Chapter 1

From C to Java

In order to attract developers, the Java creators gave the language a C/C++-like syntax. While we ultimately have to be mindful that C favors one programming paradigm—procedural programming—and Java favors another—object-oriented programming, we can leverage our knowledge of C syntax to quickly ramp up to Java.

In this chapter, we introduce the elements of C programming that translate to Java. Recall that procedural programming is a programming paradigm where a program is composed of procedural calls, i.e., functions with side-effects. Initially, we will construct Java programs that are composed of functions that contain variables, mutation, loops, etc. After we have acquainted ourselves with procedural programming in Java, we will migrate to a more object-oriented style of programming.

1.1 Statements and Expressions

A programming language is made up different language constructs that we can put together in different ways. In C as well as Java, there are two major sorts of constructs that do work in our programs:

- **Expressions** are programming language constructs that evaluate to a value. A value is simply an expression that can no longer take any more steps of evaluation.
- **Statements** are programming language constructs that carry out one or more side-effects. Side-effects include reading from a file, printing to the console, or mutating a variable.

An example of an expression is an arithmetic expression, e.g., \(3 + 5 * (4 - 2)\), which evaluates to the value 13. An example of a statement is a variable assignment, e.g., \(x = 5\), which has the effect of copying the value 5 into the variable \(x\).

We combine statements and expressions to create programs that do meaningful work. For example, the for-loop:

```java
int prod = 0;
for (int i = 1; i <= 10; i++) {
    prod = prod * i;
}
```
repeatedly updates the prod variable with the results of the expression \( \text{prod} \times 2 \). After the loop is done, \( \text{prod} \) effectively contains the result of evaluating \( 1 \times 2 \times \ldots \times 10 \) or 10!.

### 1.1.1 Expressions

Java inherits a large part of its expression language from C. Expressions include literals for several types:

- Integers, e.g., 5,
- Floating-point values, e.g., 3.5,
- Booleans, e.g., \texttt{true},
- Characters, e.g., 'c', and
- Strings, e.g., "hello world".

We can also perform:

- Arithmetic over integer and floating-point expressions with +, -, *, [], and %.
- Comparisons over integers and floating-point expressions with >, >=, <, <=, ==, and !=.
- Boolean arithmetic over booleans with !, &&, and ||.
- Bitwise operations over integer values with &, |, ^, ~, >>, <<, and >>. 
- Function calls that produce or return values.

Expressions generalize arithmetic expressions, computing by repeated evaluation and substitution. Recall that an arithmetic expression evaluates by repeatedly:

1. Finding the next sub-expression to evaluate by applying rules of precedence, e.g., multiplication comes before addition, parenthesized expressions take precedence over non-parenthesized expressions.
2. Evaluating that sub-expression to a value.
3. Substituting the newly acquired value for that sub-expression.

This process continues until the resulting expression is a value. A value is an expression that takes no further steps of evaluation. In this context, a number itself is an expression that does not evaluate further, so it is considered a value.

Arithmetic expressions consist of a single type, e.g., integers, and a small set of expression forms, e.g., addition and subtraction. Expressions in Java expand on the set of possible types as well as expression forms. Otherwise, they behave exactly like arithmetic expressions with respect to how they compute. For example, here is a more complex expression and its step-by-step evaluation to a final value:

\[
\neg(5 \times 3 > 3 + 2 \&\& !\text{true} || \text{false}) \\
\rightarrow \neg(15 > 3 + 2 \&\& !\text{true} || \text{false}) \\
\rightarrow \neg(15 > 5 \&\& !\text{true} || \text{false}) \\
\rightarrow \neg(\text{true} \&\& !\text{true} || \text{false})
\]
1.1.2 Statements

Java also inherits much of its statement language from C. In particular, Java features:

- **Local variable declaration statements**, e.g., `int x;`.

- **Variable assignment statements**, e.g., `x = 5;`. Note that we can combine variable declaration and initial assignment statements, e.g., `int x = 5;`, and like C, we favor always initializing our locals when they are declared. Java also features the same set of “shortcut” assignment statements as C, e.g., “assign equals” operators such as `+=` and pre- and post-increment and decrement operators `++` and `--`.

- **Conditional statements**, e.g.,

  ```java
  if (x < 5) {
    System.out.println("less than five");
  } else {
    System.out.println("not less than five");
  }
  ```

  where the statements constituting the if-branch are executed if the guard of the conditional (here, `x < 5`) evaluates to `true`. Otherwise, the guard evaluates to `false` and the statements of the else-branch are executed. Like C, you can elide the else branch or introduce multiple branches using `else if`.

