S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.
Department of Computer Science & Engineering

Course: Programming in Java(18CS653)

Module 4: Packages and Interfaces, Exception Handling in Java

Prof. Prasanna Patil
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Packages in JAVA

- A **java package** is a group of similar types of classes, interfaces and sub- packages.
- Package in java can be categorized in two form,
    - built-in package
    - user-defined package
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantages of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Syntax

- The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple
    {
    public static void main(String args[])
    {
       System.out.println("Welcome to package");
    }
    }
```

How to access package from another package?

- There are three ways to access the package from outside the package.
  - import package.*;
  - import package.classname;
  - fully qualified name.

1. *Using packagename.**

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

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();
}
}
```

Output:Hello

2. *Using packagename.classname*

- If you import package.classname then only declared class of this package will be accessible.
- **Example of package by import package.classname**

```
//save by A.java
package pack;
public class A
{
public void msg()
{
System.out.println("Hello");
```

```
}
}
 //save by B.java
package mypack;
import pack.A;
class B
{
public static void main(String args[])
{
A obj = new A();
obj.msg();
}
}
```

Output:Hello

3. *Using Fully Qualified name*

- If you use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to import.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

- **Example of package by import fully qualified name**

```
//save by A.java
 package pack;
public class A
{
public void msg()
{
System.out.println("Hello");
}
}

//save by B.java
package mypack;
class B
{
public static void main(String args[])
{
pack.A obj = new pack.A();      //using fully qualified name
obj.msg();
}
}
```

Output:Hello

Access Modifiers/Specifiers

- The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - private
  - default
  - protected
  - public

- **private access modifier**
  - The private access modifier is accessible only within class.
- **default access modifier**
  - If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.
- **protected access modifier**
  - The **protected access modifier** is accessible within package and outside the package but through inheritance only.
  - The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- **public access modifier**
  - The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

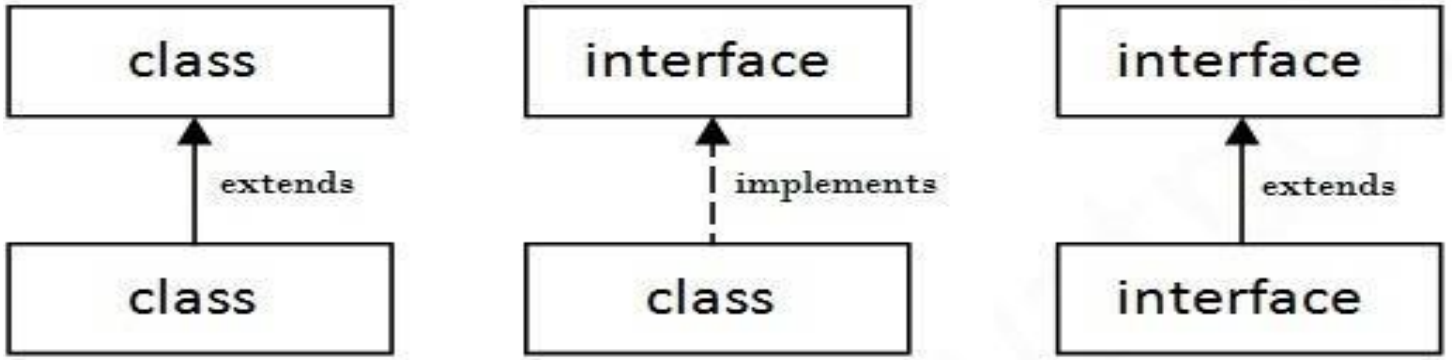
Access Modifier	within class	within package	Outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Interfaces in Java

Interfaces in java

- An **interface in java** is a blueprint of a class. It has static final variables and abstract methods.
- Interface fields are public, static and final by default, and methods are public and abstract.
- The interface in java is **a mechanism to achieve abstraction**.
- By interface, we can support the functionality of multiple inheritance.

Understanding relationship between classes and interfaces



Syntax

```
interface interfacename
{
 // final fields;
 //abstract methods
}
class classname implements interfacename
{
// body of class
}
```

```
//Example 1
interface printable
{
  int PI = 22;
  void print();
}

class Test implements printable
{
  public void print()
  {
    System.out.println("Hello");
    System.out.println("PI = "+PI);
  }
}

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

Output:

Hello

```
//Example 2
//Interface declaration: by first user
interface Drawable
{
void draw();
}

