1 JAVA - OBJECT ORIENTED

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance

1.1 JAVA – ENCAPSULATION

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

Following is an example that demonstrates how to achieve Encapsulation in Java:

```
/* File name : EncapTest.java */
public class EncapTest {
  private String name;
  private String idNum;
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int newAge) {
    age = newAge;
  }

  public String getName() {
    return name;
```

```
  }

  public void setName(String newName) {
    name = newName;
  }

  public String getIdNum() {
    return idNum;
  }

  public void setIdNum(String newId) {
    idNum = newId;
  }
}
```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program:

```
/* File name : RunEncap.java */
public class RunEncap {
  public static void main(String args[]) {
    EncapTest encap = new EncapTest();
    encap.setName("James");
    encap.setAge(20);
    encap.setIdNum("12343ms");

    System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge());
  }
}
```

This will produce the following result:

```
Name : James Age : 20
```

1.1.1 Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

1.2 JAVA – ABSTRACTION

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise, in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

1.2.1 Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name : Employee.java */
public abstract class Employee {
  private String name;
  private String address;
  private int number;

  public Employee(String name, String address, int number) {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }

  public double computePay() {
    System.out.println("Inside Employee computePay");
    return 0.0;
  }

  public void mailCheck() {
```

```
    System.out.println("Mailing a check to " + this.name
        + " " + this.address);
  }

  public String toString() {
    return name + " " + address + " " + number;
  }

  public String getName() {
    return name;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String newAddress) {
    address = newAddress;
  }

  public int getNumber() {
    return number;
  }
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way:

```
/* File name : AbstractDemo.java */
public class AbstractDemo {
  public static void main(String[] args) {
/* Following is not allowed and would raise error */
    Employee e = new Employee("George W.", "Houston, TX", 43);

    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

When you compile the above class, it gives you the following error:

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

1.2.2 Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way:

```java
/* File name : Salary.java */
public class Salary extends Employee {
  private double salary; //Annual salary

  public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
  }

  public void mailCheck() {
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName()
        + " with salary " + salary);
  }

  public double getSalary() {
    return salary;
  }

  public void setSalary(double newSalary) {
    if (newSalary >= 0.0) {
      salary = newSalary;
    }
  }

  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary / 52;
  }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```java
/* File name : AbstractDemo.java */ public class AbstractDemo {
  public static void main(String[] args) {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();

    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

This produces the following result:

```
Constructing an Employee
Constructing an Employee

Call mailCheck using  Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0


Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

1.2.3 Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the abstract keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semi colon (;) at the end. Following is an example of the abstract method.

```
public abstract class Employee {
  private String name;
  private String address;
  private int number;

  public abstract double computePay();
//Remainder of class definition
}
```

Declaring a method as abstract has two consequences:

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note: Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below:

```
/* File name : Salary.java */ public class Salary extends Employee {
  private double salary; // Annual salary
```

```
  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary / 52;
  }
//Remainder of class definition
}
```

1.2.4 Java – Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

1.2.4.1 Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface.

Example

Following is an example of an interface:

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

```
/* File name : NameOfInterface.java */

import java.lang.*;
//Any number of import statements

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

Example

```
/* File name : Animal.java */
interface Animal {
  public void eat();

  public void travel();
}
```

1.2.4.2 Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

This will produce the following result:

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

```java
/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }
}
```

When implementation interfaces, there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

1.2.4.3 Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```java
//Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);

  public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);

  public void visitingTeamScored(int points);

  public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports {
```

```
  public void homeGoalScored();

  public void visitingGoalScored();

  public void endOfPeriod(int period);

  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

1.2.4.4 Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma- separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event
```

1.2.4.5 Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;

public interface EventListener

{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

Creates a common parent: As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class: This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

1.3 JAVA – POLYMORPHISM

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian {
}

public class Animal {
}

public class Deer extends Animal implements Vegetarian {
}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d=new Deer();
Animal a=d;
Vegetarian v=d;
Object o=d;
```

All the reference variables d, a, v, o refers to the same Deer object in the heap.

