OBJECT ORIENTED PROGRAMMING WITH JAVA

Interfaces in Java – II

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Various Types of Interface

Some of Java's Most used interfaces

Iterator	• To run through a collection of objects without knowing how the objects are stored, for example, in array, list, bag, or set.
Cloneable	• To make a copy of an existing object via the `clone()` method on the class `Object`.
Serializable	• Pack a web of objects such that it can be send over a network or stored to disk. A naturally later be restored as a web of objects `Comparable`.
Comparable	• To make a total order of objects, for example, 3, 56, 67, 879, 3422, 34234

Iterator interface

The Iterator interface in the package **java.util** is a basic iterator that works on collections.

```
package java.util.*;
public interface Iterator {
     public abstract boolean hasNext();  // Check, if the list has more
     Object next();   // Return the next element
     void remove(); // optional throws exception
 }
     // use an iterator
      myShapes = getSomeCollectionOfShapes(); // Has set of objects
      Iterator iter = myShapes.iterator();
          while (iter.hasNext()) {
            Shape s = (Shape)iter.next(); // downcast
            s.draw();
      }
```

Cloneable interface

- A class `X` that implements the `Cloneable` interface tells that the objects of class `X` can be cloned.
- The interface is empty, that is, it has no method.
- Returns an identical copy of an object.
  - A shallow copy, by default.
  - A deep copy is often preferable.
- Prevention of cloning
  - Necessary, if unique attribute, for example, database lock or open file reference.
  - Not sufficient to omit to implement `Cloneable`.
    - Sub classes might implement it.
  - Clone method should throw an exception:
    - `CloneNotSupportedException`

Cloneable Interface: Example

```
public class Car implements Cloneable{
      private String make;
      private String model;
      private double price;
      public Car() {// default constructor
            this("", "", 0.0);
      }
 // give reasonable values to instance variables
      public Car(String make, String model, double price){
            this.make = make;
            this.model = model;
            this.price = price;
     }
     public Object clone(){// the Cloneable interface
            return new Car(this.make, this.model, this.price);
    }
}
```

Serializable interface

```
public class Car implements Serializable {
       // rest of class unaltered

}
      // write to and read from disk
import java.io.*;
public class SerializeDemo{
       Car myToyota, anotherToyota;
       myToyota = new Car("Toyota", "Carina", 42312);
       ObjectOutputStream out = getOutput();
       out.writeObject(myToyota);
       ObjectInputStream in = getInput();
       anotherToyota = (Car)in.readObject();
}
```

- ✓ A class X that implements the Serializable interface tells clients that X objects can be stored on a file or other persistent media.
- ✓ The interface is empty, that is, has no methods.

Comparable Interface

In the package *java.lang*.

Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
package java.lang.*;
public interface Comparable {
      int compareTo(Object o);
}
```

Comparable interface: Example

```
public class IPAddress implements Comparable{
     private int[] n; // here IP stored, e.g., 125.255.231.123
     /** The Comparable interface */
     public int compareTo(Object o){
          IPAddress other = (IPAddress) o; // downcast
          int result = 0;
          for(int i = 0; i < n.length; i++){
               if (this.getNum(i) < other.getNum(i)){
                    result = -1;
                    break;
               }
               if (this.getNum(i) > other.getNum(i)){
                    result = 1;
                    break;
               }
          }
          return result;
     }
}
```

Some Salient Points

Defining an interface

- Defining an interface is similar to creating a new class.
- An interface definition has two components: the interface declaration and the interface body.

```
interfaceDeclaration
   {
            interfaceBody
   }
```

The `interfaceDeclaration` declares various attributes about the interface such as its name and whether it extends another interface, etc.

➢ The `interfaceBody` contains the constant and method declarations within the interface.

Defining an interface

```
public interface StockWatcher
{
     final String sunTicker = "SUNW";
     final String oracleTicker = "ORCL";
     final String ciscoTicker = "CSCO";
     void valueChanged (String tickerSymbol, double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface.

Implementing an interfaces

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

```
class classname [extends superclass] [implements interface [,interface...]]
 {
      // class-body
 }
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then that method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.

Implementing interfaces: An example

Example: A class that implements, say Callback interface:

```
class Client implements Callback {
     // Implement Callback's interface

      public void callback(int p) {
      System.out.println("callback called with " + p);
    }
}
```

When you implement an interface method, it must be declared as `public`.

Implementing interfaces: An example

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

Example: The following version of `Client` implements `callback( )` and adds the method `nonIfaceMeth( )`

```
class Client implements Callback {
 // Implement Callback's interface
    public void callback(int p) { System.out.println("callback called
    with"+ p);
     }
    void nonIfaceMeth() {
             System.out.println("Classes that implement interfaces " + "may
             also define other members, too.");
    }
}
```

Partial implementations

- If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as abstract.

Example

```
abstract class Incomplete implements Callback {
     int a, b;
     void show() {
        System.out.println(a + " " + b);
     }
}
```

Here, the class `Incomplete` does not implement `callback( )` and must be declared as `abstract`. Any class that inherits *`Incomplete`* must implement `callback( )` or be declared `abstract` itself.

Nested interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a nested interface.

A nested interface can be declared as public, private, or protected.

When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Nested interfaces: Example

