Lecture 1

We discuss problems that can arise at run-time, both preventable and unpreventable. The latter apply particularly to I/O.

We discuss *defensive programming*: anticipating, detecting and responding to problems.

We consider three ways of responding to problems: default values, special values and exceptions.

We discuss Java exception-handling in depth.

Problems at run-time

- Some run-time errors are internal to your program, so you can get rid of them by debugging your code
  - null pointer exception; index out of bounds exception; division by zero exception
- But other run-time problems cannot be solved this way
  - Suppose your program is communicating with an external resource (user, file, database, Internet)
  - Suppose clients of your program (i.e. other programs that invoke your methods) are not trustworthy (e.g. if your program is placed into a library) and may supply you with problematic data

Defensive Programming

- Your code must take responsibility for protecting itself from receiving bad data or other exceptional run-time situations (e.g. network down, disk full, etc.)
  - Your code must *detect* these problems
  - Your code must *respond* to these problems sensibly
- But
  - Defensiveness brings 'code bloat'
  - Defensiveness brings 'code pollution'

Responding by returning a default

- Suppose your users type something into a `JTextField` called `tf` and you're expecting them to type month names
- What if they don't? You could substitute a default:

```
public String getMonthFromTF()
{  String str = tf.getText();
   if (! isValidMonthName(str))
   {  return "Jan";
   }
   else
   { return str;
   }
}
```

- But the client of your method now won't know whether the user typed "Jan" or typed rubbish

Responding by returning a special value

- Or you could return a special value to signal to the client code that a problem has occurred:

```
public String getMonthFromTF()
{  String str = tf.getText();
   if (! isValidMonthName(str))
   {  return null;
   }
   else
   { return str;
   }
}
```

- But special values are not always available
- And will the programmer who writes the client remember to check for the special value and take special action?

Java exception-handling

- A systematic way of representing, transmitting, capturing and handling well-defined abnormal situations
- When the method detects the abnormal situation, it creates an object, known as an exception
- It throws the exception, which may be caught and handled by an appropriate exception-handler, e.g. in the client code
- This is an abnormal return of the flow of control from the method to the client
- It cannot, in general, be ignored by the client
- But handling is separated from detecting/throwing, so less code pollution

Classes of exceptions

```
Exception
   |
   +----CloneNotSupportedException
   |
   +----IOException
   |       |
   |       +----EOFException
   |       |
   |       +----FileNotFoundException
   |       |
   |       +----MalformedURLException
   |
   +----InterruptedException
   |
   +----RuntimeException (unchecked)
           |
           +----IllegalArgumentException
           |       |
           |       +----NumberFormatException
           |
           +----IndexOutOfBoundsException
           |       |
           |       +----ArrayIndexOutOfBoundsException
           |       |
           |       +----StringIndexOutOfBoundsException
           |
           +----NullPointerException
```


Example: `Integer.parseInt`

- Suppose your users type into `JTextField tf` and you're expecting them to type integers
- The programmer who wrote `parseInt` arranged for it to throw a `NumberFormatException` if the `String` cannot be converted to an `int`
- The client code can catch & handle the exception:

```
try
{  ...
   String str = tf.getText().trim();
   int num = Integer.parseInt(str);
   ...
}
catch (NumberFormatException nfe)
{  System.out.println("Problem!");
   ...
}
```


A `try` block and its `catch` blocks

```
try
{  some statements
   some more statements
   yet more statements
}
catch (FileNotFoundException fnfe)
{  handle the exception
}
catch (EOFException eofe)
{  handle the exception
}
```

- Try to execute the `try` block. If no exceptions produced, the `catch` blocks are ignored
- If any statement in the `try` block throws an exception, the rest of the `try` block is not executed. The appropriate `catch` block is executed

`finally` blocks

```
try
{  some statements
}
catch (NumberFormatException nfe)
{  handle exception
}
finally
{  further statements
}
```

- Try to execute the `try` block
- Execute an appropriate `catch` block if an exception is thrown
- *Either way,* afterwards execute the `finally` block

Which `catch` block is appropriate?

- Java looks thro' `catch` clauses sequentially and executes the first that matches (is equal to or a superclass of)

```
catch (EOFException eofe)
{  // handle the exception
}
catch (IOException ioe)
{  // handle the exception
}
```

- So why would this be silly?

```
catch (IOException ioe)
{  // handle the exception
}
catch (EOFException eofe)
{  // handle the exception
}
```


What if no `catch` block matches?

- If none match, the exception object is passed up the *call chain*
- If no method in the call chain catches the exception, the Java run-time system catches it, displays a message & terminates the program (except for applets and applications with GUIs)