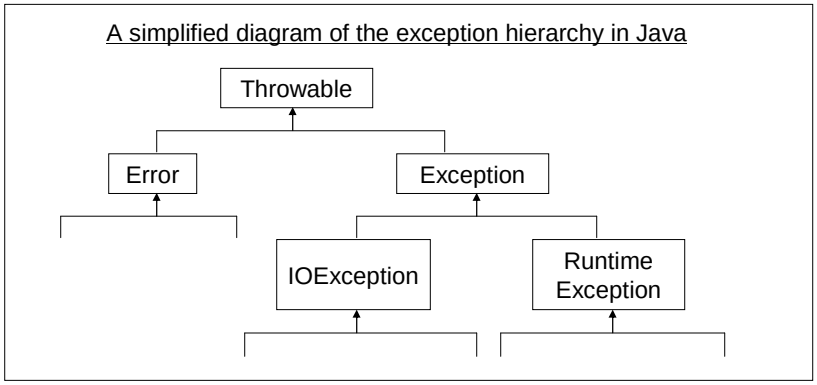
Lecture 7
Exceptions and I/O

Overview of the exception hierarchy



- All exceptions extend the class `Throwable`, which immediately splits into two branches: `Error` and `Exception`
  - `Error:` internal errors and resource exhaustion inside the Java runtime system. Little you can do.
  - `Exception`: splits further into two branches.

Focus on the `Exception` branch

- Two branches of `Exception`
  - exceptions that derived from `RuntimeException`
    - examples: a bad cast, an out-of-array access
    - happens because errors exist in your program. Your fault.
  - those not in the type of `RuntimeException`
    - example: trying to open a malformed URL
    - program is good, other bad things happen. Not your fault.
- Checked exceptions vs. unchecked exceptions
  - *Unchecked exceptions*: exceptions derived from the class `Error` or the class `RuntimeException`
  - *Checked exceptions*: all other exceptions that are not unchecked exceptions
    - If they occur, they must be dealt with in some way.
    - The compiler will check whether you provide exception handlers for checked exceptions which may occur

Declaring to throw checked exceptions

- A java method or constructor should be declared to throw exceptions under two situations:
  1. It calls another method that throws a checked exception
  2. It throws a checked exception with the `throw` statement inside its body
- Declare a method or constructor to throw an exception (or exceptions) by using `throws` clause in its header

  E.g

  Single exception:

```
public FileInputStream(String s) throws FileNotFoundException
```

  Multiple exception:

```
public Image load(String s) throws EOFException, IOException
```

- ✧ You should NOT declare an unchecked exception after `throws`! They are either outside of your control, or should be avoid completely by correcting your code.

Using `throw` to throw an exception

- Throw an exception under some bad situations

  E.g: a method named `readData` is reading a file whose header says it contains 700 characters, but it encounters the end of the file after 200 characters. You decide to throw an exception when this bad situation happens by using the `throw` statement

```
throw (new EOFException());
```

or,

```
EOFException e = new EOFException();
throw e;
```

the entire method will be

```
String readData(Scanner in) throws EOFException {
   . . .
   while(. . .) {
      if (!in.hasNext()) //EndOfFile encountered {
          if(n < len)
             throw (new EOFException());
      }
      . . .
   }
   return s; }
}
```

Using `throw` to throw an exception (cont.)

- In the method body you can only **`throw`** exceptions which are of the same type or the subtype of the exceptions declared after **`throws`** in the method header
- If you can find an appropriate exception class in the library, make an object of that class and throw it; otherwise, you design your own exception class

Creating new exception types

- Exceptions are objects. New exception types should extend `Exception` or one of its subclasses
- Why creating new exception types?
  1. describe the exceptional condition in more details than just the string that `Exception` provides

E.g. suppose there is a method to update the current value of a named attribute of an object, but the object may not contain such an attribute currently. We want an exception to be thrown to indicate the occurring of if such a situation

```
public class NoSuchAttributeException extends Exception {
   public String attrName;
   public NoSuchAttributeException (String name) {
        super("No attribute named \"" + name + "\" found");
        attrName = name;
   }
}
```

  2. the type of the exception is an important part of the exception data – programmers need to do some actions exclusively to one type of exception conditions, not others

Catching exceptions

- Checked exceptions handling is strictly enforced. If you invoke a method that lists a checked exception in its throws clause, you have three choices
  1. Catch the exception and handle it
  2. Declare the exception in your own `throws` clause, and let the exception pass through your method (you may have a `finally` clause to clean up first)
  3. Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own `throws` clause

try/catch clause (1)

- Basic syntax of try/catch block

```
try {
   statements
} catch(exceptionType1 identifier1) {
   handler for type1
} catch(exceptionType2 identifier1) {
   handler for type2
} . . .
```

