Exception Handling

PresentationPoint

- An *exception is an* abnormal condition that arises in a code sequence at run time.

- An exception is a run-time error.

- In computer languages that do not support exception handling, errors must be checked and handled manually.

- Java’s exception handling avoids these problems.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

- 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, or they can be manually generated by our code.

```
class Exc0
{
        public static void main(String args[])
        {
                  int d = 0;
                  int a = 42 / d;
                 System.out.println(“This code is safe”);
        }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws this exception.*

- This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

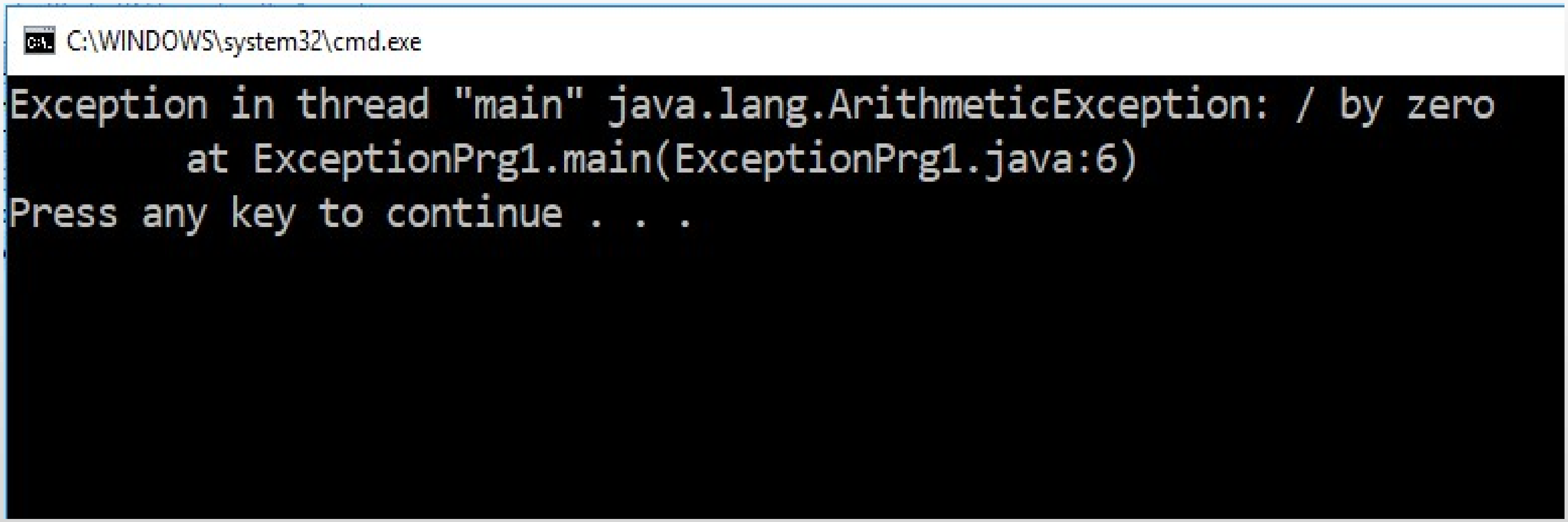
- Any exception that is not caught by our program will ultimately be processed by the default handler.

- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the output generated when this example is executed.

java.lang.ArithmeticException: / by zero

at Exc0.main(Exc0.java:4)

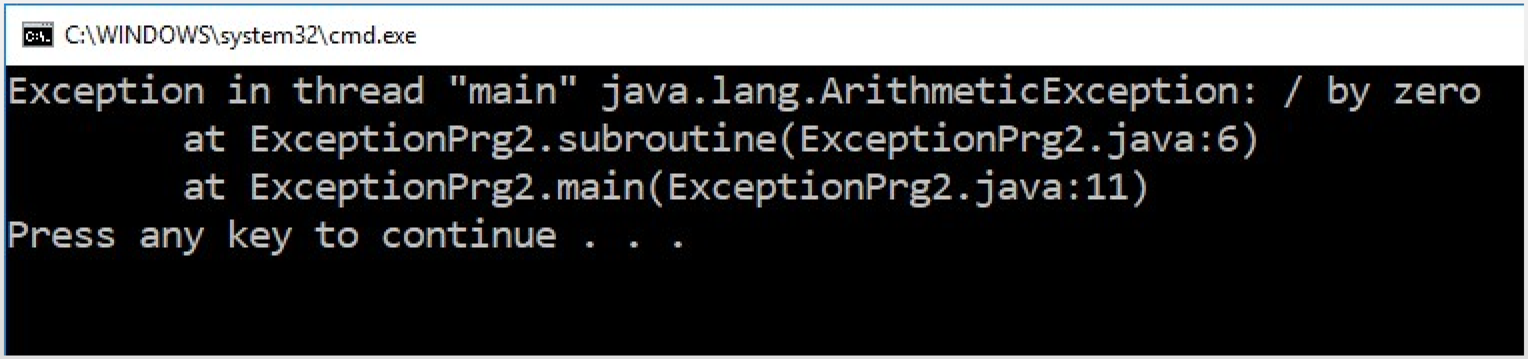


