases it may also be performed by independent software testers

The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.



JUNIT

JUnit is an open source unit testing framework for Java

JUnit provides

- a base class, TestCase, that can be extended to create a series of tests for the class that is created
- annotations to identify test methods
- assertions for testing expected results

JUnit can be easily included with Eclipse, Maven etc.

JUnit shows test progress in a bar that

- is green if the test is running smoothly
- turns red when a test fails

Test suites, Test Runners, JUNIT classes and fixtures are the most prominent features of this framework

Why JUNIT?

Open source framework

For running automated tests

Multiple tests can be run concurrently

Running tests automatically help to identify software regressions introduced by changes in the source code

Having high test coverage of code enables the developer to continue development of features without performing more manual tests



Installing And Running JUNIT

Download "Junit 4.0" jars from
<http://junit.org/junit4>

Install JUNIT jar files in eclipse

- Right Click on project
- Click on "Buildpath" and then click on "Configure Build Path"
- Include " JUNIT4.jars " into the project workspace

Annotation

- Annotation are metadata or data about data
- Introduced in Java 5
- Starts with @ symbol
- Can be attached with class, interface, methods or fields
- Provides additional information to be used by java compiler or development tool
- Example

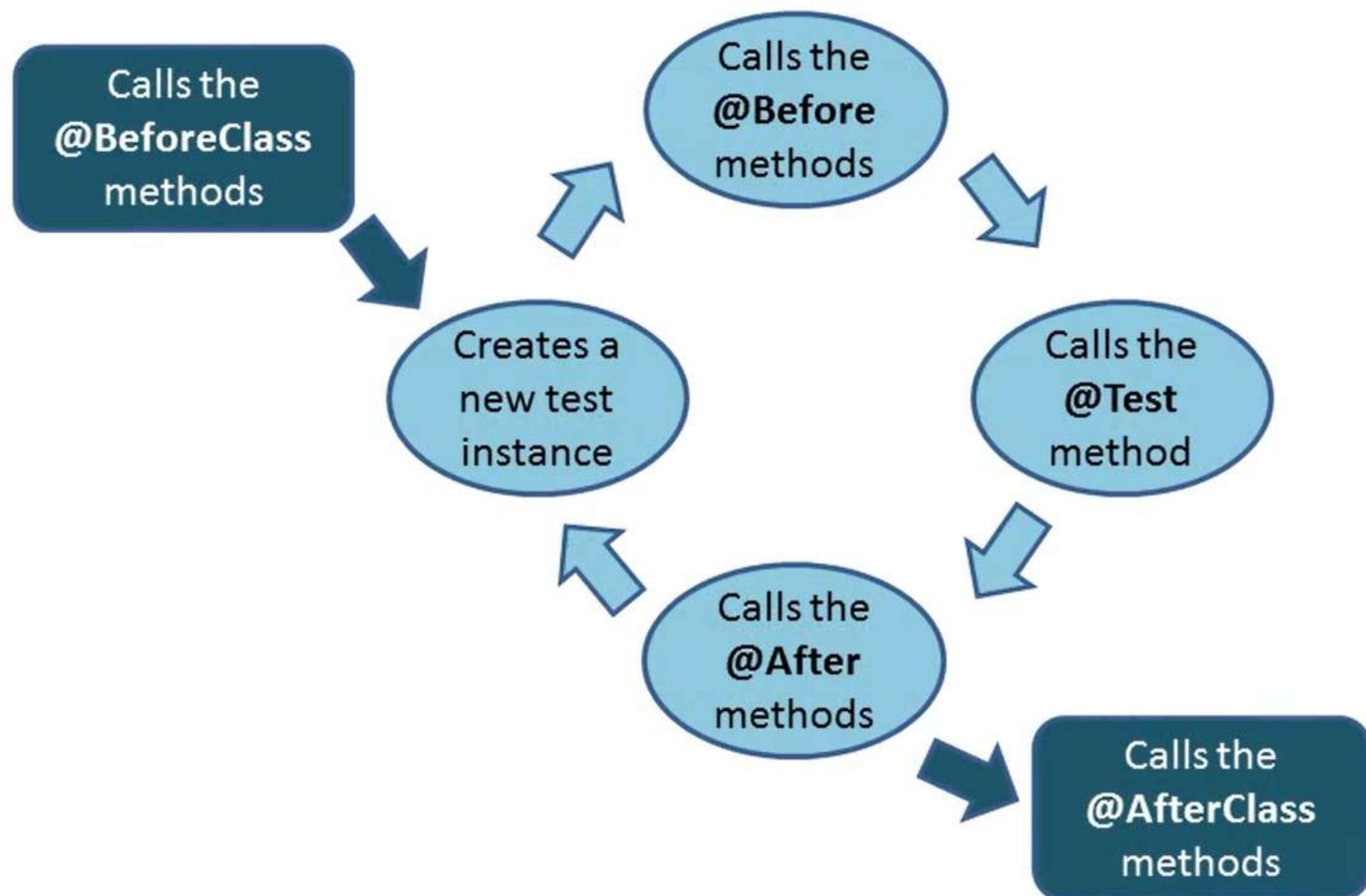
```
@Override
    – used with methods
    – informs compiler that the method annotated with this
        annotation should override the method in the parent
```

Annotation in Junit

- Junit 4.x is annotation based unit testing framework

Annotation	Description
@Test	Specifies test method
@BeforeClass	public static void no- argument method will be invoked only once before starting all the tests.
@AfterClass	public static void no- argument method will be invoked only once after finishing all the tests.
@After	Method will be invoked after each test.
@Before	Method will be invoked before each test.
@Ignore	Ignores the execution of the test method. This is useful when the code is not completely ready, so that a test method fails.

JUnit Execution



JUnit Method

To run the annotated methods, JUnit first creates an instance and then invokes the methods in that class.

All unit test methods

- should be annotated as @Test
- should be public void with no parameters
- should make assert calls to validate the result

In Eclipse to create a Junit

- Right click the package, click New, click Junit Test Case

Assert methods

- Assertions are used to check if the actual result is the same as the expected result.
- Static methods present in org.junit.Assert class are shown below

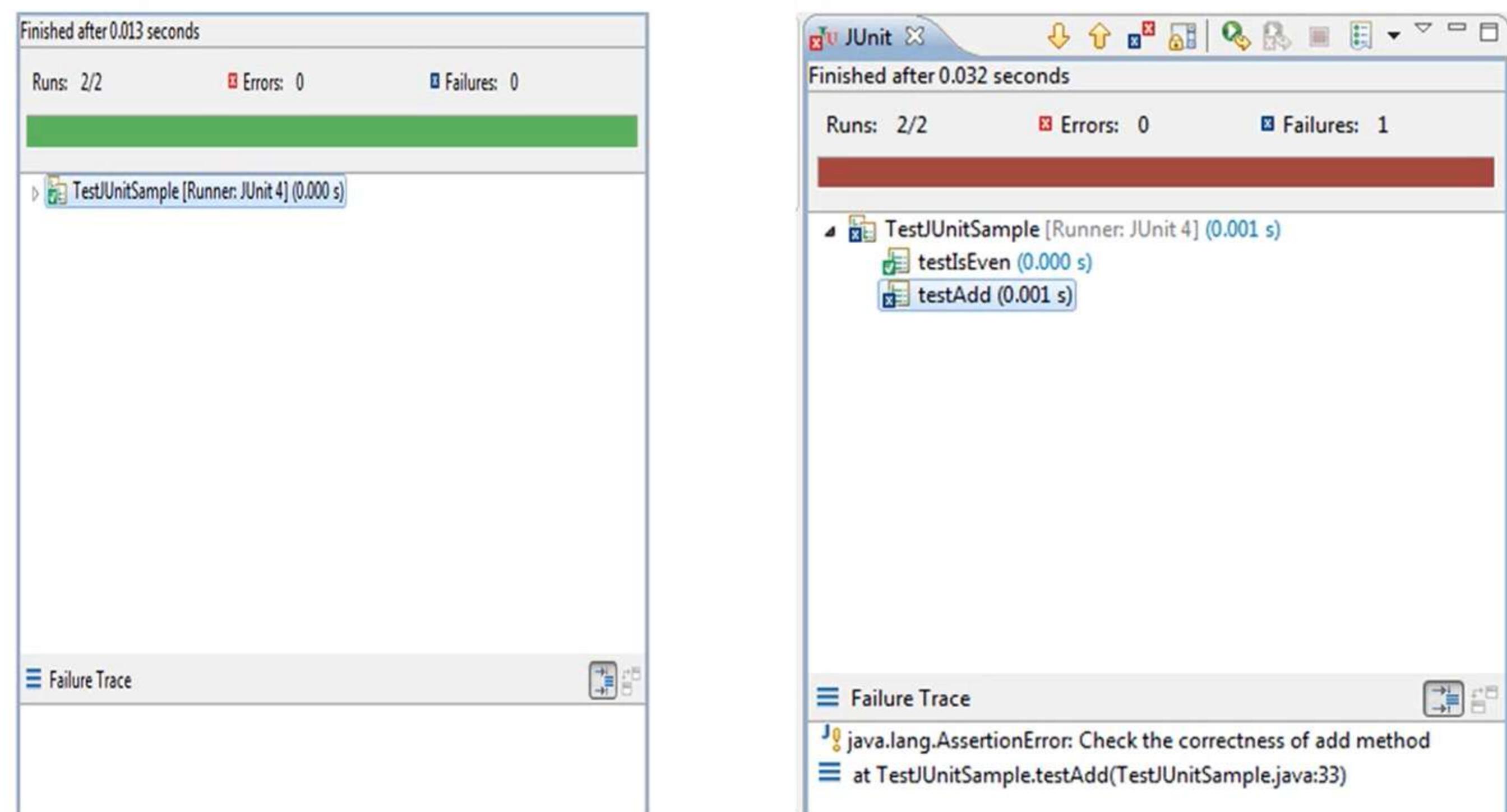
Assert Method	Description	Example
assertEquals(String, Object expected, Object actual)	Asserts actual value is equal to expected value	assertEquals("Not equal", 16, product(2,8));
assertNull(String, Object)	Asserts the object is null	Student s=null; assertNull("Value is not null",s);
assertNotNull(String, Object)	Asserts the object is not null	String s="Java"; assertNotNull("Value is null",s);
assertTrue(String, boolean)	Asserts the boolean value is true	boolean x= 15 > 10; assertTrue("Not greater",x);
assertFalse(String, boolean)	Asserts the boolean value is false	boolean x= 5 > 10; assertFalse("Greater",x);
assertSame(String, Object expected, Object actual)	Asserts two objects refer to the same object	Student s1=new Student(); Student s2=s1; assertSame("Not same",s2,s1)
assertNotSame(String, Object expected, Object actual)	Asserts two objects are not referring to the same object	Student s1=new Student(); Student s2=new Student(); assertNotSame("Same",s2,s1)

Junit - Example

```
public class JUnitSample {  
  
    public int add(int a,int b){  
        if(a>0 && b>0)  
            return a+b;  
        else  
            return 0;  
    }  
  
    public boolean isEven(int a) {  
        return (a%2==0);  
    }  
}
```

```
import org.junit.BeforeClass;  
import org.junit.Test;  
  
public class TestJUnitSample {  
  
    static JUnitSample sample=null;  
  
    @BeforeClass  
    public static void setUp() {  
        sample=new JUnitSample();  
    }  
  
    @Test  
    public void testIsEven(){  
        assertTrue("Value is not even",sample.isEven(10));  
        assertFalse("Check for odd numbers",sample.isEven(5));  
    }  
  
    @Test  
    public void testAdd(){  
        try {  
            assertEquals(5,sample.add(2, 3));  
            assertEquals(0,sample.add(-2, 3));  
            assertNotEquals(1,sample.add(-2, 3));  
        }  
        catch(Error e){  
            fail("Check the correctness of add method");  
        }  
    }  
}
```

Junit - Example



The image displays two side-by-side screenshots of a JUnit test runner interface. Both screenshots show the same test class, `TestJUnitSample`, running on JUnit 4.

