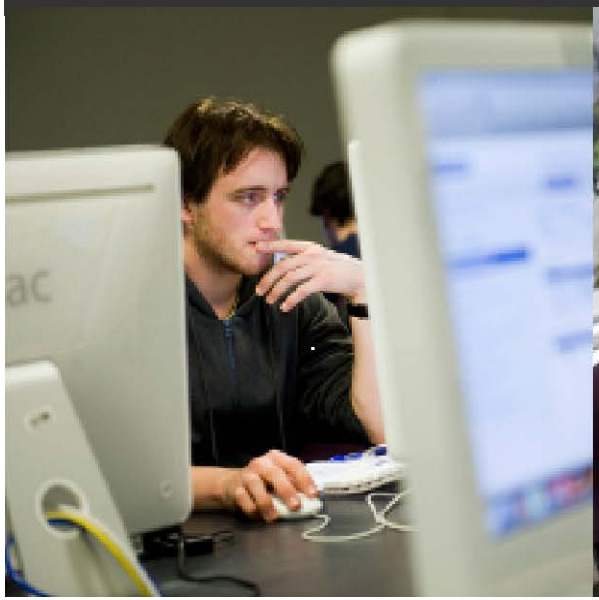
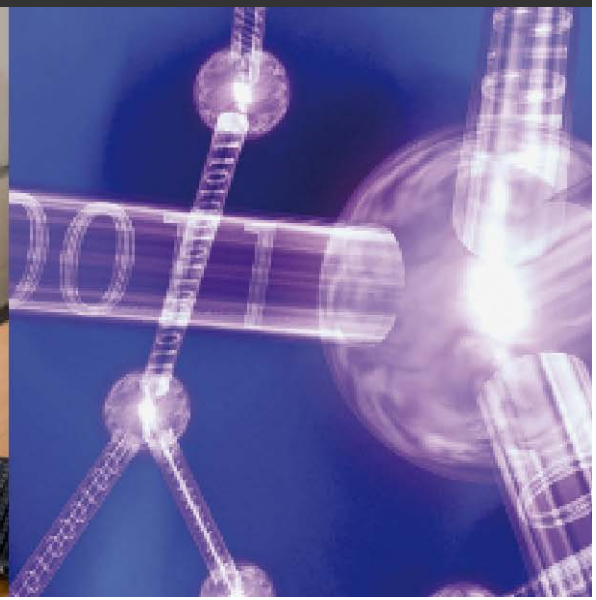
MONASH University

Information Technology

Module 9:
File Input & Output, and Exceptions

FIT2034 Computer Programming 2
Faculty of Information Technology





Case Study

- Lets look at a case study to go with today's content
- We are working on a D&D style game
- 3 Character Classes (Warrior, Mage, Rogue)
- Our task is to create a way to load pre-built characters from a text file
- Characters share some variables (HP)
- Character have unique variables (Strength, Mana, Stealth Rating)

What we start with

- BaseCharacter
  - String name
  - Int maxHP
  - Int currentHP
  - Int defence
- Warrior
  - Int strength
- Mage
  - Int magicPower
  - Int maxMana
  - Int currentMana

What we start with

- Rogue
  - Int numLockpicks
  - Int stealthRating

What we want to create

- MainDriver
  - For testing purposes (Load File, Get Character, etc)
- CharacterFactory
  - Class for loading file
  - Created Characters added to an ArrayList
  - Get a copy of a Character Object in ArrayList via index

Part 1:
Exceptions & Exception Handling

Objectives – Part 1

- Explain the purpose of exceptions in *Java*
- handle some of the common *Java* exceptions
- construct try … catch blocks, including the use of a *finally* clause
- invent, instantiate, throw and catch programmer-defined exceptions

Responding to Failure

- Sometimes validating user input is not enough
- Things can still go wrong over situations for which you have no control:
  - Inappropriate method used (e.g. used by other programmers re-using your classes)
  - Bad user input
  - Faulty equipment, device error
  - Physical limitations
- Users expect the programs to react sensibly when errors happen.
  - Errors ***at runtime*** as opposed to syntax errors or logic errors

What Do We Do About It?