//Implementation: by second user

class Rectangle implements Drawable
{
public void draw()
{
System.out.println("drawing rectangle");
}
}

class Circle implements Drawable
{
public void draw()
{
System.out.println("drawing circle");
}
}
```

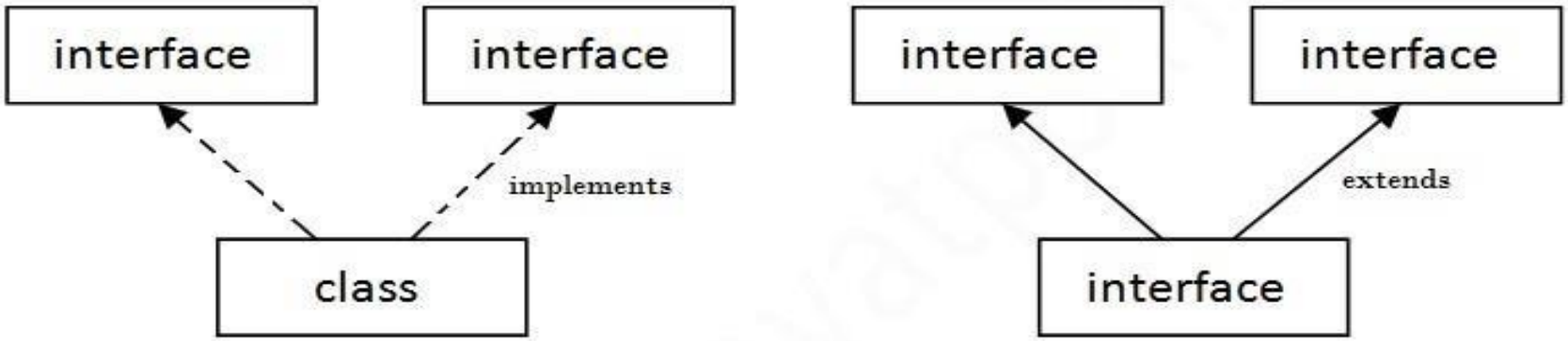
```
//Using interface: by third user
class TestInterface1
{
public static void main(String args[])
{
//In real scenario, object is provided by method e.g. getDrawable()
Drawable d=new Circle();
 d.draw();
}
}
```

Output:

drawing circle

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
//Example
interface Printable
{
void print();
}

interface Showable
{
void show();
}

class Pgm2 implements Printable,Showable
{
public void print()
{
System.out.println("Hello");
}

public void show()
{
System.out.println("Welcome");
}
}
```

```
Class InterfaceDemo
{
public static void main(String args[])
{
Pgm2 obj = new Pgm2 ();
obj.print();
obj.show();
}
}
```

Output:
Hello
Welcome

```
//Example
 interface Printable
{
void print();
}

interface Showable
{
void print();
}

class InterfacePgm1 implements Printable, Showable
{
public void print()
{
System.out.println("Hello");
}
}
class InterfaceDemo
{
public static void main(String args[])
{
InterfacePgm1 obj = new InterfacePgm1 (); obj.print();
}
}
```

Output:

Hello

- **Interface inheritance**
  - A class implements interface but one interface extends another interface .

```
interface Printable
{
void print();
}

interface Showable extends Printable
{
void show();
}

class InterfacePgm2 implements Showable
{
public void print()
{
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
} }
```

```
Class InterfaceDemo2
{
public static void main(String args[])
{
InterfacePgm2 obj = new InterfacePgm2 ();
obj.print();
obj.show();
}
}
```

Output:

```
Hello
Welcome
```

```
interface Printable
{
void print();
}
interface Displable
{
      void display();
}

interface Showable extends Printable, Displable
{
void show();
}

class InterfacePgm2 implements Showable
{
public void print()
{
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
}
public void display()
{
System.out.println("Good Bye");
}
}
class InDemo2
{
public static void main(String args[])
{
InterfacePgm2 obj = new InterfacePgm2 ();
obj.print();
obj.show();
obj.display();
}
}
```

Output

Hello

Welcome

Good Bye

Exception Handling in Java

Introduction

- It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions.
- An abnormal event can be an error in the program.

- Errors in a java program are categorized into two groups:
  - **Compile-time errors**
  - **Run-time errors**
- **Concepts of Exceptions**
- An exception is a run-time error that occurs during the execution of a java program.

Examples

- Divide by zero
- Running out of memory
- Resource allocation errors
- Inability to find files
- Problems in network connectivity

Exception handling techniques:

- Java exception handling is managed via five keywords they are:
  - 1. try.
  - 2. catch.
  - 3. throw.
  - 4. throws.
  - 5. finally.

Exception handling Statement Syntax

```
try
{
  <code>
}
catch (<exception type1> <parameter1>)
{
  // 0 or more<statements>

}
finally
{
  // finally block<statements>
}
```

1. try Block: The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.
2. catch Block: Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed .
3. finally Block: A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.