Screenshot 1 (Left): This screenshot shows a successful test run. The status bar at the top says "Finished after 0.013 seconds". Below it, the statistics are "Runs: 2/2", "Errors: 0", and "Failures: 0". A green progress bar is shown. Under the statistics, there is a tree view of the test results:

- TestJUnitSample [Runner: JUnit 4] (0.000 s)
 - (Green checkmark icon) testIsEven (0.000 s)
 - (Green checkmark icon) testAdd (0.001 s)

At the bottom, there is a "Failure Trace" section which is currently empty.

Screenshot 2 (Right): This screenshot shows a failed test run. The status bar at the top says "Finished after 0.032 seconds". Below it, the statistics are "Runs: 2/2", "Errors: 0", and "Failures: 1". A red progress bar is shown. Under the statistics, there is a tree view of the test results:

- TestJUnitSample [Runner: JUnit 4] (0.001 s)
 - (Green checkmark icon) testIsEven (0.000 s)
 - (Red X icon) testAdd (0.001 s)

At the bottom, the "Failure Trace" section contains the following information:

- java.lang.AssertionError: Check the correctness of add method
- at TestJUnitSample.testAdd(TestJUnitSample.java:33)

Test cases

In a test class, @Test annotated method tells the Junit, that method can run as a test case.

A test case tests the response of a single method to a particular set of input.

Test case comprises of testcase id, module to be tested, test data which comprises of set of data to be given as input, expected output, actual output, Result which can be either pass or fail and Comments.

Test Suite and Test Runner

Several test classes can be combined into a **Test Suite**

Running a Test Suite executes all test classes in that suite in the specified order

Test Suites can be nested

A Test Runner is used to run the Test Suite and reports results.

Test Suite

Example

```
public class FindMax{  
    public static int findMax(int a,int b){  
        return a>b ? a : b;  
    }  
}
```

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class TestFindMax{  
    @Test  
    public void testFindMax(){  
        assertEquals("Check logic of findmax method", 9,FindMax.findMax(7, 9));  
        assertEquals("Check for equality in findmax ", 9,FindMax.findMax(9, 9));  
    }  
}
```

Test Suite

Example – contd .

Creation of Test Suite

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
  
@RunWith(Suite.class)  
  
@Suite.SuiteClasses({  
    TestJUnitSample.class,  
    TestFindMax.class  
})  
  
public class TestSuiteDemo {  
  
    // This class remains empty.  
    // used only as a holder for the above annotations  
}
```

Test Runner

Example – contd .

Creation of Test Runner

```
import org.junit.runner.JUnitCore;  
import org.junit.runner.Result;  
import org.junit.runner.notification.Failure;  
public class TestRunnerDemo {  
  
    public static void main(String args[]){  
  
        Result result = JUnitCore.runClasses(TestSuiteDemo.class);  
  
        if(result.getFailureCount()==0)  
            System.out.println("Successfully Passed All test cases");  
        else  
            for (Failure failure: result.getFailures())  
                System.out.println("Hi "+failure.toString());  
  
        System.out.println("Result "+result.wasSuccessful());  
    }  
}
```

Summary

- Why Testing
- Junit Framework
- Why Junit
- Installing and Running Junit
- Understanding JUnit Framework
- Annotation and Assertion in Junit
- Testing with JUnit
- Test Suite And Test Runner



ECLIPSE IDE



Overview

Tom wants to give his car for a water wash



Overview



Vs



Manual Car Service – Past

Modern day Car Service Station - Present

In this module you will learn

- Introduction to Eclipse IDE
- Installation and Setting up of Eclipse
- Creating and Managing Java Projects
- Miscellaneous Options



What Is an Integrated Development Environment (IDE)?

An integrated development environment (IDE) is an application that facilitates application development with ease.

IDE will consist of

- Source Code Editor
- Compiler
- Debugger
- Build Automation Tools

IDE may support either a single language or multiple language

IDE

Example

- Eclipse
- NetBeans
- DreamWeaver
- VisualStudio
- Coda



Eclipse is a
widely used IDE
for Java
development



Introduction to Eclipse

Eclipse is free to download and install

It is composed of plugins.

Few of those plugins

- Java Development Tools (JDT) for Java
- PyDev for Python
- C / C++ Development Tools (CDT) for C / C++
- PHPEclipse for PHP

Eclipse platform and other plug-ins from the Eclipse foundation is released under the Eclipse Public License (EPL).

Every year new versions of Eclipse are released

Installation and Setting up of Eclipse

Different versions of eclipse are available

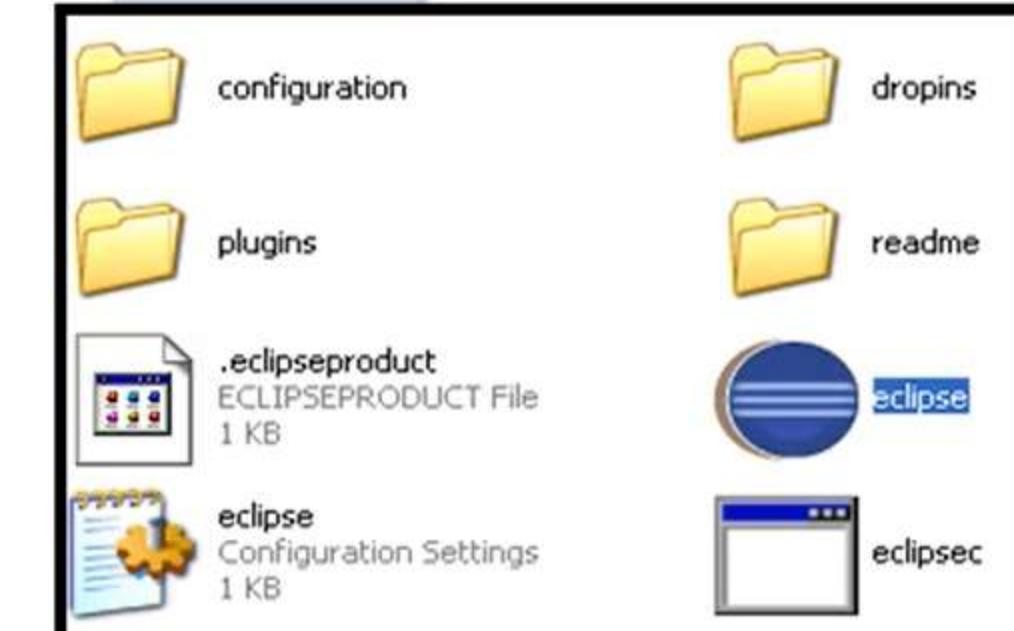
Download eclipse from <http://www.eclipse.org/downloads/>

Choose eclipse version based on the operating system

First download the zip file and extract it.