- Step 1: Detect that a problem has occurred.
- Step 2: Report the problem – so that the user knows that something unexpected has happen.
- Step 3: Deal with the problem in an appropriate way – program should then recover/continue gracefully.

Runtime Exceptions

- Sometimes an instruction fails…
- If it fails, the Java Virtual Machine will
  "throw an exception"

- the execution of the running program is interrupted
- this is called a "***runtime exception***"
- Some other part of the program in the current call-stack must have been coded to ***"catch"*** potential exceptions, otherwise the program will die.

Exceptions

- Exception
  - An event that occurs during the execution of a program that disrupts the normal flow of instructions.

- Exception handling
  - A mechanism for passing control from the point of error detection to a competent “recovery handler”.

An Example Situation Causing an Exception

- The following code causes an exception if the user enters something that isn't a number:

```
public void doSomething ()
{
   Scanner scanInput = new Scanner(System.in);
   int userInput;

   userInput = scanInput.nextInt();
}
```

What can we do?

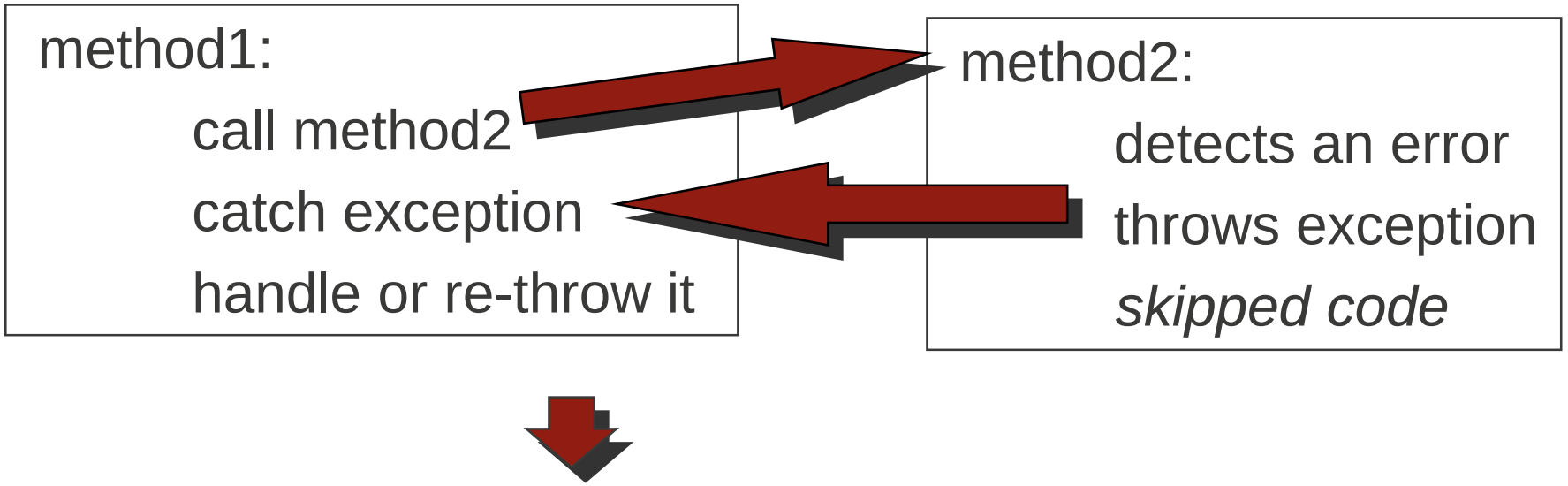
- We can catch the exception and
  - Handle the exception, or
  - Re-throw it to be handled elsewhere

Exception Handling when a call stack is involved

- Java allows every method an alternative exit path, when it is not able to complete the task in the normal way.
- Instead of returning a value, the system ***throws*** an object that contains information about the problem.
- Looks for the first ***exception handler*** that can deal with the particular error condition.
- Execution does not resume at the line after the code that called the method.