```
// This class contains a nested interface.
class A {
  public interface NestedIF {
      boolean isNotNegative(int x);
} }
// B implements the nested interface.
class B implements A.NestedIF {
       public boolean isNotNegative(int x) {
          return x < 0 ? false: true;
} }
 class NestedIFDemo {
      public static void main(String args[]) {
       // use a nested interface reference
       A.NestedIF nif = new B();
       if(nif.isNotNegative(10))
                 System.out.println("10 is not negative");
                  if(nif.isNotNegative(-12))
                            System.out.println("this won't
                 be displayed");
 } }
```

A defines a member interface called *NestedIF* and that it is declared public.

B implements the nested interface by specifying implements *A.NestedIF*

Inside the *main*( ) method, an *A.NestedIF* reference called *nif* is created, and it is assigned a reference to a *B* object. Because B implements *A.NestedIF*.

Variables in interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

```
import java.util.Random;
    interface SharedConstants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
```

This program makes use of one of Java's standard classes: Random, which provides pseudorandom numbers.

Variables in interfaces

```
class Question implements SharedConstants {
      Random rand = new Random();
       int ask() {
           int prob = (int) (100 * rand.nextDouble());
          if (prob < 30)
                    return NO; // 30%
          else if (prob < 60)
                    return YES; // 30%
          else if (prob < 75)
                    return LATER; // 15%
          else if (prob < 98)
                    return SOON; // 13%
          else
                    return NEVER; // 2%
       }
}
```

In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the classes, **Question** implements the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside the class, the code refers to these constants as if each class had defined or inherited them directly.

Variables in interfaces

```
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
          case NO:
              System.out.println("No");
              break;
           case YES:
              System.out.println("Yes");
              break;
          case MAYBE:
              System.out.println("Maybe");
              break;
          case LATER:
              System.out.println("Later");
              break;
          case SOON:
              System.out.println("Soon");
              break;
          case NEVER:
              System.out.println("Never");
              break;
          }
      }
```

```
    public static void main(String args[]) {
       Question q = new Question();
       answer(q.ask());
       answer(q.ask());
       answer(q.ask());
       answer(q.ask());
     }
   }
```

Interfaces can be extended

- An interface can inherit another using the keyword **`extends`**. The syntax is the same as for inheriting classes.

```
// One interface can extend another.
interface A {
 void meth1();
 void meth2();
}
// B now includes meth1() and meth2() --
it adds meth3().
interface B extends A {
      void meth3();
}
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Interfaces can be extended

```
// This class must implement all of A and B
class MyClass implements B {
      public void meth1() {
          System.out.println("Implement meth1().");
      }
       public void meth2() {
           System.out.println("Implement meth2().");
      }
       public void meth3() {
            System.out.println("Implement meth3().");
       }
}
class IFExtend {
     public static void main(String arg[]) {
          MyClass ob = new MyClass();
            ob.meth1();
            ob.meth2();
            ob.meth3();
      }
}
```

As an experiment, if you try removing the implementation for **meth1( )** in MyClass, it will cause a compile-time error.

Multiple Inheritance Issue

Multiple inheritance issues

- Java does not support the multiple inheritance of classes. There is a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

- For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both Alpha and Beta provide a method called **reset( )** for which both declare a default implementation? Is the version by Alpha or the version by Beta used by MyClass? Or, consider a situation in which Beta extends Alpha. Which version of the default method is used? Or, what if MyClass provides its own implementation of the method?

- To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

Multiple inheritance issues

Rules.

- First, in all cases, a class implementation takes priority over an interface default implementation.
  - Thus, if `MyClass` provides an override of the `reset( )` method, `MyClass`' version is used.
  - This is the case even if `MyClass` implements, say both `Alpha` and `Beta`. In this case, both defaults are overridden by `MyClass`' implementation.
- Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will occurs. Continuing with the example, if `MyClass` implements both `Alpha` and `Beta`, but does not override `reset( )`, then an error will occur.

Multiple inheritance issues

Rules.

- In cases, one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if `Beta` extends `Alpha`, then `Beta`'s version of `reset( )` will be used.

- It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of `super`. Its general form is shown here:

Multiple Inheritance Issues

- It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of `super`. Its general form is shown here:

```
InterfaceName.super.methodName( )
```

- For example, if `Beta` wants to refer to `Alpha`'s default for `reset( )`, it can use this statement:

```
Alpha.super.reset();
```

Questions to think…

- How a robust program can be developed in Java ?
- How Java manages different types of errors in programs so that it can avoid abnormal termination of programs?

