COURSE:- Java Programming
Sem:- I
Module: 3

By:
Dr. Ritu Jain,
Associate Professor

Abstraction: Abstract Classes and Interface

Ways to achieve Abstraction in Java

There are two ways to achieve **abstraction** in java

1. Abstract class
2. Interface

abstract keyword in java

- In Java, abstract is a optional modifier in java applicable for **classes, and methods but not variables**.
- In Java, the abstract keyword is used to define abstract classes and methods.

Abstract Methods in Java

- Sometimes, we require just method declaration in super-classes. This can be achieved by specifying the **abstract** type modifier.
- These methods are sometimes referred to as *subclass responsibility* because they have no implementation specified in the super-class. Thus, a subclass must override them to provide a method definition.
- To declare an abstract method, use this general form:
  - **abstract type method-name(parameter-list);**
- As you can see, *no method body is present*.
- *Any class that contains one or more abstract methods must also be declared abstract*
-

Abstract methods and Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- This is the case with the class **Figure** used in the preceding example. The definition of **area( )** is simply a placeholder.

Abstract Classes in Java

- **Abstract Class:** The class which is having partial implementation(i.e. not all methods present in the class have method definitions). To declare a class abstract, use this general form :

```
abstract class class-name{
        //body of class
}
```

- Due to their partial implementation, *we cannot instantiate abstract classes*.
- *Any subclass of an abstract class must either implement all of the abstract methods in the super-class or be declared abstract itself.*

Abstract Class

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- If it is extended by any subclass, its abstract methods need to be implemented.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- You cannot declare abstract constructors, or abstract static methods.

Example of abstract class and abstract method

```
abstract class Bike{
  abstract void run();
}

class Honda extends Bike{
      void run(){
                  System.out.println("running safely”);
              }

      public static void main(String args[]){
              Bike obj = new Honda();
              obj.run();
      }
}
```

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Example 2 of abstract class and abstract method

Abstract methods and Abstract Classes

- *Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.*
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Example of abstract class and abstract method and run time polymorphism

```
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

```
class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());
    figref = t;
    System.out.println("Area is " + figref.area());
  }
}
```

Practice Question 1: Abstract Class Concept

Write a Java program to create an abstract class Animal with an abstract method called sound().

Create subclasses Lion and Cat that extend the Animal class and implement the sound() method to make a specific sound for each animal.

Shape3D: Practice Question 3 for Abstract Class Concept

Write a Java program to create an abstract class Shape3D with abstract methods calculateVolume() and calculateSurfaceArea().

Create subclasses Sphere and Cube that extend the Shape3D class and implement the respective methods to calculate the volume and surface area of each shape.

Interface

Interface

- **In Java, an interface is a blueprint of a class that is used to define a set of abstract methods (methods without body) and constants that other classes must implement.**
- It is one of the key tools in Java for achieving **abstraction** and **multiple inheritance**.
- Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, ***using interface, you can specify what a class must do, but not how it does it.***
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- One class can implement any number of interfaces.
- To implement an interface, a class must implement (override) all the abstract methods of the interface.

Defining an Interface

- An interface is defined much like a class. This is the general form of an interface:

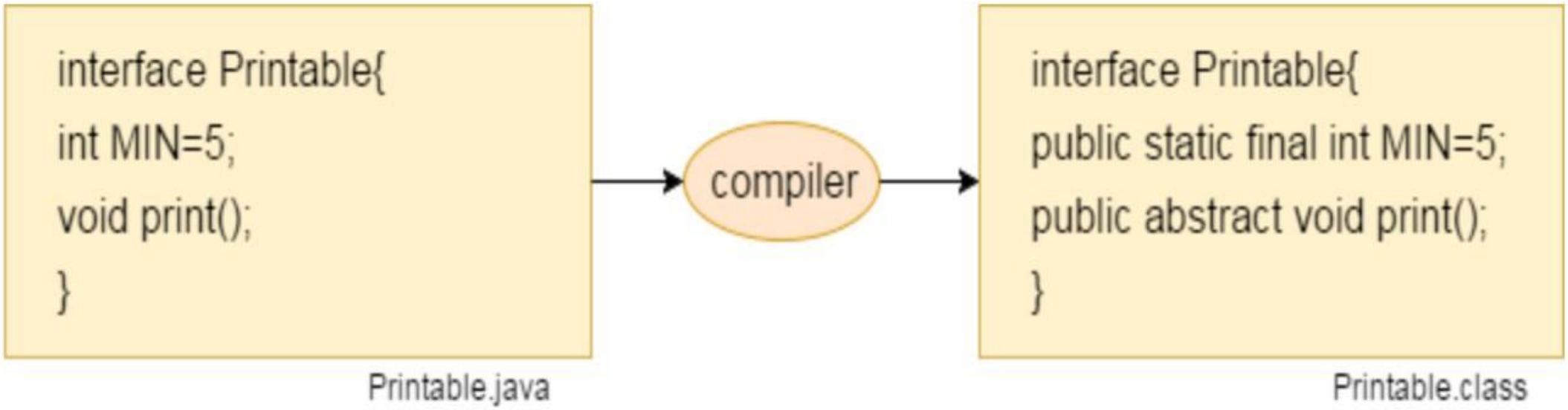
```
access_modifier interface Interface_name
{
 return-type method-name1(parameter-list); //abstract method
 type constant-varname1 = value; //constants
 // ...
 }
```

Example of how to define an interface