Catching Exceptions

Exceptions may be caught.



To be handled or re-thrown by some other method

The `try` and `catch` Keywords

- When we call a method which may throw an exception, we should place it inside a **try** block:

```
try
{      // code which may cause an exception to be thrown
       myDatabase.remove(elementNumber);
       doSomething();
}
```

- When an exception is thrown inside a **try** block, our program looks for a matching **catch** block:

```
catch (NumberOutOfRangeException exc)
{
       // do something here to deal with problems
}
```

The Exception class

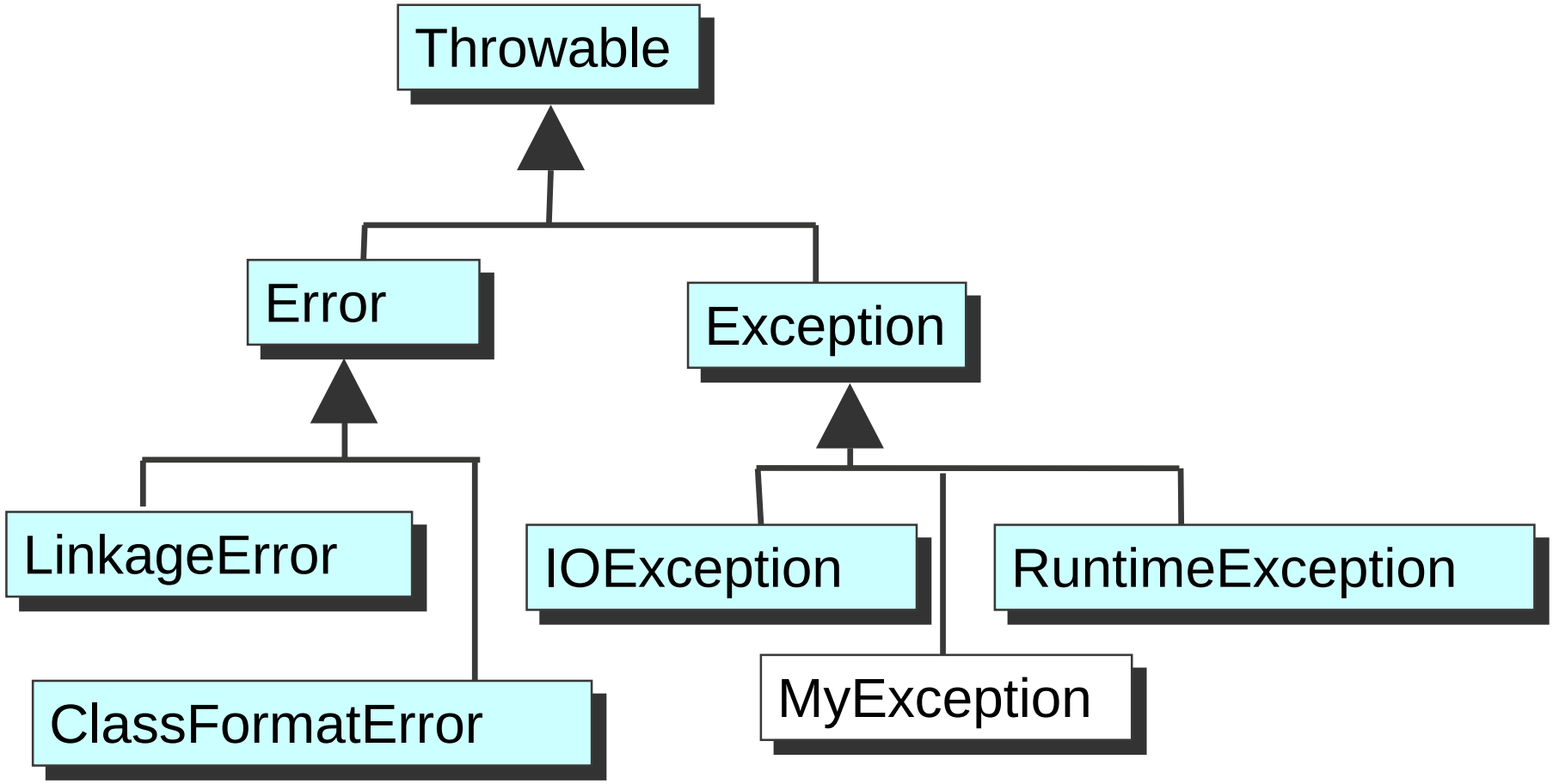
- Exceptions are objects
  (i.e. instances of class **Exception,** or one of its subclasses)
- There are a number of constructors: (See Java API docs)

