Namespace/Package

We sometimes want to group related classes and interfaces together under a single umbrella.

We group them under a *namespace*.

In Java we call it a *package*.

Namespace also solves a lot of name conflicts. E.g., both my framework and your framework may have classes called `List`, but as long as they are under different packages they can coexist distinctly.

To put a class under a package `mypackage`, put this in the beginning of the source file:

```
package mypackage;
```

In addition, choose a directory *dir* in the classpath, create a directory *dir*/`mypackage`, and put the byte code under it.

Package Scoping

With the `package` construct comes additional scoping rules:

- Classes in the same package can use protected members of each other.
- In addition to the three accesses *public*, *private*, and *protected*, there is a fourth one, *package* access.
  - You declare this access by omitting the access modifier, e.g.,
    ```
    void foo() { ... }
    ```
  - Classes in the same package can use package members of each other.
- You can declare a class that is only visible to your package. Just omit `public`. E.g.,

  ```
  class Foo { ...
  ```


Using a Package

Now the converse: how do you call up a class in someone else's package?

- You can use the full name, e.g.,

  ```
  yourpackage.List r = new yourpackage.List();
  ```

- You can just import the name of that class: this is called *single type import*. Put this in the beginning of your source file:

  ```
  import yourpackage.List;
  ```

- You can import the whole namespace: this is called *import on demand*. Put this in the beginning of your source file:

  ```
  import yourpackage.*;
  ```

  Then all class names in `yourpackage` are imported.

Package Naming

Package names can contain dots. E.g., we can have a package called `mypackage.util`, and put the byte code under *dir*/`mypackage/util`.

This package is sometimes considered to be a *subpackage* of `mypackage`. However, this is just a mental picture. Java treats `mypackage` and `mypackage.util` as *completely distinct* packages with no relation at all.

All standard Java classes (almost) are under `java.*` or its subpackages, e.g., `java.lang`, `java.util`, etc.

The package `java.lang` is always implicitly imported.

It is recommended that your packages be named in the following way. Suppose the domain name of your organization is `treblacy.org`. Suppose your package is about geometry. Then you are advised to call it `ORG.treblacy.geometry`.

This hopefully gives you a globally unique name.

Input and Output

The basic I/O facilities of Java are in package `java.io`.

Three major hierarchies:

- byte streams: `InputStream` and `OutputStream`
- char streams: `Reader` and `Writer`
- I/O exceptions: `IOException`

Java uses 16-bit Unicode chars internally, while files and devices in the real world use 8-bit octets. How do we fill the gap?

With byte streams, you have to do the conversion separately, or just stick with bytes. Not convenient for internationalization.

With char streams, the conversion is integrated. They have a default converter, and you may also provide your converter.

InputStream

`InputStream` is an abstract class.

- `int read() throws IOException`

  Returns octet as non-negative number, or `-1` if EOF.

- `int read(byte[] buf) throws IOException`

  `int read(byte[] buf, int off, int len) throws IOException`

  Reads octets into array, returns count or `-1` if EOF.

- `long skip(long count) throws IOException`

  Skips octets, returns actual count.

- `int available() throws IOException`

  Counts octets available before blocking.

- `void close() throws IOException`

  Closes the stream.

Input Streams

Useful subclasses of `InputStream`:

- `FileInputStream` opens a file for reading.
- `ByteArrayInputStream` reads from the given byte array.

The following are wrappers: they wrap around your input streams and provide extra services.

- `SequenceInputStream` concatenates several input streams.
- `PushBackInputStream` provides "unreading".
- `BufferedInputStream` provides buffering.

OutputStream

`OutputStream` is an abstract class.

- `void write(int b) throws IOException`

  `void write(byte[] buf) throws IOException`

  `void write(byte[] buf, int off, int len) throws IOException`

  Write octet(s).

- `void flush() throws IOException`

  Flushes the buffer and the downstreams, if any, of the stream.

- `void close() throws IOException`

  Flushes and closes the stream.

Output Streams

Useful subclasses of `OutputStream`:

- `FileOutputStream` opens a file and writes to it.

- `ByteArrayOutputStream` writes into an internally created byte array. When you are done writing, obtain it with:

  `byte[] toByteArray()`

These are wrappers:

- `PrintStream` provides the `print` and `println` methods for printing various data types in human readable form. `System.out` and `System.err` are `PrintStream`'s.

- `BufferedOutputStream` provides buffering.

Reader

`Reader` is the abstract class of character input streams.

- `int read() throws IOException`

  `int read(char[] cbuf) throws IOException`

  `int read(char[] cbuf, int off, int len) throws IOException`

  Reads character(s).

- `long skip(long count) throws IOException`

  Skips characters.

- `boolean ready() throws IOException`

  Determines if the stream can be read without blocking.

- `void close() throws IOException`

  Closes the stream.

Useful Readers

These are useful subclasses of `Reader`:

- `FileReader`

- `CharArrayReader`

- `StringReader` reads from a string.

These are useful wrappers:

- `PushBackReader`

- `BufferedReader` provides buffering and read-a-line service:

  `String readLine() throws IOException`

- `LineNumberReader` subclasses `BufferedReader` counts lines for you as it reads:

  `int getLineNumber()`

  `void setLineNumber(int n)`

Writer

`Writer` is the abstract class of character output streams.

- `void write(int c) throws IOException`

  `void write(char[] cbuf) throws IOException`

  `void write(char[] cbuf, int off, int len) throws IOException`

  `void write(String s) throws IOException`

  `void write(String s, int off, int len) throws IOException`

- `void flush() throws IOException`

- `void close() throws IOException`

Useful Writers

These are useful subclasses of `Writer`:

- `FileWriter`

- `CharArrayWriter` writes into an internally created char array. Obtain it with:

  ```
  char[] toCharArray()
  ```

- `StringWriter` writes into an internally created string buffer. Obtain it with `toString` or use

  ```
  StringBuffer getBuffer()
  ```

These are useful wrappers:

- `PrintWriter`

- `BufferedWriter`