java.util

- Defines a collection of useful utility classes.
- `Date`
  - represents date/time information (millisecond representation)
- `Calendar` (Java 1.1)
  - uses more familiar units - months, days, hours, minutes
- `Stack`
  - LIFO stack
- `Random`
  - generates/returns pseudo-random numbers
- `Vector`
  - dynamic array of objects (grows as needed)
  - Example:

```
Vector foo = new Vector();
foo.addElement("dog");
foo.addElement("cat");
int len = foo.size();
```

java.util

- `Hashtable`
  - implements a hashtable (associative array)
  - objects can be stored/retrieved using arbitrary keys
  - Example:

```
Hashtable foo = new Hashtable(50);
foo.put("key", "value");
String val = (String)foo.get("key");
```

- `StringTokenizer`
  - parses a string into tokens
  - Example:

```
String foo = "A short string";
StringTokenizer bar = new StringTokenizer(foo);
String first = bar.nextToken();
```

java.lang

- `Boolean, Character, Byte, Short, Integer, Long, Float, Double`
  - wrapper classes for primitive types
- `Math`
  - provides methods for floating point calculations
  - `sin, cos, tan, sqrt` ...
- `Thread`
  - support for multiple threads of control within same Java interpreter
- `Throwable, Exception, Error`
  - error/exception objects
- `String`
  - immutable string type; methods do not modify string
  - Example:

```
String foo = "A short string";
int len = foo.length();
String bar = foo.substring(2, 7);
```

java.lang

- StringBuffer
  - string whose contents can be modified
  - grows in length as necessary
  - setCharAt(), append(), insert() methods modify string contents.
  - Example:

```
StringBuffer foo = new StringBuffer();
foo.append("A");
foo.append(" test");
foo.reverse();
String bar = foo.toString();
```

- System
  - implements standard system streams: stdin, stdout, stderr
    - System.in (InputStream)
    - System.out (PrintStream)
    - System.err (PrintStream)

`java.io`

- Majority of input/output is via *streams*.
  - Can be local (memory/file) or remote (network).
  - NB. Exception - `RandomAccessFile` class.
- What is a *stream*?
  - an object from which data can be read sequentially or to which data can be written sequentially.
- Important that I/O code should implement suitable error/exception handling.

- Reading/writing byte streams:
  - `InputStream`, `OutputStream` (& their subclasses)
- Reading/writing character streams:
  - `Reader`, `Writer` (& their subclasses)

- Methods: `read()`, `write()`, `close()` ...

`java.io` - Byte Streams

- `FileInputStream, FileOutputStream`
  - read/write bytes to/from file
- `ByteArrayInputStream, ByteArrayOutputStream`
  - read/write bytes to/from array of bytes in memory
- `PipedInputStream, PipedOutputStream`
  - Work together to implement "pipes" for communication between threads.
- `FilterInputStream, FilterOutputStream`
  - filter input/output bytes
  - subclasses of these classes must be created to implement filters
- `BufferedInputStream, BufferedOutputStream`
  - subclasses of filter stream classes
  - provide I/O buffering

`java.io` - Byte Streams

- Example:

```
File fileObj = new File("wibble.txt");
FileInputStream buf = new FileInputStream(fileObj);
char chr = (char)buf.read();
```

- `File`
  - Class which represents file or directory.
  - Methods: `delete()`, `exists()`, `isFile()`, `isDirectory()` ...
  - Does **not** implement methods for manipulating file contents.

`java.io` - Character Streams

- These streams supersede the byte streams for character I/O.
- `InputStreamReader, OutputStreamWriter`
    - read/write characters to/from byte streams (doing appropriate character conversions)
- `FileReader, FileWriter`
    - read/write characters to/from file;
    - automatically create `FileInputStream/FileOutputStream` (as appropriate)
- `PipedReader, PipedWriter`
- `FilterReader, FilterWriter`
- `BufferedReader, BufferedWriter`
    - `BufferedReader` supports all standard `Reader` methods, plus `readLine()`.

`java.io` - Character Streams

- Examples:

```
FileWriter file = new FileWriter("wibble.txt");
file.write('A');
file.close();

BufferedReader buf = new BufferedReader(new
   InputStreamReader(System.in));
String line = buf.readLine();
```

java.io - RandomAccessFile

- Read/write to/from arbitrary locations within a file.
- Handles bytes, text and primitive data types.
- Independent of all other I/O classes.
- Methods: `read()`, `write()`, `readLine()`, `seek()` ...
  - `seek()` - used to select position in file at which to read/write.
- Example:

```
RandomAccessFile foo = new
     RandomAccessFile("wibble.txt", "rw");
foo.seek(30);
String bar = foo.readLine();
```

Error Handling & Exceptions

- Intercepting errors makes for robust code.
- Java handles errors via *exceptions*.
  - An exceptional event that occurs during program execution and disrupts normal flow of control.
- Exceptions are objects.
  - Instances of some subclass of: `java.lang.Throwable`
  - 2 standard subclasses:
    - `java.lang.Error`
      - Dynamic loading, memory problems, etc.
      - Treat as unrecoverable.
    - `java.lang.Exception`
      - Subclasses normally can be caught.
        - `java.io.EOFException`
          `java.lang.ArrayAccessOutOfBounds`

Throwing & Catching Exceptions

- *Throwing* an exception
  - Signal an exceptional condition.
- *Catch* an exception
  - Handle the exception, i.e. take action(s) to recover from it.

- Exceptions propagate up though block structure of a Java method and then up method call stack.
  - If exception not caught by block that throws it - propagates to next higher enclosing block and so on.
  - If not caught in method, propagates to invoking method and onwards through its block structure ...
- If no suitable handler available - system terminates.

try/catch/finally

- Used to handle exceptions within a method.
- Syntax:

```
try {
   // Code executes top to bottom
   // unless exception occurs (or break/continue/return)
}
catch (SomeException e1) {
   // Handles exception object e1 of type SomeException
   // or subclass of that type.
}
catch (AnotherException e2) {
   // Handles AnotherException or subclass.
}
finally {
   // Code here always executed
   // After completion of try block (no exceptions)
   // of after catch block executes, or exception
   // not handled, or after break/continue/return.
}
```

- First `catch` clause matching exception is executed.

Declaring Exceptions

- Any method which can cause a normal exception to occur must either:
  - Catch exception internally;
  - or, specify type of exception it may generate.
- `throws`
  - Keyword used to specify exception in method declaration.
- Example:

```
public void open_file() throws IOException {
   // Code here could generate java.io.IOException
}
```

  - Exception specified may be superclass of type actually thrown.

Generating Exceptions

- `throw`
  - Keyword used to generate exceptions within code.
  - `throw <ExceptionObject>`
- Example:

```
if (!fn.canRead())
   {throw new IOException();}
```

Defining New Exception Types

- Create a subclass of an existing exception.
- `Throwable` class includes a String - stores error message.
- Constructors needed:
  - No arguments;
  - One String argument.
- Example:

```
class MyException extends Exception {
   public MyException() { super(); }
   public MyException(String s) { super(s); }
}
```