2 From C to Java

In order to attract programmers, 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 to quickly ramp up to Java.

In this chapter, we introduce how to program in Java as if we were in C. Recall that *procedural programming* is a programming paradigm where a program is composed of procedural calls, *i.e.*, functions with side-effects. Initially our Java programs will be composed of functions that contain variables, mutation, loops, *etc.* After we have become acquainted with the slight changes in syntax between C and Java, we’ll introduce objects and move to a more object-oriented style of programming.

2.1 Statements and Expressions

A programming language is made up different *constructs* that we can put together in different ways. Part of the power of a general-purpose programming language is that with a small set of such constructs, we can build software that does anything we can imagine.

In C as well as Java, there are two major sorts of constructs we use 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.*, \(3 + 5 \times (4 - 2)\), 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 have our programs do actual 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 \(prod \times 2\). After the loop is done, \(prod\) effectively contains the result of evaluating \(1 \times 2 \times \ldots \times 10\) or 10!.

2.1.1 Expressions

The expression language of Java is identical to that of C with some minor tweaks and additions. Expressions include literal values for several types:

- *Integers* *e.g.*, \(5\),
- *Floating-point values* *e.g.*, \(3.5\),
• Booleans, e.g., `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.

Of the additions, the most convenient is that we can concatenate two strings together with the `+` operator! For example "hello " + "world" evaluates to the string "hello world".

2.1.2 Statements

The statement language of Java is also identical to C's statement language. We specify a particular set of statements to execute in-order. In particular, you can write:

• Local variable declaration statements, e.g., `int x;`.
• Variable assignment statements, e.g., `x = 5;`. Note that we can also combine variable declaration and initial assignment statements, e.g., `int x = 5;`, and like C, we favor always initialization our locals when they are declared. Java also features the same set of "shortcuts", 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 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.

2.2 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 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 whereas in Java it is ill-typed.

We will discuss compound types in more detail when we talk about object-oriented programming in Java. For our purposes of importing our knowledge of C into Java, we note that both languages share an array type although Java does not have an explicit notion of pointers.
2.3 Function Declarations

The basic programming-in-the-small constructs are very similar to C. However, code organization is significantly different. Recall that the canonical “Hello World!” program in C is:

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

In contrast, this program looks as follows in Java:

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

Note the differences:

- The “function” main is wrapped in a `public class declaration` called Hello. The name of the file, Hello.java, is the same as the class name, Hello.
- Function declarations are prepended with `public static`.
- The function signature for the main function in Java is `void main(String[])`. That is, 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 `System.out.printf` rather than simply `printf`. Alternatively, and more standard, we can also use `System.out.println` which prints a string to the console and also adds in a newline character (`'\n'`).

We’ll describe the purpose of these different bits of syntax in a later reading and lab. For now, write your Java programs where:

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 `public static`. 