Exceptions

Examples of code which shows the syntax and all that

When a method might cause a checked exception

So the main difference between checked and unchecked exceptions was that the compiler forces us to handle or propagate a checked exception, while an unchecked exception simply propagates silently until it might be handled.

What methods throw checked exceptions?

For instance, methods which deal with I/O like file handling.

Opening a file for writing

Two classes will help us write to a file: java.io.PrintWriter and java.io.FileWriter.

FileWriter can write data to a file (given in the constructor), which is fine. If we want to write text, PrintWriter is a good subclass, because it gives us convenient methods, such as println.

To create a PrintWriter which prints to a file, you need to feed the constructor a FileWriter reference:

```
PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
```

But what could go wrong here?

Things that might go wrong with files

The important part of the previous slide's code was this:

```
new FileWriter("OutFile.txt")
```

We are assuming that we can create an object which can write to this file.

What could go wrong?

- The file might exist but actually be a directory - not good ;-)
- The file cannot be created by us (filesystem/OS says NO!)
- The file exists, but we cannot open it for some reason

Let’s look at the constructor for FileWriter

From the API documentation:

```
public FileWriter(String fileName)
           throws IOException
```

The important part is the “throws” part! This constructor declares that it might throw an IOException for the reasons stated before. IOException is a checked exception (it doesn’t extend RuntimeException), so any code calling it, must take action (or the compiler will complain).

What can we do then, to please the compiler?

Calling something which declares that it might throw a checked exception gives you two choices:

The method surrounding your code must declare that it throws an exception of the same sort (or a super class to the exception), or you must handle the exception.

Declaring that your surrounding method also throws this exception is easy, simply use the `throws` keyword at the declaration, and it will be someone else's problem (the code which calls your method, that is).

If I want to handle the exception then?

If you don’t want to pass on the responsibility to the callers of your method, then you can handle the exception. This is used via the try-catch construct.

You surround the code which might throw an exception with a try-block:

```
try{
  //code which might throw an exception!
}
```

After the try-block you add a catch-block which handles the exception:

```
catch(ExceptionClassName ex){
  //code which deals with the exception
}
```

Back to the writing-file example

```
static void writeFile(){
  try{
    PrintWriter out =
       new PrintWriter(new FileWriter("OutFile.txt"));
  }catch(IOException ex){
    System.err.println("Couldn't open file for writing: " +
                                  ex.getMessage());
  }
}
```

The try-catch block handles IOExceptions if they are thrown:

```
$ javac WritingFile.java && java WritingFile
Couldn't open file for writing: OutFile.txt (Permission denied)
```

If we don’t want to handle it in the method?

If we didn’t want to handle IOExceptions in the method, we could declare that the method throws IOException. But then, the method calling our method would be in our old situation and have to handle or declare the IOException.

If the method is called from, e.g. main() then main() has to handle the IOException with a try-catch around the call to our method. Or it can declare that it throws IOException.

Note, declaring that the main method throws exception isn’t very helpful. No one can handle those exception and the program will crash instead.

Pause!

What can we do with the exception object?

An object which we catch, of type Exception, has, among others, the following methods (which it has inherited):

toString() - you know what it does!

getMessage() - it might have a message for us, as a String

printStackTrace() - it can print the whole call stack, i.e. what method calls led up to this exception being thrown?

A message example:

Our write file example did the following in the handler which caught an IOException and called it ex:

```
System.err.println("Couldn't open file for writing: " +
                    ex.getMessage());
```

An example printout was the following:

```
Couldn't open file for writing: OutFile.txt (Permission denied)
```

The message part was: “OutFile.txt (Permission denied)”

A stack trace example

```
java.io.FileNotFoundException: OutFile.txt (Permission denied)
    at java.io.FileOutputStream.open(Native Method)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:110)
    at java.io.FileWriter.<init>(FileWriter.java:63)
    at WritingFile.writeFile(WritingFile.java:11)
    at WritingFile.main(WritingFile.java:6)
```

It shows the call chain (in reverse, because it’s a stack):

main->writeFile->FW constructor->FOS constructor x 2 -> open

So, main() called writeFile() which called the constructor which eventually called open() which crashed. And we g et line numbers too :-)

Pause…

Who should handle it?