- **While-loop statements**, e.g.,

  ```java
  while (x != 0) {
    System.out.println(x);
    x += 1;
  }
  ```

  where the statements that constitute the body of the loop are executed until the guard of the loop evaluates to `false`. Java also has the do-while loop statement where the while-loop body is evaluated once before the guard is evaluated.

- **For-loop statements**, e.g.,

  ```java
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
  ```

  which act like while-loops but combine initialization and updating of a variable along with repeated execution of the loop body.
Switch statements, e.g.,

```java
switch (x) {
    case 0:
        return 1;
    case 1:
        return 1;
    default:
        return x * 2;
}
```

which performs case analysis on a integral value and jumps to the statement after that value’s corresponding case label. Like C, we have to be careful about case fall-through with cases. The cases are not genuine branches like a conditional; they are merely labels. So unless you use a break statement to exit the switch, execution flows through subsequent cases.

For example, the following switch statement:

```java
switch (x) {
    case 0:
        x = x + 1;
    case 1:
        x = x + 1;
    default:
        x = x + 1;
}
```

If `x` contains the value 0, then the switch statement will increment `x` three times, eventually storing the value 3 inside of the variable. To fix this problem, the switch should introduce break statements at the end of each case “block” as follows:

```java
switch (x) {
    case 0:
        x = x + 1;
        break;
    case 1:
        x = x + 1;
        break;
    default:
        x = x + 1;
        break;
}
```

Unlike expressions, statements do not evaluate to a value. Instead, they produce side-effects on the program. A side-effect is any change of state to the program visible to the “outside world”. The canonical example of this is variable mutation, e.g., `x = 5` changes the value stored at `x` to
5. “State” is not restricted only to the values of variables. For example, writing to the console changes the state of the screen to display more text.

1.2 Types

Like C, Java is a statically-typed language. That is, every expression in our program has a type which classifies the value that it produces. This type is known to the compiler before you run a program, and the compiler performs an analysis called type checking to ensure that all values in our program are used in a consistent manner. Such programs are called well-typed. Programs that do not have this property are called ill-typed and produce type errors at compilation time.

Types themselves can be divided into two sorts: primitive and compound types. Compound types are made up of other, smaller types. In contrast, primitive types are atomic, they cannot be decomposed into smaller types like with compound types.

Java shares a number of primitive types with C although the set of values they classify differ slightly:

- **long**: type of 64-bit (8 byte) signed integers.
- **int**: the type of 32-bit (4 byte) signed integers.
- **short**: the type of 16-bit (2 byte) signed integers.
- **byte**: the type of 8-bit (1 byte) signed integers.
- **double**: the type of 64-bit (8 byte) floating point values.
- **float**: the type of 32-bit (4 byte) floating point values.
- **boolean**: the type of boolean values, i.e., true and false.
- **char**: the type of single 16-bit Unicode characters.

Notably, Java has a richer type system than C in the sense that a boolean is not simply an integer; a boolean is its own distinguished type that cannot be mixed up with numbers. In particular, in C, the expression 1 + true is well-typed (it’s resulting value is undefined) whereas in Java, the expression is ill-typed and would be rejected at compile time.

In C, aggregate types includes arrays, pointers (which generalize arrays), and structs. Java differs from C with respect to all three of these language features:

- In Java, arrays are their own type distinct from pointers, written T[] for an array of Ts. To create an array, we use a **new** array expression which creates an array on the heap of the specified size. The syntax of the new array expression is: new <type>[<size>]. For example the variable declaration int[] arr = new int[100] declares a variable called arr that references an array (of ints) of size 100 on the heap.

Indexing into an array in Java is identical to C with an array indexing expression, e.g., arr[4] to retrieve the element at index 4 (the fifth element) of the array. Unlike C, Java performs bounds checking to ensure that the index provided to an array indexing expression is in range. If an invalid index is accessed of an array, rather than performing undefined behavior, a Java program will throw an exception signifying that an erroneous condition has occurred.
Consequently, this means that a Java array must store its length in addition to its element to perform this run-time check. We can access the length of an array directly by using a field access expression to access the \texttt{length} field of the array. Combining all this together, the following code snippet sums up the elements of an integer array called \texttt{arr}:

\begin{verbatim}
int[] arr = new int[100]; /* Load arr with some values... */
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
\end{verbatim}

- Java does not explicitly have a pointer type although certain types in Java that we’ll learn more about later (so-called “reference types”) are always implicitly handled through pointer.
- Java does not have a \texttt{struct} construct. It instead has a \texttt{class} construct which you can think of as a \texttt{struct} with additional features. \texttt{class} declarations are the cornerstone of Java programming in an object-oriented style, so we’ll defer our discussion of classes to later.