```
interface Callback
{
        final int id = 10;
       void callback(int param);
}
```

Imp. Points about Interfaces

- All methods in an interface are implicitly **public abstract** (unless they are static or default methods — Java 8+)
- Variables declared inside the interface are implicitly public, static, and final, mean they can't be changed by the implementing class. They must also be initialized.
- The Java compiler adds public and abstract keywords before the interface method.
- Moreover, it adds public, static and final keywords before data members.



Imp. Points about Interfaces

- **A class can implement multiple interfaces**, overcoming the limitation of single inheritance in Java.
- From Java 8 onwards, interfaces can contain **default methods** (with implementation) and **static methods**
- From Java 9 onwards, they can also have **private methods**
- Interfaces are designed to support dynamic method resolution at run time.
- Interfaces cannot be instantiated just like the abstract class.
- An interface cannot have instance variables, for example. Thus, the defining **difference between an interface and a class is that a class can maintain state information, but an interface cannot.**
- A class implements an interface using the **implements** keyword.
- **Each class that implements an interface must implement all of the methods.**
- Interface abstract methods **can't be marked final.**

Why Use Interfaces?

Purpose	Explanation
Abstraction	Hides implementation details and shows only essential features.
Multiple Inheritance	A class can implement multiple interfaces (unlike classes).
Loose Coupling	Interfaces allow classes to work together flexibly.
Standardization	Interfaces define a contract — all implementing classes must follow it.

Implementing Interfaces

- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]]

{

// class-body

}

- Once an **interface** has been defined, one or more classes can implement that interface.

Example of interface

```
interface Animal {
    void eat();      // abstract method
    void sleep();    // abstract method
}

class Dog implements Animal {
    public void eat() {
        System.out.println("Dog eats bones.");
    }
    public void sleep() {
        System.out.println("Dog sleeps in kennel.");
    }
}

public class InterfaceEx{
    public static void main(String[] args) {
        Animal a = new Dog(); // Interface reference, object of Dog
        a.eat();
        a.sleep();
    }
}
```

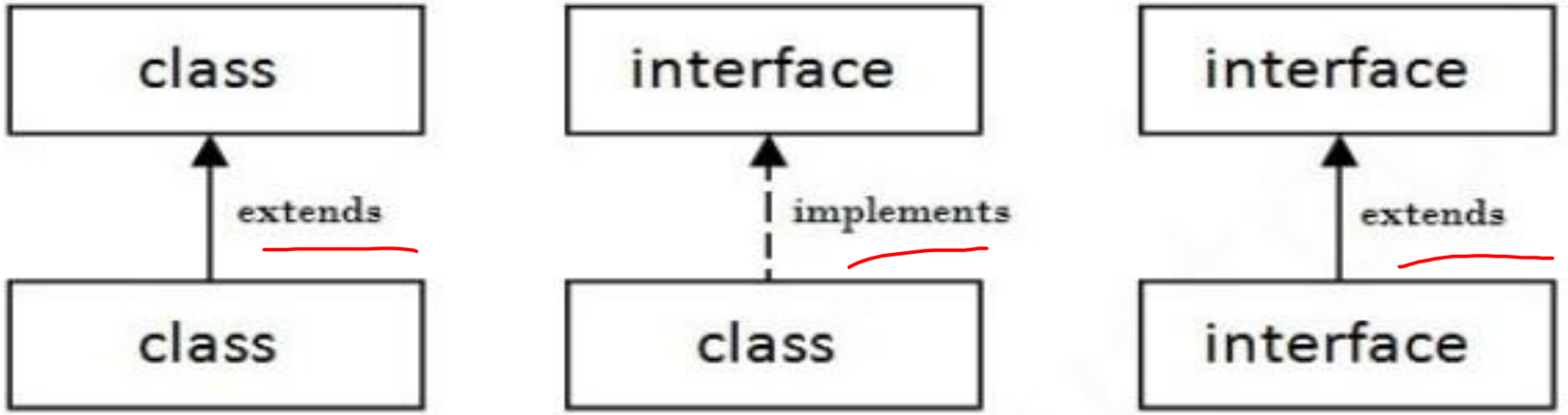
Java Interface Example

