MODULE III

Syllabus of module 3-Part1

More features of Java: Packages and Interfaces - Defining Package, CLASSPATH, Access Protection, Importing Packages, Interfaces.

Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause, Multiple catch Clauses, Nested try Statements, throw, throws and finally.

Interfaces

- Interface is basically a kind of class.
- The difference is that interface can contain only **abstract methods** and **final fields**.
- Using the keyword **interface**, a class can be fully abstracted from its implementation. i.e. using interface one can specify what the class must do without specifying how it does it.
- An **alternative** approach to support **Multiple Inheritance**
- Syntax :

```
interface Interfacename
{
        variable declaration;
        methods declaration;
}
```

- Variables are declared as follows:

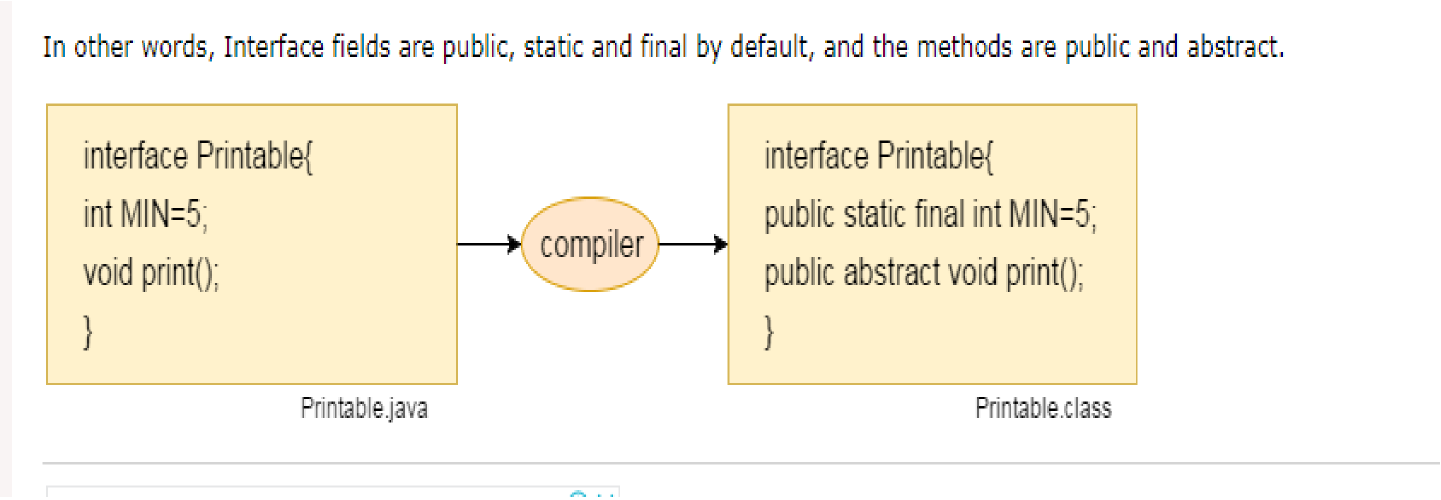
  static final type variablename=value;

- Methods are declared as follows:

  returntype methodname(parameterlist);

Interfaces

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members. We don't have to add it explicitly.



In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

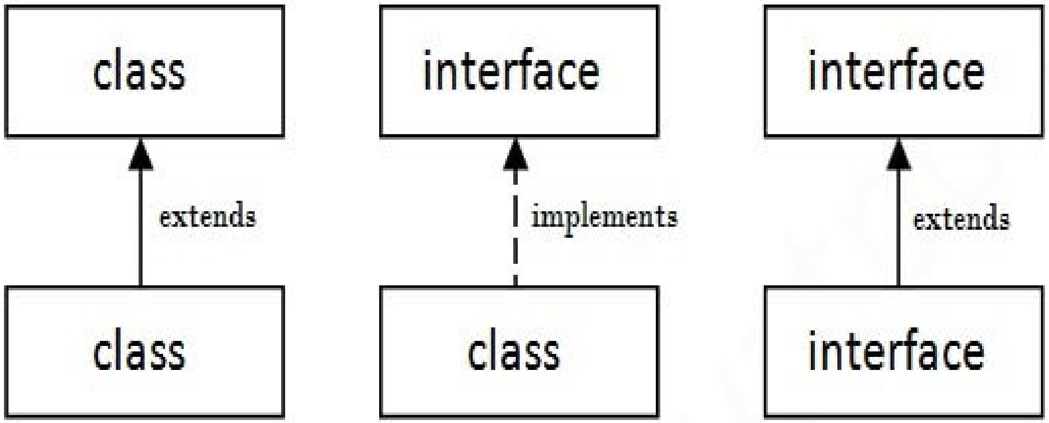
- The code for the method(**void print()** in the above example) is not included in the interface since it is abstract in nature.
- The class that implements this interface must define the code for the method

Interfaces

Class	Interface
The members of a class can be constant or variables	The **members** of an interface are always declared as **constant.** It is declared **public static final**.
The class definition can contain the code for each of its methods. That is, the methods can be abstract or non-abstract	The **methods** in an interface are **abstract** in nature. There is no code associated with them. It is later defined by the class that implements the interface.
It can be instantiated by declaring objects	It **cannot be used to declare objects**. It can only be inherited by a class
It can use various access specifiers like public, private or protected.	It can **only use** the **public access specifier.**

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.