Eclipse is now ready to start

Start the eclipse by clicking the eclipse.exe file

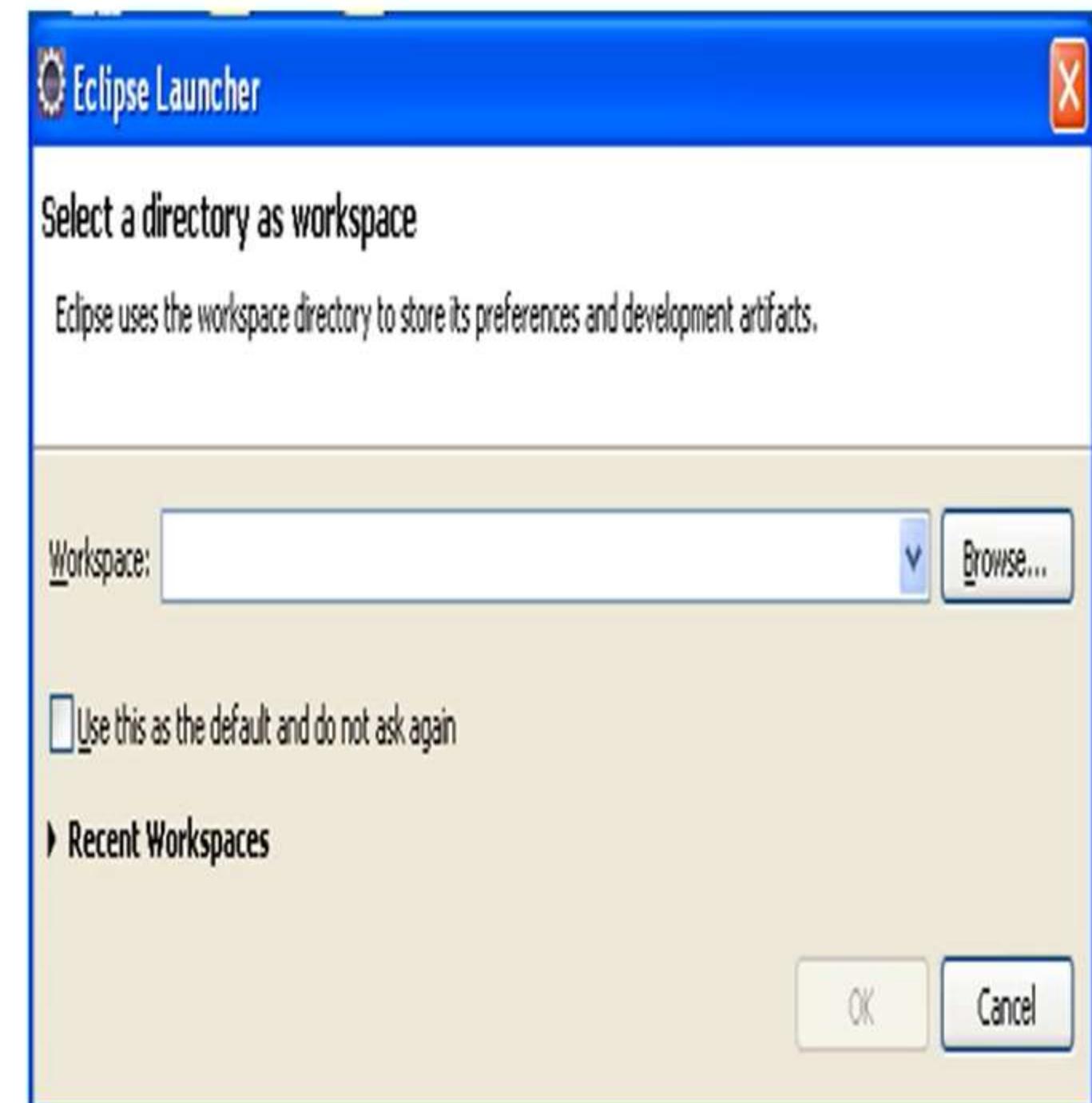


Create a workspace

Eclipse prompts for workspace

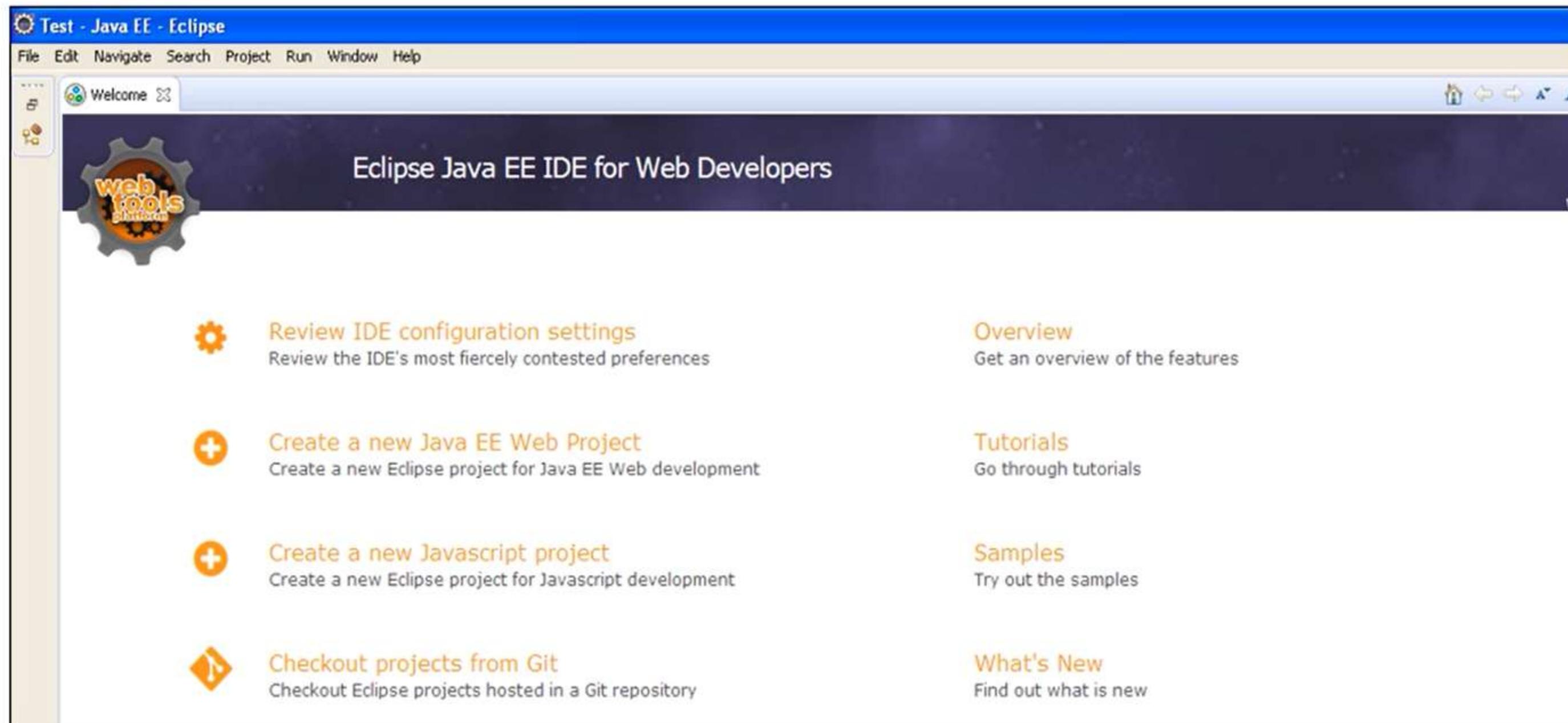
- Workspace is the location where multiple projects are created and stored
- The *workspace* is the physical location (file path) for storing certain meta-data and the development artifacts.

If the workspace does not exist, it creates one



Welcome Page

Eclipse shows Welcome page. Close this page

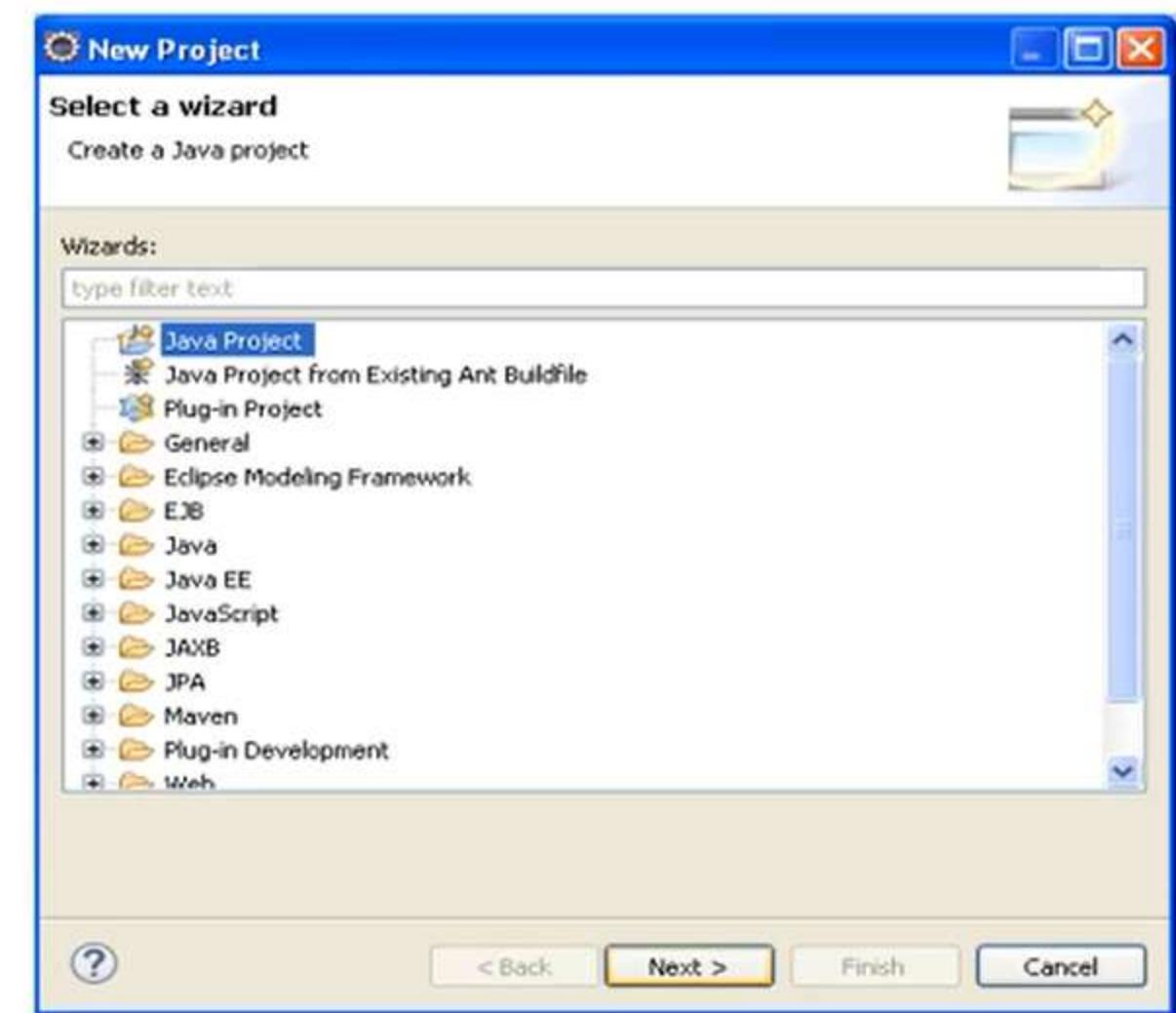
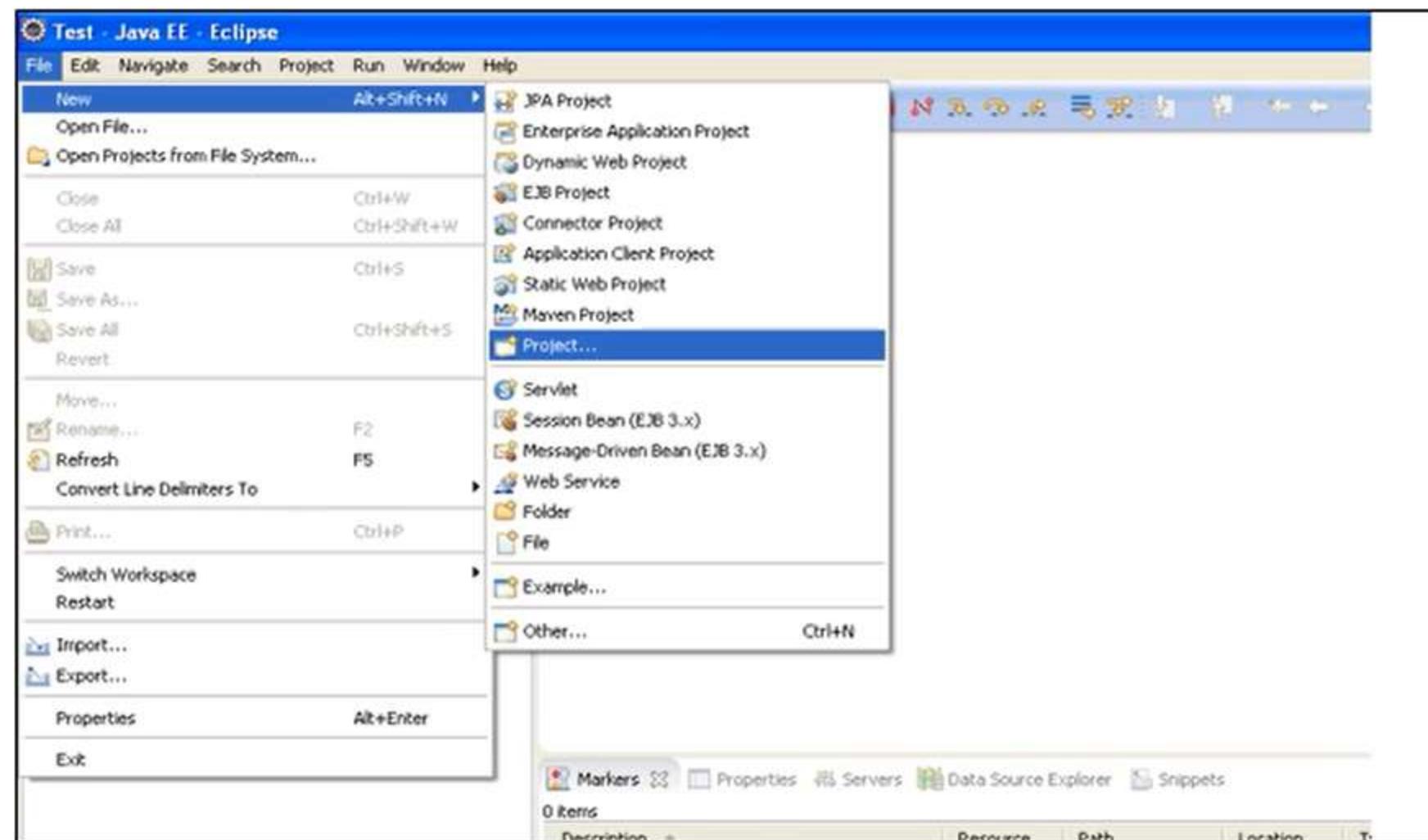