```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionPrg1.main(ExceptionPrg1.java:6)
Press any key to continue . . .
```

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

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

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



```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionPrg2.subroutine(ExceptionPrg2.java:6)
        at ExceptionPrg2.main(ExceptionPrg2.java:11)
Press any key to continue . . .
```

Using try and catch

```
 try
{


}
 catch (Exception   e)
{
}
```

Exception

ArithmeticException

ArrayIndexOutOfBoundsException

Using try and catch

```
class Exc2
 {        public static void main(String args[])
          {        int d, a;
                   try
                  {          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.");
      }
}
```

This program generates the following output:

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

```
class HandleError
{    public static void main(String args[])
    {       int a=0, b=0, c=0;
          Random r = new Random();
          for(int i=0; i<1000; i++)
          {         try
                     {         b = r.nextInt();
                              c = r.nextInt();
                              a = 12345 / (b/c);
                    }
                    catch (ArithmeticException e)
                    {         System.out.println("Division by zero.");
                              a = 0; // set a to zero and continue
                    }
                    System.out.println("a: " + a);
          } } }
```

Displaying a Description of an Exception

```
catch (ArithmeticException   e)
 {
        System.out.println("Exception: " + e);
        a = 0; // set a to zero and continue
}
```

When the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

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[5] = 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.");
   } }
```

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

```
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

- When we use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.

- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

- Thus, a subclass would never be reached if it came after its superclass.

- Further, in Java, unreachable code is an error.

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

- If we try to compile the program, we receive an error message stating that second catch statement is unreachable because the exception has already been caught.

- Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException.

- This means that the second catch statement will never execute.

- To fix the problem, reverse the order of the catch statements.

Nested try Statements

```
class NestTry
{       public static void main(String args[])
      {         try
                {           int a = args.length;
                           int b = 42 / a;
                           System.out.println("a = " + a);
                           try
                           {        if(a==1)
                                    a  = a/(a-a); // division by zero
                                    if(a==2)
                                   {           int c[] = { 1 };
                                              c[5] = 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);
             }
     }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero


C:\>java NestTry One Two
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException
```

```
/* Try statements can be implicitly nested via calls to methods. */
class MethNestTry
{       static void nesttry(int a)
      {         try
                {               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);
                }
     }  }
```

PresentationPoint

throw

It is possible to throw an exception explicitly, using the throw statement.

The general form of throw is shown here:

```
throw ThrowableInstance;
```

Nested try Statements

```
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);          }
        }
}
```

```
Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

- We can do this by including a throws clause in the method’s declaration.

- A throws clause lists the types of exceptions that a method might throw.

- Necessary for all exceptions, except those of type Error or RunTimeException.

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

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

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

```
// This program contains an error and will not compile.
class ThrowsDemo
{
          static void throwOne()
          {
                            System.out.println("Inside throwOne.");
                             throw new IllegalAccessException("demo");
           }
           public static void main(String args[])
           {
                           throwOne();
            }
}
```

```
// This is now correct.
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);     }
   }
}
```

Here is the output generated by running this example program:

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

finally

- **finally** executes a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.

- The **finally** block will execute whether or not an exception is thrown.

- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

- Used to perform certain house-keeping operations such as closing files and releasing system resources.

```
class prg2
{     public static void main(String args[])
    {
             try
            {           int a = 8/2;
            }
            catch(Exception e)
            {
                        System.out.println("error"+e);
            }
            finally
            {
                        System.out.println("finally");
            }
            System.out.println(“successful");
   }
}
```

```
class prg2
{    public static void main(String args[])
     {       try
             {
                    int a = 8/0;
             }
             catch(Exception e)
            {
                   System.out.println("error"+e);
            }
            finally
            {
                  System.out.println("finally");
           }
           System.out.println(“successful");
     }
}
```

```
class FinallyDemo1
 {      static void procA()
        {         try  {      System.out.println("inside procA");
                               throw new RuntimeException("demo");    }
                  finally
                 {         System.out.println("procA's finally");        }
       }
       public static void main(String args[])
      {      try    {       procA();    }

               catch (Exception e)
              {             System.out.println("Exception caught");
               }
      }  }
```

```
class FinallyDemo
 {
      static void procB()
      {       try
            {            System.out.println("inside procB");
                         return;
               }
             finally {  System.out.println("procB's finally");      }
        }
       public static void main(String args[])
        {
               procB();
        }
 }
```

```
class FinallyDemo
{           static void procC()
             {            try
                           {           System.out.println("inside procC");
                            }
                             finally
                            {          System.out.println("procC's finally");              }
             }
             public static void main(String args[])
             {
                          procC();
             }
 }
```

