Jb0105: Java OOP: Similarities and Differences between Java and C++*

R.G. (Dick) Baldwin

Abstract

This module compares Java and C++ for the benefit of persons having familiarity with C++ and making the transition to Java.

1 Table of Contents

- Preface (p. 1)
- Similarities and differences (p. 1)
- Miscellaneous (p. 4)

2 Preface

This module, which presents some of the similarities and differences between Java and C++, is provided solely for the benefit of those students who are already familiar with C++ and are making the transition from C++ into Java.

If you have some familiarity with C++, you may find the material in this module helpful. If not, simply skip this module and move on to the next module in the collection.

In general, students in Prof. Baldwin's Java/OOP courses are not expected to have any specific knowledge of C++.

This module is intended to be general in nature. Therefore, although a few update notes were added prior to publication at cnx.org, no significant effort has been made to keep it up to date relative to any particular version of the Java JDK or any particular version of C++. Changes have occurred in both Java and C++ since the first publication of this document in 1997. Those changes may not be reflected in this module.

3 Similarities and differences

This list of similarities and differences is based heavily on The Java Language Environment, A White Paper [1] by James Gosling and Henry McGilton and *Thinking in Java* by Bruce Eckel, which was freely available on the web when this document was first published.

*Version 1.1: Nov 17, 2012 1:04 pm -0600

†http://creativecommons.org/licenses/by/3.0/

[1] http://net.uom.gr/Books/Manuals/langenviron-a4.pdf

Java does not support **typedefs** , **defines** , or a **preprocessor** . Without a preprocessor, there are no provisions for including header files.

Since Java does not have a preprocessor there is no concept of **#define** macros or *manifest constants* . However, the declaration of named constants is supported in Java through use of the **final** keyword.

Java does not support **enums** but, as mentioned above, does support *named constants* . *(Note: the enum type* [2] *was introduced into Java sometime between the first publication of this document and Java version 7.)*

Java supports *classes* , but does not support *structures* or *unions* .

All stand-alone C++ programs require a function named **main** and can have numerous other functions, including both stand-alone functions and functions that are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called methods. However, a Java application *(not a Java applet)* does require a class definition containing a **main** method.

Global functions and global data are not allowed in Java. However, variables that are declared **static** are shared among all objects instantiated from the class in which the **static** variables are declared. *(Generally, static has a somewhat different meaning in C++ and Java. For example, the concept of a static local variable does not exist in Java as it does in C++.)*

All classes in Java ultimately inherit from the class named **Object** . This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another. All Java objects contain the eleven methods that are inherited from the **Object** class.

All function or method definitions in Java are contained within a class definition. To a C++ programmer, they may look like inline function definitions, but they aren't. Java doesn't allow the programmer to request that a function be made inline, at least not directly.

Both C++ and Java support class *(static)* methods or functions that can be called without the requirement to instantiate an object of the class.

The **interface** keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed in a Java interface definition. *(True abstract base classes can also be created in Java.)* The interface concept is not supported by C++ but can probably be emulated.

Java does not support multiple class inheritance. To some extent, the **interface** feature provides the desirable features of multiple class inheritance to a Java program without some of the underlying problems.

While Java does not support multiple class inheritance, single inheritance in Java is similar to C++, but the manner in which you implement inheritance differs significantly, especially with respect to the use of constructors in the inheritance chain.

In addition to the access modifiers applied to individual members of a class, C++ allows you to provide an additional access modifier when inheriting from a class. This latter concept is not supported by Java.

Java does not support the **goto** statement *(but goto is a reserved word)* . However, it does support labeled **break** and **continue** statements, a feature not supported by C++. In certain restricted situations, labeled **break** and **continue** statements can be used where a **goto** statement might otherwise be used.

Java does not support **operator overloading** .

Java does not support automatic type conversions *(except where guaranteed safe)* .

Unlike C++, Java has a **String** type, and objects of this type are immutable *(cannot be modified)* . *(Note, although I'm not certain, I believe that the equivalent of a Java String type was introduced into C++ sometime after the original publication of this document.)*

Quoted strings are automatically converted into **String** objects in Java. Java also has a **StringBuffer** type. Objects of this type can be modified, and a variety of string manipulation methods are provided.

Unlike C++, Java provides true arrays as first-class objects. There is a length member, which tells you how big the array is. An exception is thrown if you attempt to access an array out of bounds. All arrays are instantiated in dynamic memory and assignment of one array to another is allowed. However, when

[2] http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

you make such an assignment, you simply have two references to the same array. Changing the value of an element in the array using one of the references changes the value insofar as both references are concerned.

Unlike C++, having two "pointers" or references to the same object in dynamic memory is not necessarily a problem *(but it can result in somewhat confusing results)* . In Java, dynamic memory is reclaimed automatically, but is not reclaimed until all references to that memory become NULL or cease to exist. Therefore, unlike in C++, the allocated dynamic memory cannot become invalid for as long as it is being referenced by any reference variable.

Java does not support **pointers** *(at least it does not allow you to modify the address contained in a pointer or to perform pointer arithmetic)* . Much of the need for pointers was eliminated by providing types for arrays and strings. For example, the oft-used C++ declaration **char* ptr** needed to point to the first character in a C++ null-terminated "string" is not required in Java, because a string is a true object in Java.

A class definition in Java looks similar to a class definition in C++, but there is no closing semicolon. Also *forward reference declarations* that are sometimes required in C++ are not required in Java.