1.3.1 Java – Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {
  public static void main(String args[]) {
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move();// runs the method in Animal class
    b.move();//Runs the method in Dog class
  }
}
```

This will produce the following result:

```
Animals can move Dogs
can walk and run
```

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Consider the following example:

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
```

```
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }

  public void bark() {
    System.out.println("Dogs can bark");
  }
}

public class TestDog {
  public static void main(String args[]) {
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object
    a.move();// runs the method in Animal class
    b.move();//Runs the method in Dog class
    b.bark();
  }
}
```

This will produce the following result:

```
TestDog.java:30: cannot find symbol
symbol  : method bark()

location: class Animal
                b.bark();
                 ^
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

1.3.1.1 Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub lass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

1.3.1.2 Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

```
class Animal {

  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {

  public void move() {
    super.move(); // invokes the super class method
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {

  public static void main(String args[]) {
    Animal b = new Dog(); // Animal reference but Dog object
    b.move(); //Runs the method in Dog class
  }
}
```

This will produce the following result:

```
Animals can move Dogs
can walk and run
```

1.3.2 Java – Overload

is a feature that allows a class to have two or more methods having same name, if their argument lists are different.

Simple words mean two methods having same method name but takes different input parameters. This called static because, which method to be invoked will be decided at the time of compilation.

```
class Overload {
    void demo(int a) {
        System.out.println("a: " + a);
```

```
    }

    void demo(int a, int b) {
        System.out.println("a and b: " + a + "," + b);
    }

    double demo(double a) {
        System.out.println("double a: " + a);
        return a * a;
    }
}

class MethodOverloading {
    public static void main(String args[]) {
        Overload Obj = new Overload();
        Obj.demo(10);
        Obj.demo(10, 20);
        double result = Obj.demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

See how the methods

1.3.3 Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

```
/* File name : Employee.java */ public class Employee {
  private String name;
  private String address;
  private int number;

  public Employee(String name, String address, int number) {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }

  public void mailCheck() {
    System.out.println("Mailing a check to " + this.name
        + " " + this.address);
  }

  public String toString() {
    return name + " " + address + " " + number;
```

```
  }

  public String getName() {
    return name;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String newAddress) {
    address = newAddress;
  }

  public int getNumber() {
    return number;
  }
}
```

Now suppose we extend Employee class as follows:

```
/* File name : Salary.java */
public class Salary extends Employee {
  private double salary; //Annual salary

  public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
  }

  public void mailCheck() {
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName()
        + " with salary " + salary);
  }

  public double getSalary() {
    return salary;
  }

  public void setSalary(double newSalary) {
    if (newSalary >= 0.0) {
      salary = newSalary;
    }
  }

  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary / 52;
  }
}
```

Now, you study the following program carefully and try to determine its output:

```
/* File name : VirtualDemo.java */
public class VirtualDemo {
  public static void main(String[] args) {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
```

```
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();
    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

This will produce the following result:

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary
3600.0 Call mailCheck using Employee reference--

Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time. Invoking mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

1.4 JAVA – PACKAGES

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are:

- **java.lang** - bundles the fundamental classes
- java.io - classes for input, output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

1.4.1 Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals*:

```
/* File name : Animal.java */
package animals;

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

```
  public void travel();
}
```

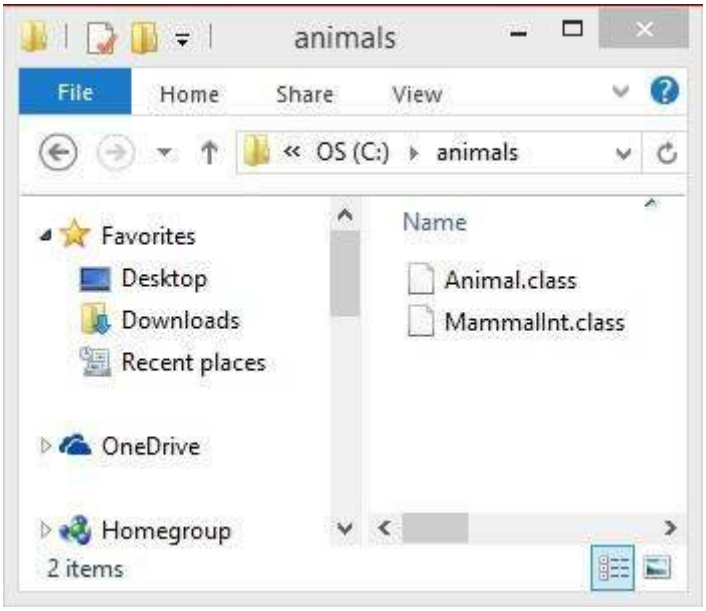
Now, let us implement the above interface in the same package *animals*:

```
package animals;

/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }
}
```