Here is the output generated by the preceding program:

```
inside procA
procA’s finally
Exception caught

inside procB
procB’s finally

inside procC
procC’s finally
```

Creating our Own Exception Subclasses

```
class MyException extends Exception
 {          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);
               }
     }
}
```

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

Create a user defined exception InvalidRegistrationNumber.

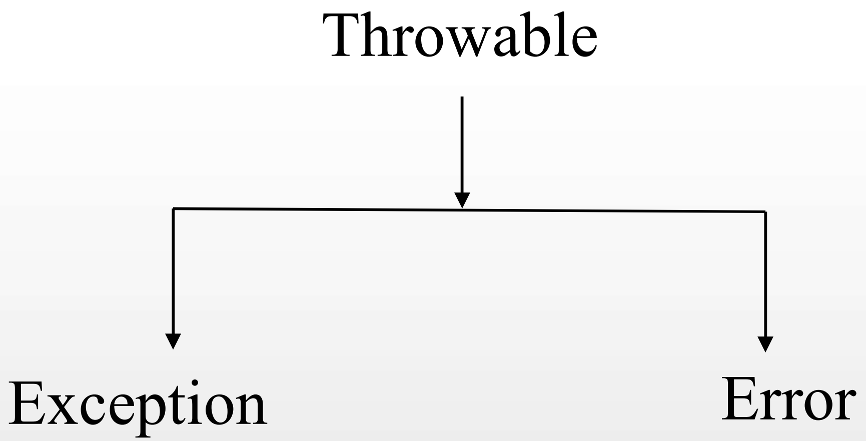
Write a java program to check whether the entered registration number is valid or not.

If not throw an exception of type InvalidRegistrationNumber.

ie 180953001 to 180953100

Ex: i/p regno =“180953050”
0/p valid registration number.