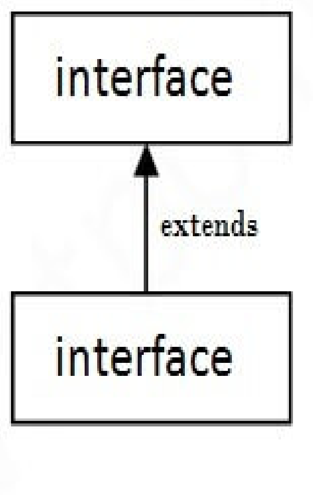


Interfaces

- An interface can also be extended. That is an interface can be subinterfaced from other interfaces.
- The new subinterface will inherit all the members of the super interface.
- General form:

```
interface name2 extends name1
{
        Body of name2
}
```

```
interface ItemConstants
{    int code=1001;
     string name="Fan";
}
interface Item extends ItemConstants
{    void display();
}
```



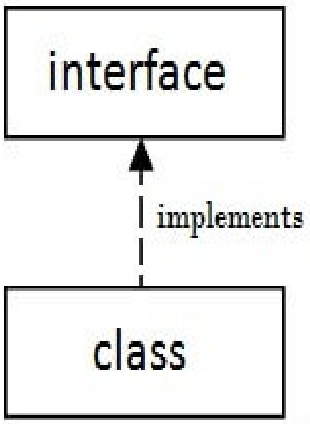
Interfaces

- We can combine several interface together into a single interface.

```
interface ItemConstants
{
        int code=1001;
        string name=”Fan”;
}
interface ItemMethods
{
        void display();
}
interface Item extends ItemConstants, ItemMethods
{
        -------------------
}
```

Interfaces

- While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in superinterfaces.
- **Subinterfaces** are still **interfaces**, **not classes**.
- It is the responsibility of any **class** that **implements** the derived interface to define all the methods.



Interfaces

- It is necessary to create a class that inherits the given interface.
- General form:

```
Class classname implements interfacename
{
        Body of classname
}
```

- More General form:

```
Class classname extends superclass implements interface1,
                                        interface2, … . . . .
{
        Body of classname
}
```

```
interface Area
{
        final static float pi=3.14F;
        float compute(float x, float y);
}
class Rectangle implements Area
{
    public float compute(float x, float y)
   {
            return x*y;
    }
}
class Circle implements Area
{
      public float compute(float x, float y)
      {
            return pi*x*x;
      }
}
```

```
class InterfaceTest
{
     public static void main(String a[])
     {
          Rectangle rect=new Rectangle();
          Circle cir=new Circle();
          Area area;
          area = rect;
          System.out.println("Area of the
rectangle : "+area.compute(10,20));

          area = cir;
          System.out.println("Area of the
circle : "+area.compute(10,0));
               }
}
```

Output:

```
               Area of the rectangle : 200
               Area of the circle : 314
```

Example: To show that Interfaces support multiple inheritance

```
interface AnimalEat
{    void eat(); }
 interface AnimalTravel
{    void travel(); }
class Animal implements AnimalEat, AnimalTravel //This is OK
{
public void eat() { System.out.println("Animal is eating");}
public void travel() {System.out.println("Animal is travelling"); } }
public class Demo {
 public static void main(String args[]) {
 Animal a = new Animal();
a.eat();                         o/p   Animal is eating
a.travel();    } }                     Animal is travelling
//Note: This will not be possible if AnimalEat and AnimalTravel are classes
```

```
class Test
{     float part1, part2;
      void getMarks(float m1, float m2)
      {
                 part1=m1;   part2=m2;
       }
      void putMarks()
      {     System.out.println("Part1="+part1);
            System.out.println("Part2="+part2);
       }
}
interface Sports
{
                 float sportWt=6.0F;
                 void putWt();
}
```

```
class Results extends Test implements Sports
{      float total;
       public void putWt()
       {  System.out.println("Sports Wt="+sportWt); }
        void display()
       {          total=part1+part2+sportWt;
                  putMarks();
                  putWt();
                  System.out.println("Total Score="+total);
       }
}
class Hybrid
{     public static void main(String a[])
      {          Results student1=new Results();
                 student1.getMarks(27.5F,33.0F);
                 student1.display();
       }
}
```

```
Output:   Part1= 27.5
          Part2 = 33.0
          Sports Wt=6.0
          Total Score = 66.5
```

Nested Interface in Java

- We can declare interfaces as member of a class or another interface. Such an interface is called as **member interface** or **nested interface**.