Now compile the java files as shown below:

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



You can execute the class file within the package and get the result as shown below.

```
$ java animals.MammalInt
Mammal eats

Mammal travels
```

1.4.2 The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss {
  public void payEmployee(Employee e) {
    e.mailCheck();
  }
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

1.4.3 The Directory Structure of Packages

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name : Car.java
package vehicle;

public class Car {
// Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as follows:

- Class name -> vehicle.Car
- Path name -> vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names. **Example**: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example:

```
// File Name: Dell.java

package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Now, compile this file as follows using -d option:

```
$javac -d . Dell.java
```

The files will be compiled as follows:

```
.\com\apple\computers\Dell.class

.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in *\com\apple\computers* as follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path- two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

1.4.4 Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):

- In Windows -> C:\> set CLASSPATH
- In UNIX -> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use:

- In Windows -> C:\> set CLASSPATH=
- In UNIX -> % unset CLASSPATH; export CLASSPATH To set the CLASSPATH variable:
- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

2 JAVA – EXCEPTIONS

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation; the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {
  public static void main(String args[]) {
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
  }
}
```

If you try to compile the above program, you will get the following exceptions.

```
C:\>javac FilenotFound_Demo.java

FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown

      FileReader fr = new FileReader(file);

                      ^

1 error
```

Note: Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions:** An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

```
public class Unchecked_Demo {

  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

If you compile and execute the above program, you will get the following exception.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
     at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

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

2.1 Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

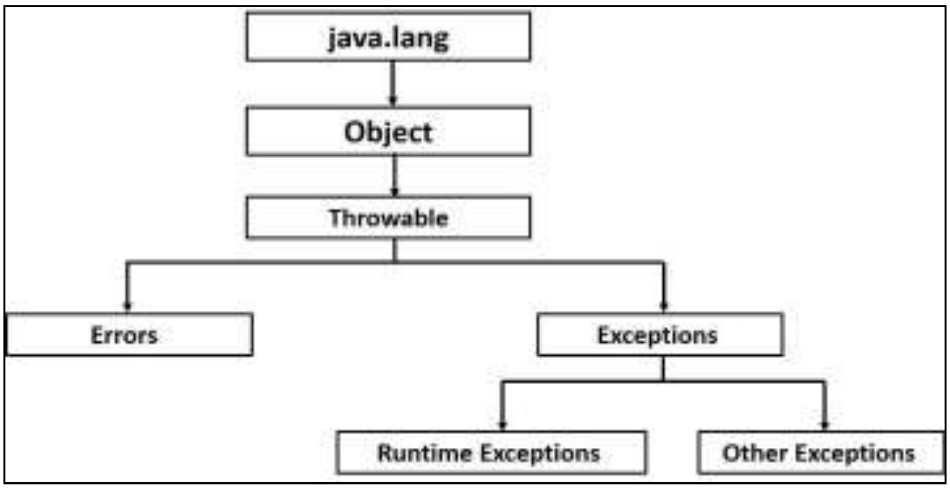
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by theruntime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.

Following is a list of most common checked and unchecked Java's Built-in Exceptions.

2.2 BUILT-IN EXCEPTIONS

Java defines several exception classes inside the standard package **java.lang**.



The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with the current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

2.3 EXCEPTIONS METHODS

Following is the list of important methods available in the Throwable class.

