The Object - Orientated Project

Software Engineering CA4

Comparing C++ and Java

(Taken from Thinking in Java - Bruce Eckel - www.eckelobjects.com)

As a C++ programmer you already have the basic idea of object-oriented programming, and the syntax of Java no doubt looks very familiar to you. This makes sense since Java was derived from C++. However, there are a surprising number of differences between C++ and Java. These differences are intended to be significant improvements, and if you understand the differences you'll see why Java is such a beneficial programming language. This section takes you through the important features that make Java distinct from C++.

1. The biggest potential stumbling block is speed: interpreted Java runs something like 20 times slower than C. Nothing prevents the Java language from being compiled and there are just-in-time compilers appearing at this writing which offer significant speed-ups. It is not inconceivable that full native compilers will appear for the more popular platforms, but without those there are classes of problems that will be insoluble with Java because of the speed issue.

2. Java has both kinds of comments like C++ does.

3. Everything must be in a class. There are no global functions or global data. If you want the equivalent of globals, make static methods and static data within a class. There are no structs or enumerations or unions. Only classes.

4. All method definitions are defined in the body of the class. Thus, in C++ it would look like all the functions are inlined, but they're not (inlines are noted later).

5. Class definitions are roughly the same form in Java as in C++, but there's no closing semicolon. There are no class declarations of the form class foo; only class definitions.

```
class aType { void aMethod( ) { /* method body */ } }
```

6. There's no scope resolution operator :: in Java. Java uses the dot for everything, but can get away with it since you can define elements only within a class. Even the method definitions must always occur within a class so there is no need for scope resolution there, either. One place where you'll notice the difference is in the calling of static methods: you say ClassName.methodName( );. In addition, package names are established using the dot, and to perform the equivalent of a C++ #include you use the import keyword. For example: import java.awt.*;.

7. Java, like C++, has primitive types for efficient access. In Java, these are boolean, char, byte , short, int,

long, float, and double. All the primitive types have specified sizes that are machine-independent for portability (this must have some impact on performance, varying with the machine). Type-checking and type requirements are much tighter in Java. For example: A. Conditional expressions can be only boolean, not integral B. The result of an expression like X + Y must be used; you can't just say "X + Y" for the side effect.

8. The char type uses the international 16-bit Unicode character set, so it can automatically represent most national characters.

9. Static quoted strings are automatically converted into String objects. There is no independent static character array string as there is in C/C++.

10. Java adds the triple right shift >>> to act as a "logical" right shift by inserting zeroes at the top end; the >> inserts the sign bit as it shifts (an "arithmetic" shift).

11. Arrays are quite different in Java. There's a read-only length member that tells you how big the array is, and run-time checking throws an exception if you go out of bounds. All arrays are created on the heap, and you can assign one array to another (the array handle is simply copied). The array identifier itself is a first-class object, with all the methods commonly available to all other objects.

12. All objects of non-primitive types can be created only via new. There's no equivalent to creating non-primitive objects "on the stack" as in C++. All primitive types can be created only on the stack, without new. There are wrapper classes for all primitive classes so you can create equivalent heap-based objects with new. (Arrays of primitives are a special case: they can be allocated via aggregate initialization as in C++, or by using new).

13. No forward declarations are necessary in Java. If you want to use a class or a method before it is defined in, you simply use it – the compiler ensures that the appropriate definition exists. Thus you don't have any of the forward referencing issues that you do in C++.

14. Java has no preprocessor. If you want to use classes in another library, you say import and the name of the library. There are no preprocessor-like macros.

15. Java uses packages in place of namespaces. The name issue is taken care of by (1) putting everything in a class and (2) a facility called "packages" that performs the equivalent namespace breakup for class names. Packages also collect library components under a single library name. You simply import a package and the compiler takes care of the rest.

16. Object handles defined as class members are automatically initialized to null. Initialization of primitive class data members is guaranteed in Java; if you don't explicitly initialize them they get a default value (a zero or equivalent). You can initialize them directly when you define them in the class, or you can do it in the constructor. The syntax makes more sense than C++, and is consistent for static and non-static members alike. You don't need to externally define storage for static members like you do in C++.

17. There are no Java pointers in the sense of C and C++. When you create an object with new, you get back a reference (which I've been calling a handle in this book). For example: String s = new String("howdy"); However, unlike C++ references that must be initialized when created and cannot be rebound to a different location, Java references don't have to be bound at the point of creation, and they can be rebound at will, which eliminates part of the need for pointers. The other reason for pointers is to select any place in memory (which makes them unsafe, which is why Java doesn't support them). Pointers are often seen as an efficient way to move through an array of primitive variables; Java arrays allow you to do that in a safer fashion. The final solution for pointer problems is native methods (discussed in Appendix A). Passing pointers to methods isn't a problem since there are no global functions, only classes, and you can pass references to objects. The

Java language promoters initially said “no pointers!” but when many programmers questioned “how can you work without pointers?” they began saying “restricted pointers.” You can make up your mind whether it’s “really” a pointer or not. In any event, there’s no pointer arithmetic.