```
interface Printable
 {        void print();
 }
class InterfaceEx1 implements Printable
  {
        public void print()
        {
                System.out.println("Hello");
        }

        public static void main(String args[])
        {
                InterfaceEx1 obj = new InterfaceEx1();
                obj.print();
         }
}
```

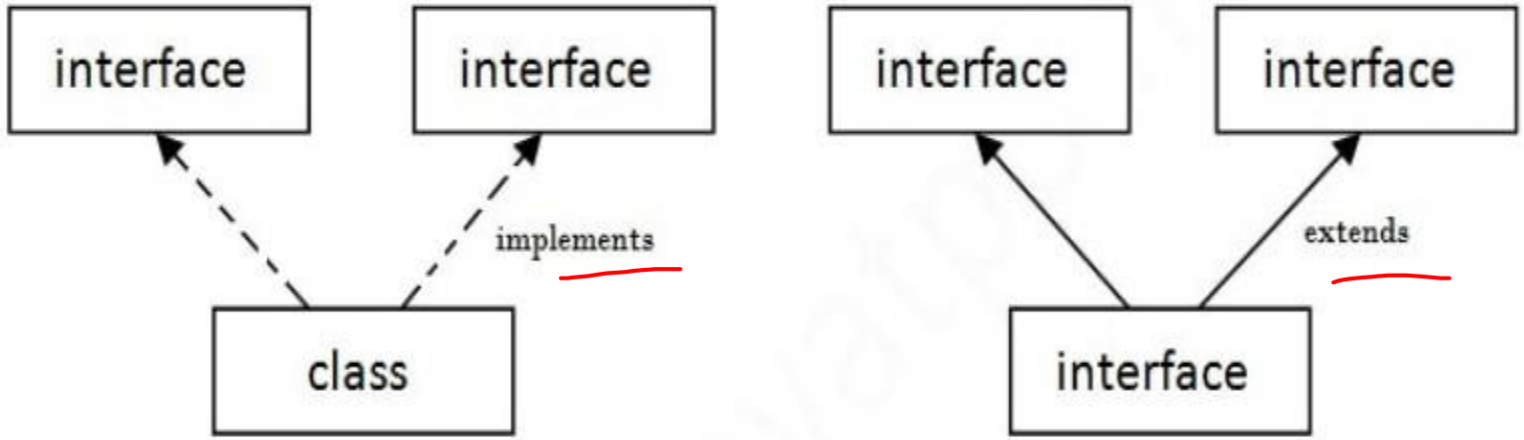
The relationship between classes and interfaces

- A class extends another class, an interface extends another interface, but a **class implements an interface**.
- **Interfaces can't implement another interface or class.**



Multiple inheritance in Java using interface

- Interface can extend one or more interfaces.
- **If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.**
- One or more than one interfaces can be implemented by a class. In addition, a java class can also inherit a single class.



Multiple Inheritance in Java

```
// Multiple inheritance in Java by interface
interface Printable
{ void print();   }
interface Showable
{  void show();  }
class InterfaceMultInhEx implements Printable, Showable
{
        public void print(){System.out.println("Hello");}
        public void show(){System.out.println("Welcome");}
        public static void main(String args[])
        {
                A2 obj = new A2();
                obj.print();
                obj.show();
        }
}
```

Interface inheritance

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

```
interface Printable{ void print();            }
interface Showable extends Printable{  void show();  }
class Inter1 implements Showable
{
        public void print(){System.out.println("Hello");}
        public void show(){System.out.println("Welcome");}

        public static void main(String args[])
        {
                Inter1 obj = new Inter1();
                obj.print();
                obj.show();
        }
}
```

If implementing class does not provide implementation of all abstract methods.

- If the implementation is unable to provide an implementation of all abstract methods, then declare the implementation class with an abstract modifier (ie make it as abstract class), and complete the remaining method implementation in the next created child classes.

Example: If implementing class does not provide implementation of all abstract methods.

