Similarities between Java and C++

Before diving too deeply into differences, let's start by reviewing what is the same, or at least nearly the same.

Note that there are a few "forward references" below to material you will read in the next few sections of this web site. It is not important while reading this material to fully understand what is described in those forward references – just a basic intuitive idea will suffice for now. You will see and understand that material very shortly.

1. Comments: Comments are identical in C++ and Java, both the single-line `//` style as well as the potentially multiline `/* … */`.

2. Primitive Types: The Java and C++ primitive types are nearly identical. The major differences are:

    a. The *Boolean* data type is called `boolean` in Java; it is called `bool` in C++.

    b. Java has both `byte` (a signed integer whose values can range from -128 to 127) and `char`. C++ has traditionally only had `char`, although newer versions starting with C++11 have additional variations of the `char` data type including `char16_t` and `char32_t`. Note that C++ variables of type `char` can be treated as integers in which case they can hold values in the -128 to 127 range.

    c. The integral types (e.g., `int`, `long`, `char`, etc.) can optionally be declared to be "`unsigned`" in C++. For example:

```
d. int counter1 = 0;          // range of values: -2^31 ≤ counter1
   ≤ 2^31-1
e. unsigned int counter2 = 0; // range of values:    0 ≤ counter2
   ≤ 2^32-1
   unsigned char val = 0;     // range of values:    0 ≤ val
   ≤ 255
```

    f. In Java, the sizes of the primitive types are a part of the language specification, hence they are guaranteed to be the same on all platforms. While the sizes of the C++ primitive types tend to be fairly consistent across platforms, there is no guarantee that, for example, a C++ `int` on machine A will be allocated the same number of bits as it will on machine B. (In 2.c, for example, I was simply assuming the currently common assignment of 4 bytes (32 bits) to `int` variables.)

3. Relational and Arithmetic Operators: No significant differences.

4. Control constructs: The major Java and C++ control constructs (e.g., `if`, `for`, `while`, `switch`, etc.) are essentially identical. Minor differences occasionally appear for newer variations such as Java's "for each" construct that will be mentioned in the section on **Arrays**.

5. `main`: Execution of a C++ or Java program is the same in that the runtime system will look for an appropriate entry point named `main` and initiate execution of your program by causing control to start at the first statement of the identified `main` function. The primary differences include:

    a. Every Java class is allowed to have a `main` method, and the runtime system decides which to run based on how you launch the program. In C++, there must be exactly one entry point called `main`, and it must be a function declared at global scope outside all classes. (See **Identifiers at global scope** in the navigation panel on the left.)

    b. The prototype for all Java `main` methods must be:

```
public static void main(String[ ] args)
```

    whereas the prototype for the one C++ `main` function can be *either*:

```
int main( )
```

    or

```
int main(int argc, char* argv[ ])
```

    If the latter, `argv` plays the same role as does `args` in the Java case. Because the C++ runtime system cannot determine the length of an array (as will be described in the **Arrays** section), `argc` contains the length of the `argv` array on entry to `main`. You will learn about the differences between Java's "`String[ ] args`" and C++'s "`char* argv[ ]`" in the **Character strings** section.

c. Note that the C++ `main` method must return an integer value. Typically you just "`return 0;`", but you can return other values (typically error codes) which might be retrievable for use by the host operating system.

As a comparative example (output to the screen using `std::cout` will be discussed in the **Keyboard/Screen I/O** section):

Java	C++

`public class Example`
`{`
`  public static void main(String[] args)`
`  {`
`  for (int i=0 ; i<args.length ; i++)`
`  System.out.println(args[i]);`
`  }`
`}`
	`#include <iostream>`
	`using namespace std;`
	
	`int main(int argc, char* argv[])`
	`{`
	`  for (int i=1 ; i<argc ; i++)`
	`  cout << argv[i] << '\n';`
	`  return 0;`
	`}`

If I execute the Java program from a linux command line as:

```
java Example chris book pat
```

I will see the same output as I would if I executed the C++ program from a linux command line as:

```
example chris book pat
```

This common output would be:

```
chris
book
pat
```

Notes:

i. This assumes the C++ program was compiled and linked into an executable program called "`example`".

j. The `for` loop in the C++ code starts at `i=1` because the name of the program launched ("`example`" in this case) is stored in argv[0].

k. The "`#include`" directive you see in the C++ code is very roughly analogous to Java's "`import`" statement. Some differences include the fact that "`import`" is a statement that the Java compiler handles, whereas "`#include`" is handled by a "preprocessor". (You will read about the preprocessor in the **Separate compilation** section.)