8 EXCEPTIONS

Objectives

- Define exceptions
- Use `try`, `catch` and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions

Exceptions

- The exception class defines mild error conditions that your program encounters.
- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class file you are interested in loading is missing
- An error class defines serious error conditions


```
1   public class HelloWorld {
2     public static void main (String args[]) {
3     int i = 0;
4
5     String greetings [] = {
6         "Hello world!",
7         "No, I mean it!",
8         "HELLO WORLD!!"
9       };
10
11    while (i < 4) {
12      System.out.println (greetings[i]);
13      i++;
14    }
15  }
16  }
```



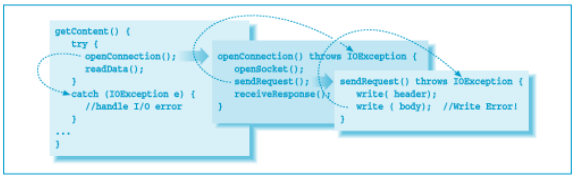
```
Hello world!
No, I mean it!
HELLO WORLD!!
java.lang.ArrayIndexOutOfBoundsException
        at HelloWorld.main(HelloWorld.java:12)
Exception in thread "main" Process Exit...
```


The try and catch Statements


```
try {
    // code that might throw a particular exception
}
catch (MyExceptionType e) {
    // code to execute if a MyExceptionType exception is thrown
}
catch (Exception e) {
    // code to execute if a general Exception exception is thrown
}
```

Call Stack Mechanism

- If an exception is not handled in the current try/catch block, it's thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.



Call Stack Mechanism

- Consider a case where a method calls another method named openConnection(), and this, in turn calls another method named sendRequest(). If an exception occurs in sendRequest(), it is thrown back to openConnection(), where a check is made to see if there is a catch for that type of exception. if no catch exists in openConnection(), the next method in the call stack, main(), is checked. if the exception is not handled by the last method on the call stack, then a runtime error occurs and the program stops executing.

The `finally` Statement

What if we have some clean up to do before we exit our method from one of the catch clauses? To avoid duplicating the code in each catch branch and to make the cleanup more explicit, Java supplies the finally clause. A finally clause can be added after a try and any associated catch clauses. Any statements in the body of the finally clause are guaranteed to be executed, no matter why control leaves the try body:

next slide


```
try {
   // Do something here
}
catch ( FileNotFoundException e ){
   ...
}
catch ( IOException e ) {
   ...
}
catch ( Exception e ) {
   ...
}
finally {
   // Cleanup here
}
```


If the statements in the try execute cleanly, or even if we perform a return, break, or continue, the statements in the finally clause are executed. To perform cleanup operations, we can even use try and finally without any catch clauses:

```
try {
 // Do something here
 return;
}
finally {
 System.out.println("Do not ignore me!");
}
```



```
public class HelloWorldRevisited {
   public static void main (String args[]) {
                int i = 0;
                String greetings [] = {
                        "Hello world!",
                        "No, I mean it!",
                        "HELLO WORLD!!" };
      while (i < 4) {
            try {
               System.out.println (i+" "+greetings[i]);
            } catch (ArrayIndexOutOfBoundsException e){
              System.out.println("Re-setting Index Value");
              break;
            } finally {
              System.out.println("This is always printed");
            }
          i++;
      } // end while()
   } // end main()
}
```

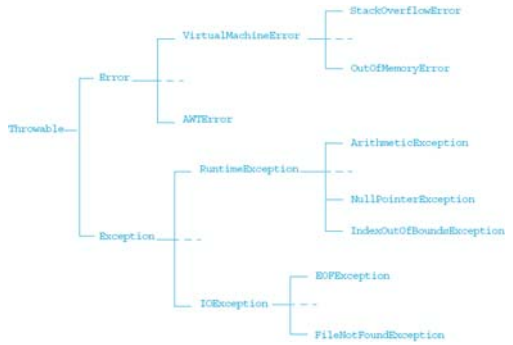
Exception Categories

Exceptions 8

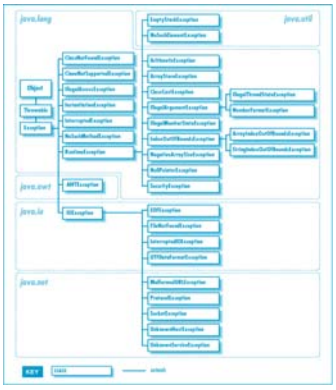
►`Error` indicates a severe problem from which recovery is difficult, if not impossible. An example is running out of memory. A program is not expected to handle such conditions.

►`RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly.An `ArrayOutOfBoundsException` exception, for example, should never be thrown if the array indices do not extend past the array bounds. This would also apply, for example, to referencing a null object variable.