- **Interface in a class**
  Interfaces (or classes) can have only public and default access specifiers when declared outside any other class . This interface declared in a class can either be default, public, protected not private.

Example 1

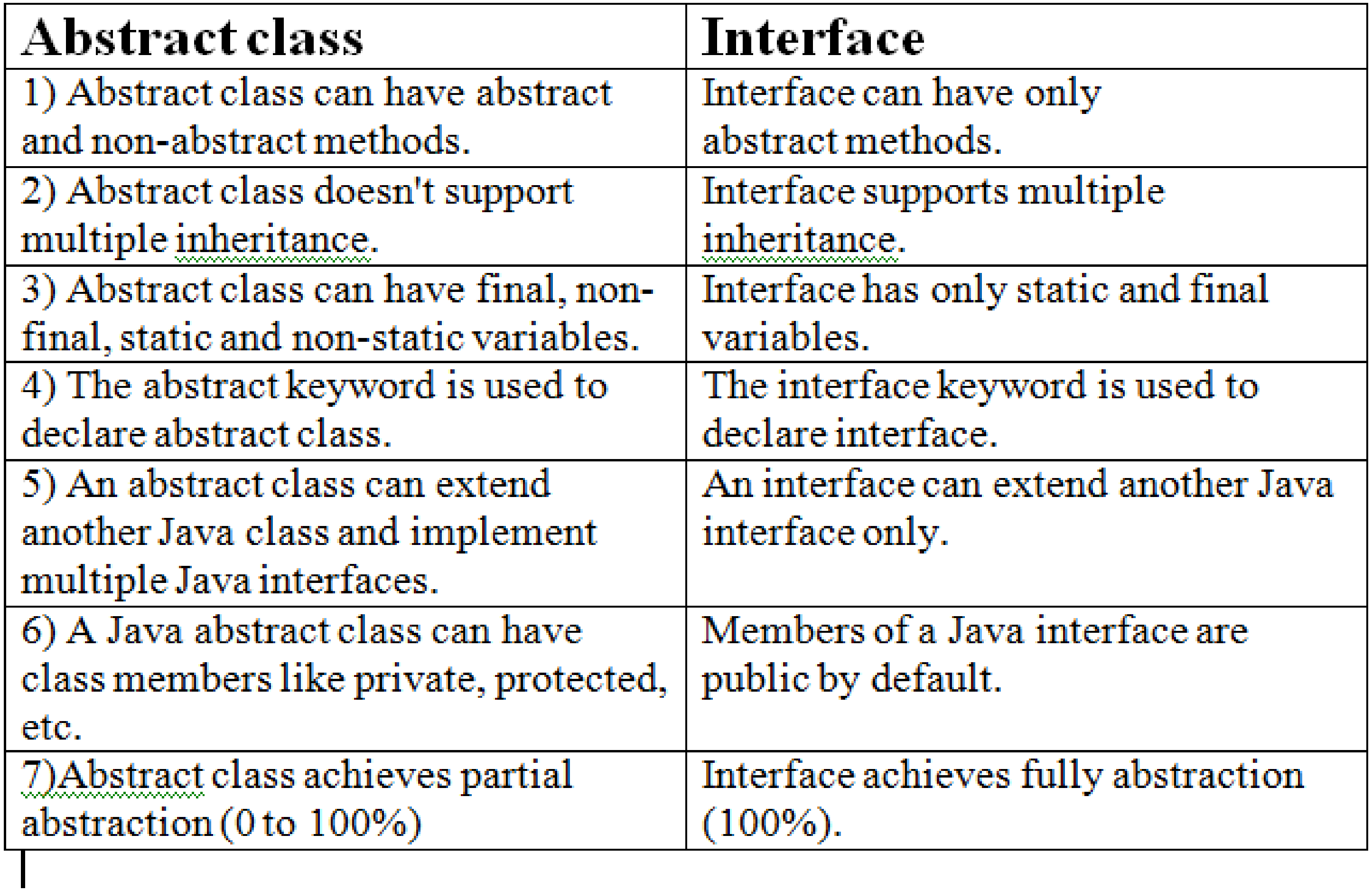
```
class NestedInterface {
interface myInterface {
void demo();
                       }
class Inner implements myInterface {
public void demo() {
System.out.println("Welcome to Nested Interface"); }
}
public static void main(String args[]) {
Inner obj=new NestedInterface().new Inner();
obj.demo();
}             // o/p Welcome to  Nested Interface
  }
```