```
interface Bank {
    void deposit();
    void withdraw();
    void loan();
    void account();
}
abstract class Dev1 implements Bank {
    public void deposit()
    {    System.out.println("Your deposit Amount :" + 100);    }
}
abstract class Dev2 extends Dev1 {
    public void withdraw()
    {.  System.out.println("Your withdraw Amount :" + 50);     }
}
class Dev3 extends Dev2 {
    public void loan() {}
    public void account() {}
}
```

```
class InterfaceEx {
    public static void main(String[] args)
    {
        Dev3 d = new Dev3();
        d.account();
        d.loan();
        d.deposit();
        d.withdraw();
    } }
```

Runtime Polymorphism through Interface

- *Interface can facilitate run-time polymorphism in java.*
  - Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Ex: Runtime Polymorphism through Interface

```
interface FigureI{
  final static float pi = 3.14F;
  void area();
  void perimeter();
}
class CircleI implements FigureI
{
    double radius;
    Circle(double r)
      {    radius = r;   }
    public void area()
      {      System.out.println("Area is "+(pi*radius*radius));     }
    public  void perimeter()
      {      System.out.println("Perimeter is "+(2*pi*radius));     }
 }
class SquareI implements FigureI
{
      double side;
      Square(double s)
         { side =s; }
      public void area()
          {      System.out.println("Area is "+ side*side);     }
       public  void perimeter()
          {      System.out.println("Perimeter is "+ 4*side);     }
}
```

```
class Interface_Runtime
 {
   public static void main(String args[])
    {
      FigureI c;
      c= new CircleI(5.5);
      c.area(); c.perimeter();

      c = new SquareI(5.5);
      c.area();
      c.perimeter();


    }
 }
```

Interface

- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.
- To define a default method, precede its declaration with **default**. It is not necessary for an implementing class to override default method.

.

```
interface Vehicle {
    void start();
    default void fuelType() {   // default method
        System.out.println("Petrol or Diesel");
    }
    static void serviceInfo() { // static method
        System.out.println("Service every 6 months");
    }
}
class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");
    }
}
public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();
        c.fuelType();
        Vehicle.serviceInfo(); // static method called using interface name
    }
}
```

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	**Interface are implicitely abstract**.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) Members of **abstract class have default access implicitely.**	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Packages

Need of Package

- In the preceding lectures, the name of each example class was taken from the **same name space**. This means that a **unique name** had to be used for each class **to avoid name collisions.**
- After a while, without some way to manage the name space, you could **run out of convenient, descriptive names for individual classes.**

Packages

- Java provides a **mechanism for partitioning the class name space into more manageable chunks.** This mechanism is the **package**.
- The package is both a **naming and a visibility control mechanism**.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.
- *Packages* are containers for classes, interfaces and sub-packages that are used to keep the name space compartmentalized.
- It is mechanism in java to encapsulate a group of classes, interfaces and sub-packages.

Types of Packages

- Package in java can be categorized in two form,
  - **built-in package and**
  - **user-defined package**.
- There are many built-in packages such as java.lang, java.util, java.io, java.net, java.awt, javax.swing

Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g wrapper classes which defines primitive data types, math operations). This package is automatically imported.

2) java.io: Contains classed for supporting input / output operations.

3) java.util: Contains collections framework, utility classes which implement data structures like Linked List, Dictionary and support for Date / Time operations.

4) java.awt: Contain classes for implementing the components for graphical user interfaces (like button , menus etc).

6) **java.net:** Contain classes for supporting networking operations.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

import package.*;

- If you use package.* then all the classes and interfaces of this package will be accessible but not classes and interfaces of subpackages. Hence, you need to import sub packages as well.
- The import keyword is used to make the classes and interface of another package accessible to the current package.
- Note: Sequence of the program must be package then import then class.

Example of importing a built-in package using Fully Qualified name

```
public class InputExample {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.print("Enter your age: ");
        int age = sc.nextInt();
        System.out.println("You are " + age + " years old.");
    }
}
```

Example of importing a built-in package using import package.classname;

```
import java.util.Scanner;   // Import the Scanner class

//or import java.util.*;   // Importing all classes from java.util package

 class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);   // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();   // Read user input
    System.out.println("Username is: " + userName);   // Output user input
  }
}
```

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.
- While the default package is fine for short, sample programs, it is inadequate for real applications

User Defined Package

- Java uses file system directories to store the packages:
- Syntax for creating a package:

```
package mypackage;
public class myclass
        {
        }
```

- A package can contain any number of classes and interfaces.
- Packages are usually defined using a hierarchical naming pattern with levels in the hierarchy separated by periods(.).
  - Directory →sub directory
- The general form of a multileveled package statement is shown here:

  package *pkg1*[.*pkg2*[.*pkg3*]];

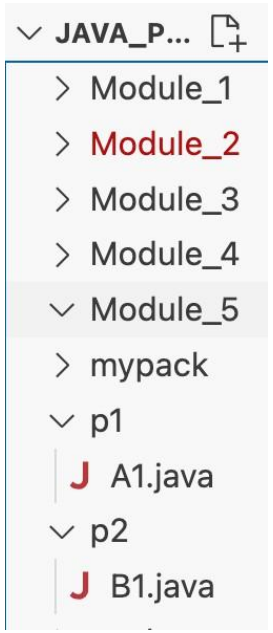
- Eg: package a.b.c;
  - needs to be stored in **a/b/c** in a Mac OS environment

Packages and Member Access

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code. The class is Java's smallest unit of abstraction.

How to implement concept of package

- Create two folders p1, p2
- Create a java file A1.java in folder p1
- Create a java file B1.java in folder p2



A1.java in folder p1

p1 > A1.java > ...