Why did we settle with handling the IOException in the writeFile() method?

Because, we knew that we couldn’t do much more than anyone else could. If we are not allowed to write the file, there’s not much we can do in the method to fix that.

Problem is that now, main(), which called writeFile() doesn’t know that something really bad has happened. Perhaps that’s fine. But if it was really important to write that file, even main should be notified.

So should we declare that writeFile() throws IOException then?

Perhaps passing the problem on isn’t perfect

If we had written the code like this instead:

```
static void writeFile() throws IOException{
  PrintWriter out = new PrintWriter(new
                    FileWriter("OutFile.txt"));
}
```

Then, main(), would’ve been forced to write try-catch around the call to writeFile(). Perhaps that’s not so convenient (topic for discussion!).

What alternatives do we have? The compiler is very particular about rules!

Throwing an exception

We can throw an exception ourselves, using the keyword `throw` (as an imperative, without the trailing s).

You have to say what you want to throw, and the thing you want to throw is an exception object. So this works:

```
throw new IOException("A message about the problem");
```

We can also create an exception which wraps another exception:

```
throw new RuntimeException("A message", ex);
```

(where ex is a reference to the other exception, like an IOException)

Throwing a NullPointerException

This seems useful, right? We can now enforce rules by documenting that we'll throw exceptions if people don't obey our rules. Considered this:

```
public Member(String name, String email){
  // Name is NOT allowed to be null!!!
  if(name == null){
    throw new NullPointerException("Name cannot be null");
  }
  this.name  = name;
  this.email = email; // we can live with null emails ;-)
}
```

Catching the IOException and rethrowing as other

So, we could go back to our writeFile() method and re-write it so that it doesn’t declare that it throws any exception, handle any IOExceptions, and if we catch one, re-throw it as a RuntimeException (which doesn’t need to be checked):

```
static void writeFile(){
  try{
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
  }catch(IOException ex){
    throw new RuntimeException("Problem writing: "+ex.getMessage(), ex);
  }
}
```

What about main ???

If main now calls writeFile() and an IOException is caught and re-thrown as a RuntimeException (unchecked), main will have some problems.

Unless main has a general exception handler, it will crash. So let’s write a simplified general exception handler for main:

```
public static void main(String[] args){
  try{
    writeFile();
  }catch(Exception e){
    System.err.println("Something went wrong: " + e.getMessage());
    System.exit(1);
  }
}
$ javac WritingFile.java && java WritingFile
Something went wrong: Problem writing: OutFile.txt (Permission denied)
```

You should probably log the stacktrace as well

It is possible to log the stacktrace in an error log, without showing the user the whole stacktrace. This is very helpful for developers who want to investigate what went wrong.

Pause

The finally clause

There is another clause which you can use with a try-statement. The finally clause let’s you add code to be executed regardless of whether an exception occurred or not. Consider (pseudo code, this is not real Java!):

```
try{
  turnOnWater();
  waterLawn();
} catch(BrokenPipeException bpe){
  callThePlumber();
}finally{
  turnOffWater();
}
```

Resources should be closed

Don’t leave the water running ;-)

If we open a file, for instance, and try to do something with it, something might go wrong and we get an exception.

A finally clause can be used to make sure that we remember to close the file regardless of whether an exception was thrown or not.

This works even without the catch clause (but usually they are seen together).

```
try{
 something which may fail miserably
 } finally{
 something which must run regardless of any failure
}
```

Some words of advice

Don’t just catch and ignore. It makes your code compile and run, but nothing is reporting or trying to deal with the problem.

Try to avoid to catch too broad an exception type. Usually, you want to do different things for different types of problems. Some code can often throw more than one type of exception. So you should catch and handle them separately in most cases.