Create Java Project

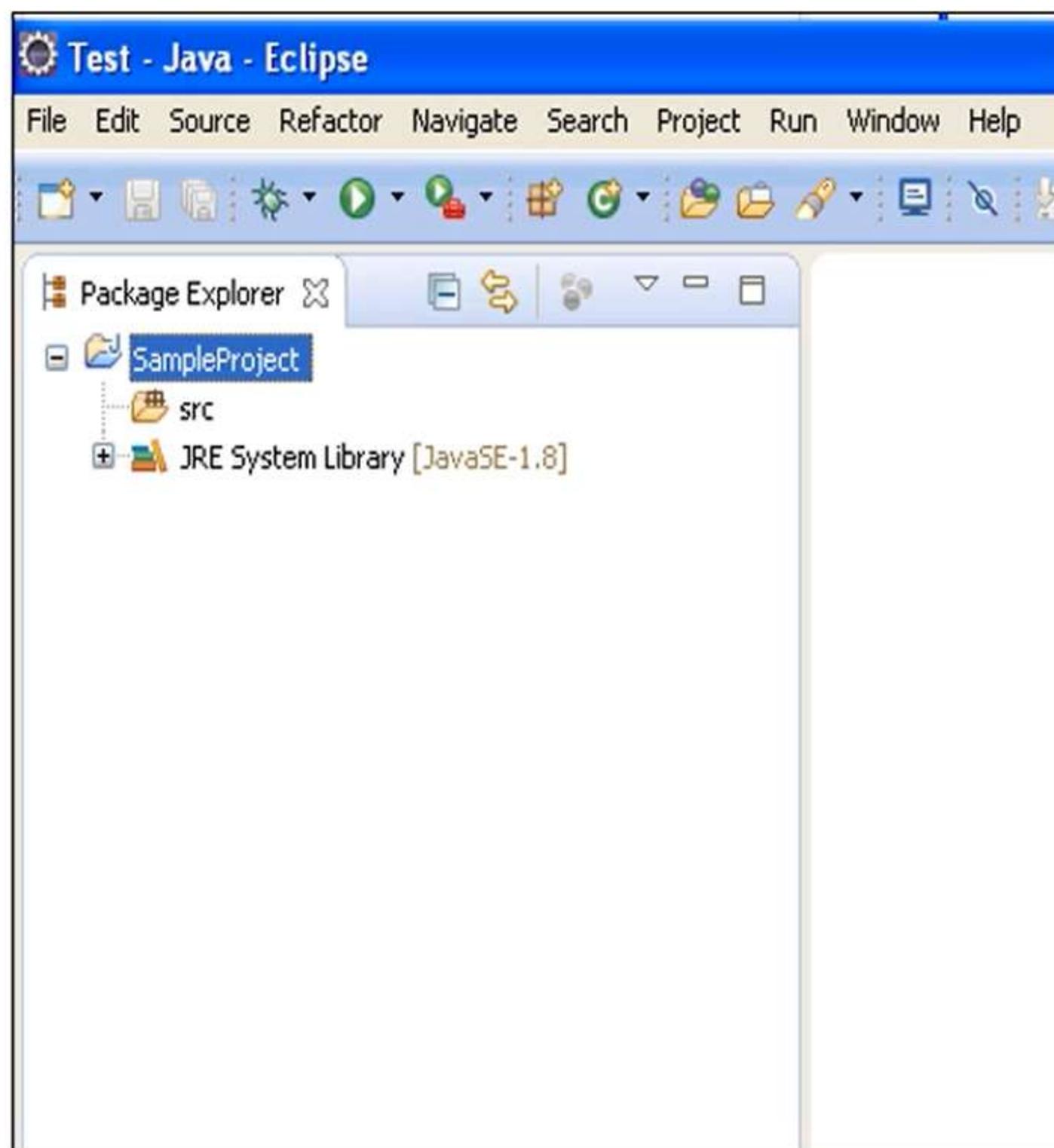
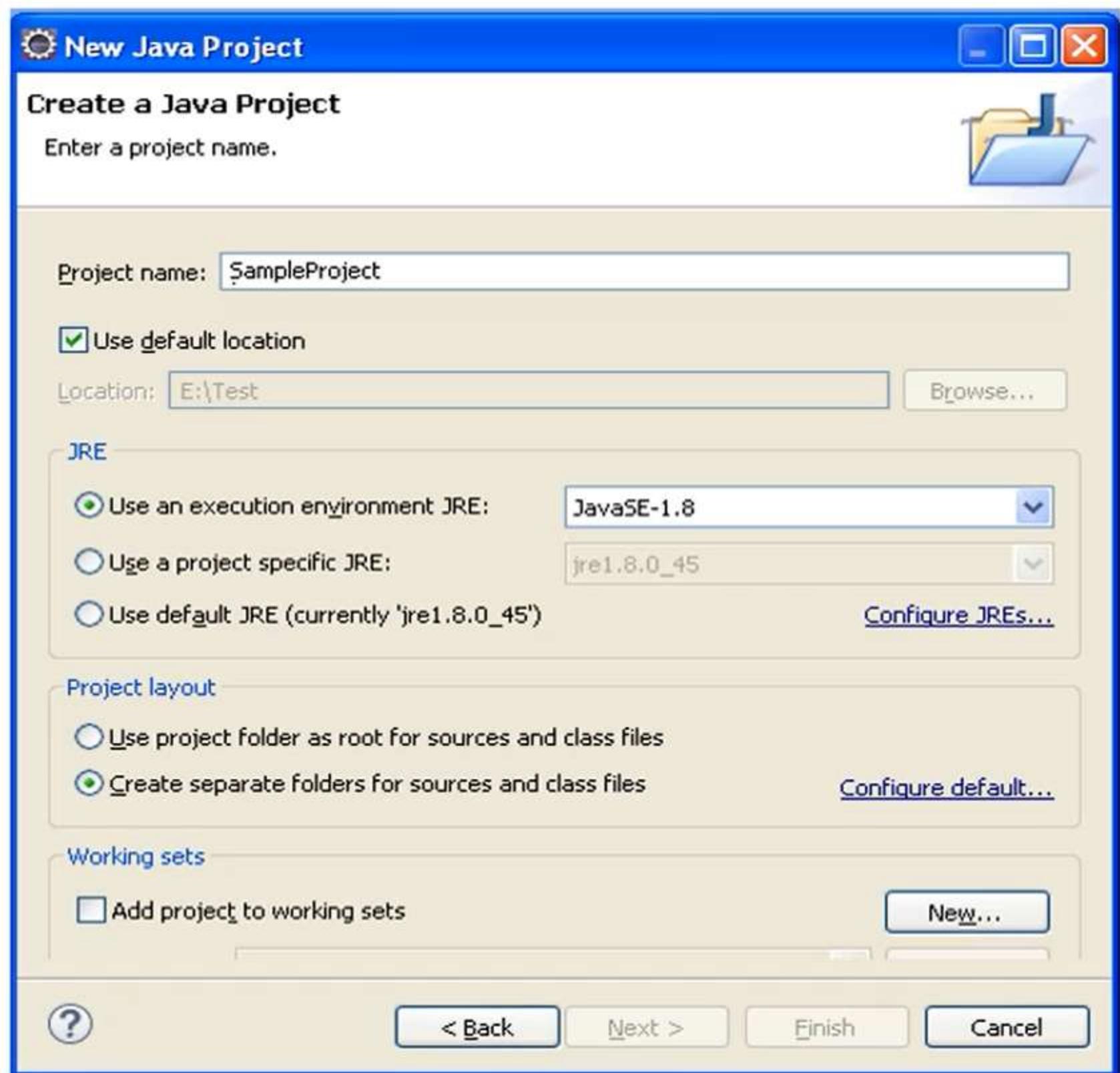
Select File -> New -> Java project from the menu

Enter the Project Name and Click Finish

A Java Project is created and opened in Java Perspective



Create Java Project



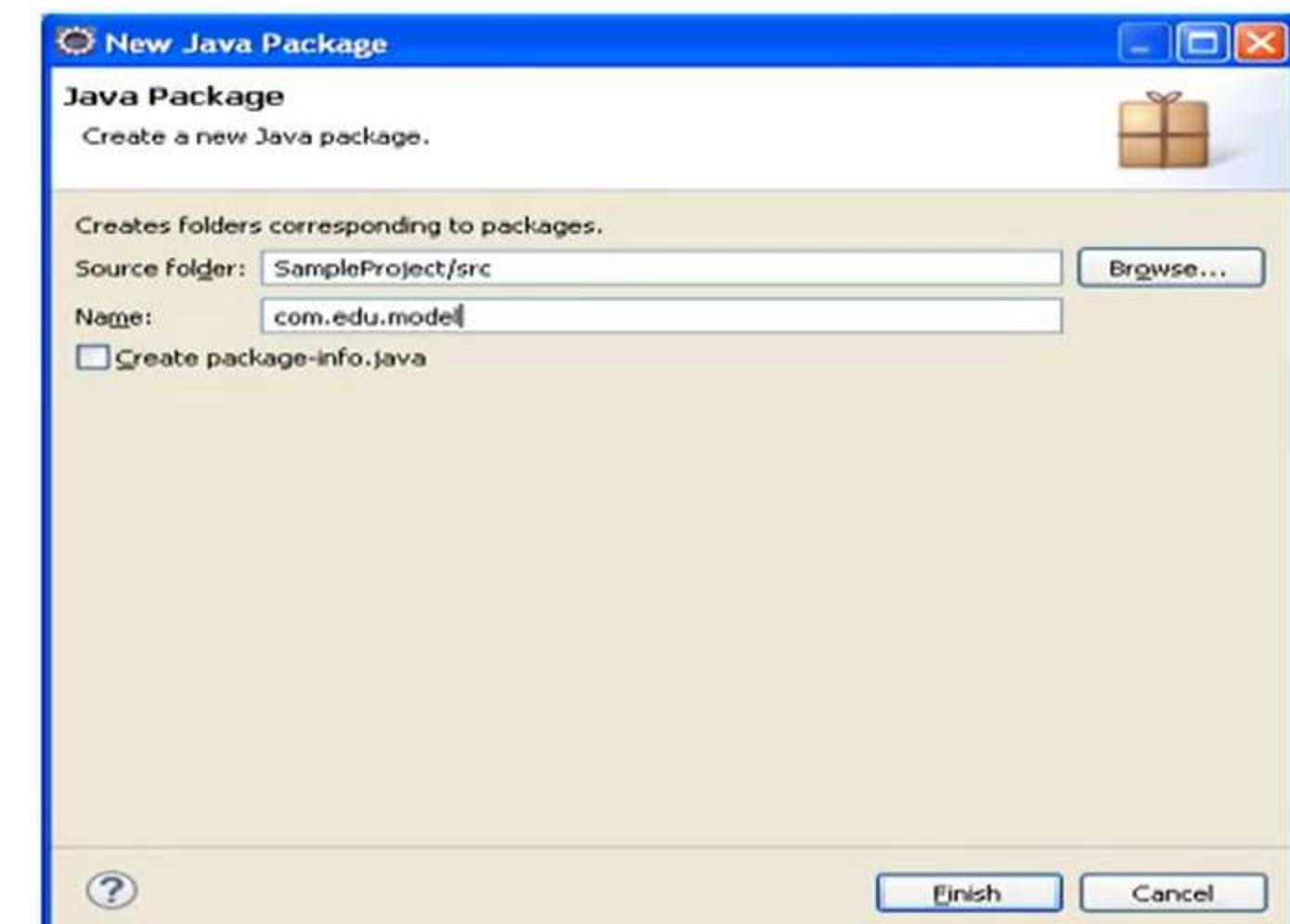
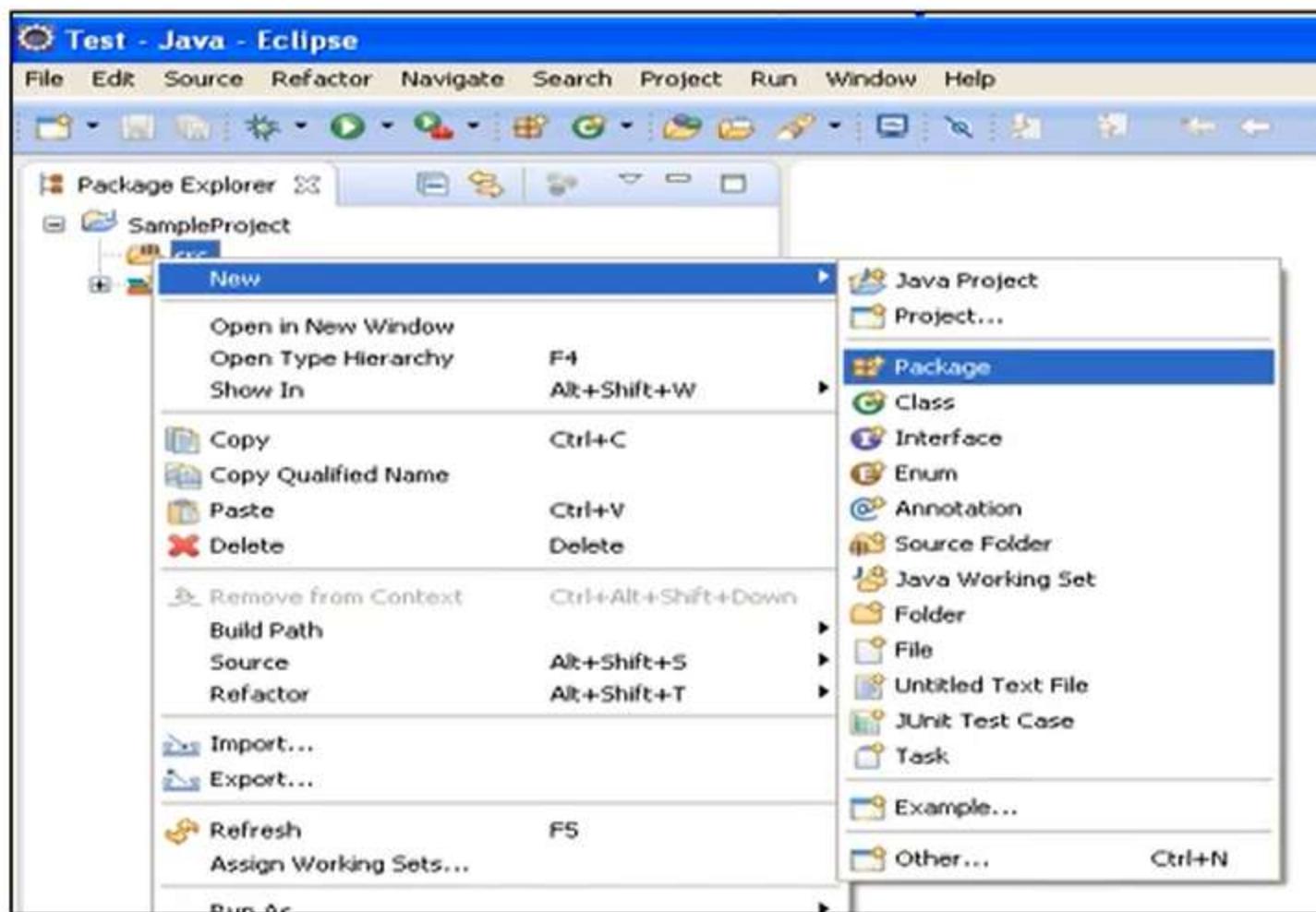
Create a package