Ex: i/p regno =“180911050”
0/p invalid registration number : 180911050



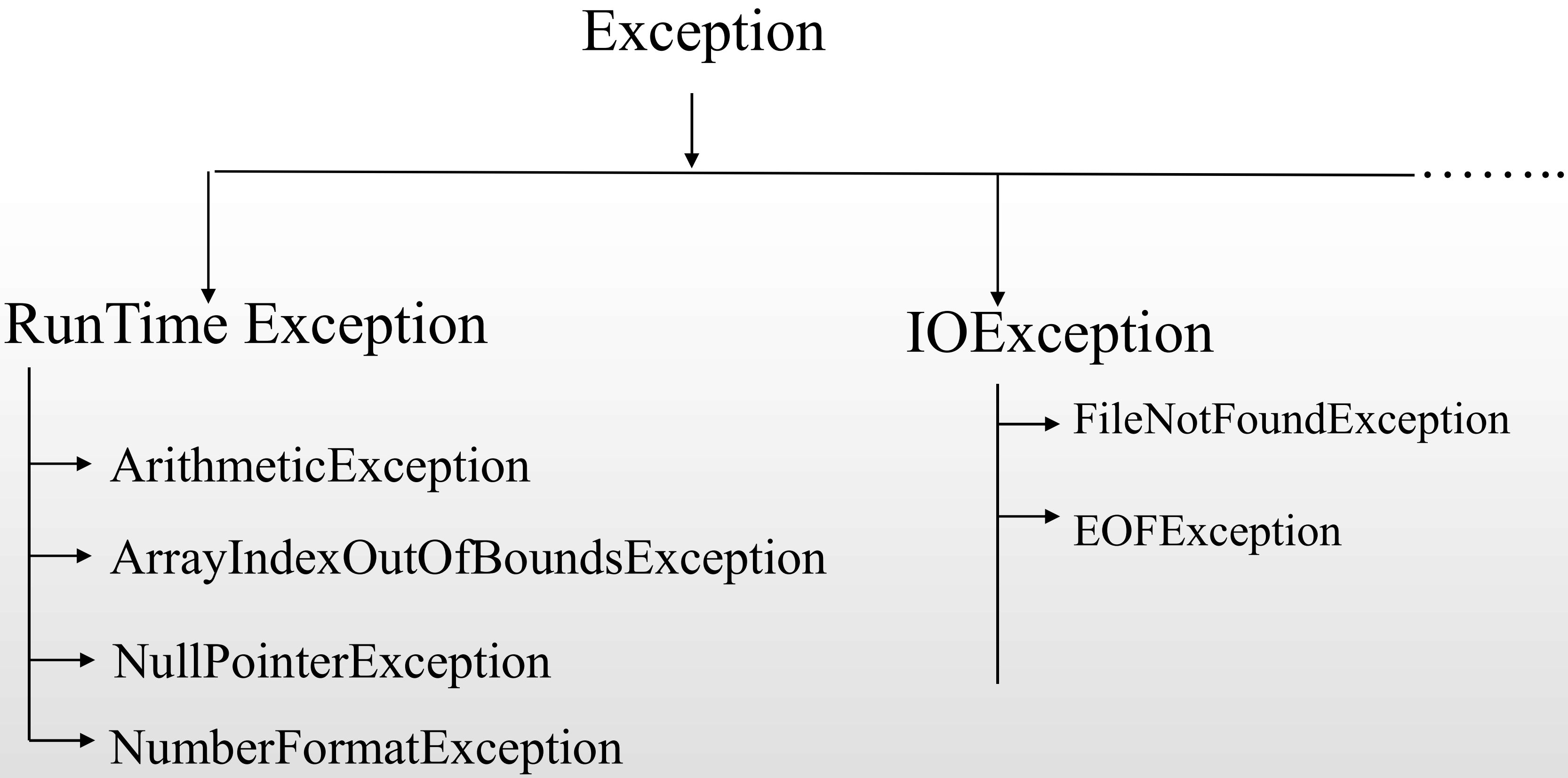
Throwable

Exception

Error

Exception : caused by programs.
are recoverable

Error : due to lack of resource
Virus attack –JVM crash
heap memory
Not recoverable



Exception

....