Sr. No.	Methods with Description

1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

2.4 CATCHING EXCEPTIONS

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

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

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java import java.io.*;
public class ExcepTest {
  public static void main(String args[]) {
    try {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This will produce the following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

2.5 Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

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

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
   return -1;
}
```

2.6 Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it:

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);

   throw ex;
```

2.7 The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException:

```
import java.io.*;

public class className {
  public void deposit(double amount) throws RemoteException {
  // Method implementation
    throw new RemoteException();
  }
//Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

2.8 The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
import java.io.*;

public class className {
  public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
// Method implementation
  }
```

```
//Remainder of class definition
}
```

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
{
   //The finally block always executes.
}
```

Example

```
public class ExcepTest {

  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    } finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This will produce the following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6

The finally statement is executed
```

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

2.9 THE TRY-WITH-RESOURCES

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {
  public static void main(String args[]) {
    FileReader fr = null;
    try {
      File file = new File("file.txt");
      fr = new FileReader(file);
      char[] a = new char[50];
      fr.read(a); // reads the content to the array
      for (char c : a)
        System.out.print(c); //prints the characters one by one
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        fr.close();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

```
try(FileReader fr=new FileReader("file path"))
   {
   //use the resource
   }catch(){
      //body of catch
    }
   }
```

Following is the program that reads the data in a file using try-with-resources statement.

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

  public static void main(String args[]) {
    try (FileReader fr = new FileReader("E://file.txt")) {
      char[] a = new char[50];
      fr.read(a); // reads the content to the array
      for (char c : a)
        System.out.print(c); //prints the characters one by one
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement AutoCloseable interface and the close() method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the deceleration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

2.10 User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{

}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
  private double amount;
  public InsufficientFundsException(double amount)
  {
    this.amount = amount;
  }
  public double getAmount()
  {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
import java.io.*;

public class CheckingAccount {
  private double balance;
  private int number;

  public CheckingAccount(int number) {
    this.number = number;
  }

  public void deposit(double amount) {
    balance += amount;
```

```
  }

  public void withdraw(double amount) throws InsufficientFundsException {
    if (amount <= balance) {
      balance -= amount;
    } else {
      double needs = amount - balance;
      throw new InsufficientFundsException(needs);
    }
  }

  public double getBalance() {
    return balance;
  }

  public int getNumber() {
    return number;
  }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
 // File Name BankDemo.java
public class BankDemo {
  public static void main(String[] args) {
    CheckingAccount c = new CheckingAccount(101);
    System.out.println("Depositing $500...");
    c.deposit(500.00);
    try {
      System.out.println("\nWithdrawing $100...");
      c.withdraw(100.00);
      System.out.println("\nWithdrawing $600...");
      c.withdraw(600.00);
    } catch (InsufficientFundsException e) {
      System.out.println("Sorry, but you are short $" + e.getAmount());
      e.printStackTrace();
    }
  }
}
```

Compile all the above three files and run BankDemo. This will produce the following result:

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...

Sorry, but you are short $200.0

InsufficientFundsException

        at CheckingAccount.withdraw(CheckingAccount.java:25)

        at BankDemo.main(BankDemo.java:13)
```

2.11 COMMON EXCEPTIONS

In Java, it is possible to define two catergories of Exceptions and Errors.

- **JVM Exceptions:** These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
- **Programmatic Exceptions:** These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

3 JAVA – INNER CLASSES

3.1 NESTED CLASSES

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

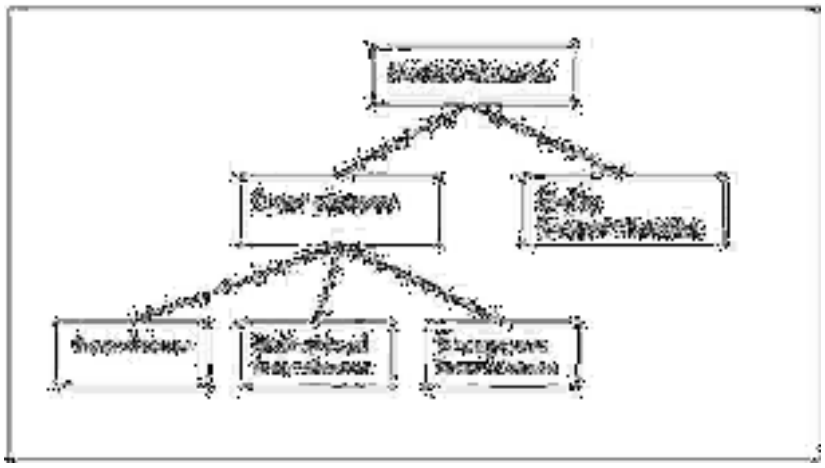
Syntax

Following is the syntax to write a nested class. Here, the class **Outer_Demo** is the outer class and the class **Inner_Demo** is the nested class.