Thank You

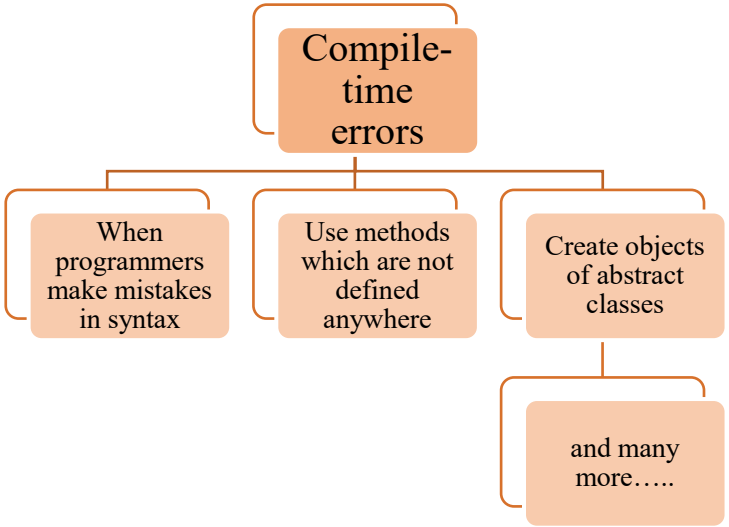
OBJECT ORIENTED PROGRAMMING WITH JAVA

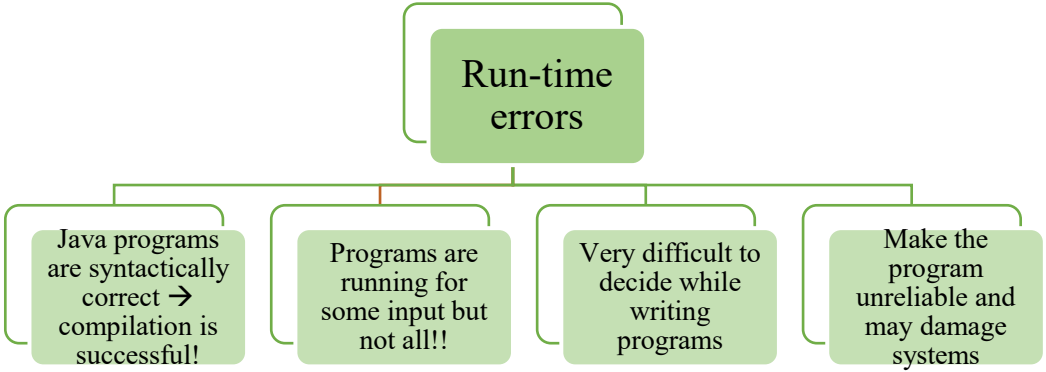
Exception Handling in Java – I

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Concept of Errors in Java

Errors in Java programs





Compile-time errors: Examples

```
Class Error {
     Public static void main (string args [ ]) {
          system.out.print("Can you find errors in me?")
     }
}
class AnotherError {
     public void insert( ){
          System.out.print("To insert a text");
     }

     abstract void delete( ){
           System.out.print("To delete a text");
      }

}
```

Some common compile-time errors

- Missing semicolons.
- Missing (or mismatch of brackets) in classes and methods.
- Misspelling of identifiers or keywords.
- Missing double quotes in strings.
- Use of undeclared variables.
- Incomplete types in assignment / initialization.
- Bad references to objects.

and many more ….

Run-time errors: Examples

```
class Error {
      public static void main (String args [ ]) {
            int a = Integer.parseInt(args[0]};
            int b = Integer.parseInt(args[1]};
            int c = a/b;
            System.out.println("Value of c =" + c);
       }
}
```

javac Error.java	→	Error.class
java Error 1 2	→	Value of c = 0
java Error -1 -2	→	Value of c = 0
java Error 0 1	→	Value of c = 0
java Error 1 0	→	?????

Run-time trrors: Examples

```
class Error {
      public static void main (String args [ ]) {
            int a = Integer.parseInt(args[0]};
            int b = Integer.parseInt(args[1]};
            int c = a/b;
            System.out.println("Value of c =" + c);
       }
}
```

java Error 1.5 2.5 → ???????

Some common run-time errors

- A user has entered invalid data.
- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incomplete class or type.
- Trying to cast an instance of a class to one of its subclasses.
- Trying to illegally change the state of a thread.
- Attempting use a negative size for an array.
- Null object reference.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

and many more ….

So errors are due to users, programmers and physical resources which are very difficult to taken into consideration while writing the programs …

Errors and exceptions in Java

Whenever an Error or Exception occurs, Java Run-Time Environment throws an object corresponding to that.

Examples:

Java Run-Time Environment throws an object called `IllegalArumentType`

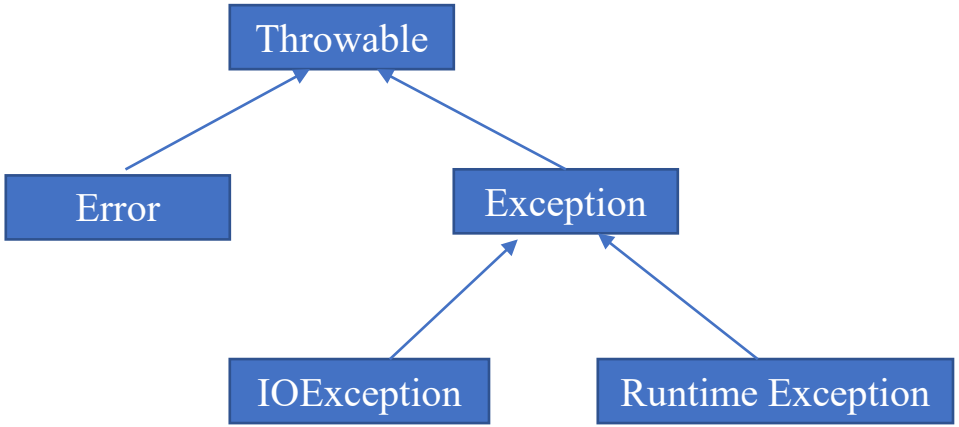
when a method `m(int x, int y)` is called as `m(1.5, 4);`

