Lecture 41

We look at some very simple file handling. We take a file (any file!) and read it in byte by byte. And we write out the bytes to another file. Hence, the program copies one file to another file.

The program allows us to introduce some important file handling library classes, `File`, `FileInputStream` and `FileOutputStream`.

But the real point of the lecture is to further illustrate and deepen our understanding of Java exception handling. In particular, we draw the distinction between checked and unchecked exceptions and see the catch-or-specify principle for checked exceptions.

Checked and unchecked exceptions

- We know that an exception can either be caught or passed up the call chain
- But there are two kinds of exception:
  - *Unchecked exceptions:* In the Java library, these are of class `RuntimeException` and its subclasses
  - *Checked exceptions:* In the Java library, these are all other subclasses of `Exception`
- If you want a checked exception to be passed up the call chain, you must explicitly say so

Checked exceptions: catch or specify principle

- Every (checked) exception must either be *caught* in an appropriate `catch` clause or, if you want it to be passed up the call chain, it must be *specified* in a `throws` clause (N.B. `throws` not `throw`)

```
public void myMethod throws IOException
{  try
   {  some statements that might throw
      CloneNotSupportedException or
      IOException
   }
   catch (CloneNotSupportedException cnse
   {  handle the exception
   }
}
```

- Your code must include one or the other; the *compiler* enforces the principle: compiler error if not

Reading & writing bytes

- The `File` class: objects that represent files/directories on the host file system. (These are objects in memory; they may or may not correspond to actual files/directories)
- `File`'s interface: `exists`, `isDirectory`, `isFile`, `canRead`, `canWrite`
- `FileInputStream` for reading bytes: constructor opens the file; `read` returns -1 if reading beyond end of file; program termination closes the file (but can use `close`); opening/reading/closing may throw `IOException`
- `FileOutputStream` for writing bytes: constructor opens the file; program termination closes the file (but can use `close`); opening/writing/closing can throw `IOException`

`FileCopier` **version 1 — Won't compile!**

```
import java.io.*;
public class FileCopier
{  public void copy()
   {  FileInputStream fis = null;
      FileOutputStream fos = null;
      int nextByte = 0;
      boolean eof = false;
      fis = new FileInputStream(source);
      fos = new FileOutputStream(target);
      do
      {  nextByte = fis.read();
         eof = (nextByte == -1);
         if (! eof)
         {  fos.write(nextByte);
         }
      } while (! eof);
      fis.close();
      fos.close();
   }
   private File source;
   private File target;
}
```


`FileCopier` **version 2 — specifies exception**

```
import java.io.*;
public class FileCopier
{  public void copy() throws IOException
   {  // body as before
   }
   private File source;
   private File target;

   public static void main(String[] args)
      throws IOException
   {  FileCopier copier = new FileCopier(
         new File("original.txt"),
         new File("copy.txt"));
      copier.copy();
   }
}
```

The method that calls `copy` (in this case `main`) must also either catch or specify (here, it specifies).

`FileCopier` version 3 — catches exception

```
public void copy()
{  // Variables as before
   try
   {  // 'heart' of the code as before
   }
   catch (IOException ioe)
   {  System.out.println(ioe.getMessage());
   }
   finally
   {  try
      {  if (fis != null)
         {  fis.close();
         }
         if (fos != null)
         {  fos.close();
         }
      }
      catch (IOException ioe)
      {  System.out.println(ioe.getMessage()
      }
   }
}
```


Rolling your own exceptions

- When one class throws exceptions to another, it can often make your program more self-documenting if you define your own classes of exceptions and throw these instead. . .

```
public class FileExistsException
   extends Exception
{
   public FileExistsException()
   {  super();
   }

   public FileExistsException(String msg)
   {  super(msg);
   }
}
```

- Inherits, e.g., `getMessage` and `printStack-Trace`

Throwing your own exceptions

- Suppose we define `FileExistsException` and `FileNotExistsException`.

- We can explicitly create objects from these classes and `throw` them (N.B. `throw` not `throws`)

```
public void copy()
   throws FileNotExistException,
      FileExistsException, IOException
{  // Variables as before
   if (! source.exists())
   {  throw new FileNotExistException();
   }
   if (target.exists())
   {  throw new FileExistsException();
   }
   // The rest as in version 2
}
```


Unchecked exceptions

- The catch or specify principle, and its enforcement by the compiler, only applies to checked exceptions

- If you want an unchecked exception to be passed up the call chain, specifying it with a `throws` clause is optional

- If you omit the `throws` clause, the compiler won't complain

- Since you're let off the hook, you may end up writing code that is less robust. E.g. if you overlook catching `NumberFormatException`s, the compiler won't remind you:

```
String str = tf.getText().trim();
int num = Integer.parseInt(str);
```


Guidelines for exception handling

- Define your own classes of exception to make your program more self-documenting

- If you define your own exceptions, 'always' make them checked exceptions

- Do not allow too much 'passing the buck' up the call chain – if the handler is too distant from where the exception is first thrown your program is harder to understand

- Keep `try` blocks quite large to avoid code pollution

- Do not use exceptions as a way of doing normal flow of control — they are often more complicated and less efficient that using `if`s, etc., and so should be used only for abnormal situations