Example:

```
// 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");
        }
}
```

- Output

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

Multiple catch Blocks:

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

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

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

                try
                {
                        int a=args.length;
                        System.out.println("a="+a);
                         int b=50/a;
                         int c[]={1}
                }
                catch (ArithmeticException e)
                {
                      System.out.println ("Division by zero");
                }

                catch (ArrayIndexOutOfBoundsException e)
                {
                       System.out.println (" array index out of bound");
                }
      }
}
```

OUTPUT

Division by zero

array index out of bound

Nested try Statements

- These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try-block.

```
try
{
   // statements
   // statements

   try
   {
      // statements
      // statements
   }
   catch (<exception_two> obj)
   {
      // statements
   }

   // statements
   // statements
}
catch (<exception_two> obj)
{
   // statements
}
```

```
class Nested_Try
{
   public static void main (String args [ ] )
   {
      try
      {
           int a = Integer.parseInt (args [0]);
           int b = Integer.parseInt (args [1]);
           int quot = 0;

           try
           {
              quot = a / b;
              System.out.println(quot);
           }
           catch (ArithmeticException e)
           {
              System.out.println("divide by zero");
           }
       }
       catch (NumberFormatException e)
       {
           System.out.println ("Incorrect argument type");
       }
   }
}
```

- Output

```
java Nested_Try 2 4 6
```

```
java  Nested_Try 2 4  aa
```

```
java Nested_Try 2 4 0
```

throw Keyword

- throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown.
- Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.
- Syntax :

  throw *ThrowableInstance*

Creating Instance of Throwable class

- There are two possible ways to get an instance of class Throwable,
  - 1. Using a parameter in catch block.
  - 2. Creating instance with **new operator.**

new NullPointerException("test");

```
class Test
{
        static void avg()
        {
                try
                {
                        throw new ArithmeticException("demo");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught");
                }
        }
        public static void main(String args[])
        {
                avg();
        }
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

throws Keyword

- Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws keyword.**
- **Syntax :**

type method_name(parameter_list) ***throws*** ***exception_list***

```
{
        //definition of method
}
```

```
class Test
{
        static void check() throws ArithmeticException
        {
                System.out.println("Inside check function");
                throw new ArithmeticException("demo");
        }

        public static void main(String args[])
        {
                try
                {
                        check();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("caught" + e);
                }
        }
}
```

finally

- The finally clause is written with the try-catch statement. It is guaranteed to be executed after a catch block or before the method quits.

```
try
{
   // statements
}
```

```
catch (<exception> obj)
{
   // statements
}
finally
{
   //statements
}
```

```
class Finally_Block
{
   static void division ( )
   {
      try
      {
         int num = 34, den = 0;
         int quot = num / den;
      }
      catch(ArithmeticException e)
      {
         System.out.println ("Divide by zero");
      }
      finally
      {
         System.out.println ("In the finally block");
      }
   }
class Mypgm
{
   public static void main(String args[])
   {
         Finally_Block    f=new Finally_Block();
         f.division ( );
   }
}
```

OUTPUT

```
Divide by zero
In the finally block
```

Java's Built-in Exceptions

- Inside the standard package java.lang, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException.
- These exceptions need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in java.lang are listed in Table 10-1.
- Table 10-2 lists those exceptions 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. These are called checked exceptions.

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.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
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.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

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.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating Your Own Exception Subclasses

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- Just define a subclass of Exception (which is, of course, a subclass of Throwable).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The Exception class does not define any methods of its own.
- It does, of course, inherit those methods provided by Throwable.
- Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.
- They are shown in Table 10-3.
- You may also wish to override one or more of these methods in exception classes that you create.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

TABLE 10-3 The Methods Defined by **Throwable**

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
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);
}
}
}
```

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

Chained Exceptions

- The chained exception feature allows you to associate another exception with an exception.
- This second exception describes the cause of the first exception.
- For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero.
- However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.
- Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.
- Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

- To allow chained exceptions, two constructors and two methods were added to **Throwable.**
- The constructors are shown here:

  Throwable(Throwable *causeExc)*

  Throwable(String *msg, Throwable causeExc)*
- In the first form, *causeExc is the exception that causes the current exception. That is, causeExc* is the underlying reason that an exception occurred.
- The second form allows you to specify a description at the same time that you specify a cause exception.

- The chained exception methods added to Throwable are getCause( ) and initCause( ).

  Throwable getCause( )

  Throwable initCause(Throwable *causeExc)*
- The getCause( ) method returns the exception that underlies the current exception.
- If there is no underlying exception, null is returned.
- The initCause( ) method associates *causeExc with* the invoking exception and returns a reference to the exception.
- Thus, you can associate a cause with an exception after the exception has been created.

```
// Demonstrate exception chaining.
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " + e.getCause());
}
}
}
```

Output:
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause

Using Exceptions

- Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- It is important to think of **try, throw, and catch** as clean ways to handle errors and unusual boundary conditions in your program's logic.
- Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions.
- Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.
- One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching.