Using arguments from the command line

Many Java applications take input from the user. Often, this input is provided interactively, while the application is running—but sometimes, it's sufficient to provide all of the necessary input when launching the application; this second option can be implemented via command line arguments.

The Java application launcher command line

To review, the basic syntax when running a Java application via the Java application launcher is

```
java [options] classname [args]
```

(Remember: The square brackets indicate that the contents are optional.)

Command-line arguments

The `main` method of a Java application class is always declared with a single parameter, an array of `String` objects. If no command line arguments are specified to the Java application launcher, an array with zero elements will be passed to `main`; otherwise, the contents of the array will be populated automatically: The first argument will be the first element of the array (at index position 0); the second argument will be the second element of the array (at index position 1); and so on.

There are some important implications of this approach:

- Java arrays are homogeneous and strongly typed; thus, even if a command line argument is a number (for example), it will be passed as a `String` in the array. If our application needs to treat such a value as a number, it will have to perform any necessary conversion.
- Command line arguments are delimited by spaces. For example, in

  ```
  java SomeClass A sentence made up of several words.
  ```

  There are actually 7 command line arguments specified for the Java application; these will be passed to the `main` method of `SomeClass` in an array with the contents `{"A", "sentence", "made", "up", "of", "several", "words."}`. If the desired behavior is to treat the entire sentence (in this case) as a single argument, it must be enclosed in quotes on the command line, e.g.

  ```
  java SomeClass "A sentence made up of several words."
  ```

  In this invocation, the command line arguments passed to the `main` method of `SomeClass` will consist of an array with a single element: `{"A sentence made up of several words."}`.

Note: The `length` field of any array can be used to find out how many elements are in that array. So (for example), if the `String[]` parameter in the declaration of `main` is called `args`, then `args.length` is the number of elements in the `args` array. Thus, if no arguments are passed on the command line, `args.length` will have a value of zero (and any attempts to read or write a value to `args[0]`, `args[1]`, etc. will result in an error). If we're expecting arguments for a given application to be passed on the command line, and we're not using a library to handle and parse them (e.g. Apache Commons CLI), it's essential that we check the length of the array passed to `main` before we try to read any of the values in that array.

Iterating over command line arguments

In general terms, *iteration* is the repetition of one or more tasks for every item in a collection. Though we haven't paid much attention to it, the array of strings passed to the `main` method is (again in general terms) such a collection. So, though we'll look at arrays and iteration in much more detail in the near future, we're going to get a small taste of them now, by looking deeper at the command line arguments.

The first thing to remember is that the single parameter to the `main` method is an array of `String` objects. So while command line arguments may be numeric in nature, they will nonetheless be passed as strings to `main`. Related to that is the fact that arrays in Java (as in many other languages) are *homogeneous* – that is, they can only contain elements of the same type. So, for example, we can have a `String[]` (an array of `String` objects), or an `int[]` (an array of `int` values), but not an array that contains both `String` and `int`. (There are other kinds of collections that aren't as restrictive; however, as a rule, we should be wary of mixing very different types in a collection.)

There are 2 basic ways to iterate over an array in Java:

- Use a traditional *for* loop, where we use a counter that (generally) starts at 0 and ascends to *`array`*`.length - 1` (the index of the last element in the array) in increments of 1. For each value of the counter, we retrieve and process the item at the corresponding index position in the array.

  In Java, this method uses the `for` statement, which has the syntax

  ```
  for (initialize; condition; update)
    statement;
  ```

  or

  ```
  for (initialization; condition; update) {
    statement;
    ...
  }
  ```

  For most purposes, the 2nd form (which allows multiple statements to be executed iteratively) is preferable. It is far less prone than the 1st to inadvertent errors that can be introduced when modifying existing code.

In the above *`initialize`*, *`condition`*, and *`update`* are *placeholders*: We will replace them with the appropriate variable declaration, assignment, test, and update statements for our purposes.

- *`initialize`* is used to initialize the value of the counter variable; we usually declare the counter variable itself as well (i.e. we declare and use a new variable for the counter, rather than re-using an already declared variable).
- *`condition`* is a a Boolean-valued expression (i.e. one that can be evaluated as `true` or `false`) that is tested prior to each iteration. If it evaluates to `true`, iteration proceeds; otherwise, iteration halts (or never even starts), and execution jumps to the next statement after the `for` statement. Typically, this condition tests the counter variable to ensure that it is still within the limits of the array.
- *`update`* is a statement used to update the counter variable – usually incrementing it by 1.

To iterate over a `String[]` variables named `args` (such as the array of command line parameters passed to the `main` method), we could do the following:

```
for (int i = 0; i < args.length; i++) {
  System.out.printf("Arg # %d is %s%n", i, args[i]); // Or whatever.
}
```

- Use an *enhanced for* loop, also called a *for-each* loop, where we iterate over the array without using a counter. This feature has been available in Java since Java 5, and has a simple, elegant syntax. In general, this is the preferred method, *unless we need to update the array contents or use the counter for something other than accessing the array elements*.

  The syntax of the *for-each* loop is similar to that of the regular *for* loop, but no counter variable is involved. Instead, we simply specify the collection to iterate over, and a variable to hold each value in turn:

  ```
  for (variable-declaration : collection)
    statement;
  ```

  or

  ```
  for (variable-declaration : collection) {
    statement;
    ...
  }
  ```

  Again, we have some placeholders in this syntax:

  - *`variable-declaration`* is exactly what it says: We declare a local variable that will hold each value of the collection in turn, as we iterate over the collection. So, for example, if we're iterating over a `String[]`, we would declare a `String` variable here. Note the difference between this variable and the one used for the previous version of the `for` statement: in the

previous, the variable used is an `int`-type counter, and values are retrieved from the array by using the counter as an index into the array; in this form, the variable is the same types as the array elements, and each element is automatically assigned to this variable as iteration proceeds.

- *`collection`* is the array (or other type of collection) over which we're iterating.

For example, we could iterate over the `String[]` argument passed to the `main` method this way:

```
for (String arg : args) {
  System.out.printf("Argument: %s%n", arg); // Or whatever.
}
```

Note that we don't have a counter variable to include in our processing in this type of iteration; we only have a copy of each array element in turn. A non-obvious (and non-trivial) implication of this is that—depending on the array's element type—we may not be able to modify the array contents as we iterate over it. For example, if the array elements are of an intrinsic type (e.g. `int`, `float`, `char`), each value will be copied in turn into the local variable; if we change the value of the local variable, it has no effect on the source of that value (the array elements). If the array elements are object references, then the copy in the local variable is also a reference to an object, and we may be able to change object's state—unless the references are to immutable objects, such as `String`.

References

- H. Schildt, "More Data Types and Operators," Java: A Beginner's Guide (8th ed.), ch. 5, pp. 135–184.
- Apache Commons™ CLI™
- Java SE API Documentation: `String.format(String format, Object… args)`
- Java Language Specification: Conditional Operator