`public Exception()` Constructs an **`Exception`** with no specified detail message.
`public Exception(String s)` Constructs an **`Exception`** with the specified detail message. Parameters: s – a description giving details about the problem

The Exception class

- The Exception class provides several methods useful for debugging purposes:
  - **printStackTrace()** – shows the call-stack at the moment when the exception occurred
  - **getMessage()** – returns the string detailing the problem represented by the exception.

The Exception hierarchy (part of it)



Examples of Common Exceptions that arise

- Exceptions that inherit from **`RuntimeException`**:
  - A null pointer access.
  - An out-of-bound array access.
  - A bad cast
- Exceptions that do not inherit from **`RuntimeException`**:
  - Trying to read past the end of a file
  - Trying to open a malformed URL

Example: What We Have Done in Past Weeks

- Validation code, in a method:

```
public boolean remove(int elementNumber)
{
   if (elementNumber < 0 || elementNumber > count)
   {
      return false;
   }
   ...      // Code to adjust array to remove the
element
}
```

- In the caller:

```
...
if (remove(4) == false)
      System.out.println("Element does not exist");
```

A problem with that approach

- Doing this for every method can make code unnecessarily long, and hard to read.
- You end-up always *programming for failure* in the caller
- It is better to *program for success.*

Solution: Writing code to generate exceptions

- Often we want to write a class so that it could generate (i.e. throw) exceptions
  - In general, a Java method can throw an exception if it encounters a situation it cannot handle.
- The caller can be written with an assumption of success
  - Wrap the code inside a try-catch block, and deal with exceptions in a uniform way

The throws clause and throw statement

- Methods can throw their own exceptions to report problems
- A method that throws a (checked) exception must declare this fact in its signature (to “advertise”)

```
public void remove (int elementNumber) throws
Exception
{
   if (elementNumber < 0 || elementNumber > count)
   {
      throw new Exception();
   }
   ...
}
```

Checked vs. Unchecked

Terminology:

- Most exceptions are **checked exceptions**.
  - Named so because the compiler checks that you have considered the possibility of the exception

- **Unchecked exceptions** can happen at any time and do not have to be declared.
  - Unchecked exceptions in Java are implemented by the RuntimeException and Error classes.

Checked vs Unchecked Exceptions

- Exceptions deriving from the classes **`Error`** or **`RuntimeException`** are *unchecked* exceptions.

- A method you write that throws exceptions must declare all the *checked* exceptions that it could possibly generate while executing, e.g.
  - In the **FileReader** class, public int **read**() throws IOException
  - In the **Scanner** class, public **Scanner**(File source) throws FileNotFoundException
- *Unchecked* exceptions are either beyond your control or result from conditions that you should not have allowed in the first place.

Unchecked Exceptions – Examples

- Examples
  1. **NullPointerException (subclass of RuntimeException)**
  2. **ArrayIndexOutOfBounds (subclass of RuntimeException)**
  3. **ClassCastException (subclass of RuntimeException)**
- These happen because of programming errors
- These exceptions are generated and thrown by the *Java Virtual Machine (JVM)* automatically.

