Java Essentials

Language Basics

Java is an object-oriented programming language providing features that support

- Data abstraction
- Code reuse
- Modular development of large systems
- Automated low-level memory management
- Type-safety & Security
- Error-handling
- Multiple threads of computation
- Portability

While its syntax can be occasionally awkward, the benefits of the language over time have substantially outweighed its weaknesses. This purpose of this document is to briefly summarize the essential elements of Java.

Executing a Java program that has been entered into a text file, say MyProgram.java, using a Unix command shell (a program for entering commands in the Unix operating system) requires two steps.

Compiling  `javac MyProgram.java`

Running  `java MyProgram`

The `javac` command converts the Java source code to an intermediate low-level language called Java bytecode. The bytecode will be placed into a file called MyProgram.class, (a Java class file). Assuming that the source code contained a method (aka procedure/function) called `main`, the `java` command will execute the code in the `main` method.

Example: Hello World.

There’s a lot of syntactic overhead involved in writing this simple program. This is not an uncommon situation with programming languages that are designed for writing large, modular systems. You’ll get used to it quickly.

The next few topics talk about the non-object-oriented features of Java. After covering them, we’ll revisit them from the perspective of object-oriented programming.

Data Types in Java

Programs store and manipulate data (sometimes called state). Data is held in variables. Java requires that every variable be declared, that is, be given an explicit type, before it can be used. For example:

```
int age;       // A simple integer value
float speed;  // A numeric value with a fractional component
char grade;   // A single character
String name;  // A sequence of characters
int[] scores; // A one-dimensional array of integers
```

A variable can be declared without being assigned a value, but a value can be provided in the declaration:

```
int age = 20;  // Assigns 20 to the variable age
float speed = 3.14f; // Assigns 3.14 to the variable speed
```

*Note that we omit the `.class` file suffix.*
int age = 21; // A simple integer value
float speed = 47.25; // A numeric value with a fractional component
char grade = 'A'; // A single character
String name = "Zeta"; // A sequence of characters
int[] scores = {97, 85, 100}; // A one-dimensional array of integers

Java provides four categories of types: 

*primitive types*, *array types*, *class types* and *enum types*.

### Primitive Types

The primitive types are:

- A collection of types for storing integer values, including: 
  `byte, short, int, long`
- Two types for storing decimal values, including:  
  `float, double`
- A type for logical values: `boolean`. A variable of type `boolean` has two possible values: `true` and `false`
- A type for storing a single character: `char`

All other types are essentially built upon these primitive types in one of two ways. An *array type* is a fixed-sized collection of values of some other type. Each element of the collection is accessed using an integer *index*:

```
int[] scores = {97, 85, 100}; // A one-dimensional array of integers
System.out.println(scores[1]); // Prints the value 85: arrays start at index 0
```

Each of these primitive share some characteristics:

- Simple storage requirements: A value has a fixed size that can fit into a single word (or less) of memory
- Values can be explicitly represented in the language: `37`, `-41`, `3.14159`, `false`, `’Z’, ’w’, ....` These representations of specific values are called *literals*.

The `String` type is *not* primitive, although is unique among the non-primitive types that its values can be represented in Java by literals: "Hi there!", "Is 3 < 2?", "gibberish". We’ll talk a lot more about strings later.

### Array Types

All values stored in an array must be of the same type[*] as the declared element type of the array.

Example: Sum: A program to add two numbers:

- Version 0: Numbers are hard-coded
- Version 1: Numbers are stored in variables
- Version 2: Items are read using a `Scanner`
- Version 3: Items are read using `args[]` parameter to `main`
- Version 4: More than two arguments are allowed; length of array is used in for loop
- Version 5: Iterator version of for loop is used.

The Java class `Arrays` provides several utility methods for copying, searching, and sorting arrays.

### Operators

Java provides a number of built-in *operators* (functions represented by 1 or 2 symbols) that perform basic computations. Here is a (non-exhaustive) list of frequently used binary (two argument) operators:

**Arithmetic binary operators** `+, -, *, /, %`

[*] Or, for class types, must be a subtype of that type.
Relational: ==, !=, <, <=, >, >=

Logical: &&, || (note: don't confuse with & and |)

Assignment: =, +=, -=, *=, /=, ...

Common unary operators include

Arithmetic: -, ++, – (prefix and postfix)

Logical: !

Notes

- There is no exponentiation operator in Java. The symbol `^`, often used for exponentiation in other languages, is the bitwise or operator in Java.
- For positive arguments, the remainder operator `%` is the same as the mathematical `mod` function, but for negative arguments it is not. For example, \(-8 \mod 3 = 1\) while \(-8\%3 = 2\). Here's the rule: \(c = a\%b\) means
  1. \(a = kb + c\),
  2. \(0 \leq |c| < |b|\), and
  3. \(\text{sign}(c) = \text{sign}(a)\) (i.e.: \(a\) and \(c\) are both positive, both negative, or both 0)
- Java provides an unambiguous operator precedence for all of its operators. Higher precedence operators (e.g., * or /) are evaluated before lower precedence operators (e.g., + or -); operators of equal precedence are evaluated from left to right. For example, \(3 + 2 * 5 - 7 / 4\) is equivalent to \(((3 + (2 * 5)) - (7/4))\).
- The logical operator && has precedence over ||; also, these operators use short-circuit evaluation: that is, once the value of the logical expression can be determined, no further evaluation takes place. For example, if \(x\) and \(y\) are boolean variables and \(x\) has the value false, the expression \(x && y\) will return false without evaluating \(y\). This is useful in expressions such as \((n \neq 0) && (k/n > 3)\), since it will prevent division by 0 if \(n\) equals 0.