You can also access the nested interface using the class name as –

Example 2

```
class Test {
 interface myInterface {      void demo();    }
}
 class Sample implements Test.myInterface {
public void demo() {
 System.out.println(“Welcome to Nested Interface");
 }
 public static void main(String args[]) {
 Sample obj = new Sample();
obj.demo();             // o/p  Welcome to Nested Interface
}
     }
```

Abstract class vs. Interfaces



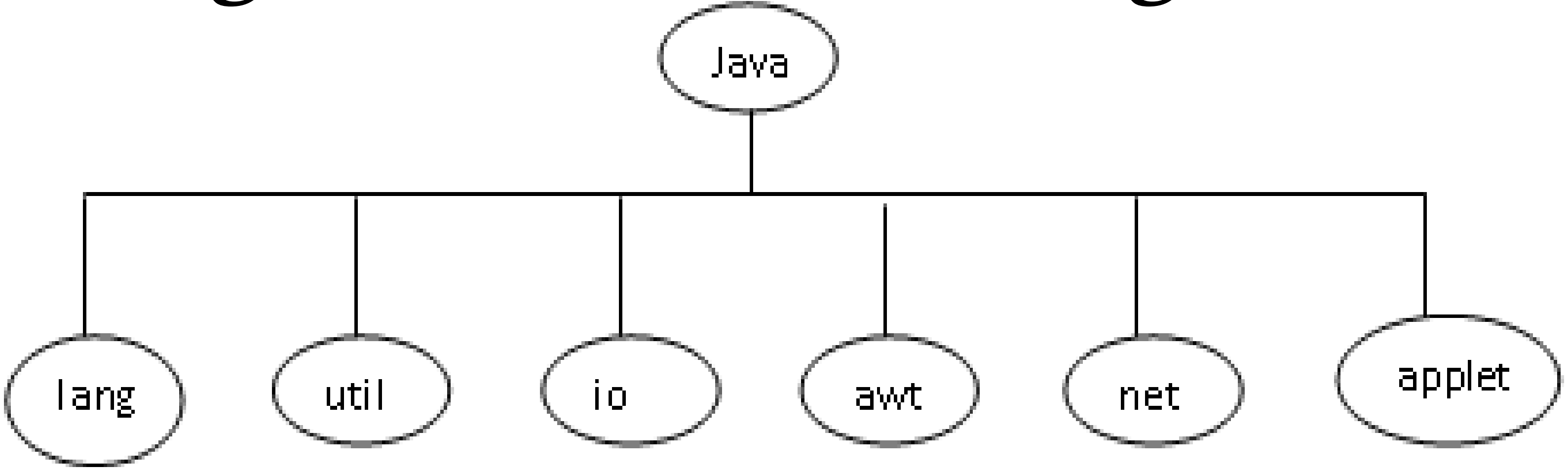
Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
5) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
6) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
7)Abstract class achieves partial abstraction (0 to 100%)	Interface achieves fully abstraction (100%).

Note: Static functions are also allowed in interfaces

Packages

- A package is a group of classes that are defined by a name. That is, if you want to declare many classes within one element, then you can declare it within a package.
- Packages group variety of classes and/or interfaces together.
- Packages allow us to use classes from other programs without physically copying them into the program under development.
-
- Packages act as containers for classes.
- Java packages are classified into two types:
  - Java API Package
  - User defined package

Packages : Java API Packages



Package name	Contents
java.lang	They are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions
java.util	Language utility classes such as vectors, hash tables, random numbers, date etc.
java.io	They provide facilities for the input and output of data.
java.awt	Set of classes for implementing GUI. They provide classes for windows, button, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers
java.applet	Classes for creating and implementing applets.

Packages : Java API Packages

- Two ways of accessing the classes stored in a package.
  - Use the fully qualified class name of the class that we want to use
    - Example: java.awt.Color
  - Import the package
    - Syntax 1: import packagename.classname;
    - Syntax 2: import packagename.*;
    - Import statements must appear at the top of the file, before any class declarations.
    - Example : import java.awt.Color;
    - Example: import java.awt.*;
    - It brings all classes of java.awt package.

Packages : User Defined Packages

- Steps to create user defined packeges:
  - Declare the package at the beginning of a file using the form :

    package packagename;
  - Define the class that is to be put in the package and declare it public
  - Create a subdirectory under the directory where the main source files are stored. Subdirectory name must match the package name exactly.
  - Store the listing as the classname.java file in the subdirectory created
  - Compile the file. This create .class file in the subdirectory

Packages : User Defined Packages

- Example:

```
package firstPackage;
public class FirstClass
{
        (body of class)
}
```

- This file should be saved as a file called **FirstClass.java** , and located in a directory named **firstPackege** under the current working directory.
- A Java package file can have more than one class definitions. In such cases, only one of the classes may be declare public and that class name with .java extension is the source file name.
- If we omit the **package** statement, the class names are put into the default package, which has no name.