Creating Your Own Exception Types

```
public class NumberOutOfRangeException extends Exception
{
   /**
    * Create a new exception with the illegal number
    * as an argument.
    */
   NumberOutOfRangeException(int number)
   {
      super("The number " + number + " is out of range");
   }
}
```

Java-Provided Exceptions

- Java already defines a number of exceptions which can be thrown (both checked and unchecked)
- These are specified in the *Java API*
- They are there for programmers to use. Examples:
  - **IllegalArgumentException** (unchecked)
  - **IOException** (checked)
  - Browse the API for more examples

Re-throwing a caught exception

```
catch(NumberOutOfRangeException problem)
{
   System.err.println("someone else's problem");
   throw problem;
}
```

- The exception searches back up the call stack to find an appropriate handler

Catching Multiple Different Exceptions

```
...
int itemNumber = getInputFromUser();
try {
   database.remove(itemNumber);
}
catch (NumberOutOfRangeException noorExc) {
  System.out.println("An error occurred: " + noorExc);
}
catch (Exception exc) {
  System.out.println("An error occurred: " + exc);
}

System.out.println("Now for something exciting...");
...
```

Important Points to Note

- You can have as many catch blocks as you need
- Avoid using a generic catch block – try to be specific
- The first catch block that can accommodate the type of the exception will be the one that is executed
  - So place more-specific exception types.
- When an exception is thrown, control is passed to the appropriate **catch** block, and then to the statement following the **try-catch** structure
- Exceptions can be re-thrown to be handled elsewhere

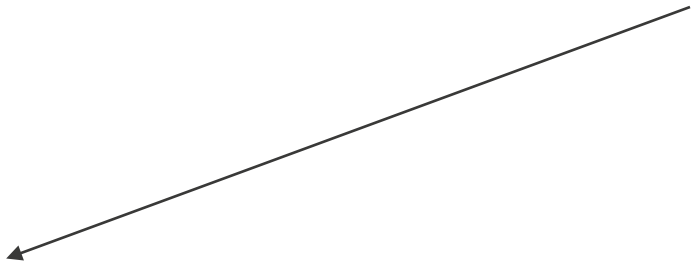
The finally Keyword

- When an exception is caught, control is passed to a corresponding **catch** block and any remaining code still inside the **try** block is not executed.
- The optional **finally** clause is always executed after a **try** block, independently of how the **try** block was terminated.
- **finally** blocks should be used to clean up after an operation
  - For example, to close any opened files, place this code in the finally block instead of in the try block.

try and finally

```
public void someMethod()
{
    ....
    try
    {
        // some code
    }
    catch
    {
    ....
    }
    finally
    {
        // clean up
    }
    ....
}
```

always executed!



When to Throw Exceptions

- Use Exceptions for a method when :
  - Something truly exceptional/abnormal happens
    - Eg. When searching for an item in an array, a method should not throw an exception if the item is not present – since that result is within the set of expected results, and is not considered abnormal.
- In short,
  - Think of yourself as the author of *this* class only
  - Anything that could go wrong, that you have no control over within *this* class, deserves an **Exception**

Part 2:

Input & Output

Objectives – Part 2

On completion of this session you should be able to:

- explain the terms: streams, append, input, output, serialization, de-serialization;
- explain the differences between text and binary files;
- create, open and close disk files;
- read from and write to sequential text files;
- read objects from and write objects to binary format files;
- explain the consequence of recompiling a class which has previously had objects of that type serialized to a file.

The Java IO package

- Provided as part of the JDK
  - Contains classes and interfaces to use when dealing with different types of input and output
- Required
  - `import java.io.*;`
  - `try/catch()` blocks to handle input / output exceptions

The File Class

The java.io package includes:

- **classes used to extract data from files**
- **classes used to place data within files**
- **class used to provide Java programs with mechanisms for interacting with the file system**

```
Class File:

java.lang.Object
      |     java.io.File

All Implemented Interfaces:
   Comparable<File>,  Serializable
```

The File class

- Represents information *about* an existing particular file, or directory
  - Derived directly from class Object
- Uses for this class:
  - List existing files and directories
  - Check for existence of files *and* directories
  - Used by other input/output objects that handle transferring of data
- Restrictions
  - **Cannot write data to or read data from any files**

Example: Creating a new File instance

new File (String pathname)

Creates a new File instance by converting the given pathname string into an abstract pathname.

