Error Handling in Java

- Run time errors
  - It is difficult to recover gracefully from run-time errors that occur in the middle of a program.
  - At the point where the problem occurs, there often isn’t enough information in that context (the method) to resolve the problem.
  - So in Java, that method hands off the problem out to a higher context (a calling method) where someone is qualified to make the proper decision.
  - There is no need to check for errors at multiple places (after each call to access a file, for instance). The code to handle a given error can be put in a single location in the code (the exception handler).
  - If the error can be resolved in the immediate context where it occurs, it is NOT called an exception.

Exception semantics - 1

- When an error occurs inside a method, the method creates an exception object.
  - could be in a library method or a user-defined method

- The exception object contains information about the error, including:
  - the type of the exception and
  - the state of the program when the error occurred (the call stack)

- Creating an exception and reporting it to the runtime system is called throwing an exception.

Exception semantics - 2

- When a method throws an exception,
  - the current path of execution is interrupted, and
  - the runtime system attempts to find an appropriate place to continue executing the program.

- The runtime system searches the call stack for an appropriate exception handler
  - the call stack: the list of methods that have been called and are waiting for the current method to return.
  - A calls B that calls C that calls D: The call stack contains A, B, C and D with D on the top.
Exception semantics - 3

• The runtime system is looking for a previous method call that is embedded in a block that has an exception handler associated with it.
  ✦ It starts at the top of the call stack and goes down (in reverse order in which the methods were called)

• The runtime system is searching for an appropriate exception handler
  ✦ An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
  ✦ "matching" is the same as is used for function calls, a thrown exception matches any superclass of its type.

Exception semantics - 4

• The first exception handler encountered that matches the exception is said to catch the exception.

• If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system terminates the program.
  ✦ And usually the exception is output to the screen

Exception simple example

```java
// 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]);
            System.out.println("After element access");
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
    }
}
```

• What part of the code throws the exception?
• Output

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

Exception syntax: how to throw an exception

• To throw an exception, use the keyword throw.
• To create an exception, use the appropriate constructor.

```java
if (t==null)
    throw new NullPointerException();
```

• This will cause the enclosing method to be exited.
  ✦ If the error can be handled inside the method, there is generally no need to throw an exception.
• Exception classes can be found in the API website: see java.lang.Exception
Exception syntax: how to catch an exception

• To catch an exception, use the try-catch block.
• Surround the code that might generate an exception in the try
• Make an exception handler (a catch clause) for every exception type you want to catch.

```java
try {
    // Code that calls methods that might throw exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
} catch(Type3 id3) {
    // Handle exceptions of Type3
}
// etc...
```

What can you do with an exception?

• printStackTrace().
  ✦ This produces information about the sequence of methods that were called to get to the point where the exception happened.
  ✦ By default, the information goes to the standard error stream
• getMessage()
  ✦ like toString() for exception classes.
  ✦ a printable description of what went wrong

Exception syntax: how to catch an exception

• Each catch clause is like a little method that takes one argument of a particular type.
• The parameter (id1, id2, and so on) can be used inside the handler, just like a method argument.
• If the handler catches an exception, its catch block is executed, and the flow of control proceeds to the next statement after (outside) the try/catch.
  ✦ only the first matching catch clause is executed.

The exception specification: being civil

• In Java, you are (strongly!) encouraged to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method
  ✦ Then the caller can know exactly what catch clauses to write to catch all potential exceptions.
• The exception specification states which exceptions are thrown by a method.

```java
void f() throws TooBig, TooSmall, DivZero {
    //...
```
  ✦ Also use the @throws tag in the javadoc comment to describe these in more detail (when/why each one is thrown).
• Catch or specify requirement: If the method throws exceptions, it must handle them or specify them in the signature.
  ✦ Otherwise it’s a compiler error.
Catch or Specify: example

```java
//Note: This class won't compile by design!
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {
    private Vector<Integer> vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(new Integer(i));
        }
    }

    public void writeList()  {
        PrintWriter out = new PrintWriter(
            new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + vector.elementAt(i));
        }
        out.close();
    }
}
```

Catch or Specify: solution 1

```java
//Note: This class won't compile by design!
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {
    private Vector<Integer> vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(new Integer(i));
        }
    }

    public void writeList() throws IOException {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + vector.elementAt(i));
        }
        out.close();
    }
}
```

Catch or Specify: solution 2

```java
public void writeList() {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + vector.elementAt(i));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    if (out != null)
        out.close();
}
```

The finally block

- The finally block is an additional block you can add to the try catch.
- The finally block ALWAYS executes when the try block exits
  ✦Whether it exited through an exception handler or just normal termination.
Runtime Exceptions: an exception to the rule

- RuntimeExceptions are a special (sub)class of Exceptions.
  ✦ They are thrown automatically by Java in certain contexts
  ✦ This is part of the standard run-time checking that Java performs for you
- These exceptions are “unchecked exceptions”, they do not need to conform to the “Catch or specify rule.
  ✦ Methods are not required to indicate if they might throw one
  ✦ Methods are not required to try to catch them
- What if they are not caught?
  ✦ If a RuntimeException gets all the way out to main() without being caught, printStackTrace() is called for that exception as the program exits

Why are RuntimeExceptions not required to be caught?
- They are generally caused by programmer errors (bugs)
  [These exceptions are very useful during testing]
- There may be no graceful way to recover from these bugs

What are some examples of RuntimeExceptions?
- NullPointerException
- ClassCastException
- See the API website for more

You can create your own exceptions

- If one of the Java Exceptions is not appropriate for your program, you can create your own Exception classes
- The class must inherit from an existing exception class
  ✦ preferably one that is close in meaning to your new exception (although this is not always possible).

```java
class SimpleException extends Exception {} 
public class SimpleExceptionDemo { 
    public void f() throws SimpleException { 
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException(); 
    } 
    public static void main(String[] args) { 
        SimpleExceptionDemo sed = new SimpleExceptionDemo();
        try { 
            sed.f(); 
        } catch(SimpleException e) { 
            System.err.println("Caught it!"); 
        } 
    } 
} 
```