Java Run-Time Environment throws an object called `ArithmeticException`

when a = x/(b – c); and b = c;

Errors and exceptions in Java

To handle the common possible errors and exceptions, Java offers a class hierarchy



All these classes (and subclasses) are defined in java.lang package

Exception Handling Mechanism

Exception handling in Java

- Java provides Run Time Error Management to deal with errors and exceptions.
- During the execution of a program, when an exceptional condition arises, an object of the respective exception classes is created and thrown in with method which caused the exception.
- That method may choose to catch the exception and then can guard against premature exit or may have a block of code to execute.
- Java exception handling is managed via five keywords:

```
try {     ….      }
catch {     ….      }
throw
throws
finally {     ….      }
```

Exception handling in Java

```
method
{
        try{
                statement(s) that may  cause
                exception(s)
                throw exception object
        }
        catch {
                statement(s) that handle the
                exception(s)
        }
        catch {
                statement(s) that handle the
                exception(s)
        }
        fially {
                statement(s) that execute on
                exit of the exception handling
        }
        .
}
```

Try block

Catch block 1

Catch block 2

Finally block

Exception handling in Java

Simple try-catch	try with multiple catch	Multiple exceptions	Exception with exit code	Throwing own exception	Nested try-catch block
try with a single catch	try with multiple catch	Multiple exception with one catch	try-catch with finally	Throws/ throw in try-block	try within another try

Simple `try-catch` block

Simple try-catch

```
method
{
        .
        try{
                statement(s) that may  cause
                exception(s)

        }
        catch {
                statement(s) that handle the
                exception(s)
        }
                .
}
```

Try block

Catch block 1

Simple try-catch: Example

```
class DivideZero {
      static int anyFunction ( int x, int y ) {
          int a = x/y;
          return (a);
      }
      public  static void main (String args [  ] ) {
          int result = anyFunction (25, 0) ;
          // Exception occurs here as y = 0
          System.out.println ( " Result : " + result );
      }
}
```

Simple try-catch: Example

Run time report while running the Application DivideZero

C:\> java DivideZero // To run the Application DivideZero

System produce the output as :

```
java.lang.ArithmeticException: / by zero
                    at DivideZero.AnyFunction    (DivideZero.Java: 3)
                    at DivideZero.main           (DivideZero.Java: 7)
```

Simple try-catch: Example

The Java program with try for any run-time error and catch it …

```
class DivideZero {
    static int anyFunction (int x, int y ){
            try {
                int a = x/y;
                return(a);
            }
            catch (ArithmeticException e) {
                System.out.println ( "a = x/y is bypassed… Enter y as non-
                zero" );
            }
    }
…… main method in the next slide
```

Continued to the next slide …

Simple try-catch: Example

Continued from the previous slide…

```
    public static void main (String args[ ] {
        int a,b, result;
        system.out.print("Enter any two integers : ");
        a = System.in.read( );
        b = System.in.read( );
        result  = any Function (a, b);
        System.out.println ( "Result : " + result);
    }
}
```

try-catch: Making program robust

```
Class TestException {
     public static void main (String args[ ] {
        int a, b, c;
        int x, y;
        try {
            x = a / (b-c);
        }
        catch (ArithmeticException e) {
             System.out.println(" b = c: Divide by zero error….!");
        }
        y = a / (b + c);
        System.out.println ( "y = " + y);
     }
 }
```

Is the problem robust?

try-catch: Making program robust

```
class CommandLineInput {
     public static void main (String args[ ] {
          int number, count;
          for (int i = 0; i < args.length; i++)  {
             number = Integer.parseInt(args[i]);
             System.out.println ( "Number at " + i + args.[i]);
          }
     }
}
```

What are the vulnerabilities in this program?

try-catch: Making program robust

```
class CommandLineInput {
        public static void main (String args[ ] {
            int number, count;
             for (int i = 0; i < args.length; i++)  {
                   number = Integer.parseInt(args[i]);
                  System.out.println ( "Number at " + i + args.[i]);
             }
        }
   }
```

Will the program work with the following test cases?

C:\> java CommandLineInput *1 2 3*

C:\> java CommandLineInput *1 -2 3*

C:\> java CommandLineInput *1.5 2 3.9*

C:\> java CommandLineInput *1 Java -0.5*

try-catch: Making the program robust

```
class CommandLineInput {
     public static void main (String args[ ] {
          int number, InvalidCount = 0; validCount = 0;
          for (int i = 0; i < args.length; i++)  {
                try {
                     number = Integer.parseInt(args[i]);
                } catch (NumberFormatException e)  {
                      inavlidCount++;
                      System.out.println ( "Invalid number at " + i + args.[i]);
                   }
                validCount++;
                System.out.println ( "Valid number at " + i + args.[i]);
          }
          System.out.println ( "Invalid entries: " + inValidCount);
          System.out.println ( "Valid entries: " + validCount);
     }
  }
```

C:\> java CommandLineInput 10 22.34 55 Java K 69 2012

Question to think…

- How to make more simple yet effective program against multiple errors?
- Is there any other feature(s) for exception handling?

Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Exception Handling in Java – II

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

try- with Multiple catch

try with Multiple catch

```
method
{
            .
            try{
                        statement(s) that may  cause
                        exception(s)
                        throw exception object
            }
            catch {
                        statement(s) that handle the
                        exception(s)
            }
            .
            .
            .
            catch {
                        statement(s) that handle the
                        exception(s)
            }
            .
}
```

Try block

Catch block 1

Catch block n

try with Multiple catch: Why?

```
class MultiCatch  {
    public static void main (String args[ ]) {
        int i = args.length;
        String myString = new String [i];
        if(args[0].equals("Java"))
            System.out.println("First word is Java !");
        System.out.println( " Number of arguments = " + i );
        int x = 18/ i;
        int y[ ] = {555, 999};
        y[i] = x;
    }
}
```

Identify the source of exceptions in the above program, if any.

try with Multiple catch: Why?

```
class MultiCatch  {
     public static void main (String args[ ]) {
          int i = args.length;
          String myString = new String [i];
          if(args[0].equals("Java"));
               System.out.println("First word is Java !");
           System.out.println( " Number of arguments = " + i );
          int x = 18/ i;
          int y[ ] = {555, 999};
          y[ i ] = x;
     }
}
```

C:\> java MultiCatch Java Welcome
C:\> java MultiCatch Welcome to Java World !
C:\> java MultiCatch

try with Multiple catch: Remedy

```
class MultiCatch  {
    public static void main (String args[ ]) {
        int i = args.length;                    // No of arguments in the command line
        String myString = new String [i];
        // If i = 0 then  myString null pointer error
        // #1 //  if(args[0].equals("Java"));
                System.out.println("First word is Java !");
        System.out.println( " Number of arguments = " + i );
        // # 2 //              int x = 18/ i;
        int y[ ] = {555, 999};              // y is an array of size 2 and index are 0,1
        // #3 //    y[ i ] = x;            // Index is out-of-range may occur if i > 1
   }
}
```

try with Multiple catch: Remedy

```
class MultiCatch  {
    public static void main (String args[ ]) {
           int i = args.length;              // Number of arguments in the command line
         try {
            String myString = new String [i];
            if(args[0].equals("Java"));      // #1 : If i = 0 then  myString null pointer error
                System.out.println("First word is Java !");
           System.out.println( " Number of arguments = " + i )
           int x = 18/ i;              // # 2 : Divide by zero error//
           int y[ ] = {555, 999};         // y is an array of size 2 and index are 0,1
           y[ i ] = x;     // #3 :  Index is out-of-range may occur if i > 1
        }
        catch (NullPointerException  e ) {                          // To catch the error at #1
              System.out.println ( " A null pointer exception : " + e );
        }
        catch (ArithmeticException  e ) {            // To catch the error at #2
            System.out.println ( " Divide by 0 : "+ e );
        }
        catch (ArrayIndexOutofBoundsException e ) { // To catch the error at  #3
            System.out.println ("Array Index OoB : " + e);
        }
    }
}
```

Multiple Errors with Single catch

Multiple errors with single catch

```
method
{
        .
        try{
                statement(s) that may  cause
                exception(s)

        }
        catch {
                statement(s) that handle the
                exception(s)
        }
                .
}
```

Try block

Catch block

Multiple errors with single catch: An Example

```
class ExceptionTest {
    public int j;
    static void main (String   args[ ] ) {
       for (int i = 0; i < 4; i++ ) {
         switch (i) {
            case 0 :
               int zero = 0;
               j = 999/ zero;
              break;
            case 1:
                 int b[ ] = null;
                 j = b[0] ;
                break;
             case 2:
                  int c[ ] = new int [2];
                  j = c[10];
                  break;
              case 3:
                  char ch = "Java".charAt(9);
                  break;
         }
       }
    }
}
```

Multiple errors with single catch: An Example

```
class ExceptionTest {
    public int j;
    static void main (String   args[ ] ) {
       for (int i = 0; i < 4; i++ ) {
       switch (i) {
            case 0:
                int zero = 0;
                j = 999/ zero; //Divide by zero
                break;
           case 1:
                 int b[ ] = null;
                 j = b[ 0]; //Null pointer error
                 break;
            case 2:
                 int c = new int [2] ;
                 j = c[10]; // Array index is out-of-bound
                break;
             case 3:
                char ch = "Java".charAt(9) ; // String index is out-of-bound
                break;
          }
      }
   }
}
```

Multiple errors with single catch: An Example

```
class ExceptionTest {
    public int j;
    static void main (String   args[ ] ) {
        for (int i = 0; i < 4; i++ ) {
         try {
             switch (i) {
                case 0 :
                  int zero = 0;
                  j = 999/ zero; // Divide by zero
                   break;
                case 1:
                  int b[ ] = null;
                  j = b[0] ; // Null pointer error
                  break;
                case 2:
                   int c = new int [2] ;
                   j = c[10]; // Array index is out-of-bound
                   break;
                case 3:
                   char ch = "Java".charAt(9) ;// String index is out-of-bound
                   break;
             }
         } catch (Exception e) {
              System.out.println("In Test case#"+i+ "\n");
              System.out.println (e.getMessage() );          }
    }
  }
}
```