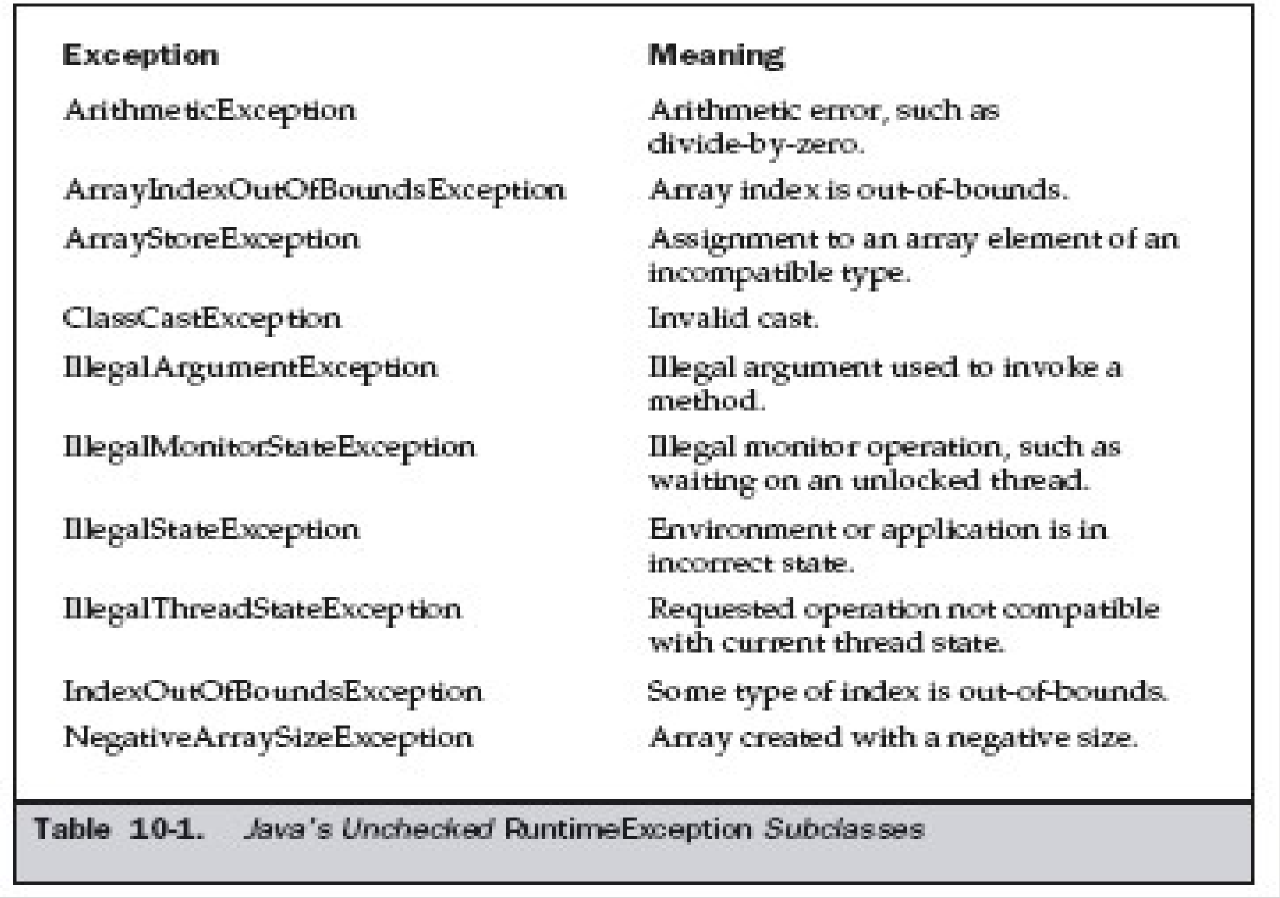
RunTime Exception

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NumberFormatException

IOException

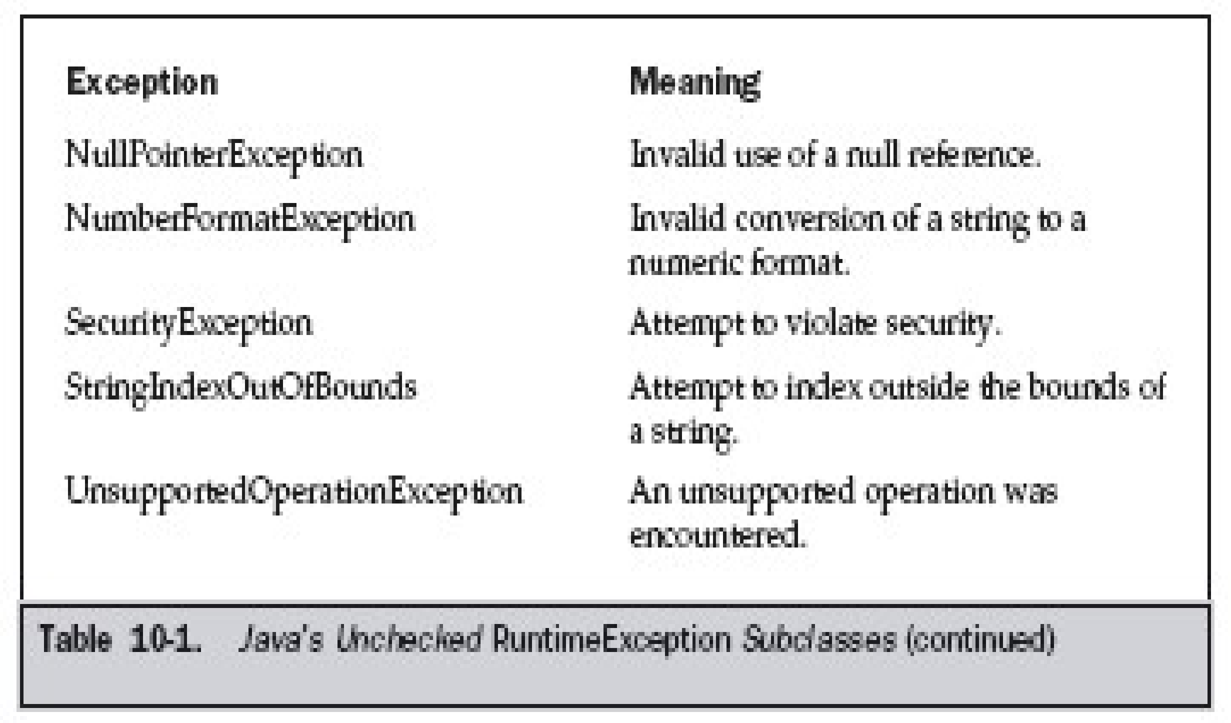
- FileNotFoundException
- EOFException

Java’s Built-in Exceptions



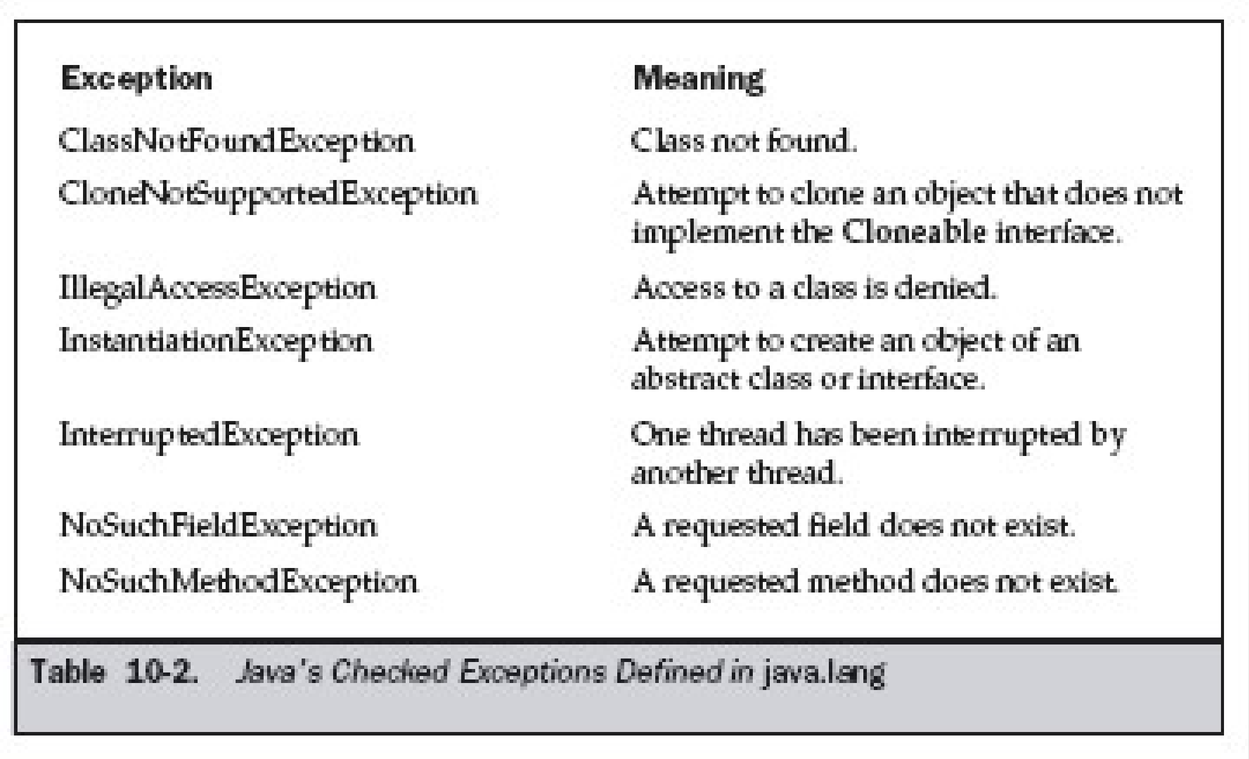
Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke 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.

Table 10-1. *Java's Unchecked* RuntimeException *Subclasses*



Exception	Meaning
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.

Table 10-1. *Java's Unchecked* RuntimeException *Subclasses* (continued)



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

Checked vs Unchecked Exceptions

Checked Exception:

- Exception that are checked by compiler whether programmer is handling or not such type of exceptions are called checked exception.

- If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

- In the case of checked exception compiler will check whether we are handling exception or not. If the programmer is not handling, then we will get compile time error.

Checked vs Unchecked Exceptions

Checked Exception:

- Consider the program to read and prints first three lines of it.
- The program doesn't compile, because FileReader() throws a checked exception *FileNotFoundException*.
- It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

```
import java.io.*;
class prg
{
    public static void main(String[] args)
   {
        FileReader file = new FileReader("C:\\test\\a.txt");