```
class Outer_Demo{
   class Nested_Demo{

   }
}
```

Nested classes are divided into two types:

- **Non-static nested classes:** These are the non-static members of a class.
- **Static nested classes:** These are the static members of a class.



3.2 INNER CLASSES (NON-STATIC NESTED CLASSES)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are:

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

```
class Outer_Demo {
  int num;

  //Accessing he inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();

    //inner class
    private class Inner_Demo {
      public void print() {
        System.out.println("This is an inner class");
      }
    }
    inner.print();
  }
}

public class My_class {
  public static void main(String args[]) {
    //Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();
    //Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

Here you can observe that **Outer_Demo** is the outer class, **Inner_Demo** is the inner class, **display_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

If you compile and execute the above program, you will get the following result.

```
This is an inner class.
```

3.3 Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer=new Outer_Demo();
Outer_Demo.Inner_Demo inner=outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

```
class Outer_Demo {
  //private variable of the outer class
  private int num = 175;

  //inner class
  public class Inner_Demo {
    public int getNum() {
      System.out.println("This is the getnum method of the inner class");
      return num;
    }
  }
}

public class My_class2 {
  public static void main(String args[]) {
    //Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();
    //Instantiating the inner class
    Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
    System.out.println(inner.getNum());
  }
}
```

If you compile and execute the above program, you will get the following result.

```
The value of num in the class Test is: 175
```

3.4 Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

```
public class Outerclass {

  public static void main(String args[]) {
    Outerclass outer = new Outerclass();
    outer.my_Method();
  }

  //instance method of the outer class
  void my_Method() {
    int num = 23;

    //method-local inner class
    class MethodInner_Demo {
      public void print() {
        System.out.println("This is method inner class " + num);
      }
    }//end of inner class

    //Accessing the inner class
    MethodInner_Demo inner = new MethodInner_Demo();
    inner.print();
  }
}
```

If you compile and execute the above program, you will get the following result.

```
This is method inner class 23
```

3.5 Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

```
AnonymousInner an_inner = new AnonymousInner(){
   public void my_method(){

   ........

   ........

   }
};
```

The following program shows how to override the method of a class using anonymous inner class.

```
abstract class AnonymousInner {
  public abstract void mymethod();
}

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

```
    AnonymousInner inner = new AnonymousInner() {
      public void mymethod() {
        System.out.println("This is an example of anonymous inner
class");
      }
    };
    inner.mymethod();
  }
}
```

If you compile and execute the above program, you will get the following result.

```
This is an example of anonymous inner class
```

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

3.6 Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument:

```
obj.my_Method(new My_Class(){
   public void Do(){

   }
});
```

The following program shows how to pass an anonymous inner class as a method argument.

```
interface Message {
  String greet();
}

public class My_class {
  public static void main(String args[]) {
//Instantiating the class
    My_class obj = new My_class();
//Passing an anonymous inner class as an argument
    obj.displayMessage(new Message() {
      public String greet() {
        return "Hello";
      }
    });
  }

  //method which accepts the object of interface Message
  public void displayMessage(Message m) {
    System.out.println(m.greet() + ", This is an example of anonymous
```

```
inner calss as an argument");
   }
}
```

If you compile and execute the above program, it gives you the following result.