### 1.3 Function Declarations

The basic programming-in-the-small constructs of Java are very similar to C. However, the way that we organize our Java programs differs significantly from C programs. Recall that the canonical “Hello World!” program in C is written as:

\begin{verbatim}
// In hello.c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
\end{verbatim}

In contrast, the equivalent Java program is written as:

\begin{verbatim}
// In Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.printf("Hello World!\n");
    }
}
\end{verbatim}

Note the differences:

- The “function” \texttt{main} is wrapped in a \texttt{public class declaration} called \texttt{Hello}. The name of the file, \texttt{Hello.java}, is the same as the class name, \texttt{Hello}.
- Function declarations are prepended with the modifiers \texttt{public static}. 


• The function signature for the main function in Java is \texttt{void main(String[])}. That is, \texttt{main} must be a function that takes an array of strings (the command-line arguments to the program) and returns nothing.

• To print text to the console, we use \texttt{System.out.printf} rather than simply \texttt{printf}. Alternatively, and more standard, we can also use \texttt{System.out.println} which prints a string to the console and also adds a newline character ('\n') to the end of the string.

All of these additional pieces of syntax—the class declaration, the \texttt{public static} modifiers—have important meaning to our program. However, they do not directly impact our ability to write basic procedural programs in Java, so we’ll defer discussion of them to later chapters. For now, we will write our Java programs as follows:

1. All functions are wrapped in a public class declaration whose name is the same as the source filename (minus the .java part).
2. All functions are prepended with \texttt{public static}.

1.4 Procedural Programming

With all of these elements—expressions, statements, and functions—we can adapt our \textit{procedural programming} model of programming in C to Java. Procedural programming is a programming paradigm where our programs are composed of procedures—functions with side-effects. If our program solves a problem, we \textit{decompose} that problem into smaller problems to solve, reflecting this decomposition in the functions that we author. For example, the program \texttt{SumArgs} found in section 1.4 prints the sum of the command-line arguments given to the program.

The program decomposes the problem of printing the sum of the command-line arguments by first transforming the arguments from an array of strings to an array of integers (\texttt{transformArgs}) and then summing up the elements of an array of integers (\texttt{sum}). This decomposition is reflected in \texttt{main} by the order of the calls to the helper functions. In our procedural style, we ideally want our \texttt{main} functions to directly reflect our high-level strategy for solving the problem at hand.

Even though we will ultimately program in an object-oriented style in Java, we will still use these procedural decomposition techniques to structure how our top-level \texttt{main} function looks and behaves.

1.5 Program Compilation

We used the GNU Compiler Collection (\texttt{gcc}) to compile our C programs to executable programs. Recall that \textit{compilation} is the process of translating a program between different forms, in this case, from textual source code to executable machine code. To compile our Java programs, we invoke the Java compiler, \texttt{javac}, from the command-line similarly to \texttt{gcc}. If the \texttt{Hello.java} source file is in the current directory, we can compile it using \texttt{javac} as follows:

\$> ls
Hello.java
1.5. PROGRAM COMPILATION

```java
public class SumArgs {
    public static int[] transformArgs(String[] args) {
        int[] ret = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            ret[i] = Integer.parseInt(args[i]);
        }
        return ret;
    }

    public static int sum(int[] arr) {
        int ret = 0;
        for (int i = 0; i < arr.length; i++) {
            ret += arr[i];
        }
        return ret;
    }

    public static void main(String[] args) {
        System.out.println(sum(transformArgs(args)));
    }
}
```

Figure 1.1: Sample procedural program that sums up the command-line arguments given to the program. To pass command-line arguments to a Java program, you specify them after the class name when invoking java, e.g., java SumArgs 1 2 3.

$> javac Hello.java
$> ls
Hello.class Hello.java

However, javac does not produce a program that we can run directly! Note that the output of compiling Hello.java is a class file called Hello.class. Class files are compiled Java programs in an intermediate instruction set called Java bytecode. These class files are not directly executable; instead, we must use an interpreter program, java, which executes the program contained in the class file.

$> java Hello
Hello World!

Why does Java produce class files rather than executable machine code directly? Java bytecodes are a machine-independent representation of a Java program. This allows us to compile a Java program once locally and then distribute the resulting class files that constitute our program to anyone, irrespective of the operating system they are running. As long as they have an appropriate version of the Java runtime on their machine (which contains the java program), they can execute our program without any additional steps of compilation or configuration!
Exercise (Compilation Pipeline)  Write a Java program in a file called MyGreeting.java that prints out a modified greeting of the form "Hello, from <your name here>"! Use this simple program to ensure that you understand how a Java program is organized, that you can compile a program from the command-line properly, and you know how to submit your reading exercises for grading.