`finally` in `try`-catch

finally in try-catch

```
method
{
        try{
                statement(s) that may  cause
                exception(s)
                throw exception object
        }
        catch {
                statement(s) that handle the
                exception(s)
        }
        finally {
                statement(s) that handle the
                exception(s)
        }
        .
}
```

Try block

Catch block

Finally block

finally in try-catch : An example

```
class FinallyDemo {
     public static void main (String [ ] args ) {
        int i = 0;
        String greetings [ ] = { "Hello Twinkle !", "Hello Java !", "Hello World ! "};
        while ( i < 4) {
          try {
            System.out.println (greetings [i++] );
          }catch (Exception e ) {
              System.out.println (e.toString );// Message of exception e in String format
          }
         finally {
             System.out.println("You should quit and reset index value < 3 :");
        }
     } // while ( )
   } // main ( )
} // class
```

`throw` in `try`-catch

throw in try-catch

```
method
{
        try{
                statement(s) that may  cause
                exception(s)
                throw exception object
        }
        catch {
                statement(s) that handle the
                exception(s)
        }
        .
        .
        .
        .
        .
}
```

Try block

Catch block

throw in try-catch : Mechanism

- 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.
- This is done by `throw` in the method declaration.
- A **`throw`** clause lists the types of exceptions that a method might throw.
- All other exceptions that a method can throw must be declared in the **`throw`** clause.
- If they are not, a compile-time error will result.

throw in try-catch : An Example

```
import java.lang.Exception;
class MyException extends Exception {
       MyException (String message) {
             super(message);
       }
}
class TestMyException {
      public statice void main (String args[ ])      {
      int x = 5;  y = 1000;
        try {
                 float z = (float) x / (float) y;
                if (z < 0.01)  throw new MyException ("Given data are not proper");
            }
       catch (MyException e) {
              System.out.println(e.getMessage());      }
       finally {
              System.out.println ("It prints always!");
       }
  }
}
```

throw in try-catch: An Example

```
class ThrowDemo {
   static void demoProc ( ) {
      try {
         throw new InterruptedException( "An Interrupt occurred" );
               // An exception object is created
       ……..   // Some code for demoProc( ) is here
     } catch (InterruptedException e) {
          System.out.println("Exception is caught in demoProc( )" );
          throw e; // Exception will be thrown to the caller of demoProc( )
       }
    }
    public static void main (Strings [ ] args ) {
     try {
            demoProc ( );
      } catch (Exception e) {
             System.out.println ( " Exception thrown  by demoProc( ) is caught here" );
        }
     }
 }
```

throws in try-catch : Mechanism

The general form of a method declaration that includes a `throws` clause:

```
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.

throws in try-catch : Mechanism

```
class InterestCalculator{
   public static void main(String args[ ] ) throws Exception {
       Float principalAmount = new Float(0);
       Float rateOfInterest = new Float(0);
       int numberOfYears = 0;
     try {
       DataInputStream in = new DataInputStream(System.in);
       String tempString;
       System.out.println("Enter Principal Amount: ");
       System.out.flush();
       tempString = in.readLine();
       principalAmount = Float.valueOf(tempString);
       System.out.println("Enter Rate of Interest: ");
       System.out.flush();
       tempString = in.readLine();
       rateOfInterest = Float.valueOf(tempString);
       System.out.println("Enter Number of Years: ");
       System.out.flush();
       tempString = in.readLine();
       numberOfYears = Integer.parseInt(tempString);
     }
     catch (Exception e) {}
        float interestTotal = principalAmount*rateOfInterest*numberOfYears;
       System.out.println("Total Interest = " + interestTotal);
    }
}
```

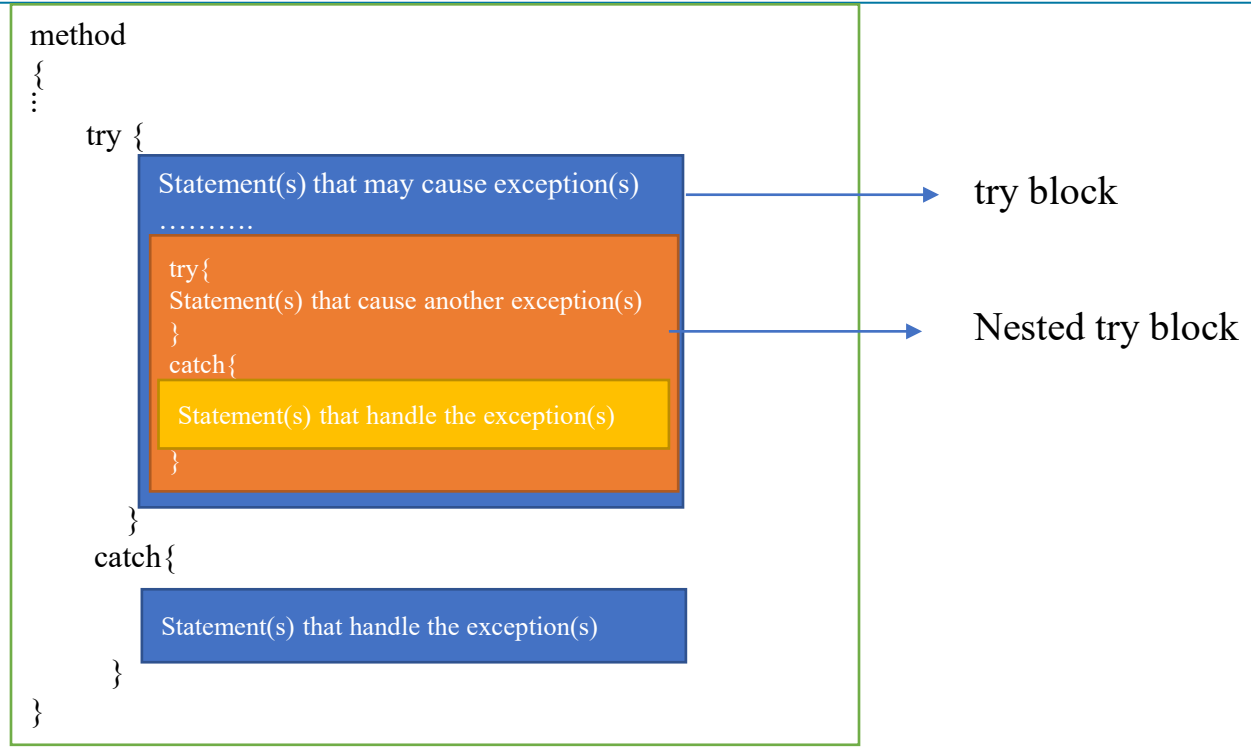
Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Exception Handling in Java – III

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Nested try-catch



Nested try-catch : An example