A package is a namespace that groups related classes

Create a new package

- Right click the “src” folder in the project -> New - > Package

Enter the package name and press Finish

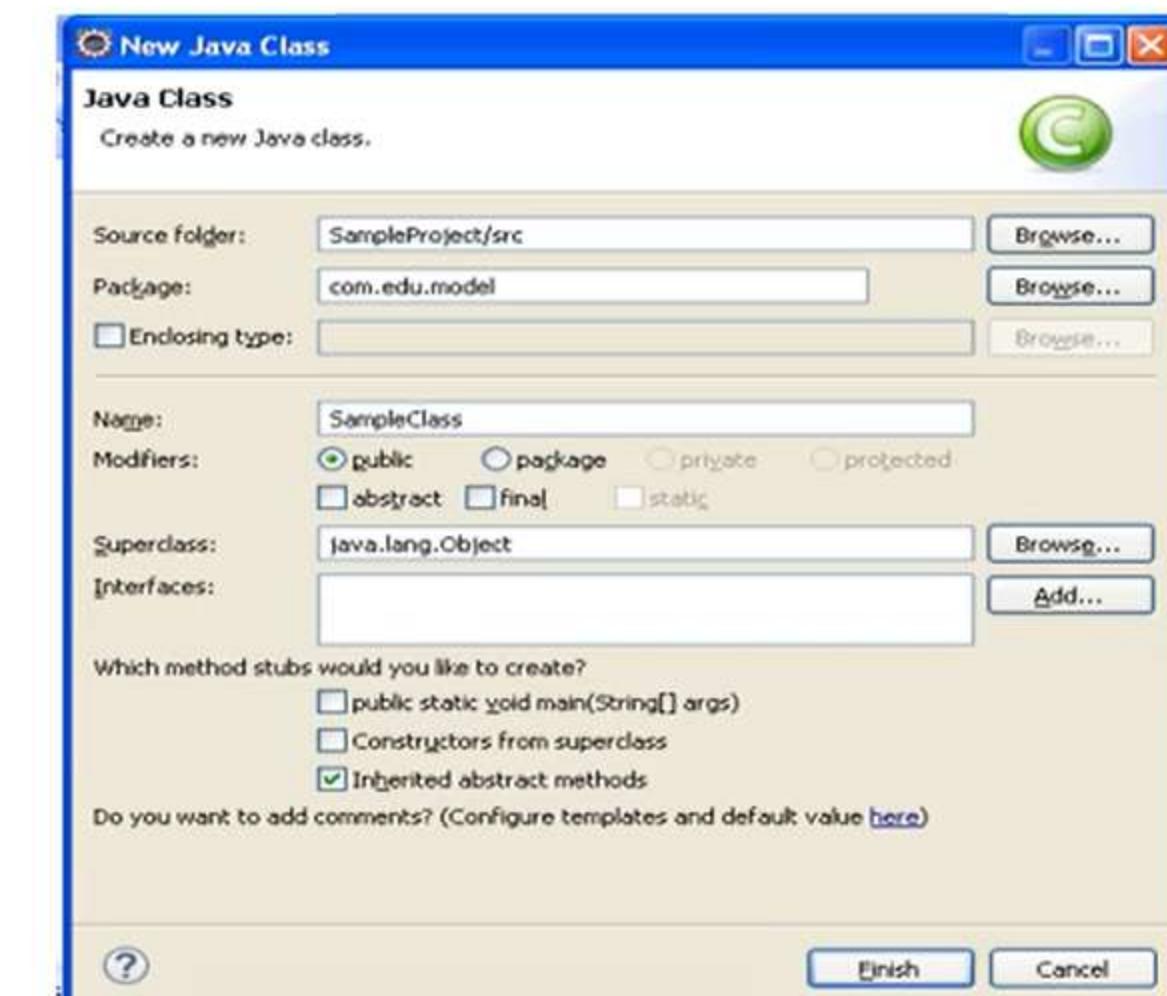
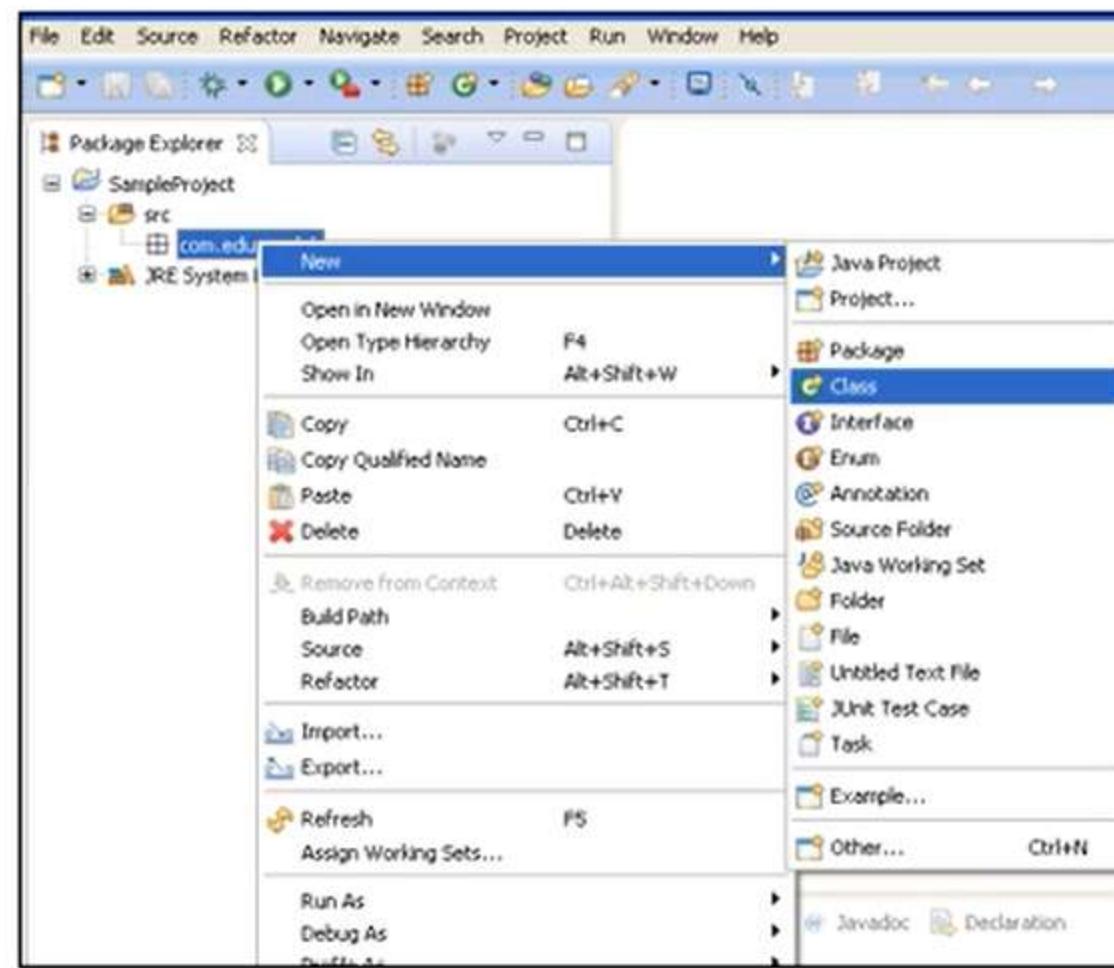