Example:

```
String fileName = "C:\\FIT2034.txt"
File f = new File(fileName);
```

Example: File class methods

```
public static void main(String [] args)
{
  File f = new File("Path and fileName go here");
  if( f.isFile() && f.canRead() && f.canWrite() )
          System.out.print("File exists and can be used");
  else
     System.out.print("File no good for purposes");
}
```

f is a new File instance that *represents* a file or a directory that may or may not already exist

Text Data vs Binary Data

- All files are stored as binary data in the form of bytes

- **Text files** are human and text editor readable because they consist only of ASCII characters

- **Binary files** are not humanly readable because they use all 256 possible values of a byte.
- Binary files are readable only by programs that know how to interpret all the bytes

Text Files

- A Text File is a file where the contents are humanly readable
  - a code-file is text file.
  - HTML files are text files
- We can use a Scanner object to read the contents of humanly readable text files
  - Scanners must be connected to an InputStream object
- We can use a PrintStream object to write contents to humanly readable text files
  - A PrintStream must be connected to an OutputStream object

Binary Data

Example Binary Files:

- Compiled Class files
- ZIP files
- MP3 files
- JPG files

- Data is sometimes organised at the bit-level, such as in compressed or encrypted files.

Opening a text file to read with Scanner

- **Scanner** has several constructors
- We are familiar with this:

  Scanner scan = new Scanner(**System.in**)
- **in** is a variable of type **InputStream**
- We can also provide a **File** as the parameter:

```
Scanner inFile = new Scanner( new File(
  "myfile.txt" ) );
```

- We can even do the following longer way:

```
inFile = new Scanner ( new FileInputStream (
  new File ( "myfile.txt" ) ) );
```

- Then use scanner methods as usual to get input

Writing to a text file using PrintStream

- We are familiar with using System.out to display to the screen.
- **out** is a variable of type **PrintStream**
- To create a PrintStream that writes into a file, we can do either of the following:

```
PrintStream outFile = new PrintStream (
    new File( “output.txt” ) );

outFile = new PrintStream ( new FileOutputStream (
    new File (“out.txt”) ) );
```

Example: Writing to a text file

```
public void writeFileDemo() throws IOException
{
   String firstName = "Fred", lastName = "Flintstone",
     address = "25 Rocky Road";
   int postcode = 3800;

   PrintStream outFile = new PrintStream ( new File ("Details.txt")
);

   // First line: first and last name, separated by a space…
   outFile.println(firstName + " " + lastName);

   // Second line: the Address
   outFile.println(address);

   // Third line: the Postcode (a number, converted to characters)
   outFile.println(postcode);

   outFile.close();
}
```

Appending to a text file

- By default, opening a file for writing destroys any existing file with same name
- To keep the existing file, but to add to the end, file needs to be opened in **append mode**
- This can only be done using the FileOutputStream constructed using the 2-parameter constructor, with ‘true’ as second parameter:

```
FileOutputStream fos = new FileOutputStream( “logfile.txt”,
    true );
```

Then you connect the PrintStream to it:

```
PrintStream outFile = new PrintStream ( fos );
```