Packages : User Defined Packages

- Java supports the concepts of package hierarchy.
- General form : package *pkg1*[.*pkg2*[.*pkg3*]];
- Example : package firstPackage.secondPackage;
  - Store this package in a subdirctory names firstPackage/secondPackage

- Syntax for Accessing a User Defined Package:

```
import package1[.package2][.package3].classname;
                    or
import packagename.*;
```

ClassA.java

```
package package1;
public class ClassA
{
        public void displayA()
        {
                System.out.println("Class A");
        }
}
```

ClassB.java

```
package package2;
public class ClassB
{
        Protected int m=10;
         public void displayB()
         {
                System.out.println("Class B");
                System.out.println("m="+m);
         }
}
```

PackageTest2.java

```
import package1.ClassA;
import package2.*;
class packageTest2
{
        public static void main(String args[])
        {
                ClassA objectA=new ClassA();
                ClassB objectB=new ClassB();
                objectA.displayA();
                objectB.displayB();
        }
}
```

Output:

```
        Class A
        Class B
        m=10
```

Packages : User Defined Packages

- When we import multiple packages it is likely that two or more packages contain classes with identical names.
- Example:

```
package pack1;
public class Student
{............................}
```

```
package pack2;
public class Student
{............................}
```

- When we import and use these packages like:

```
import pack1.*;
import pack2.*;
Student Student1;// error
```

```
import pack1.*;
import pack2.*;
pack1.Student Student1;
pack2.Student Student2;
```

Correct

Packages : User Defined Packages

- It is possible to subclass a class that has been imported from another package.

ClassB.java

```
package package2;
public class ClassB
{
        protected int m=10;
        public void displayB()
        {
           System.out.println(“Class B”);
           System.out.println(“m=”+m);
        }
}
```

Output:

```
Class B
m=10
Class C
m=10
n=20
```

PackageTest3.java

```
import package2.ClassB;
class ClassC extends ClassB
{               int n=20;
                void display()
                {
                System.out.println(“Class C”);
                System.out.println(“m=”+m);
                System.out.println(“n=”+n);
                }
}
class PackageTest3
{        public static void main(String args[])
        {
                ClassC objectC=new ClassC();
                objectC.displayB();
                objectC.displayC();
        }
}
```

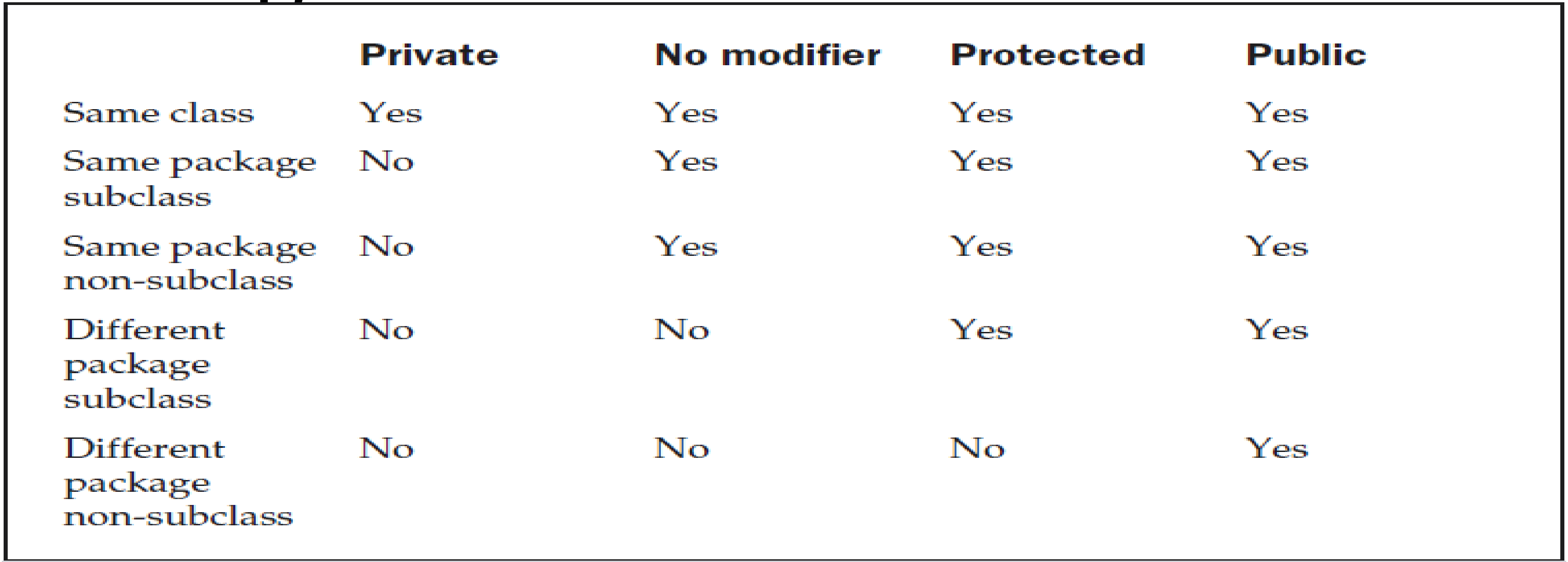
Packages : User Defined Packages

- If we want to create a package with multiple public classes in it, follow the steps:
  - Decide the name of the package
  - Create a subdirectory with this name under the directory where the main source files are stored.
  - Create classes that are to be placed in the package in separate source files and declare the package statement