       BufferedReader fileInput = new BufferedReader(file);

       // Print first 3 lines.
       for (int counter = 0; counter < 3; counter++)
       System.out.println(fileInput.readLine());

       fileInput.close();
  }
}
```

- To fix the program:
  - we either need to specify list of exceptions using throws, or
  - need to use try-catch block.

```
import java.io.*;
 class prg
 {
        public static void main(String[] args)
        {
             try
             {
                       FileReader file = new FileReader("C:\\test\\a.txt");
                       BufferedReader fileInput = new BufferedReader(file);

                       // Print first 3 lines.
                      for (int counter = 0; counter < 3; counter++)
                     System.out.println(fileInput.readLine());
                     fileInput.close();
              }
               catch(Exception e) {  }
        }
 }
```

```
import java.io.*;
 class prg
 {
        public static void main(String[] args) throws IOException
        {
              FileReader file = new FileReader("C:\\test\\a.txt");
              BufferedReader fileInput = new BufferedReader(file);

              // Print first 3 lines.
              for (int counter = 0; counter < 3; counter++)
               System.out.println(fileInput.readLine());

               fileInput.close();
        }
 }
```

Unchecked Exceptions

- Exceptions that are not checked at compiled time.
- In C++, all exceptions are unchecked.
- It is up to the programmers to be civilized, and specify or catch the exceptions.