Expressions and Control Structures in Java

Java allows the computation of values by building expressions. Expressions use variables, operators, and method calls to produce a value—the *value of the expression*. This value can then be assigned to a variable or passed to/returned from a method. Expressions that produce a boolean value (true/false) are called *boolean (or logical) expressions* and can be used with conditional or looping constructs to determine the flow of control of the program. Some examples:

```java
// Assume x has been declared as an int and y as a double
if (x != 0) y = 1.0/x;
else y = 0;

while (x > 0) {
    System.out.println("x is positive");
    x--; // decrement x
}
```

There are two conditional statements: *if* and *switch*. The *if* statement has the form

```java
if (booleanExpression) // There is exactly 1 "if" clause
    statement;
else if (booleanExpression) // There can be 0 or more "else if" clauses
    statement;
else // There can be at most 1 "else" clause
    statement;
```
The single "statement" can be replaced with a block—a sequence of statements enclosed in brackets:

```java
if (booleanExpression) {
    statement1;
    statement2;
    ...
}
```

In many situations, the condition being tested is the value of a variable of type `int`. In this case, one can use the `switch` statement:

```java
// Assume that x has been declared to be an int
switch (x) {
    case 0:
        System.out.println("Your card is a club");
        break;
    case 1:
        System.out.println("Your card is a diamond");
        break;
    case 2:
        System.out.println("Your card is a heart");
        break;
    case 3:
        System.out.println("Your card is a spde");
        break;
    default:
        System.out.println("Illegal value!");
        break;
}
```

Switch statements can also be used for enumeration types and (with some restrictions) Strings.

In addition to the `if` and `switch` statements, which provide for conditional flow of control, Java also provides several statements that provide for repeated execution of a block of code loops. We’ve already seen one of these, the `for` statement. We’ll come back to it shortly, but first let’s see the two simpler looping constructs: the `while` and `do-while` statements.

There are also two flavors of `while` loop:

- `while ( continue? ) statement;`
- `do statement while ( continue? );`

The latter form is guaranteed to execute the statement at least once, since the check happens after the statement.

The `for` statement has two basic forms

- `for ( oneTimeInitialization, continue?, updateWork ) statement;`
- `for ( arrayValueType identifier : arrayName ) statement;`

The latter form is frequently used in situations in which we wish to access (but not modify) each element of the array in order.

The other fundamental control flow constructs provided by Java are the `break`, `continue`, and `return` statements. The `break` statement is used to within a `switch`, `for`, `while`, or `do-while` control structure to exit that structure and to transfer control to the first line of code after the exited structure. If instead, the goal is to stop executing the `current iteration` of a loop and begin the next iteration of the loop, then the `continue` statement is

---

1 There is no point to allowing a `continue` statement in a `switch` structure, just as there’s no point to allowing a `break` statement in an `if` structure.
used. Finally, as we’ll see later, the `return` transfers control from an executing method back to the code that invoked that method.

So far, we have the ability to write programs that use primitive types and arrays using the `main()` method of a class. If such a program gets to be too long, we can decompose it into smaller pieces using methods. Here’s an example of a program that reads a list of integers from the command line and finds the smallest one.

```java
// Sample code using static methods

class StaticMethods {

    // Sample code using static methods
    
    class StaticMethods {

        public static int posOfMin(int[] nums) {
            int pos = 0;
            for (int i = 1; i < nums.length; i++)
                if (nums[i] < nums[pos]) pos = i;
            return pos;
        }

        public static void main(String[] args) {
            // quit if no integers entered on command line
            if (args.length == 0) System.exit(0);

            // create a list of integers
            int[] numbers = new int[args.length];
            int i = 0;
            // Convert each command line (String) value to an int
            for (String s : args) numbers[i++] = Integer.valueOf(s);

            // Find the location in the array of the smallest int
            int loc = posOfMin(numbers);
            System.out.println("The smallest value is " + numbers[loc] + "; it is in position " + loc);
        }

    }

    // Sample code using static methods

}
```

We can create as many methods as we need to keep each method at a manageable (easy to understand, debug, maintain) size. Like `main()`, `posOfMin()` is declared `static`. We’ll explain why in the next handout. Also, `posOfMin()` computes and returns a value: the location in the array of the smallest element. We indicate that the method returns an integer value by using the Java keyword `int` just before the name of the method.

Much of the real power of Java (and other object-oriented languages) stems from the ability to supplement the primitive types with types of our own making. In Java, these are called `class types` and they are the topic of the next handout.