    ```
    package packagename;
    ```

    at the top of each source file
  - Switch to the subdirectory created earlier and compile each source file. When compiled, the package would contain .class files of the all source files.

Packages : Access Protection



	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1. *Class Member Access*

- **Private –** only within class
- **Default –** it is visible to subclasses as well as to other classes in the *same package*.
- **Protected –** If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.
- **Public –** can be accessed from anywhere

```
package p1;
public class Protection
{       int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
        public Protection()
        {       System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

```
package p1;
class Derived extends Protection
{
        Derived()
        {
                System.out.println("derived
                constructor");
                System.out.println("n = " + n);
                // System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

```
package p1;
class SamePackage
{       SamePackage()
        {   Protection p = new Protection();
            System.out.println("same package constructor");
            System.out.println("n = " + p.n);
            // System.out.println("n_pri = " + p.n_pri);
            System.out.println("n_pro = " + p.n_pro);
             System.out.println("n_pub = " + p.n_pub);
        }
}
```

Protection.java

Derived.java

SamePackage.java

```
package p2;
class Protection2 extends p1.Protection
{
        Protection2()
        {       System.out.println("derived other package constructor");
                // System.out.println("n = " + n);
                // System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

Protection2.java

```
package p2;
class OtherPackage
{
        OtherPackage()
        {
                p1.Protection p = new p1.Protection();
                System.out.println("other package constructor");
                // System.out.println("n = " + p.n);
                // System.out.println("n_pri = " + p.n_pri);
                // System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

OtherPackage.java

Packages : User Defined Packages

- When we import a package into a file, all public classes are imported.
- To hide few classes from external access, declare those as non public.
- Example:

```
package p1;
public class X
{
      //body of X
}
class Y
{
      //body of Y
}
```

- The class Y can be seen and used only by the other classes in the same package.

```
import p1.*;
X objectX;              //correct
Y objectY;              //error. Y is not available
```

Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- Type of Errors :
  - Compile time errors
    - These errors are detected and displayed by the Java compiler
    - Most of the compile time errors are due to typing mismatch
    - Ex: Missing ;, Mismatch of brackets, Misspelling of keywords and identifiers, Missing double quotes in string etc.
  - Run time errors
    - Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.
    - Ex: dividing an integer by zero, Accessing an element that is out of the bounds of an array, trying to save an element into an array of incompactible type etc.
    - When such errors are encountered, Java generates an error message and aborts the program.

Exception Handling : Exception Types

- **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses :
  - **Exception**
    - This class is used for exceptional conditions that user programs should catch.
    - Used to create our own custom exception types.
    - There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
  - **Error**
    - These exceptions are not expected to be caught under normal circumstances by our program.
    - These are used by the Java run-time system to indicate errors having to do with the run-time environment
    - Ex: Stack overflow

Exception Handling

- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- Exceptions can be generated by
  - The **Java run-time system** : due to the violation of the rules of Java language or constraints of the Java execution environment
  - **Manually generated** by our code

Exception Handling

- Uncaught Exceptions are handled by Java runtime system.

```
class Exc0
{
        public static void main(String args[])
        {
                int d = 0;
                int a = 42 / d;
        }
}
```

```
Output:
        java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:6)
```

```
class Exc1
{
        static void subroutine()
        {
                int d = 0;
                int a = 10 / d;
        }
        public static void main(String args[])
        {
                Exc1.subroutine();
        }
}
```

```
Output:
        java.lang.ArithmeticException: / by zero
        at Exc1.subroutine(Exc1.java:6)
        at Exc1.main(Exc1.java:10)
```

Exception Handling

- Java exception handling is managed via five keywords:
  - **try** : Program statements that we want to monitor for exceptions are contained within a **try** block.
  - **throw** : To manually throw an exception.
  - **throws** : Any exception that is thrown out of a method must be specified by a **throws** clause.
  - **catch** : to catch exceptions and handle it in some rational manner.
  - **finally** : Any code that absolutely must be executed before a method returns is put in a **finally** block.

Exception Handling

- General Form :

```
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 before try block ends
}
```

ExceptionType is the type of exception that has occurred.