```
class NestedTry {
        public static void main(String args[]) {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
            if(a==1)
                a = a/(a-a);
            if(a==2) {
                int c[ ] = { 1 };
                c[2] = 99;
            }
        }
}
```

Nested try-catch : An example

```
class NestedTry {
    public static void main(String args[]) {
        int a = args.length;
        int b = 42 / a; // If no command-line args are present, then it generates a divide-by-zero exception
        System.out.println("a = " + a);
        if(a==1)
            a = a/(a-1);// If one command-line arg is used, then a divide-by-zero exception
        if(a==2) {
            int c[ ] = { 1 };
            c[a] = 99;// If two command-line args are used, then an out-of-bound exception
        }
    }
}
```

```
C:\> java NestedTry 1 2 3
C:\> java NestedTry 1 2
C:\> java NestedTry 1
C:\> java NestedTry
```

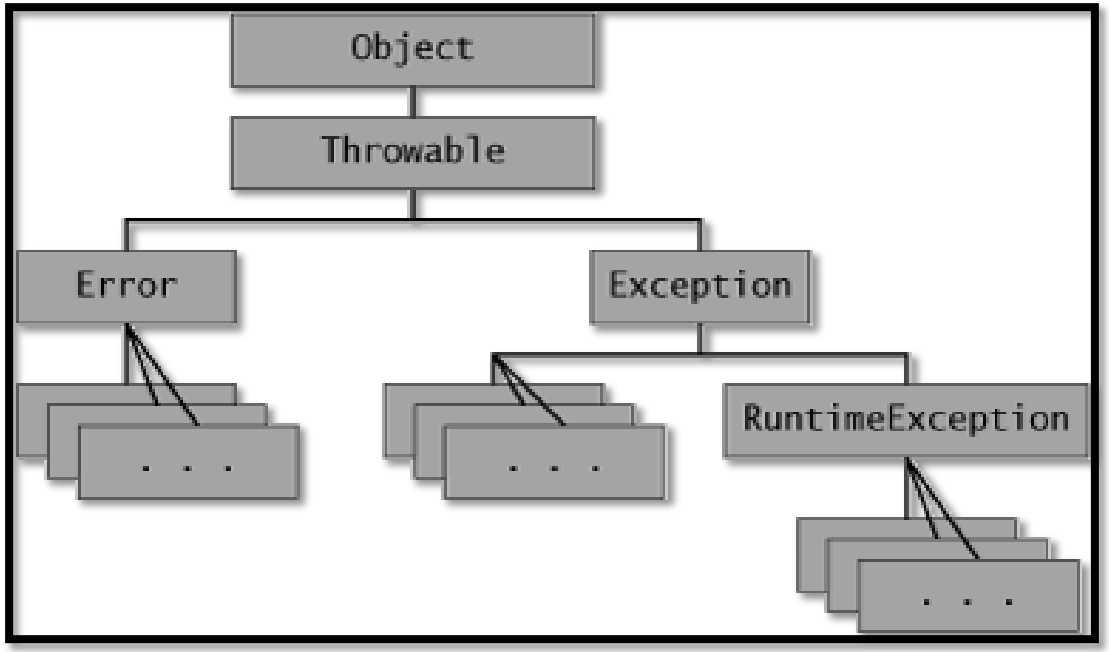
Nested try-catch : An example

```
class NestedTry {
      public static void main(String args[]) {
          try {
              // To catch divide-by-zero
             int a = args.length;
             int b = 42 / a;
            // divide-by-zero exception
             System.out.println("a = " + a);
             if(a==1)
              a = a/(a-a);
          // another divide-by-zero exception
              try {
                // nested try block
                   if(a==2) { // If two command-
                    line args are used, then an
                    out-of-bounds exception
                   int c[ ] = { 1 };
                   c[a] = 99;
                  }
```