```
1  package p1;
2
3  public class A1 {
4      public void msg(){
5          System.out.println(x:"hello");
6      }
7  }
8
```

B1.java in folder p2

You can write either:

- import p1.*; or
- import p1.A1; or
- p1.A1 obj= new p1.A1();

p2 > B1.java > ...

```
1   package p2;
2   import p1.A1;
3   public class B1 {
        Run | Debug
4       public static void main(String[] args) {
5           A1 obj=new A1();
6           obj.msg();
7       }
8
9   }
```

How to Compile and execute Java files in packages

- Compile A1.java
- Compile B1.java and execute it

```
dr.ritujain@DrRitus-Air java_prg_vs % pwd
/Users/dr.ritujain/java/java_prg_vs
dr.ritujain@DrRitus-Air java_prg_vs % javac p1/A1.java
dr.ritujain@DrRitus-Air java_prg_vs % javac p2/B1.java
dr.ritujain@DrRitus-Air java_prg_vs % java p2.B1
hello
```

Role of Access Modifiers

A1.java in package p1

p1 > A1.java > ...

```
1   package p1;
2
3   public class A1 {
4       int i=10;
5       public int j=10;
6       protected int k=15;
7       public void msg(){
8           System.out.println(x:"hello");
9       }
10  }
11
```

Role of Access Modifiers: B1.java in package p1

p2 > B1.java > C1 > c1Meth()

```
1  package p2;
2  import p1.*;
3  public class B1 {
       Run | Debug
4      public static void main(String[] args) {
5          A1 obj=new A1();
6          obj.msg();
7          //System.out.println(obj.i); //not visible outside pkg
8          System.out.println("value of j is "+obj.j);
9
10         C1 obj2=new C1();
11         obj2.c1Meth();
12     }
13 }
14 class C1 extends A1{
15  public void c1Meth(){
16     System.out.println("value of k is "+k);
17  }
18 }
```

Compiling and executing Java files in packages

```
dr.ritujain@DrRitus-Air java_prg_vs % javac p1/A1.java
dr.ritujain@DrRitus-Air java_prg_vs % javac p2/B1.java
dr.ritujain@DrRitus-Air java_prg_vs % java p2.B1
hello
value of j is 10
value of k is 15
dr.ritujain@DrRitus-Air java_prg_vs %
```

```
//Example of package that import the packagename.*
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Exception Handling

Error

- Error may be broadly classified into two categories:
  - **Compile time error** (eg: syntactical errors, misspelt keywords, missing semicolons, missing braces, etc)
  - **Run time error** (eg: division by zero, out of bound accessing an Array, invalid data input, negative size of an array, etc)

Example of Runtime Error: Uncaught Exception

```
1.  class Exc0
2.  {   public static void main(String args[])
3.      {       int d = 0;
4.              int a = 42 / d;
5.      }
6.  }
```

O/P:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Second Example of Runtime Error: Uncaught Exception

```
1.  class Exc1 {
2.          static void subroutine() {
3.                  int d = 0;
4.                  int a = 10 / d;
5.          }
6.          public static void main(String args[]) {
7.                  Exc1.subroutine();
8.          }
9.  }
```

O/P:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

Types of Exception

- All exception types are subclasses of the built in class Throwable.
- **Error:** exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Eg: Stack overflow