Exception Handling : try and catch

```
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 will not be printed.");
                }
                catch (ArithmeticException e)
                {
                        System.out.println("Division by zero.");
                }
                System.out.println("After catch statement.");
        }
}
```

```
Output:
        Division by zero.
        After catch statement.
```

Exception Handling : try and catch

- Enclose the code that we want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that we wish to catch.
- Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Execution never returns to the **try** block from a **catch**.
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism

Exception Handling : Multiple catch clauses

```
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.");
        }
}
```

- More than one exception could be raised by a single piece of code.
- To handle this type of situation, we 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.

Exception Handling : Multiple catch clauses

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

- The exception subclasses must come before any of their superclasses.

Exception Handling : nested try

- A **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

```
class NestTry
{      public static void main(String args[])
       {   try
           {       int a = args.length;
                   int b = 42 / a;      // no command-line args,  generate a divide-by-zero exception.
                  System.out.println("a = " + a);
                  try     // nested try block
                   {
                       if(a==1)  a = a/(a-a);    // 1 cmd-line arg, generate a divide-by-zero exception
                       if(a==2)
                       {       int c[] = { 1 };
                               c [42] = 99;     // generate an out-of-bounds exception
                        }
             }
             catch(ArrayIndexOutOfBoundsException e)
             {       System.out.println("Array index out-of-bounds: " + e);    }
           }
           catch(ArithmeticException e)
           {       System.out.println("Divide by 0: " + e);     }
         }
}
```

Exception Handling : Implicitly nested try

```
class MethNestTry
{           static void nesttry(int a)
          {        try     // implicitly nested try block
                   {         if(a==1)       a = a/(a-a); // division by zero
                             if(a==2)
                             {         int c[] = { 1 };
                                       c[42] = 99; // generate an out-of-bounds exception
                              }
                    }
                   catch(ArrayIndexOutOfBoundsException e)
                   {          System.out.println("Array index out-of-bounds: " + e);        }
                    }
                    public static void main(String args[])
                   {         try
                            {         int a = args.length;
                                      int b = 42 / a;
                                      System.out.println("a = " + a);
                                      nesttry(a);
                             }
                            catch(ArithmeticException e)
                            {         System.out.println("Divide by 0: " + e);       }
                    }
}
```

Exception Handling : throw

- To throw an exception explicitly
- Syntax : **throw *ThrowableInstance*;**

  ThrowableInstance is an object of type **Throwable** or a subclass of **Throwable**.
- The flow of execution stops immediately after the **throw** statement.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the 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.

```
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);
                }
        }
}
```

Unchecked Exception

Output:

```
Caught inside demoproc.
Recaught:java.lang.NullPointerExcept
ion: demo
```

Exception Handling : throws

- A **throws** clause lists the types of exceptions that a method might throw.
- All exceptions (except those of type **Error** or **RuntimeException**, or any of their subclasses) that a method can throw must be declared in the **throws** clause.
- Syntax :

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

Exception Handling : throws

```
class ThrowsDemo
{
        static void throwOne()
        {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                throwOne();
        }
}
```

Checked Exception

This program contains an error and will not compile.

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

Output:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Exception Handling : finally

- **finally** creates a block of code that will be executed after a **try/catch** block has completed.
- **finally** block is used to handle any exception generated within a **try** block which is not caught by any of the previous catch statements.
- 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, the **finally** clause is also executed just before the method returns.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Exception Handling : finally

- Syntax :

```
try
{
        ...................
}
finally
{
        .................
}
```

```
try
{
        ...................
}
catch(){...............}
catch(){...............}
catch(){...............}
.
.
finally
{
        .................
}
```