```
Hello This is an example of anonymous inner class as an argument
```

3.7 Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows:

```
class MyOuter {

   static class Nested_Demo{

   }

}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

```
public class Outer {
  public static void main(String args[]) {
    Outer.Nested_Demo nested = new Outer.Nested_Demo();
    nested.my_method();
  }

  static class Nested_Demo {
    public void my_method() {
      System.out.println("This is my nested class");
    }
  }
}
```

If you compile and execute the above program, you will get the following result.

```
This is my nested class
```

4 JAVA – DATA STRUCTURES

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:

- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties

All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework, which is discussed in the next chapter.

4.1 THE ENUMERATION

The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.

For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

To have more detail about this interface, check The Enumeration.

4.1.1 The Enumeration Interface

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table:

Sr. No.	Methods with Description

1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

4.1.2 Example

Following is an example showing usage of Enumeration.

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {

  public static void main(String args[]) {
    Enumeration days;
    Vector dayNames = new Vector();
    dayNames.add("Sunday");
    dayNames.add("Monday");
    dayNames.add("Tuesday");
    dayNames.add("Wednesday");
    dayNames.add("Thursday");
    dayNames.add("Friday");
    dayNames.add("Saturday");
    days = dayNames.elements();
    while (days.hasMoreElements()) {
      System.out.println(days.nextElement());
    }
  }
}
```

This will produce the following result:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

4.2 THE BITSET

The BitSet class implements a group of bits or flags that can be set and cleared individually.

This class is very useful in cases where you need to keep up with a set of Boolean values; you just assign a bit to each value and set or clear it as appropriate.

For more details about this class, check The BitSet.

4.2.1 The BitSet Class

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

Sr. No.	Constructor and Description
1	**BitSet( )** This constructor creates a default object
2	**BitSet(int size)** This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero

4.2.2 Example

The following program illustrates several of the methods supported by this data structure:

```
import java.util.BitSet;

public class BitSetDemo {

  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for (int i = 0; i < 16; i++) {
      if ((i % 2) == 0) bits1.set(i);
      if ((i % 5) != 0) bits2.set(i);
    }
    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);
    // AND bits bits2.and(bits1);
```

```
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

This will produce the following result:

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}
bits2 AND bits1:
{2, 4, 6, 8, 12, 14}
bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
bits2 XOR bits1:
{}
```

4.3 The Vector

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a Vector object can be accessed via an index into the vector.

The nice thing about using the Vector class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

For more details about this class, check The Vector.

4.3.1 The Vector Class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.

- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr. No.	Constructor and Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c

4.3.2 Example

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;

public class VectorDemo {

  public static void main(String args[]) {
    // initial size is 3, increment is 2
    Vector v = new Vector(3, 2);
    System.out.println("Initial size: " + v.size());

    System.out.println("Initial capacity: " + v.capacity());
    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Capacity after four additions: " +
v.capacity());
    v.addElement(new Double(5.45));
```

```
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
    System.out.println("First element: " +
        (Integer) v.firstElement());
    System.out.println("Last element: " +
        (Integer) v.lastElement());
    if (v.contains(new Integer(3)))
      System.out.println("Vector contains 3.");
    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();
    System.out.println("\nElements in vector:");
    while (vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

This will produce the following result:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5

Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.
```

4.4 The Stack

The Stack class implements a last-in-first-out (LIFO) stack of elements.

You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.

When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.

For more details about this class, check The Stack.

The Stack Class

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines the following methods:

4.4.1 Example

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;

public class StackDemo {

  static void showpush(Stack st, int a) {
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }


  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }


  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }
  }
}
```

This will produce the following result:

```
stack: [ ]
push(42) stack:
[42] push(66)
stack: [42, 66]
push(99)

stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]