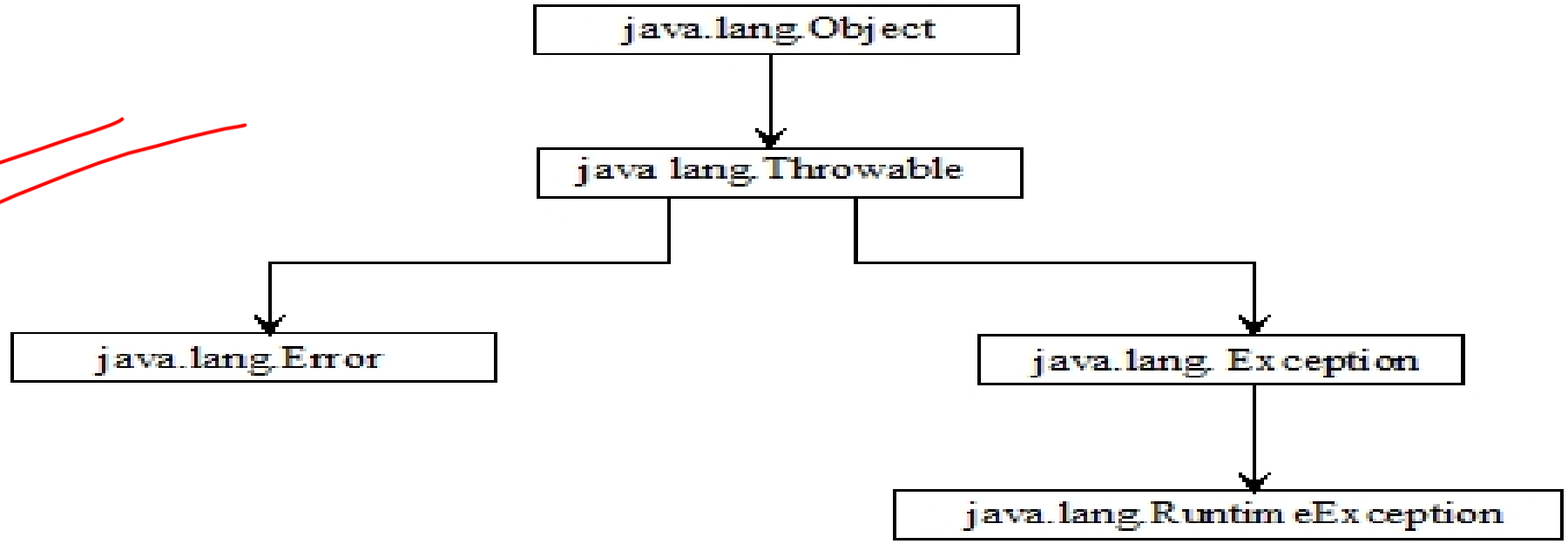
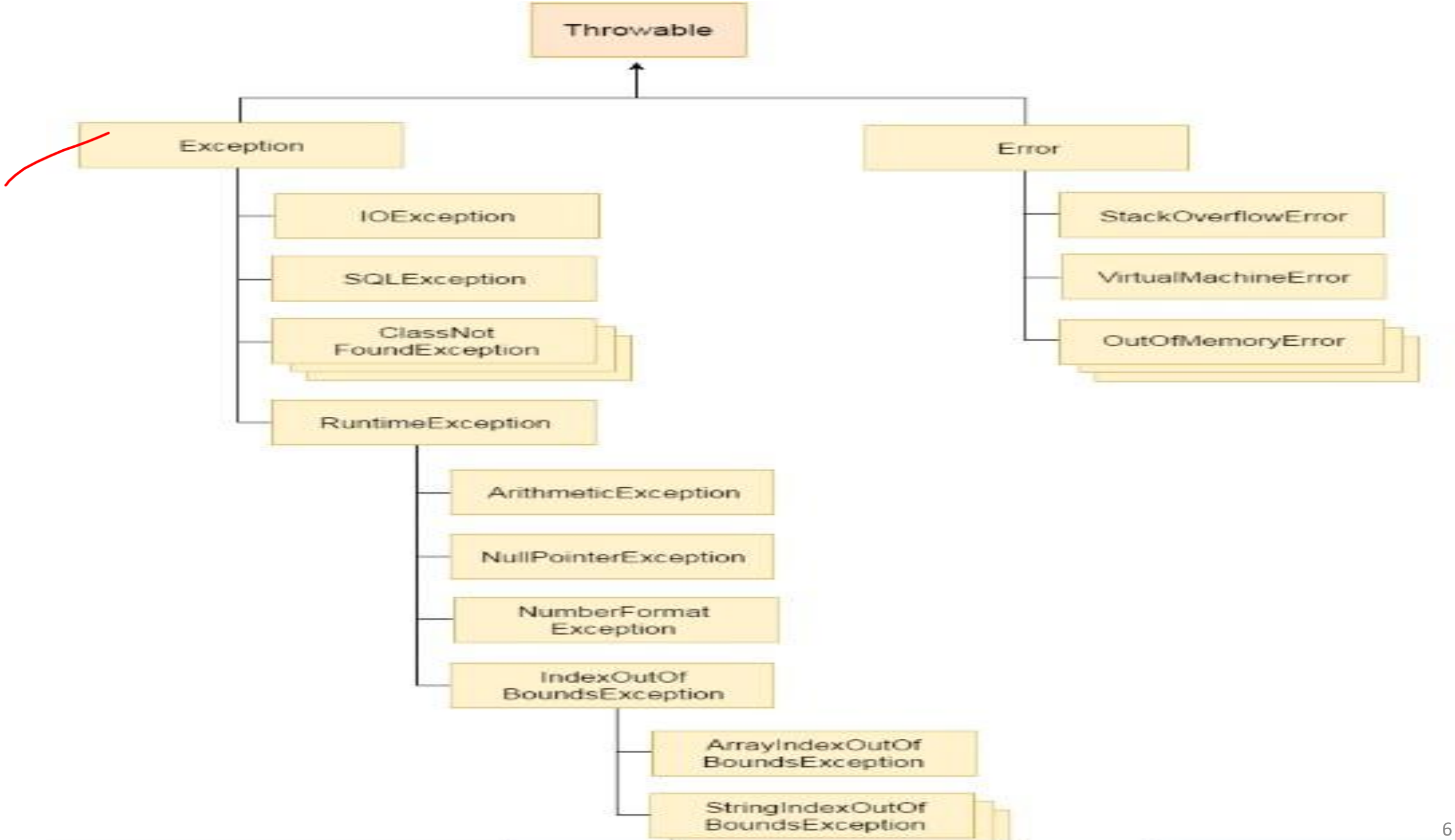