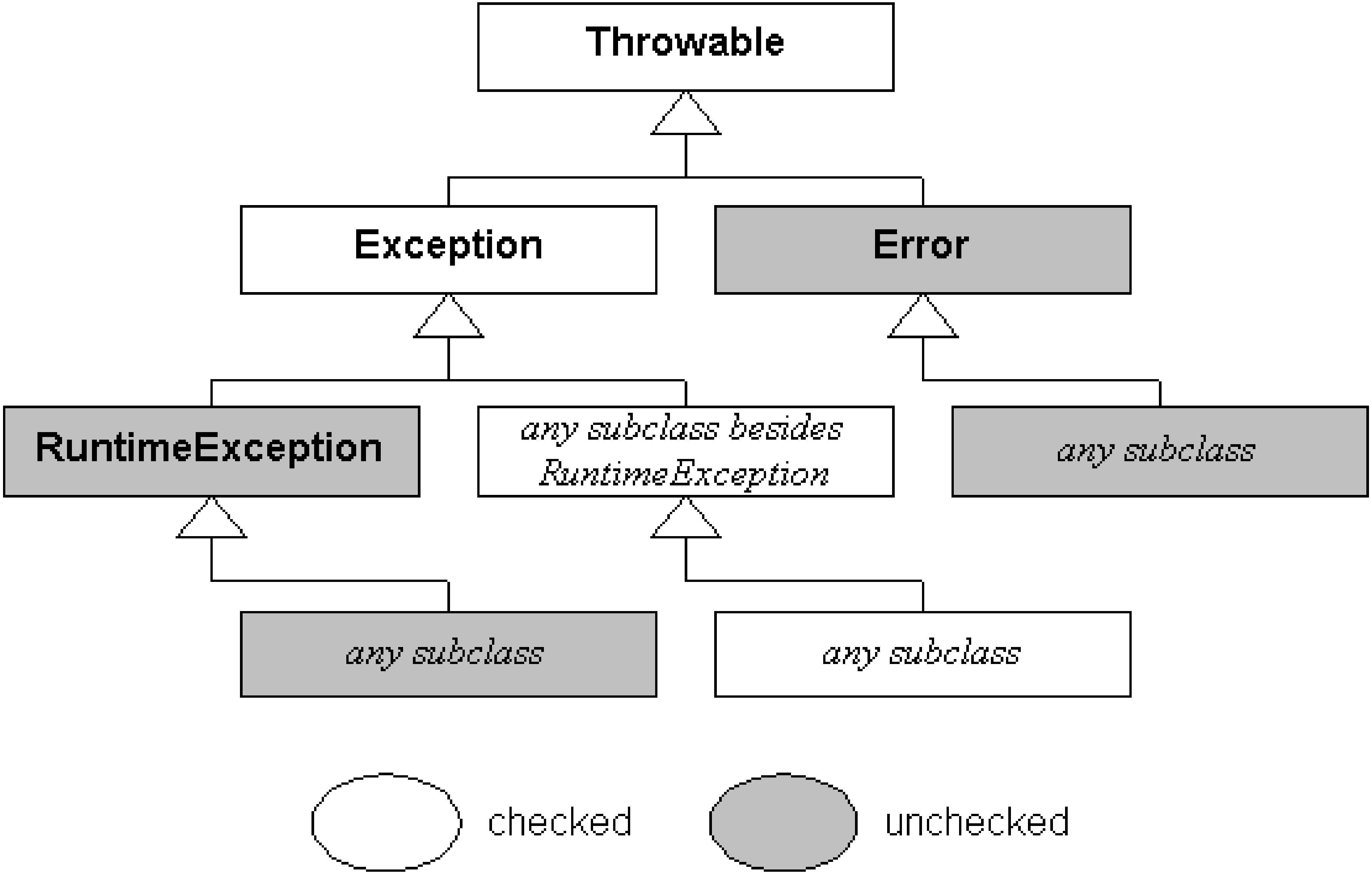
```
class FinallyDemo
{
        // Through an exception out of the method.
        static void procA()
        {       try
                {
                System.out.println("inside procA");
                throw new RuntimeException("demo");
                }
                finally
                {
                System.out.println("procA's finally");
                }
        }
        // Return from within a try block.
        static void procB()
        {       try
                {
                System.out.println("inside procB");
                 return;
                }
                finally
                {System.out.println("procB's finally");
                }
        }
        // Execute a try block normally.
        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 :

```
    inside procA
    procA’s finally
    Exception caught
    inside procB
   procB’s finally
    inside procC
     procC’s finally
```

Exception Handling : Built in Exceptions

- **java.lang**, package contains several exception classes.
- Two types of exceptions:
  - Unchecked exceptions
    - Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.
    - They need not be included in any method’s **throws** list.
    - The compiler does not check to see if a method handles or throws these exceptions.
  - Checked exceptions :
    - Some exceptions are defined by **java.lang** that must be included in a method’s **throws** list if that method can generate one of these exceptions and does not handle it itself.



Throwable

Exception

Error

RuntimeException

any subclass besides RuntimeException

any subclass

any subclass

any subclass

checked

unchecked

Java’s Unchecked RuntimeException Subclasses

Exception	Meaning
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 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.

Java’s Checked Exception Subclasses

Exception	Meaning
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.

Exception Handling : User Defined Exceptions

```
class MyException extends Exception
{
        private int detail;
        MyException(int a)
        {
                detail = a;
        }
        public String toString()
        {
                return "MyException[" + detail + "]";
        }
}
```

```
Output :
    Called compute(1)
    Normal exit
    Called compute(20)
    Caught MyException[20]
```

```
class ExceptionDemo
{
        static void compute(int a) throws MyException
        {
                System.out.println("Called compute(" + a + ")");
                if(a > 10)
                        throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[])
        {
                try
                {
                        compute(1);
                        compute(20);
                }
                catch (MyException e)
                {System.out.println("Caught " + e);
                }
        }
}
```