Chapter 7: Command line Argument Processing

Parameter	Details
args	The command line arguments. Assuming that the `main` method is invoked by the Java launcher, `args` will be non-null, and will have no **`null`** elements.

Section 7.1: Argument processing using GWT ToolBase

If you want to parse more complex command-line arguments, e.g. with optional parameters, than the best is to use google's GWT approach. All classes are public available at:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

An example for handling the command-line `myprogram -dir "~/Documents" -port 8888` is:

```
public class MyProgramHandler extends ToolBase {
   protected File dir;
   protected int port;
   // getters for dir and port
   ...

   public MyProgramHandler() {
       this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }
       });
       this.registerHandler(new ArgHandlerInt() {
            @Override
            public String[] getTagArgs() {
               return new String[]{"port"};
            }
            @Override
            public void setInt(int value) {
               this.port = value;
            }
       });
   }
   public static void main(String[] args) {
      MyProgramHandler myShell = new MyProgramHandler();
      if (myShell.processArgs(args)) {
         // main program operation
         System.out.println(String.format("port: %d; dir: %s",
            myShell.getPort(), myShell.getDir()));
      }
      System.exit(1);
   }
}
```

`ArgHandler` also has a method `isRequired()` which can be overwritten to say that the command-line argument is required (default return is **`false`** so that the argument is optional.

Section 7.2: Processing arguments by hand

When the command-line syntax for an application is simple, it is reasonable to do the command argument

processing entirely in custom code.

In this example, we will present a series of simple case studies. In each case, the code will produce error messages if the arguments are unacceptable, and then call `System.exit(1)` to tell the shell that the command has failed. (We will assume in each case that the Java code is invoked using a wrapper whose name is "myapp".)

A command with no arguments

In this case-study, the command requires no arguments. The code illustrates that `args.length` gives us the number of command line arguments.

```
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.err.println("usage: myapp");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked");
    }
}
```

A command with two arguments

In this case-study, the command requires at precisely two arguments.

```
public class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: myapp <arg1> <arg2>");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked: " + args[0] + ", " + args[1]);
    }
}
```

Note that if we neglected to check `args.length`, the command would crash if the user ran it with too few command-line arguments.

A command with "flag" options and at least one argument

In this case-study, the command has a couple of (optional) flag options, and requires at least one argument after the options.

```
package tommy;
public class Main {
    public static void main(String[] args) {
        boolean feelMe = false;
        boolean seeMe = false;
        int index;
        loop: for (index = 0; index < args.length; index++) {
            String opt = args[index];
            switch (opt) {
            case "-c":
               seeMe = true;
               break;
            case "-f":
```

```
                feelMe = true;
                break;
            default:
                if (!opts.isEmpty() && opts.charAt(0) == '-') {
                    error("Unknown option: '" + opt + "'");
                }
                break loop;
            }
        }
        if (index >= args.length) {
            error("Missing argument(s)");
        }

        // Run the application
        // ...
    }

    private static void error(String message) {
        if (message != null) {
            System.err.println(message);
        }
        System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
        System.exit(1);
    }
}
```

As you can see, processing the arguments and options gets rather cumbersome if the command syntax is complicated. It is advisable to use a "command line parsing" library; see the other examples.