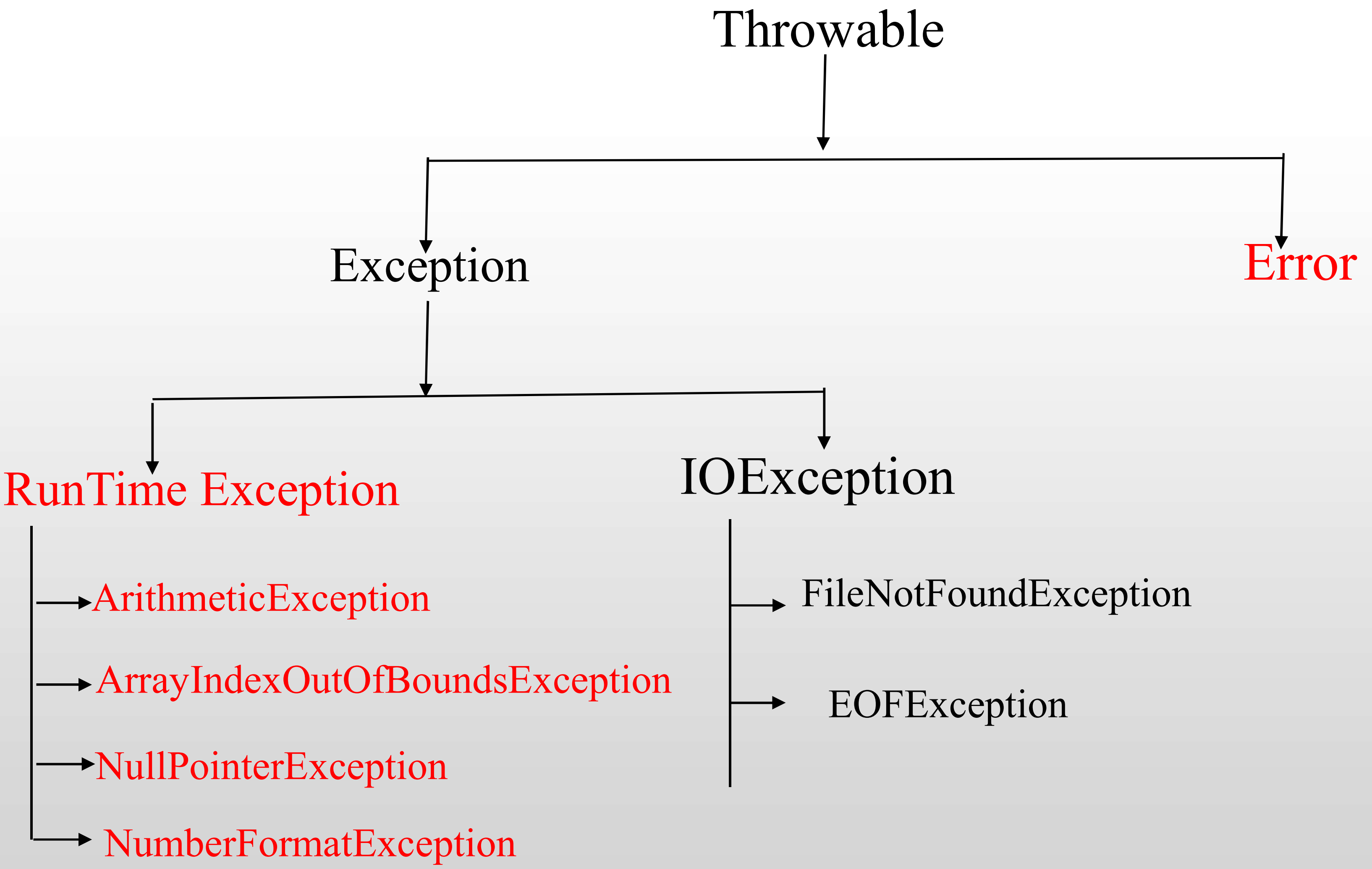
In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

- Below program compiles fine, but throws *ArithmeticException* when run.

- Compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class program
{
   public static void main(String args[])
    {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

Unchecked Exceptions



Throwable

Exception

Error

RunTime Exception

IOException

ArithmeticException

ArrayIndexOutOfBoundsException

NullPointerException

NumberFormatException

FileNotFoundException

EOFException

Unchecked Exceptions

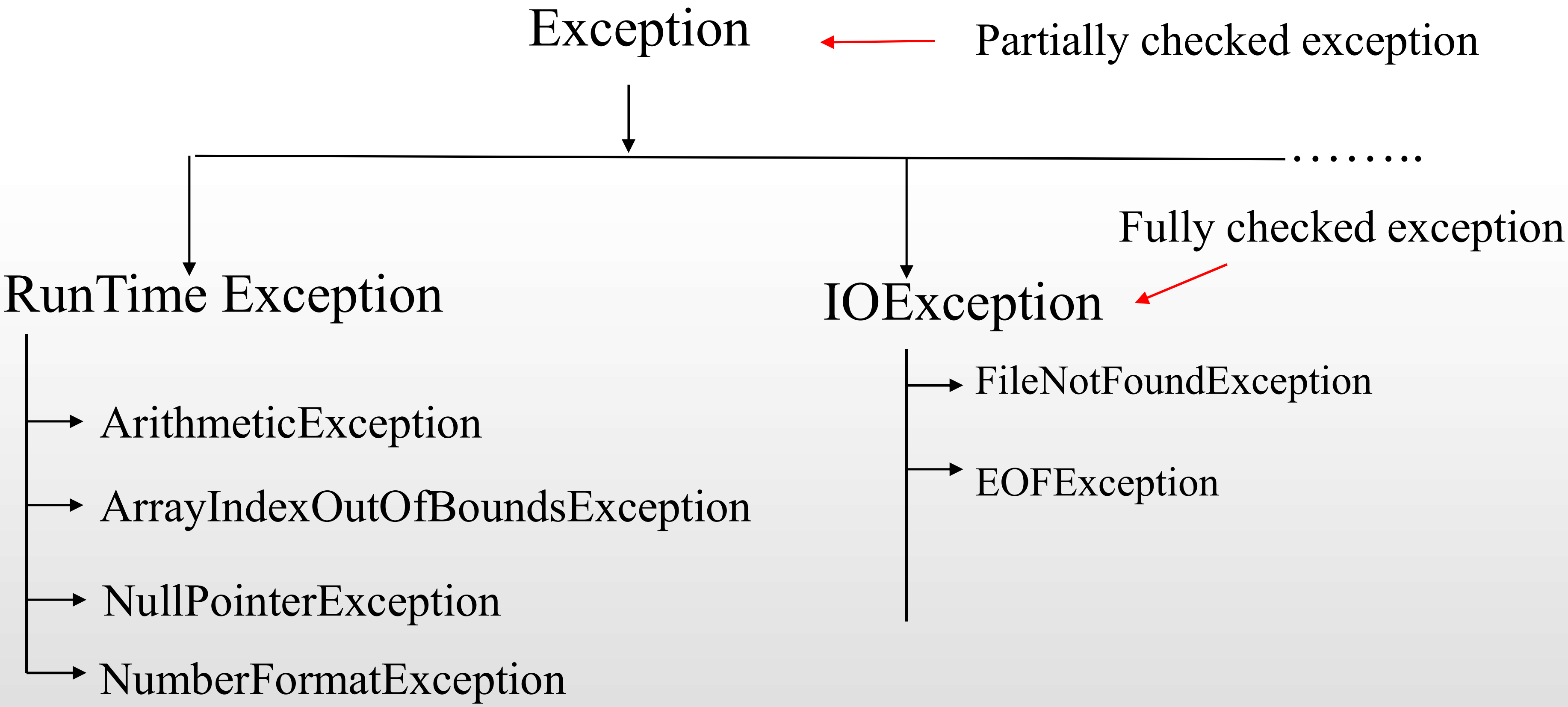
- Runtime exception and its child class, Error and its child classes are unchecked exception.

- Except this remaining are checked exceptions.

Fully Checked vs partially Checked Exceptions

- A checked exception is said to be fully checked if and only if all its child classes also checked.
- Ex: IOException

- A checked exception is said to be partially checked if and only if some of its child classes are unchecked.
- Ex : Exception



Exception

Partially checked exception

……..

RunTime Exception

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NumberFormatException

IOException

Fully checked exception

- FileNotFoundException
- EOFException