pop -> empty stack
```

4.5 THE DICTIONARY

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.

This is useful in cases where you want to be able to access data via a particular key rather than an integer index.

Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.

For more details about this class, check The Dictionary.

4.5.1 The Dictionary Class

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

The abstract methods defined by Dictionary are listed below:

Sr. No.	Methods with Description
1	**Enumeration elements( )** Returns an enumeration of the values contained in the dictionary.
2	**Object get(Object key)** Returns the object that contains the value associated with the key. If the key is not in the dictionary, a null object is returned.
3	**boolean isEmpty( )** **Returns true if the dictionary is empty, and returns false if it contains at least one key.**
4	**Enumeration keys( )** **Returns an enumeration of the keys contained in the dictionary.**
5	**Object put(Object key, Object value)** **Inserts a key and its value into the dictionary. Returns null if the key is not already in the dictionary; returns the previous value associated with the key if the key is already in the dictionary.**
6	**Object remove(Object key)** **Removes the key and its value. Returns the value associated with the key. If the key is not in the dictionary, a null is returned.**
7	**int size( )** **Returns the number of entries in the dictionary.**

The Dictionary class is obsolete. You should implement the Map interface to obtain key/value storage functionality.

4.5.2 Example

Map has its implementation in various classes like HashMap. Following is an example to explain map functionality:

```
import java.util.*;

public class CollectionsDemo {

  public static void main(String[] args) {
    Map m1 = new HashMap();
    m1.put("Zara", "8");
    m1.put("Mahnaz", "31");
    m1.put("Ayan", "12");
    m1.put("Daisy", "14");
    System.out.println();
    System.out.println(" Map Elements");
    System.out.print("\t" + m1);
  }
}
```

This will produce the following result:

```
ap Elements
        {Mahnaz=31, Ayan=12, Daisy=14, Zara=8}
```

4.6 The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.

The specific meaning of keys with regard to hash tables is totally dependent on the usage of the hash table and the data it contains.

For more detail about this class, check The Hashtable.

4.6.1 The Hashtable Class

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Following is the list of constructors provided by the HashTable class.

Sr. No.	Constructor and Description
1	**Hashtable( )** This is the default constructor of the hash table it instantiates the Hashtable class.
2	**Hashtable(int size)** This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size.
3	**Hashtable(int size, float fillRatio)** **This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.**
4	**Hashtable(Map<? extends K, ? extends V> t)** **This constructs a Hashtable with the given mappings.**

4.6.2 Example

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;

public class HashTableDemo {

  public static void main(String args[]) {
    // Create a hash map
    Hashtable balance = new Hashtable();
    Enumeration names;
    String str;
    double bal;
    balance.put("Zara", new Double(3434.34));
    balance.put("Mahnaz", new Double(123.22));
```

```
    balance.put("Ayan", new Double(1378.00));
    balance.put("Daisy", new Double(99.22));
    balance.put("Qadir", new Double(-19.08));

    // Show all balances in hash table.
    names = balance.keys();
    while (names.hasMoreElements()) {
      str = (String) names.nextElement();
      System.out.println(str + ": " + balance.get(str));
    }
    System.out.println();
    // Deposit 1,000 into Zara's account
    bal = ((Double) balance.get("Zara")).doubleValue();
    balance.put("Zara", new Double(bal + 1000));
    System.out.println("Zara's new balance: " +
balance.get("Zara"));
  }
}
```

This will produce the following result:

```
Qadir: -19.08

Zara: 3434.34

ahnaz: 123.22

Daisy: 99.22

Ayan: 1378.0

Zara's new balance: 4434.34
```

4.7 The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

For more detail about this class, check The Properties.

4.7.1 The Properties Class

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

Properties define the following instance variable. This variable holds a default property list associated with a Properties object.

```
Properties defaults;
```

Following is the list of constructors provided by the properties class.

Sr. No.	Constructors and Description
1	**Properties( )** This constructor creates a Properties object that has no default values.
2	**Properties(Properties propDefault)** Creates an object that uses propDefault for its default values. In both cases, the property list is empty.

4.7.2 Example

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;


public class PropDemo {

  public static void main(String args[]) {
    Properties capitals = new Properties();
    Set states;
    String str;

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Show all states and capitals in hashtable.
    states = capitals.keySet();
    // get set-view of keys
    Iterator itr = states.iterator();
    while (itr.hasNext()) {
      str = (String) itr.next();
      System.out.println("The capital of " +
          str + " is " + capitals.getProperty(str) + ".");
```

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


    // look for state not in list -- specify default
    str = capitals.getProperty("Florida", "Not Found");
    System.out.println("The capital of Florida is "
        + str + ".");
  }
}
```

This will produce the following result:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.

The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```