Exceptions

The Standard Way for Object to Return Work

- Object 1 gives object 2 some work.
  - It calls a method of object 2.
  - It waits until object 2 has finished.
- Object 2 may well give some of the work to object 3, and so on, down to object 10.
- We have a long chain of objects waiting for the one at the end of the chain to finish.

Shorting Out The Chain

- An ***exception*** is a way of cutting out the middle men.
- Object 10 can get back to object 1 directly.
  - Bypassing all the middle men.
- Used in emergencies
  - Object 10 had detected a serious problem.
  - Trying `readDouble` when the input contains letters, for example.

Also Convenient

- An exception is also a convenient way of returning from a method in some cases.
- When something unexpected happens.
  - Trying to read from a file and reaching the end of file.
  - An end of file exception is thrown.
- Do not overuse!

Mental Model of an Exception

- Object 10 writes down a special message on a piece of paper
  - folds it into the shape of a plane
  - and throws it towards object 9.
- Object 9 can catch it or let it sail over his head.
- If this happens, the plain flies towards object 8.
  - It is a powered and guided paper aeroplane.

Mental Model (2)

- This process continues until an object catches it or it reaches the `main` object.
- If `main` does not catch it then we have a runtime error
  - Unhandled exception.

The Special Message

- The special message is also an object
  - An exception object.
- It will contain information on why the exception object was thrown.
- It can be used in the `catch` block by the object that caught it.

Why Worry About Exceptions

- If we use library code that might throw an exception, we cannot avoid exceptions.
- We cannot stop an exception from being thrown.
- Part of our code must be prepared to catch the exception.
- The compiler will not let us avoid this.

Simple Exception Catching

- The FormatIOX package is a variant of FormatIO that uses exceptions.
- Example, catching the `EndOfFileException` thrown by `FileIn` and `StringIn`.
- Any method that might throw this exception must be inside a `try` block.
- The code to deal with the exception if it thrown must be inside a `catch` block.

Exception Example

```
try
{
   line = fin.readLine();
}

catch (EndOfFileException x)
{
   System.err.println(“Unexpected end of file”);
}
```

Example Explained

- The `try` block is the compound statement after the word try.
- The `readLine` method might throw an `EndOfFileException` exception.
  - Therefore it must be inside the `try` block.
- The code to handle the exception is inside the `catch(EndOfFileException x)` block.
  - In this case it ignores the exception object `x`.

Two Pathways In The Code

- If no exception is thrown:
  - Our code executes all the instructions in the `try` block.
  - The `catch` block is ignored.
- If an exception is thrown:
  - Our code only does those bits of the `try` block that happen before the exception.
  - It then jumps straight to the `catch` block.

Catching More than One Exception

- A `NumberFormatException`, number format exception might also be thrown.
- Trying to read a number when the input contains `“Fred”`, for example.
- We can have several `catch` blocks, one after another.

Catching Two Exceptions

```
try
{
   line = fin.readLine();
}
catch (EndOfFileException x)
{
   System.err.println(“Unexpected end of file”);
}
catch (NumberFormatException x)
{
   System.err.println(“Number Format Error”);
}
```

Scope of a Variable

Scope is Visibility

- The scope of a variable is the part of the program where it can be used.
- Its scope is the compound statement where it is defined.
  - A compound statement is also called a ***block***.
- The scope includes all inner blocks.

Scope Example

```
String line = "";

FileIn fin = new FileIn("..\\radius.txt");
try
{
    line = fin.readLine();
}
catch(EndOfFileException x){}

StringIn sin = new StringIn(line);
```

Scope Example Explained

- The variable `line` is declared in the main block.
- It can be used throughout the code shown.
  - Including the inner `try` block.
- `fin` is also defined in the main block.
- It can be used anywhere after it is defined.

Example That Does Not Work

- If we define the variable `line` inside the `try` block.
- We cannot use it outside that block.
- It no longer exists by the time we reach the `sin` definition.
- The following code will generate a syntax error.

Wrong Scope

```
FileIn fin = new FileIn("..\\radius.txt");
try
{
    String line = fin.readLine();
}
catch(EndOfFileException x){}

StringIn sin = new StringIn(line);
```

Keeping The Compiler Happy

Combining Scope and Exceptions

- Here is an initial version of code to read a word from `fin`.

```
String word = fin.readWord();
con.print(word);
```

- This will not work because we have ignored the `EndOfFileException` exception.
- We must catch `EndOfFileException`.

Catching The Exception

```
try
{
String word = fin.readWord();
}
catch(EndOfFileException x) {}
con.print(word);
```

- This will not work because `word` is out of scope by the time we reach `con.print`.
- We must define `word` before the `try` block.

Uninitialised Variable

```
String word;
try
{
String word = fin.readWord();
}
catch(EndOfFileException x) {}
con.print(word);
```

- The compiler now complains about `word` being uninitialised.

Two Paths Through try-catch

- `word` is uninitialised because there are 2 routes through a `try-catch`.
- We could go right through the `try` block.
  - `word` will get a value.
- We could exit the `try` block early, jumping to the `catch`.
  - `word` does not get a value.
- We must initialise `word` before the `try`.

Initialised Variable

```
String word = "";
try
{
String word = fin.readWord();
}
catch(EndOfFileException x) {}
con.print(word);
```

- Now the compiler is happy!