Example: Reading from text file

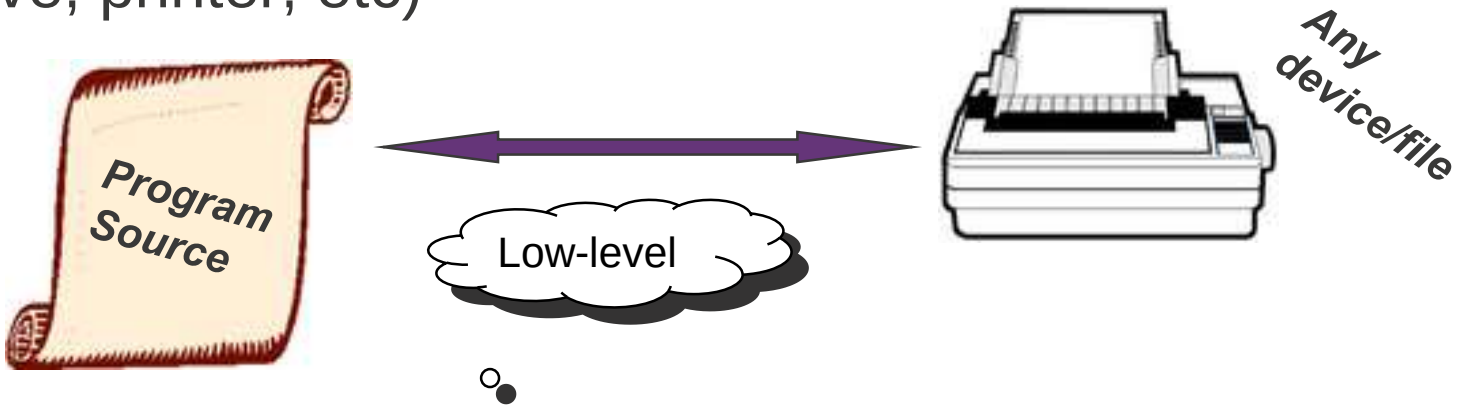
```
public void readFileDemo() throws IOException
{
   Scanner inFile = new Scanner ( new File ("Details.txt") );

   String firstName, lastName, address;
   int postcode;

   firstName = inFile.next();          // reads first token from file
   lastName = inFile.next();           // reads next token from file
   inFile.nextLine();
   address = inFile.nextLine();        // Reads whole of second line.

   postCode = inFile.nextInt();        // Reads the next token, as int

   System.out.println(firstName + " " + lastName + " lives at " +
      address + " " + postCode);

   inFile.close();
}
```

Streams

- I/O takes place through objects representing *streams* of bytes (or pairs of bytes) to and from physical devices
- There are five (5) categories of I/O classes in Java:
  - Low-level Byte Stream classes: **`FileInputStream / FileOutputStream`**
  - High-level Byte Stream classes: **`FilterInputStream / FilterOutputStream`**
  - Low-level Character Stream classes: **`FileReader / FileWriter`**
  - High-level Character Stream classes: **`BufferedReader / BufferedWriter`**
  - Direct File Input/Output classes: **`FileReader / FileWriter`**

Streams : Low-level

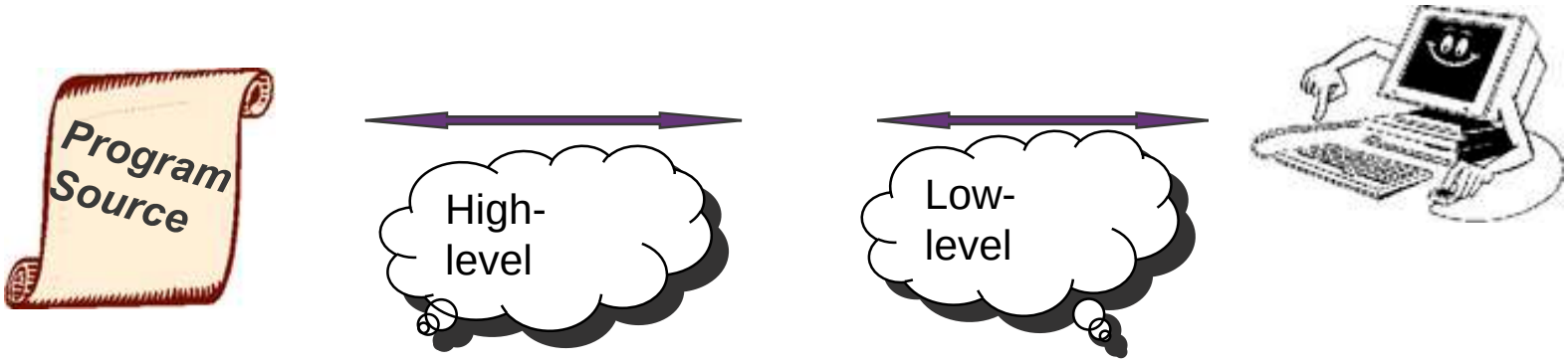
- Low-level streams establish a connection (a pathway) between your program and any particular device, (a hard drive, printer, etc)




```
FileOutputStream fos = new
  FileOutputStream("fileName");

fos.write(byteArray);       // writes contents of
                            // byteArray[ ] to file

fos.close();
```