  - If no exception occurs during the execution of the statements in the try clause, it finishes successfully and all the catch clauses are skipped
  - If any of the code inside the try block throws an exception, either directly via a throw or indirectly by a method invoked inside it
    1. The program skips the remainder of the code in the try block
    2. The catch clauses are examined one by one, to see whether the type of the thrown exception object is compatible with the type declared in the catch.
    3. If an appropriate catch clause is found, the code inside its body gets executed and all the remaining catch clauses are skipped.
    4. If no such a catch clause is found, then the exception is thrown into an outer try that might have a catch clause to handle it
  - A catch clause with a superclass exceptionType cannot precede a catch clause with a subclass exceptionType

try/catch clause (2)

- *Example*

```
public void read(String fileName) {
   try {
      InputStream in = new FileInputStream(fileName);
      int b;

      //the read() method below is one which will throw an IOException
      while ((b = in.read()) != -1) {
         process input
      }
   } catch (IOException e) {
      exception.printStackTrace();
   }
}
```

another choice for this situation is to do nothing but simply pass the exception on to the caller of the method

```
public void read(String fileName) throws IOException {
   InputStream in = new FileInputStream(fileName);
   int b;
   while ((b = in.read()) != -1) {
         process input
   }
}
```

- If you call a method that throws a checked exception, you must either handle it or pass it on. Check the Java API documentation to see what exceptions will be thrown!

`try/catch` clause (3)

- You can throw an exception in a `catch` clause. Typically you do this because you want to change the exception type to indicate the failure of your subsystem

```
try {
   access the database
} catch (SQLException e) {
   Throwable se = new ServletException("database
error");
   se.setCause(e);
   throw se;
}
```

  - When the exception is caught, the original exception can be retrieved. This ***chaining*** technique allows you to throw high-level exceptions in subsystems without losing details of the original failure

```
Thowable e = se.getCause();
```

`finally` clause (1)

- You may want to do some actions whether or not an exception is thrown. `finally` clause does this for you

```
Graphics g = image.getGraphics();
try {
    //1
    code that might throw exceptions
    //2
} catch (IOException e) {
    //3
    show error dialog (// some code which may throw exceptions)
    //4
} finally {
    g.dispose(); (// some code which will not throw exceptions)
    //5
} //6
```

  - No exception is thrown: 1, 2, 5, 6
  - An exception is thrown and caught by the `catch` clause
    - The `catch` clause doesn't throw any other exception: 1, 3, 4, 5, 6
    - The `catch` clause throws an exception itself: 1, 3, 5 , and the exception is thrown back to the caller of this method
  - An exception is thrown but not caught by the `catch` clause: 1, 5, and the exception is thrown back to the caller of this method

`finally` clause (2)

- You can use a `finally` clause without a `catch` clause
- Sometimes the `finally` clause can also thrown an exception

Example

```
public boolean searchFor(String file, String word)
  throws StreamException
{
    Stream input = null;
    try {
        some code which may throw an StreamException
    } finally {
        input.close(); // this may throw an IOException
    }
}
```

  - If the try clause throws a `StreamException` and the `finally` clause throws an `IOException`, the original exception is lost and the `IOException` is thrown out instead – a situation you'd better avoid

The I/O package - overview

- The `java.io` package defines I/O in terms of *streams* – ordered sequences of data that have a *source* (input streams) or a *destination* (output streams)
- Two major parts:
  1. *byte streams*
     - 8 bits, data-based
     - input streams and output streams
  2. *character streams*
     - 16 bits, text-based
     - readers and writers
- Check out Java API documentation for details about java.io

Byte streams

- Two parent `abstract` classes: `InputStream` and `OutputStream`
- Reading bytes:
  - `InputStream` class defines an abstract method

    ```
    public abstract int read() throws IOException
    ```

    - Designer of a concrete input stream class overrides this method to provide useful functionality.
    - E.g. in the `FileInputStream` class, the method reads one byte from a file
  - `InputStream` class also contains nonabstract methods to read an array of bytes or skip a number of bytes
- Writing bytes:
  - `OutputStream` class defines an abstract method

    ```
    public abstract void write(int b) throws IOException
    ```
  - `OutputStream` class also contains nonabstract methods for tasks such as writing bytes from a specified byte array
- Close the stream after reading of writing to it to free up limited operating system resources by using `close()`

Example code1:

```
import java.io.*;
class CountBytes {
   public static void main(String[] args) throws IOException {
      FileInputStream in = new FileInputStream(args[0]);
      int total = 0;
      while (in.read() != -1)
         total++;
      in.close();
      System.out.println(total + " bytes");
   }
}
```

Example code2:

```
import java.io.*;
class TranslateByte {
   public static void main(String[] args) throws IOException {
      byte from = (byte)args[0].charAt(0);
      byte to = (byte)args[1].charAt(0);
      int x;
      while((x = System.in.read()) != -1)
         System.out.write(x == from ? to : x);

   }
}
```

If you run "`java TranslateByte b B`" and enter text `bigboy` via the keyboard the output will be: `BigBoy!`

Character streams

- Two parent `abstract` classes for characters: `Reader` and `Writer`. Each support similar methods to those of its byte stream counterpart–`InputStream` and `OutputStream`, respectively
- The standard streams—`System.in`, `System.out` and `System.err`—existed before the invention of character streams. So they are byte streams though logically they should be character streams.