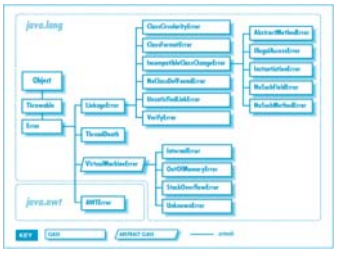
►Other exceptions indicate a difficulty at runtime that is usually caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions. Because these usually occur as a result of user error, you are encouraged to handle them.



Exceptions 8



Exceptions 8



Exceptions 8

Common Exceptions

►`ArithmeticException`

```
int i = 12 / 0 ;
```

►`NullPointerException`

```
Date d = null ;
System.out.println(d.toString()) ;
```

►`NegativeArraySizeException`

►`ArrayIndexOutOfBoundsException`

►`SecurityException`

- Access a local file
- Open a socket to the host that is not the same host that served the applet
- Execute another program in runtime environment

Exceptions 8

The Handler or Declare Rule

►Handle exceptions by using the `try-catch-finally` block

►Declare that the code causes an exception by using the `throws` clause

►A method may declare that it `throws` more than one exception

►You do not need to handle or declare runtime exceptions or errors.

Exceptions 8

Method Overriding and Exceptions

The overriding method:

- Can throw exceptions that are subclasses of the exceptions being thrown by the overridden method

For example, if the superclass method throws an `IOException`, then the overriding method can throw an `IOException`, a `FileNotFoundException` (a subclass of `IOException`), but not an Exception (the superclass of `IOException`)


```
public class TestA {
   public void methodA() throws RuntimeException {
      // do some number crunching
   }
}
public class TestB1 extends TestA {
   public void methodA() throws ArithmeticException {
      // do some number crunching
   }
}
public class TestB2 extends TestA {
   public void methodA() throws Exception {
      // do some number crunching
   }
}
```



```
public class TestMultiA {
   public void methodA()
                  throws IOException, RuntimeException {
      // do some IO stuff
   }
}
public class TestMultiB1 extends TestMultiA {
   public void methodA()
    throws FileNotFoundException, UTFDataFormatException,
           ArithmeticException {
      // do some number crunching
   }
}
```



```
import java.io.* ;
import java.sql.* ;

public class TestMultiB2 extends TestMultiA {
   public void methodA()
    throws FileNotFoundException, UTFDataFormatException,
           ArithmeticException, SQLException {
      // do some IO, number crunching, and SQL stuff
   }
}
```



```
public class ServerTimedOutException extends Exception {
      private int port;
      public ServerTimedOutException (String reason,int port){
         super(reason);
         this.port = port;
      }
      public int getPort() {
          return port;
      }
}
```

To throw an exception of the above type, write

```
throw new ServerTimedOutException("Could not connect",60) ;
```



```
public void connectMe(String serverName)
                        throws ServerTimedOutException  {
  int success;
  int portToConnect = 80 ;
  success = open(serverName,portToConnect) ;
  if( success == -1 )
    throw new ServerTimedOutException("Could not connect",
                                      portToConnect);
}
public void findServer() {
      try {
         connectMe(defaultServer) ;
      }  catch (ServerTimedOutException e) {
         System.out.println("Server timed out, trying alternative") ;
         try{
            connectMe( alternativeServer) ;
         } catch (ServerTimedOutException e1) {
            System.out.println("Error : " + e1.getMessage() +
                               "connecting to port" + e1.getPort()) ;
         }
      }
}
```

```
1  // Fig. 14.10: UsingExceptions.java
2  // Demonstrating the getMessage and printStackTrace
3  // methods inherited into all exception classes.
4  public class UsingExceptions {
5     public static void main( String args[] )
6     {
7        try {
8           method1();
9        }
10       catch ( Exception e ) {
11          System.err.println( e.getMessage() + "\n" );
12
13          e.printStackTrace();
14       }
15    }
16    public static void method1() throws Exception
17    {
18       method2();
19    }
20
21    public static void method2() throws E
22    {
23       method3();
24    }
25    public static void method3() throws E
26    {
27       throw new Exception( "Exception thrown in method3" );
28    }
29 }
```

Call **method1**, which calls **method2**, which calls **method3**, which throws an exception.

getMessage prints the **String** the **Exception** was initialized with.

printStackTrace prints the methods in this order:

method3
method2
method1
main

(order they were called when exception occurred)


```
Exception thrown in method3
java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:28)
        at UsingExceptions.method2(UsingExceptions.java:23)
        at UsingExceptions.method1(UsingExceptions.java:18)
        at UsingExceptions.main(UsingExceptions.java:8)
```