Streams: High-level

- High level streams modify an ***existing stream*** that has been created by a low-level stream class.
- High-level stream ***cannot*** set up a connection between a program and a data source.




```
DataOutputStream filterOS = new DataOutputStream(
  new FileOutputStream("fileName"));
```

- DataOutputStream has methods – writeChar, writeInt, writeDouble, writeFloat etc. It has more “high-level” methods than FileOutputStream. (See APIs)

Serialization

- Serialization is the process of converting an object to a sequence of bytes, and then regenerating the original object.
- Provides – lightweight persistence
  - You can take a serializable object and write it to disk, then restore that object when the program is run again: it produces the effect of persistence

Serialization

- Objects to be serialised must implement the ***`Serializable`*** interface
- ***`Serializable`*** has no methods
- Many Java API classes already implement this interface
- References within objects to be serialised are serialised also
  - This is referred to as a web of objects.

Serialization and De-Serialization

- Serialization is the process of converting an object into a sequence of bytes which represent the state of an object at that moment.

- The opposite process is called **de-serialization**
  - **Reads** in a sequence of bytes representing an object,
  - **Restores** it into memory such that it has the exact state that it had when it was serialized,
  - **Re-establishes** all links to other objects, so composite objects can be fully restored.

Writing / Reading objects

- There are 2 requirements for writing objects to a file:

1. The class must implement the **`Serializable`** interface:
   - to allow objects of that class type to be written to disk
2. An object of type **`ObjectOutputStream`** is required.
   - This provides the **`writeObject()`** method which accepts any object, and will convert it into a sequence of bytes, and send these into the stream.

- To retrieve the object, an object of type **`ObjectInputStream`** is required.

To Serialize an object to a stream

```
Customer aCustomer = new Customer(...);
...
{
    FileOutputStream fileOut =
      new FileOutputStream("myFile.ser");

    ObjectOutputStream out =
      new ObjectOutputStream(fileOut);

    out.writeObject(aCustomer);  // serializes
}
```

Effects of Serialization

- When an object is serialized, all attributes that are not declared transient will be written out.
- Transient means ‘not persistent’
- Example Attributes:

```
private transient OracleDBConnection
  dbConn;
private String myName;
private Car myCar;
```

The object referred to by **dbConn** variable will not be serialized, whereas the String referred to by **myName** will be.

De-serialization of an object from stream

```
Customer aCustomer;

…

{
  FileInputStream fileIn = new
  FileInputStream("myFile.ser");
  ObjectInputStream in = new
  ObjectInputStream(fileIn);

  // de-serialize – requires a typecast AND must catch
  // ClassCastException:
  try {
    aCustomer = (Customer) in.readObject();
  } catch (ClassCastException cce) { ... };
}
```

Input / Output Exceptions

- In compiling and/or running programs which deal with files and streams, you may come across either checked and/or runtime exceptions
  - IOException
  - FileNotFoundException
  - EOFException
- These must be handled by your code
- Include the throws clause or try/catch blocks