Figure 5.1 :Exception Class hierarchy in Java

Types of Exception

- **Checked Exception:** Exceptions that must be handled or declared by the application code where it is thrown.
  - Inherits Exception class but not Runtime Exception
- **Unchecked Exception:** any exception that does not need to be declared or handled by application code where it is thrown
  - Includes any class that inherits Runtime Exception

	Checked Exceptions	Unchecked Exceptions
Definition	Checked at compile-time	Not checked at compile-time
Exception Types	IOException, ClassNotFoundException, etc.	NullPointerException, ArrayIndexOutOfBoundsException, etc.
Handling Requirement	Must be caught or declared using throws keyword	Optional to catch or declare using throws keyword
Exception Handling	Handled using try-catch blocks or throws keyword	Not required to handle explicitly
Flow Control	Program flow is interrupted and transferred to catch block	Program execution is halted with an error message
Examples	File operations, database operations, etc.	Logical errors, invalid arguments, etc.


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

ExceptionHandling

- It is observed that Java's default run time handler displays the detail of an exception and execution suspended as soon as an error encountered, but it is often more desirable to handle the exception yourself and continue running.
- The **try** key word can be used to specify a block of code that should be guarded against all exceptions. The code that you want to monitor against runtime errors is enclosed in try block.
- try block: Always enclose the statements with curly braces.

Catch block

- Immediately following a try block, you should include a **catch** clause which specifies the exception type that you wish to catch.
- Scope of catch clause is restricted to the statements enclosed in immediately preceding try statement.
- Goal of catch clause:
  - To resolve the exceptional condition and than continue as if the error has never happened.

Syntax

```
try
{
        // block of code to monitor for errors
} catch (ExceptionType1 exOb)
  {
        // exception handler for ExceptionType1
  }
 catch (ExceptionType2 exOb)
  {
        // exception handler for ExceptionType2
  }
```

Basic Exception Handling Program

```
class Exc2 {
    public static void main(String args[]) { int d, a;
        try { // monitor a block of code.
                    d = 0;
                    a = 42 / d;
                    System.out.println("This is an example of exception handling.");
            } catch (ArithmeticException e){ // catch divide-by-zero error
                    System.out.println("Division by zero.");
            }
            System.out.println("After catch statement.");
    }
} O/P:
Division by zero.
After catch statement.
```

Command line argument in Java

```
public class SumCommandLine {
    public static void main(String[] args) {
        // Check if two arguments are provided
        if (args.length < 2) {
            System.out.println("Please provide two numbers as command line arguments.");
            return;
        }

        // Convert string arguments to integers
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);

        int sum = num1 + num2;
        System.out.println("Sum = " + sum);
    }
}
```

Command line argument in Java

```
public class Example {
    public static void main(String[] args) {
        // args is an array of Strings that holds command line arguments
        System.out.println("Number of arguments: " + args.length);

        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}
```

Multiple catch clause

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.

Multiple catch clause

```
// Demonstrate multiple catch statements.
class MultiCatch
{
  public static void main(String args[]) {
   try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
      } catch(ArithmeticException e)
        {
            System.out.println("Divide by 0: " + e);
         } catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index oob: " + e);
        }
     System.out.println("After try/catch blocks.");
  }
}
```

Output of MultiCatch

C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.