Create a class

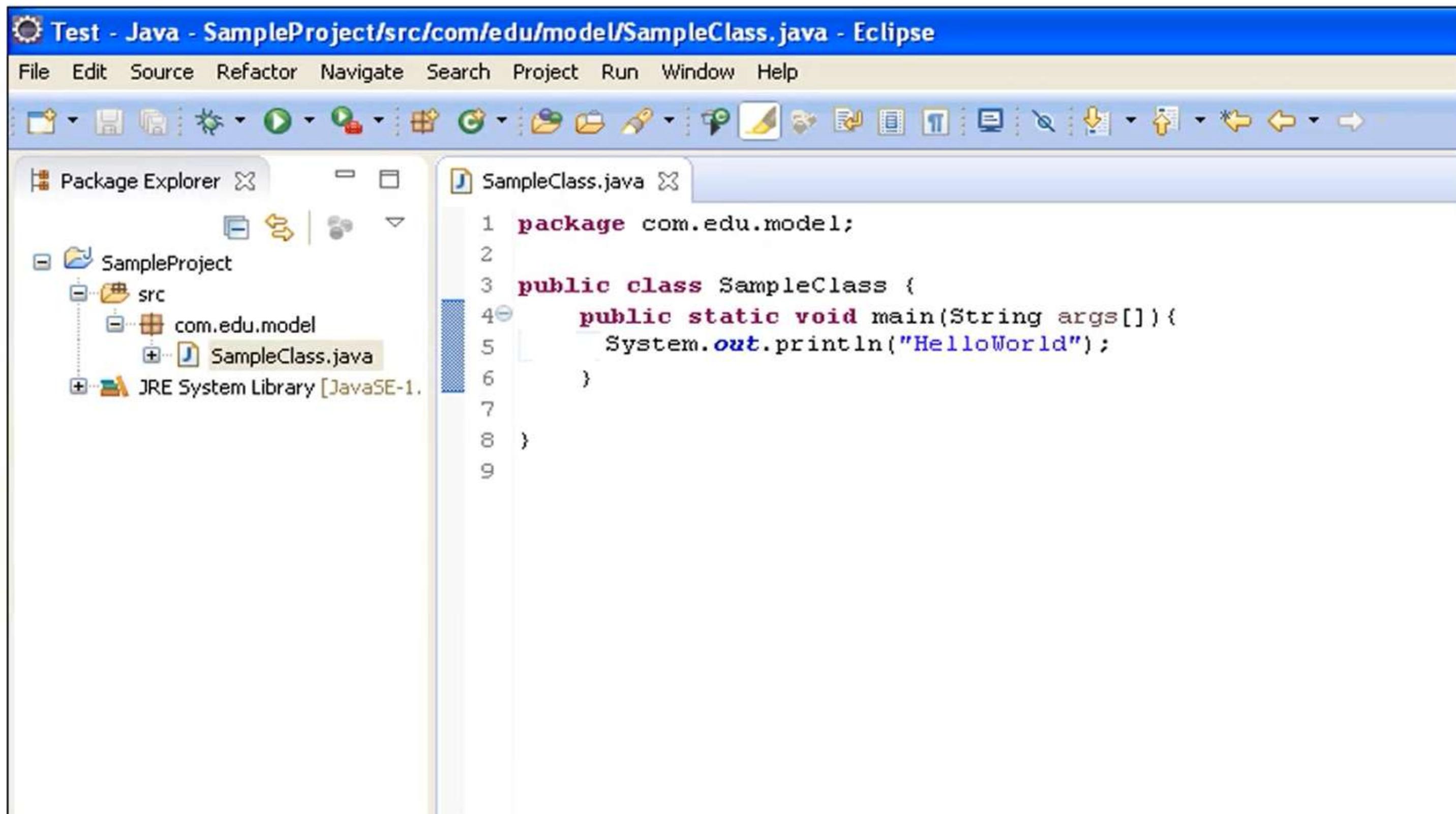
Right click the package -> New -> Class

Enter the class name, say SampleClass

Select the *public static void main (String[] args)* checkbox.(optional)



Create a class



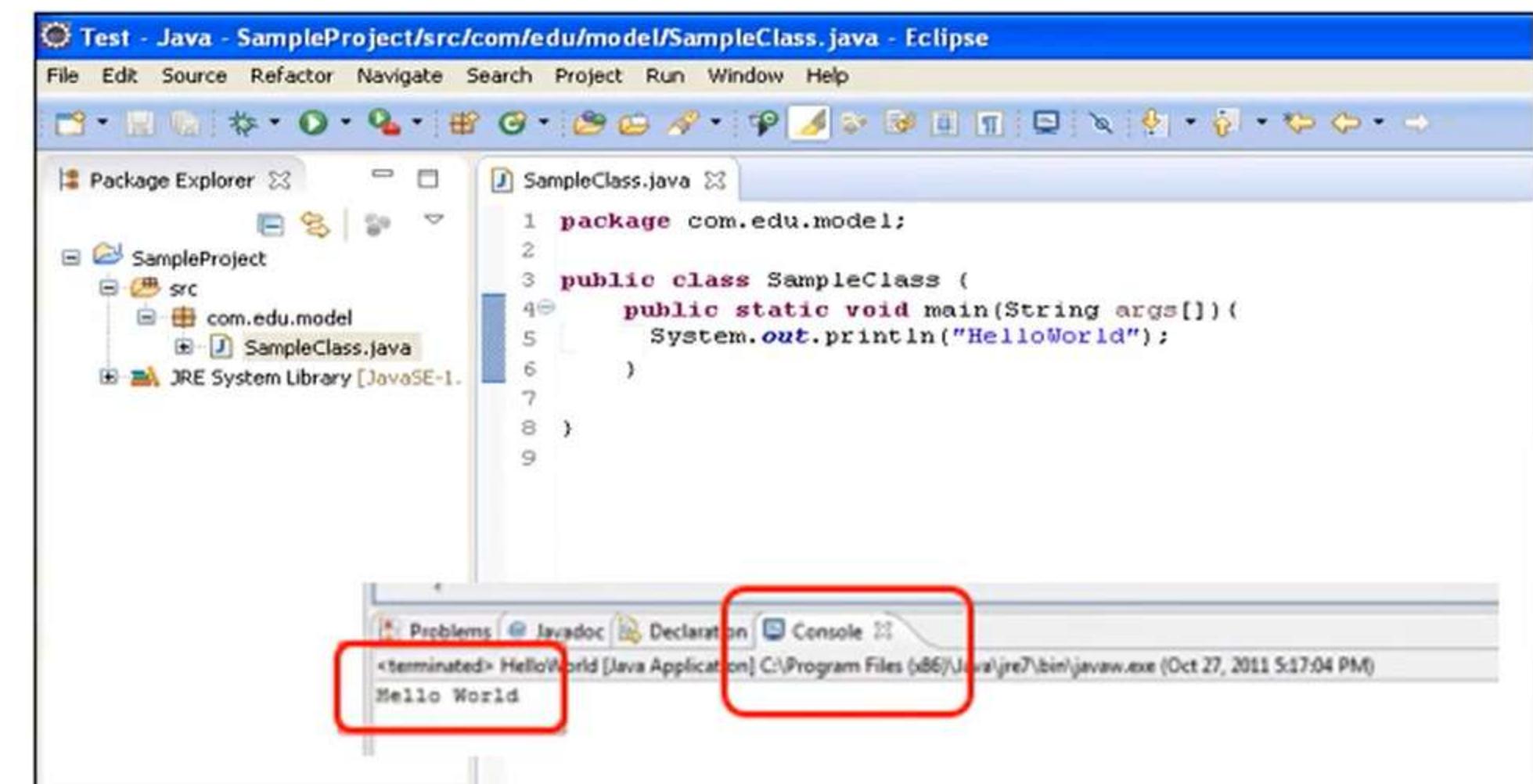
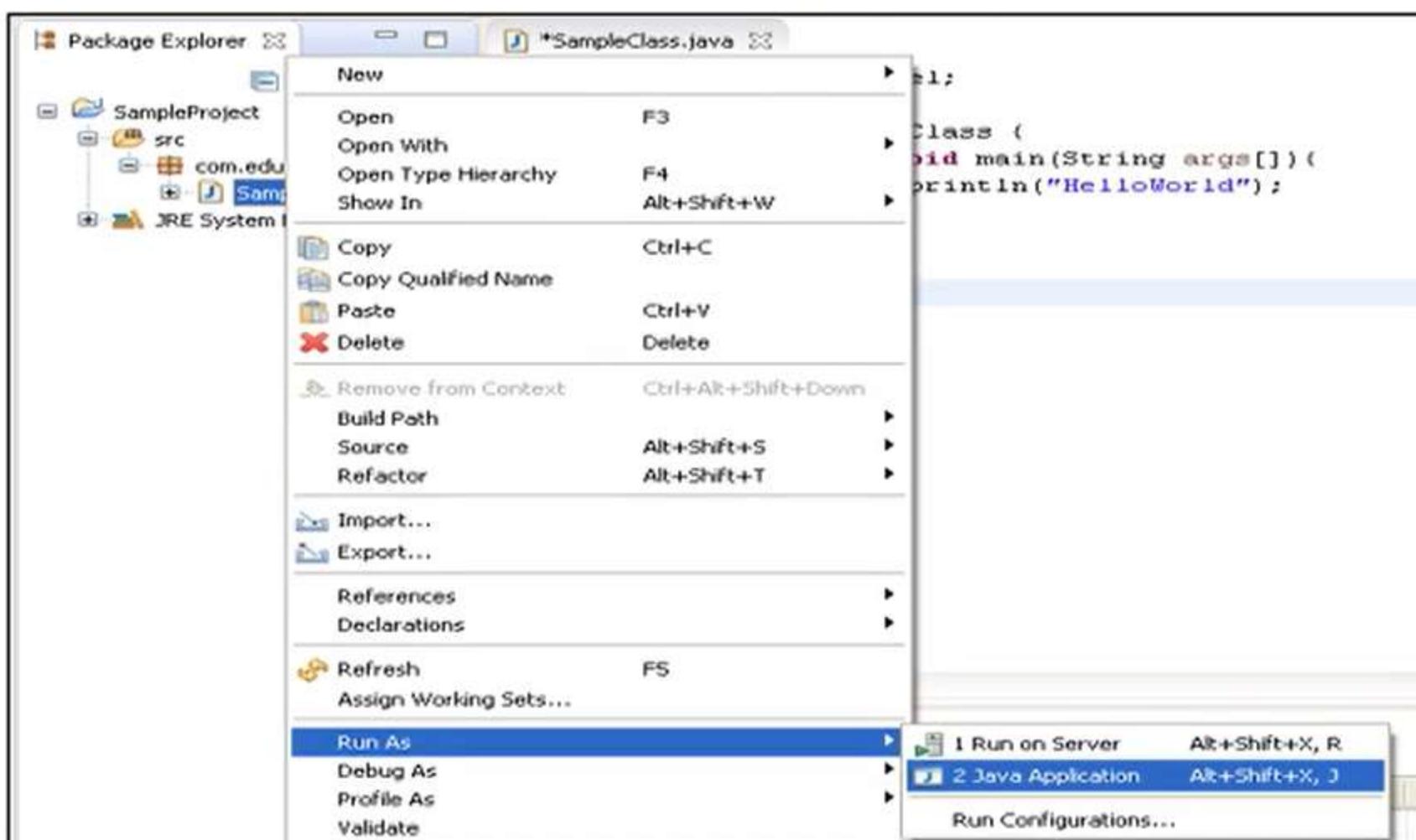
The screenshot shows the Eclipse IDE interface. The title bar reads "Test - Java - SampleProject/src/com/edu/model/SampleClass.java - Eclipse". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar below has various icons for file operations like Open, Save, Cut, Copy, Paste, Find, and Run. The left side features the "Package Explorer" view, which displays the project structure: "SampleProject" containing "src" (with "com.edu.model" package and "SampleClass.java" file selected), and "JRE System Library [JavaSE-1.8]". The right side is the "SampleClass.java" editor window, showing the following Java code:

```
1 package com.edu.model;
2
3 public class SampleClass {
4     public static void main(String args[]) {
5         System.out.println("HelloWorld");
6     }
7
8 }
9
```