```
          catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array
                index out-of-bounds: " + e);
           }
       } catch(ArithmeticException e) {
                System.out.println("Divide by 0:"+e);
           }
    }
}
```

Built-in Exception Handling in Java

Exception classes in Java.lang.Throwable



Exception classes in Java.lang.Throwable

RuntimeException sub classes:

- ArithmeticException
- ArrayIndexOutofBoundException
- ArrayStoreException
- ClassCasteException
- IlegalArgumentException
- SecurityException
- IncompatibleClassChangeError
- IndexOutofBoundException
- NegativeArraySizeException
- NullPointerException
- NumberFormatException
- StringIndexOutofBoundException

Exception classes in Java.lang.Throwable

Exception sub classes:

- ClassNotFoundException
- DataFormatException
- IllegalAccessException
- InstantiationException
- InterruptedException
- NoSuchMethodException
- RuntimeException

Exception classes in Java.lang.Throwable

Error Classes:

- ClassNotFoundException
- DataFormatException
- IllegalAccessException
- InstantiationException
- InterruptedException
- NoSuchMethodException
- RuntimeException

Exception classes in Java.lang.Throwable

- ClassCirculatoryError
- ClassFormatError
- Error
- IllegalAccessError
- IncompatibleClassChangeError
- InstantiationError
- LinkageError
- NoCassDefFoundError
- NoSuchMethodError
- NoSuchFieldError
- OutofMemoryError
- StackOverflowError
- Throwable
- UnknownError
- UnsatisfiedLinkError
- VerifyError
- VirtualMachineError

Exception methods

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

User Defined Exceptions

Declaring your own exception

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

- Syntax

```
class MyException extends Exception {
 . . .
}
```

Own exception: Example

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount) {
      this.amount = amount;
   }
   public double getAmount(){
      return amount;
   }
}
```

Own exception: Example

```
// File Name CheckingAccount.java
import java.io.*;
public class CheckingAccount
{
   private double balance;
   private int number;
   public CheckingAccount(int number) {
     this.number = number;
   }
   public void deposit(double amount)   {
     balance += amount;
   }
public void withdraw(double amount) throws
InsufficientFundsException {
      if(amount <= balance){
         balance -= amount;
      }
```

```
      else {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance() {
         return balance;
   }
   public int getNumber(){
         return number;
   }
}
```

Own exception: Example

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

```
Depositing $500...
Withdrawing $100...
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
      at
CheckingAccount.withdraw(CheckingAccount.java:25)
      at BankDemo.main(BankDemo.java:13
```

Final Round of Wrapping

Example program for `Scanner` : Maximum

```
import java.util.Scanner;

public class MaximumCalculator {
    public static void main(String args[]) {
        Scanner scnr = new Scanner(System.in);
        // Calculating the maximum two numbers in Java
        System.out.println("Please enter two numbers to find maximum of two");
        int a = scnr.nextInt();
        int b = scnr.nextInt();
        if (a > b) {
              System.out.printf("Between %d and %d, maximum is %d \n", a, b, a);
        }
        else {
            System.out.printf("Between %d and %d, maximum number is %d \n", a, b, b);
        }
    }
}
```

Example program with Scanner and array

```
import java.util.*;
 class SimpleArrayList{
 public static void main(String args[]){
     int sum = 0;
     float avg = 0;
     ArrayList <Integer> l = new ArrayList<Integer>();
     System.out.println("Enter the input ");
          Scanner input = new Scanner(System.in);
          while (input.hasNextInt()) {
               l.add(input.nextInt());
          }
          for (int i = 0; i < l.size(); i++) {
              sum = sum+l.get(i);
          }
          avg = sum/(l.size());
          System.out.println("Average : " + avg);
  }
}
```

```
C:\Users\Desktop\Java\SimpleArrayList>Ent
er the input
5
6
4^Z
Average :  5.0
```

Note:
Press Ctrl+Z to stop scanning.

Input with DataInputStream : Calculator Program

```
import java.io.*;

class InterestCalculator{
   public static void main(String args[ ] ) {
        Float principalAmount = new Float(0);
        Float rateOfInterest = new Float(0);
        int numberOfYears = 0;
        DataInputStream in = new DataInputStream(System.in);
        String tempString;
        System.out.println("Enter Principal Amount: ");
        System.out.flush();
        tempString = in.readLine();
        principalAmount = Float.valueOf(tempString);
        System.out.println("Enter Rate of Interest: ");
        System.out.flush();
        tempString = in.readLine();
        rateOfInterest = Float.valueOf(tempString);
        System.out.println("Enter Number of Years: ");
        System.out.flush();
        tempString = in.readLine();
        numberOfYears = Integer.parseInt(tempString);
        // Input is over: calculate the interest
        float interestTotal = principalAmount*rateOfInterest*numberOfYears;
        System.out.println("Total Interest = " + interestTotal);
    }
}
```

```
C:\Users\Desktop\Java\InterestCalculator>
Enter Principal Amount:
100.0
Enter Rate of Interest:
12.5
Enter Number of Years:
2
Total Interest = 25.0
```

Question to think…

- Java is well known for the distributed programming. How?
- How Java programming is to develop many sophisticated Internet programs?


Thank You