Conversion between byte and character streams

- The conversion streams `InputStreamReader` and `OutputStreamReader` translate between Unicode and byte streams
  - `public InputStreamReader(InputStream in)`
  - `public InputStreamReader(InputStream in, String encoding)`
  - `public OutputStreamWriter(OutputStream out)`
  - `public OutputStreamWriter(OutputStream out, String encoding)`
- `read` method of `InputStreamReader` read bytes from their associated `InputStream` and convert them to characters using the appropriate encoding for that stream
- `write` method of `OutputStreamWriter` take the supplied characters, convert them to bytes using the appropriate encoding and write them to its associated `OutputStream`
- Closing the conversion stream also closes the associated byte stream – may not always desirable

Working with files

- Sequential-Access file: the `File` streams—`FileInputStream`, `FileOutputStream`, `FileReader` and `FileWriter`—allow you to treat a file as a stream to input or output sequentially
  - Each file stream type has three types of constructors
    - A constructor that takes a `String` which is the name of the file
    - A constructor that take a `File` object which refers to the file
    - A constructor that takes a `FileDescriptor` object
- Random-Access file: `RandomAccessFile` allow you to read/write data beginning at the a specified location
  - a *file pointer* is used to guide the starting position
  - It's not a subclass of `InputStream`, `OutputStream`, `Reader` or `Writer` because it supports both input and output with both bytes and characters

Example of RandomAccessFile

```
import java.io.*;
class Filecopy {
   public static void main(String args[]) {
      RandomAccessFile fh1 = null;
      RandomAccessFile fh2 = null;
      long filesize = -1;
      byte[] buffer1;

      try {
         fh1 = new RandomAccessFile(args[0], “r”);
         fh2 = new RandomAccessFile(args[1], “rw”);
      } catch (FileNotFoundException e) {
         System.out.println(“File not found”);
         System.exit(100);
      }

      try {
         filesize = fh1.length();
         int bufsize = (int)filesize/2;
         buffer1 = new byte[bufsize];
         fh1.readFully(buffer1, 0, bufsize);
         fh2.write(buffer1, 0, bufsize);
      } catch (IOException e) {
         System.out.println("IO error occurred!");
         System.exit(200);
      }
   }
}
```

The `File` class

- The `File` class is particularly useful for retrieving information about a file or a directory from a disk.
  - A `File` object actually represents a path, not necessarily an underlying file
  - A `File` object doesn't open files or provide any file-processing capabilities
- Three constructors
  - `public File( String name)`
  - `public File( String pathToName, String name)`
  - `public File( File directory, String name)`
- Main methods
  - `boolean canRead() / boolean canWrite()`
  - `boolean exists()`
  - `boolean isFile() / boolean isDirectory() / boolean isAbsolute()`
  - `String getAbsolutePath() / String getPath()`
  - `String getParent()`
  - `String getName()`
  - `long length()`
  - `long lastModified()`

Add more efficiency

- `BufferedReader` reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

```
BufferedReader (Reader in)
```

- For example:
  - ✧ to wrap an `InputStreamReader` inside a `BufferedReader`

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in));
```

  - ✧ to wrap a `FileReader` inside a `BufferedReader`

```
BufferedReader in
  = new BufferedReader(new FileReader("fileName"));
```

  then you can invoke `in.readLine()` to read from the file line by line

```
import java.io.*;
public class EfficientReader {
   public static void main (String[] args) {

      try {
         BufferedReader br = new BufferedReader(new FileReader(args[0]));

         // get line
         String line = br.readLine();

         // while not end of file… keep reading and displaying lines
         while (line != null) {
              System.out.println("Read a line:");
              System.out.println(line);
              line = br.readLine();
         }
         // close stream
         br.close();
      } catch(FileNotFoundException fe) {
              System.out.println("File not found: “+ args[0]");
      } catch(IOException ioe) {
              System.out.println("Can’t read from file: “+args[0]);
      }
   }
}
```

Supplemental reading

- Handling Errors with Exceptions
  http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html
- Reading and Writing
  http://java.sun.com/docs/books/tutorial/essential/io/index.html
- File Access and Permissions
  http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava1/data.html