Execute the Program

Right click the class file -> Run As -> Java Application

- Output is seen in the console view

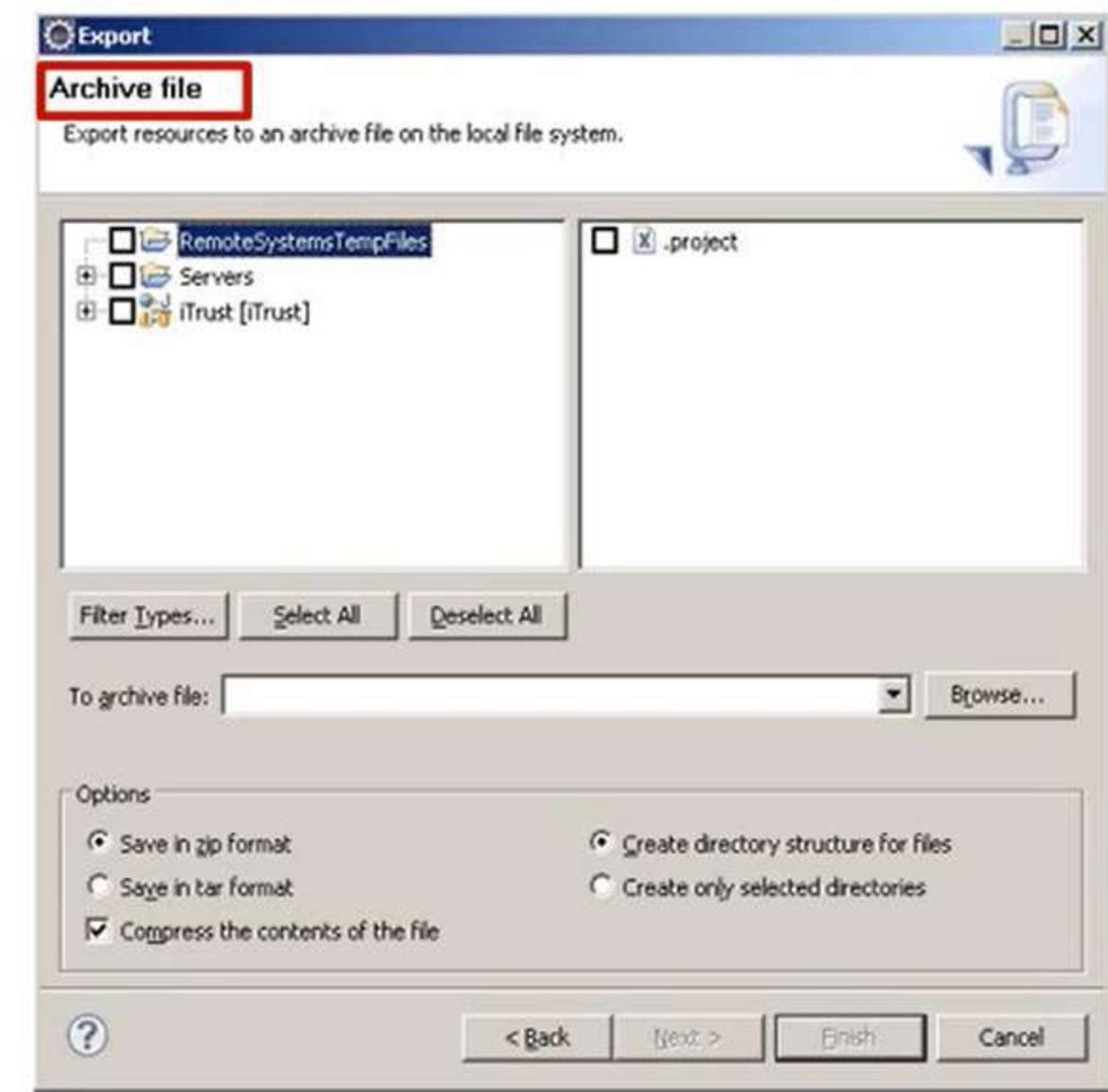
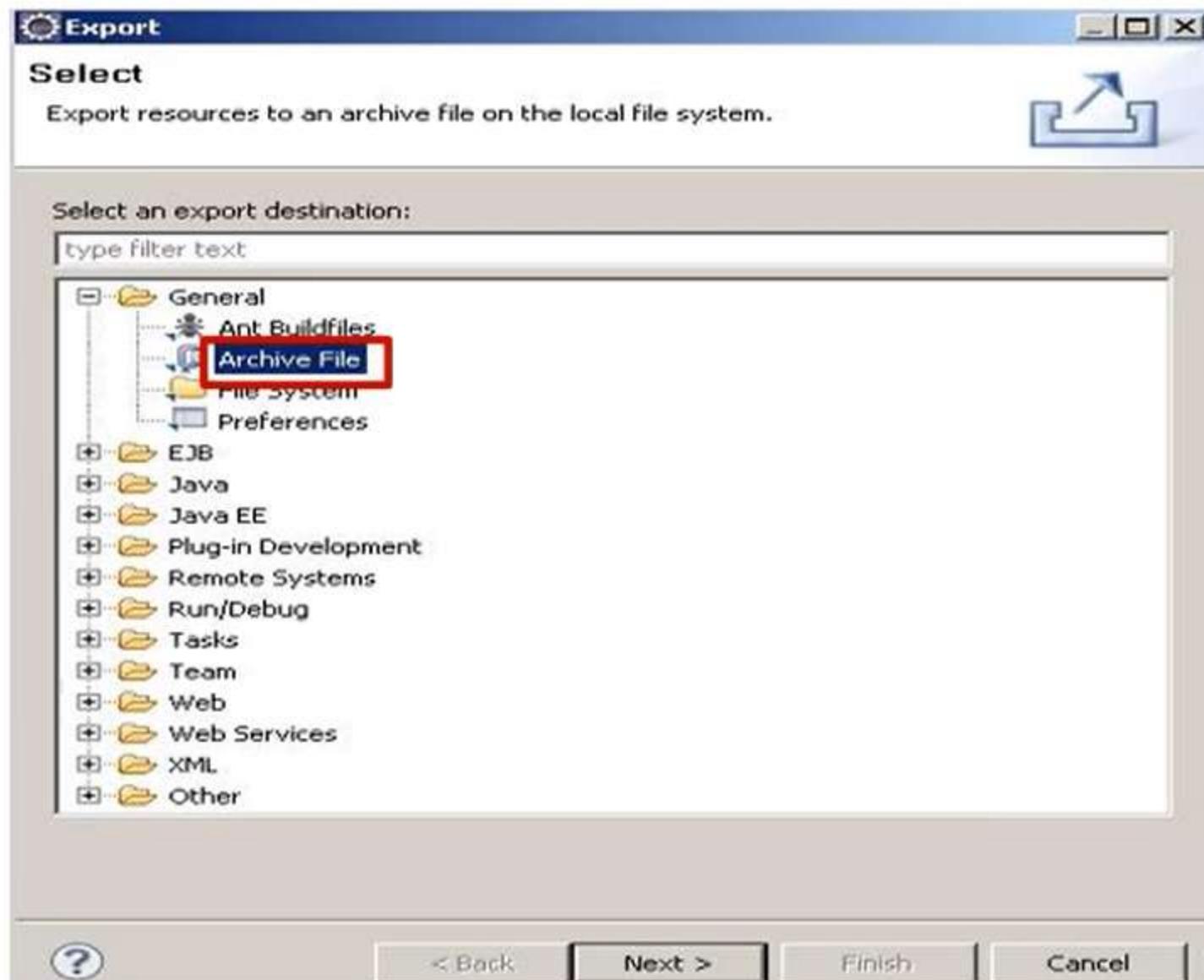


Also the class can be executed by clicking the Run icon  on the toolbar

Miscellaneous Options – Export Project

Can Export an entire Eclipse project into an archive file,

- Go to File -> Export.



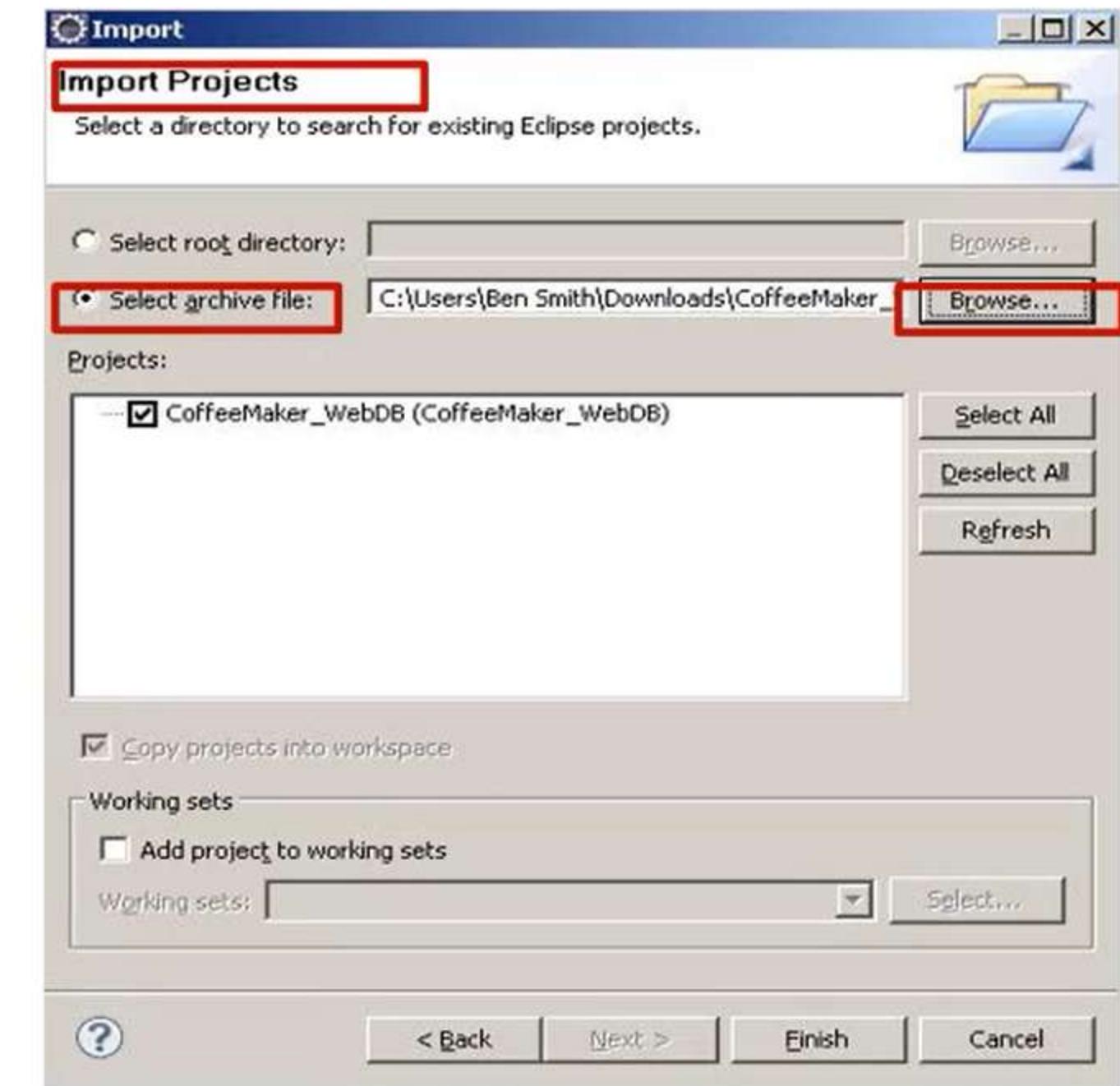
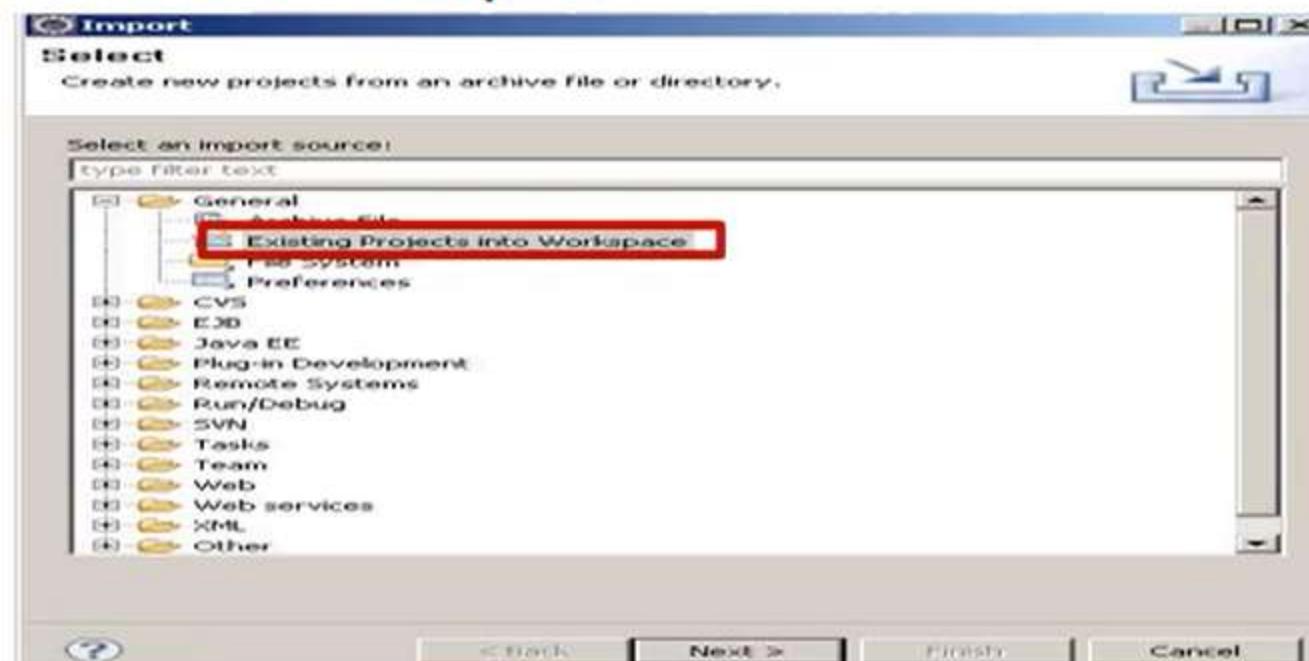
Click Next

