

## Lecture 16

### Coursework 2

- ❑ Troubleshooting sessions begin today at 5pm in the IBM lab
- ❑ Next week I'll cover the details about the marking process

### Feedback

- ❑ Please fill in the feedback forms and hand them in at the end of the lecture

## Exception Handling

You may have come across some run-time errors while doing your coursework

```
class RobotData
{
    ...
    private int[] xCoordinates;

    RobotData()
    {
        // xCoordinates = new int[maxSquares];
    }
    ...
}
```

Exception in thread "Systematic" java.lang.NullPointerException  
at recordSquare.<init>(Compiled Code)  
at Systematic.controlRobot(Compiled Code)

## Exception Handling

As things stand, Java handles its own errors

But Java also allows the programmer to handle run-time errors

- ❑ You can make your run-time errors meaningful
- ❑ You can trace errors through your program

In this lecture we will look at Java's exception (error) handling system

## Exceptions and Exception Types

Run-time errors occur for various reasons

- ❑ User enters invalid input
- ❑ Program attempts to access an out-of-bounds array element
- ❑ Program tries to access a non-existent object

When these happen a Java *exception* object is generated

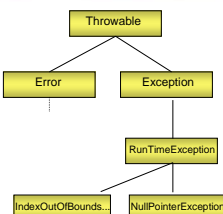
A Java exception is an instance of a class derived from *Throwable*

It is possible to create your own exception classes by extending the class (or sub-class of) *Throwable*

## Exception Types

### Sub-classes of *Throwable*

- ❑ **Error**
  - Describes internal System errors (JVM errors)
- ❑ **Exception**
  - Describes errors caused by your program and external circumstances
  - These are the ones the programmer can 'handle'



There are also sub-classes of these which we will not worry about

## Understanding Exception Handling

Java's exception handling is based on three operations

- i. **claiming an exception**
- ii. **throwing an exception**
- iii. **catching an exception**

## i. Claiming Exceptions

An executing statement either belongs to *main()* or some method called from *main...*

or from a method called from a method called from *main* etc.

The *main* method is invoked by the *System*

In general, every method must state the type of exceptions it can encounter

This is called *claiming an exception*; which simply tells the compiler what can go wrong

## ii. Throwing Exceptions

When a statement causes an error, the method containing the statement creates an *exception* object and passes it to the system

The exception object contains information about the error

- Error type; state of program when error occurred etc.

This process is called *throwing an exception*

## iii. Catching Exceptions

After a method throws an exception, the Java RTS begins the process of finding the code to handle the error

The code that handles the error is called the *exception handler*

This is found by searching back through a chain of methods, starting from the current method

The handler attempts to match the type of the exception thrown against exception handling code

This is called *catching an exception*

## i. Claiming Exceptions (details)

To claim an exception in a method is to say what can go wrong during execution

Use the *throws* keyword in the method declaration

```
public int myMethod () throws Exception
```

The method can throw any number of different types of exception

- Allowing any number of different error types to be detected

```
methodDeclaration throws ExceptionType1, ExceptionType2
```

The default (all along) is

```
methodDeclaration throws RuntimeException
```

- which is a sub-class of Exception

## ii. Throwing Exceptions (details)

In the method which claimed the exception, you can throw an exception of that type

To throw an exception, use the *throw* keyword

```
public static int divides (int x, int y) throws Exception
{
    if (y ==0) throw new Exception ("can't divide by zero");
    return (x / y);
}
```

The class in which this method is defined does not change

## iii. Catching Exceptions (details)

When calling a method which explicitly claims an exception (eg. *divides*), you must use the *try-catch* block to wrap the calling statement

```
try
{
    result = divides(10,0); // Eg. statement that may throw exception
    System.out.println ("Result = " + result);
}
catch (Exception e) // (One or more) exception types
{
    System.out.println("O'dear" + e);
}
```

If no exceptions arise during the *try* clause, the *catch* clause is skipped (like an *if-else*)

## Catching Exceptions cont...

If one of the statements inside the try block throws an exception, Java skips the remaining statements and searches for an exception handler

- The line `System.out.println("Result = " + result)`
- Will be missed out in the event of an exception in divides

If the exception type matches the one listed in the catch clause, the code for the catch clause is executed

- The statement `System.out.println("O'dear" + e)` is executed

If the exception type does not match the one in the catch clause(s), Java exits this method and passes the exception to the method which invoked this method...

This *chaining* continues until a handler is found

```
main(...)
{
  try { ... invoke method1();
        System.out.println("Cool, it works!");
      }
  catch (Exception1 e1) {
    System.out.println(e1);
  }
}

method1()
{
  try { ... invoke method2();
        System.out.println("Rock-n-roll!");
      }
  catch (Exception2 e2) {
    System.out.println(e2);
  }
}

method2()
{
  try { ... invoke method3();
        System.out.println("Wicked, sorted.");
      }
  catch (Exception3 e3) {
    System.out.println(e3);
  }
}

method3() throws Exception1, Exception2, Exception3
{
  Exception1 error1 = new
    Exception1("Nightmare! whole thing is knackered");
  Exception2 error2 = new
    Exception2("My robot has died.");
  Exception3 error3 = new
    Exception3("I am sure it worked a minute ago..!");
  ... // some code which may throw error 1, 2 or 3
}
```

## Scenarios

No errors

method3 throws error3

method3 throws error2

method3 throws error1

## Throwable Objects

Contain a number of useful instance methods

- `public String getMessage()`
  - returns the detailed message of the Throwable object
- `public String toString()`
  - returns the short message of the Throwable object
- `public void printStackTrace()`
  - prints the Throwable object and its trace information

e.g. `e3.printStackTrace()`

## Creating Exception Classes

You can create your own exception classes derived from the Exception class – this example counts the number of errors

```
class MyException extends Exception
{
  private int count = 0;

  MyException(String s)
  {
    super("Your error is..." + s);
    count++;
  }

  public String toString()
  {
    return ("Error number: " + count + super.getMessage());
  }
}
```

## Re-throwing Exceptions

When an exception occurs in a method, the method exits immediately if the exception is not caught by the method

If you want to perform some activities before exiting, you can catch the exception in the method and re-throw it to the real handler

```
try
{ // statements; }
catch (TheException e)
{ // perform operations before exit;
  throw e;
}
```

Shutdown system safely

## The *finally* Clause

You might want some code to be executed regardless of whether exceptions occur

To do this use the *finally* clause

```
try
{ // statements; }
catch (TheException e)
{
    // handler code;
}
finally
{
    // get the last word;
}
```

This is always  
executed