Exception subclass must come before any of their super class

```
class SuperSubCatch {
public static void main(String args[])
  { try {
        int a = 0;
        int b = 42 / a;
     } catch(Exception e)
        {
        }
        catch(ArithmeticException e) { // ERROR - unreachable
           System.out.println("This is never reached.");
        }
    }
}
```

Syntax

```
try
{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
finally {
        // block of code to be executed after try block ends
}
```

finally

- When exceptions are thrown, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.
- The **finally** keyword is designed to address this contingency.

finally

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- **The finally block will execute whether or not an exception is thrown.**
- **If an exception is thrown, the finally block will execute even if no**
catch statement matches the exception.
- **Any time a method is about to return to the caller from inside a try/ catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.**

finally

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
// Demonstrate finally.
class FinallyDemo {
    static void procA()
    {   try {   System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {         System.out.println("procA's finally");              }
    }
    static void procB()
    {       try { System.out.println("inside procB");   return; }
            finally {  System.out.println("procB's finally"); }
    }
  static void procC()
    {
        try {           System.out.println("inside procC");         }
        finally {           System.out.println("procC's finally");              }
    }
    public static void main(String args[])
    {
        try {   procA();      }
        catch (Exception e) { System.out.println("Exception caught"); }
        procB();
        procC();
    }
}
```

Output of previous program

Output of FinallyDemo:

inside procA

procA's finally

Exception caught

inside procB

procB's finally

inside procC

procC's finally

throw statement

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways you can obtain a **Throwable** object:
  - using a parameter in a **catch** clause, or
  - creating one with the **new** operator.

throw statement

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Eg. Of throw statement

```
class ThrowDemo
{       static void demoproc()
            {
                try {
                        throw new NullPointerException("demo");
                    } catch(NullPointerException e)
                        { System.out.println("Caught inside demoproc.");
                          throw e; // rethrow the exception
                    }
            }
          public static void main(String args[])
           {     try {
                        demoproc();
                    } catch(NullPointerException e)
                      {
                        System.out.println("Recaught: " + e);
                      }
           }
}
```

O/P: Caught inside demoproc. Recaught: java.lang.NullPointerException: demo

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or Runtime Exception.
- **Syntax:**

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

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

Checked Exceptions

- These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the *throws* keyword.
- Ex: ClassNotFoundException, SocketException, SQLException, IOException, FileNotFoundException

Example: Checked Exceptions

```
//Incorrect program… won't compile
class CheckedExc_Eg {
public static void main(String[] args)
{
        FileReader file = new FileReader("C:\\test\\a.txt");
       BufferedReader fileInput = new BufferedReader(file);
      // Printing first 3 lines of file "C:\test\a.txt"
      for (int counter = 0; counter < 3; counter++)
                System.out.println(fileInput.readLine());
              fileInput.close();
   }
}
```

- The function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

Example: Checked Exceptions

```
//Correct Version
import java.io.*;
class CheckedExc_Eg {
public static void main(String[] args)  throws IOException
    {
          FileReader file = new FileReader("C:\\test\\a.txt");
         BufferedReader fileInput = new BufferedReader(file);
        // Printing first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
                    System.out.println(fileInput.readLine());
         fileInput.close();
    }
}
```

Unchecked Exceptions

- These are the exceptions that are not checked at compile time.
- Exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.
- Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile because *ArithmeticException* is an unchecked exception.

```
class UncheckedExc_Ex {

    public static void main(String args[])
    {
        // Here we are dividing by 0
              int x = 0;
        int y = 10;
        int z = y / x;
    }
}
```

Unchecked Exception

- In short unchecked exceptions are runtime exceptions that are not required to be caught or declared in a throws clause.
- These exceptions are usually caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.
- Unchecked exceptions include all subclasses of the RuntimeException class, as well as the Error class and its subclasses.

Examples of Exceptions

1. **ArithmeticException:** thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **NullPointerException:** is raised when referring to the members of a null object.
4. **NumberFormatException:** is raised when a method could not convert a string into a numeric format.
5. **RuntimeException:** This represents an exception that occurs during runtime.
6. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size

User-defined exception

- In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception.

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {            super(str);        }
}
 public class TestCustomException1
{
    static void validate (int age) throws InvalidAgeException{
     if(age < 18){                    throw new InvalidAgeException("age is not valid to vote");         }
     else {           System.out.println("welcome to vote");           }
    }
   public static void main(String args[])
   {  try
      {          validate(13);          }
      catch (InvalidAgeException ex)
      {             System.out.println("Caught the exception");
                    System.out.println("Exception occured: " + ex);
      }
      System.out.println("rest of the code...");
   }
}
```

```
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Thanks!!!