Click **Finish**. Eclipse will automatically export and zip the project into an archive file and store it in the location specified.

Miscellaneous Options – Import Project

Can Import an entire Eclipse project from an archive file

- Go to File -> Import.



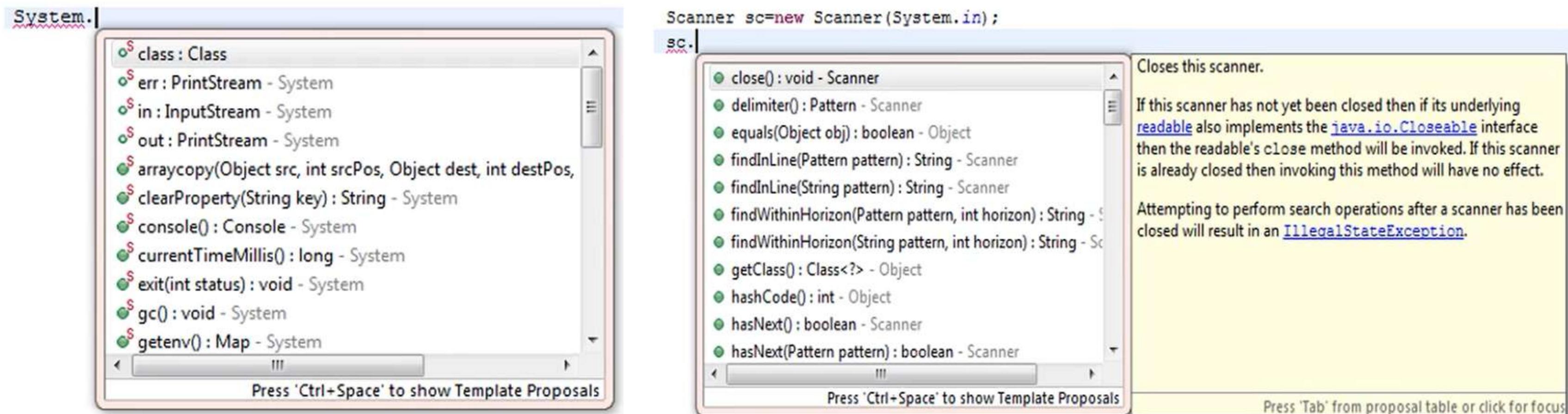
- Select **Existing Projects into Workspace**.
- Click the radio button next to **Select archive file** and click the **Browse** button
- Choose the file path for the zip file
- Click finish 

Miscellaneous Options – Code Assist

When typing the code in the text editor, eclipse provides code assist, termed as Intellisense.

Code Assist is

- Either triggered automatically (when we type a dot after variable or class name)
- Or forced by pressing Ctrl + Space on code



Miscellaneous Options – Eclipse Shortcuts

Eclipse IDE has many shortcuts that help in fast development

Few shortcuts that are often used:

Shortcut	Description
CTRL SPACE	Type assist
CTRL SHIFT F	Format code
CTRL /	Comment a line
F3	Go to the declaration of the variable
F4	Show type hierarchy of a class
ALT SHIFT Z	Enclose block in try-catch
CTRL F11	Run last run program
CTRL 1	Quick fix code

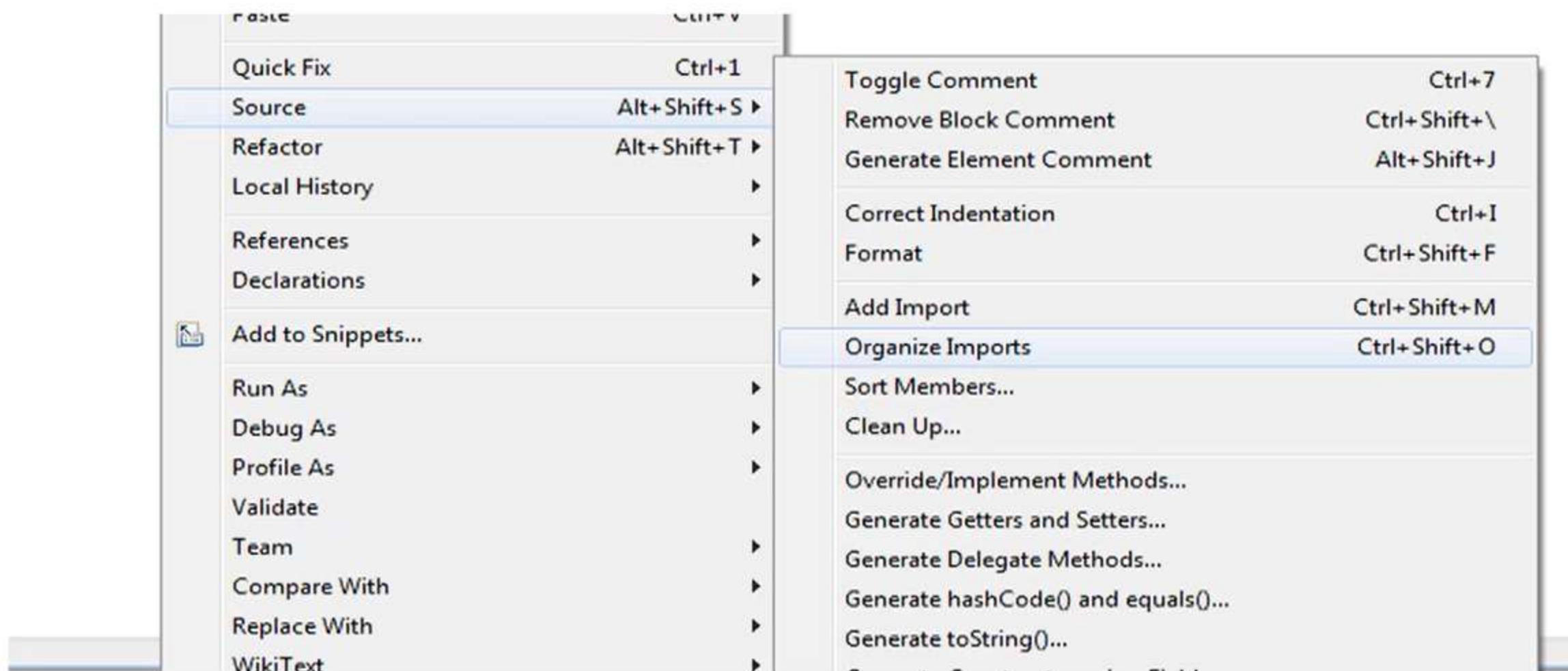
Miscellaneous Options – Organize Imports

Using Eclipse IDE, all the classes needed for a file can be imported automatically

Right click -> Source -> Organize Imports

This will import the class files needed

If unwanted class files are imported, those will be deleted



Miscellaneous Options – Auto Format

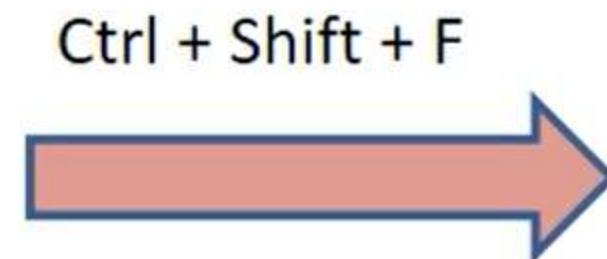
Eclipse provides auto format feature for the code

To make a code readable and maintainable , it should be properly aligned and indented

In Eclipse, press **Ctrl+Shift+F** on code to format it.

- If a portion of code is selected, it formats the selection.
- Else it formats the entire source file.

```
public class TestDemo {  
  
    public void add(int a,int b)  
    {  
        int c; c=a+b; int d; d=a-b;  
        int x=a*b;  
    }  
  
}
```



```
public class TestDemo {  
  
    public void add(int a, int b) {  
        int c;  
        c = a + b;  
        int d;  
        d = a - b;  
        int x = a * b;  
    }  
  
}
```

Miscellaneous Options – Fix Compilation Errors



In Eclipse, the code editor shows compilation error, if any, as the code is typed

IDE provides suggestion on how to fix the compilation error

A screenshot of the Eclipse IDE code editor. A cursor is positioned at the end of the line 'int c=a+b;'. A red error icon with a white 'X' is located to the left of the cursor. A light blue tooltip box is overlaid on the code, containing the error message 'Type mismatch: cannot convert from long to int'.

```
public long add(long a, long b)
{
    int c=a+b;
}
```

A screenshot of the Eclipse IDE code editor showing the same code as above. The cursor is now at the end of the line 'int c=a+b;'. A red error icon with a white 'X' is still present. A tooltip box is open, displaying the error message 'Type mismatch: cannot convert from long to int' and a suggested fix: 'Cast to int'.

```
public long add(long a, long b)
{
    int c=a+b;
}
```

Summary

- Introduction to Eclipse IDE
- Installation and Setting up of Eclipse
- Creating and Managing Java Projects
- Miscellaneous Options

