Fundamentals of Java

Files and exceptions

Files

To use the File I/O API:

```
import java.io.*;

File f = new File("hello.txt");
```

- This does ***not*** create a new file!
- It creates a reference to a file location.

File methods

Method	Description
exists()	Tests whether the file exists
isDirectory()	Tests whether this file is a directory
isFile()	Tests whether this file is a regular file
listFiles()	If this is a directory, returns an array of its contents
mkdir()	Creates a directory at this path
mkdirs()	Creates a directory at this path, including any missing parent directories
renameTo(File)	Renames this file

Class File provides methods for everything except for reading and writing files.

Reading and Writing

Java uses the same approach to read from:

- a file
- the keyboard
- a URL
- a network socket

AND the same approach to write to:

- a file
- the terminal window
- a URL (via the POST method)
- a network socket

InputStream and Reader

Use class InputStream to read binary data.

```
abstract class InputStream {
    int read() ...
    int read(byte[] b) ...
    int read(byte[] b, int off, int len) ...
    void close() ...
}
```

Use class Reader to read text data.

```
abstract class Reader {
    int read() ...
    int read(char[] cbuf) ...
    int read(char[] cbuf, int off, int len) ...
    void close() ...
}
```

Subclasses of InputStream

Subclass	Description
ByteArrayInputStream	Reads data from an array
FileInputStream	Reads data from a file
StringBufferInputStream	Reads data from a StringBuffer
BufferedInputStream	Buffers input for efficiency

`System.in` is a `BufferedInputStream` wrapped around a `FileInputStream` wrapped around a `FileDescriptor`.

Subclasses of Reader

Subclass	Description
CharArrayReader	Reads data from an array
FileReader	Reads data from a file
StringReader	Reads data from a String
BufferedReader	Buffers input for efficiency, and also provides line scanning.
InputStreamReader	Reads from an InputStream

```
in = new BufferedReader(
  new InputStreamReader(
    new FileInputStream(new File("hi.
txt"))));

// Returns null at end-of-file
String line = in.readLine();
```

OutputStream and Writer

Use class OutputStream to write binary data.

```
abstract class OutputStream {
    void write(byte b) ...
    void write(byte[] b) ...
    void write(byte[] b, int off, int len) ...
    void close() ...
}
```

Use class Writer to write text data.

```
abstract class Reader {
    void write(int c) ...
    void write(char[] cbuf) ...
    void write(char[] cbuf, int off, int len) ...
    void close() ...
}
```

Subclasses of OutputStream

Subclass	Description
ByteArrayOutputStream	Writes data to an array
FileOutputStream	Writes data to a file
PrintStream	Provides print() and println()
BufferedOutputStream	Buffers output for efficiency

`System.out` is a `PrintStream` wrapped around a `BufferedOutputStream` wrapped around a `FileDescriptor`.

Subclasses of Writer

Subclass	Description
CharArrayWriter	Writes data to an array
FileWriter	Writes data to a file
PrintWriter	Provides print() and println()
BufferedWriter	Buffers output for efficiency
StringWriter	Writes data to a String

```
out = new PrintWriter(
   new FileWriter("file.txt");
);
out.println("A line");
```

Read loop - #1

```
BufferedReader in = ...;

String line;

line = in.readLine();
while (line != null) {
    System.out.println("read line: " + line);
    line = in.readLine();
}

in.close();
```

Is there a way to avoid the duplicate **`readLine`**?

Read loop - #2

```
BufferedReader in = ...;

String line;

while ((line = in.readLine()) != null) {
    System.out.println("read line: " + line);
}

in.close();
```

Exceptions

File operations can fail for various reasons:

- File not found
- Permission denied
- Insufficient disk space
- etc.

Java provides a general framework for error handling called *Exceptions*.

Exceptions example

```
try {
    File f = new File("myfile.txt");
    FileReader in = new FileReader(f);
    BufferedReader bin = new BufferedReader(in);
    String line = in.readLine();
    System.out.println(line);
}
catch (FileNotFoundException e) {
    System.err.println("Could not find file");
}
catch (IOException e) {
    System.err.println(e);
    e.printStackTrace();
}
```

Documentation on exceptions

Methods that potentially throw exceptions are declared as such. For example:

```
public String readLine() throws IOException
```

Refer to the Javadoc documentation for a class to see what exceptions a method throws:

http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html#readLine()

Checked vs Unchecked exceptions

A "checked" exception must be handled otherwise the compiler will produce an error.

An "unchecked" exception is not checked by the compiler and may crash your program.

Checked exceptions:

`IOException`, `InterruptedException`...

Unchecked exceptions:

`NumberFormatException`, `NullPointerException`...

Handling exceptions

There are two ways to deal with a checked exception.

#1 - Catch the exception

```
void save() {
    try {
        FileOutputStream out = new ...
        ...
        out.close();
    }
    catch (IOException e) {
        System.err.println("Cannot
save.");
    }
}
```

#2 - Re-throw the exception

```
void save() throws IOException {
    FileOutputStream out = new
...
    ...
    out.close();
}
```

Defining an exception class

You may define an exception class for your own purposes.

- Checked exceptions are subclasses of Exception.
- Unchecked exceptions are subclasses of RuntimeException.

For example:

```
public class ValidationException extends Exception {
    public ValidationException(String message) {
        super(message);
    }
}
```

Throwing your own exception

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) throws ValidationException {
        if (name.trim().isEmpty())
            throw new ValidationException("Name is empty");
        this.name = name;
    }

    public void grow(int amount) throws ValidationException {
        if (amount <= 0)
            throw new ValidationException("Amount must be positive");
        age += amount;
    }
}
```

Catching your exception

```
try {
    Person p = new Person("John", 22);
    p.grow(-10); // will throw an exception
}
catch (ValidationException e) {
    System.err.println(e.getMessage());
}
```

Finally

A finally block is always executed after a try block:

```
try { ... code which may fail ...}
catch (Exception e) { ... }
finally { ... code that is always executed ... }
```

Finally example

```
PrintWriter out = null;
try {
    out = new PrintWriter(new FileWriter("file.txt"));
    out.println("line 1");
    out.println("line 2");
}
catch (IOException e) {
    System.err.println("Failed to save file: " + e);
}
finally {
    // happens on both success and failure
    if (out != null)
        out.close(); // what if this fails?
}
```

If the final out.close() fails, it will also throw an exception. In practice, you will need another try/catch block around this line (unfortunately!)