The scope resolution operator (::) required in C++ is not used in Java. The dot is used to construct all fully-qualified references. Also, since there are no pointers, the pointer operator (->) used in C++ is not required in Java.

In C++, static data members and functions are called using the name of the class and the name of the static member connected by the scope resolution operator. In Java, the dot is used for this purpose.

Like C++, Java has primitive types such as **int** , **float** , etc. Unlike C++, the size of each primitive type is the same regardless of the platform. There is no unsigned integer type in Java. Type checking and type requirements are much tighter in Java than in C++.

Unlike C++, Java provides a true **boolean** type. *(Note, the C++ equivalent of the Java boolean type may have been introduced into C++ subsequent to the original publication of this document.)*

Conditional expressions in Java must evaluate to **boolean** rather than to integer, as is the case in C++. Statements such as

if(x+y)...

are not allowed in Java because the conditional expression doesn't evaluate to a **boolean** .

The **char** type in C++ is an 8-bit type that maps to the ASCII *(or extended ASCII)* character set. The **char** type in Java is a 16-bit type and uses the Unicode character set *(the Unicode values from 0 through 127 match the ASCII character set)* . For information on the Unicode character set see http://www.unicode.org/ [3] .

Unlike C++, the $\gg$ operator in Java is a "signed" right bit shift, inserting the sign bit into the vacated bit position. Java adds an operator that inserts zeros into the vacated bit positions.

C++ allows the instantiation of variables or objects of all types either at compile time in static memory or at run time using dynamic memory. However, Java requires all variables of primitive types to be instantiated at compile time, and requires all objects to be instantiated in dynamic memory at runtime. Wrapper classes are provided for all primitive types to allow them to be instantiated as objects in dynamic memory at runtime if needed.

C++ requires that classes and functions be declared before they are used. This is not necessary in Java.

The "namespace" issues prevalent in C++ are handled in Java by including everything in a class, and collecting classes into packages.

C++ requires that you re-declare static data members outside the class. This is not required in Java.

In C++, unless you specifically initialize variables of primitive types, they will contain garbage. Although local variables of primitive types can be initialized in the declaration, primitive data members of a class cannot be initialized in the class definition in C++.

In Java, you can initialize primitive data members in the class definition. You can also initialize them in the constructor. If you fail to initialize them, they will be initialized to zero *(or equivalent)* automatically.

[3] http://www.unicode.org/

Like C++, Java supports constructors that may be overloaded. As in C++, if you fail to provide a constructor, a default constructor will be provided for you. If you provide a constructor, the default constructor is not provided automatically.

All objects in Java are passed by reference, eliminating the need for the copy constructor used in C++.

(In reality, all parameters are passed by value in Java. However, passing a copy of a reference variable makes it possible for code in the receiving method to access the object referred to by the variable, and possibly to modify the contents of that object. However, code in the receiving method cannot cause the original reference variable to refer to a different object.)

There are no destructors in Java. Unused memory is returned to the operating system by way of a *garbage collector* , which runs in a different thread from the main program. This leads to a whole host of subtle and extremely important differences between Java and C++.

Like C++, Java allows you to overload functions *(methods)* . However, default arguments are not supported by Java.

Unlike C++, Java does not support templates. Thus, there are no generic functions or classes. *(Note, generics similar to C++ templates were introduced into Java in version 5 subsequent to the original publication of this document.)*

Unlike C++, several "data structure" classes are contained in the "standard" version of Java. *(Note, the Standard Template Library was introduced into the C++ world subsequent to the original publication of this document.)*

More specifically, several "data structure" classes are contained in the standard class library that is distributed with the Java Development Kit (JDK). For example, the standard version of Java provides the containers **Vector** and **Hashtable** that can be used to contain any object through recognition that any object is an object of type **Object** . However, to use these containers, you must perform the appropriate upcasting and downcasting, which may lead to efficiency problems. *(Note, the upcasting and downcasting requirements were eliminated in conjunction with the introduction of "generics" into Java mentioned earlier.)*

Multithreading is a standard feature of the Java language.

Although Java uses the same keywords as C++ for access control: **private** , **public** , and **protected** , the interpretation of these keywords is significantly different between Java and C++.

There is no **virtual** keyword in Java. All non-static methods use dynamic binding, so the virtual keyword isn't needed for the same purpose that it is used in C++.

Java provides the **final** keyword that can be used to specify that a method cannot be overridden and that it can be statically bound. *(The compiler may elect to make it inline in this case.)*

The detailed implementation of the exception handling system in Java is significantly different from that in C++.

Unlike C++, Java does not support operator overloading. However, the (+) and (+=) operators are automatically overloaded to concatenate strings, and to convert other types to string in the process.

As in C++, Java applications can call functions written in another language. This is commonly referred to as *native methods* . However, applets cannot call native methods.

Unlike C++, Java has built-in support for program documentation. Specially written comments can be automatically stripped out using a separate program named **javadoc** to produce program documentation.

Generally Java is more robust than C++ due to the following:

- Object handles *(references)* are automatically initialized to null.
- Handles are checked before accessing, and exceptions are thrown in the event of problems.
- You cannot access an array out of bounds.
- The potential for memory leaks is prevented *(or at least greatly reduced)* by automatic garbage collection.

4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0105: Java OOP: Similarities and Differences between Java and C++
- File: Jb0105.htm
- Originally published: 1997
- Published at cnx.org: November 17, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-