18. Java has constructors, similar to constructors in C++. You get a default constructor if you don’t define one, and if you define a non-default constructor, there’s no automatic default constructor defined for you, just like C++. There are no copy-constructors, since all arguments are passed by reference.

19. There are no destructors in Java. There is no “scope” of a variable per se, to indicate when the object’s lifetime is ended – the lifetime of an object is determined instead by the garbage collector. There is a finalize( ) method that’s a member of each class, like a destructor, but finalize( ) is called by the garbage collector and is supposed to be responsible only for releasing resources. If you need something done at a specific point, you must create a special method and call it, not rely upon finalize( ). Put another way, all objects in C++ will be (or rather, should be) destroyed, but not all objects in Java are garbage collected. Because Java doesn’t support destructors, you must be careful to create a cleanup method if necessary, and to explicitly call all the cleanup methods for the base class and member objects in your class.

20. Java has method overloading that works virtually identically to C++ function overloading.

21. Java does not support default arguments.

22. There’s no goto in Java. The one unconditional jump mechanism is the break label or continue label, which is used to jump out of the middle of multiply-nested loops.

23. Java uses a singly-rooted hierarchy, so all objects are ultimately inherited from the root class Object. In C++ you can start a new inheritance tree anywhere, so you end up with a forest of trees. In Java you get a single ultimate hierarchy. This can seem restrictive, but it gives a great deal of power since you know that every object is guaranteed to have at least the Object interface. C++ appears to be the only OO language that does not impose a singly-rooted hierarchy.

24. Java has no templates or other implementation of parameterized types. There is a set of collections: Vector, Stack and Hashtable that hold Object references, and through which you can satisfy your collection needs, but these collections are not designed for efficiency like the C++ Standard Template Library (STL). For a more complete set of collections, there’s a freely-available library called the Generic Collection Library for Java (www.ObjectSpace.com) which shows signs of eventually being incorporated into the standard Java language.

25. Garbage collection means memory leaks are much harder to cause in Java, but not impossible. However, many memory leaks and resouce leaks may be tracked to a badly written finalize( ) or not releasing a resource at the end of the block where it is allocated (a place where a destructor would certainly come in handy). The garbage collector is a huge improvement over C++, and makes a lot of programming problems simply vanish. It may make Java unsuitable for solving a small subset of problems that cannot tolerate a garbage collector, but the advantage of a garbage collector seems to greatly outweigh this potential drawback.

26. Java has built in multithreading support. There’s a Thread class that you inherit to create a new thread (you override the run( ) method). Mutual exclusion occurs at the level of objects using the synchronized keyword as a type qualifier for methods. Only one thread may use a synchronized method of a particular object at any one time. Put another way, when a synchronized method is entered, it first “locks” the object against any other synchronized method using that object, and “unlocks” the object only upon exiting the method. There are no explicit locks; they happen automatically. You’re still responsible for implementing more sophisticated synchronization between threads by creating your own “monitor” class. Recursive synchronized methods work correctly. Time slicing is not guaranteed between equal priority threads.

27. Instead of controlling blocks of declarations like C++ does, access specifiers (public, private and protected) are placed on each definition for each member of a class. Without an explicit access specifier, the element defaults to "friendly," which means it is accessible to other elements in the same package (equivalent to them all being friends) but inaccessible outside the package. The class, and each method within the class, has an access specifier to determine whether it's visible outside the file. Sometimes the private keyword is used less in Java because "friendly" access is often more useful than excluding access from other classes in the same package (however, with multithreading the proper use of private is essential). The Java protected keyword means "accessible to inheritors and to others in this package." There is no equivalent to the C++ protected keyword which means "accessible to inheritors only" (private protected used to do this, but it was removed).

28. Nested classes. In C++, nesting a class is an aid to name hiding and code organization (but C++ namespaces eliminate the need for name hiding). Java packaging provides the equivalence of namespaces, so that isn't an issue. Java 1.1 has inner classes which look just like nested classes. However, an object of an inner class secretly keeps a handle to the object of the outer class that was involved in the creation of the inner-class object. This means that the inner-class object may access members of the outer-class object without qualification, as if those members belonged directly to the inner-class object. This provides a much more elegant solution to the problem of callbacks, solved with pointers to members in C++.

29. Because of inner classes described in the previous point, there are no pointers to members in Java.

30. No inline methods. The Java compiler may decide on its own to inline a method, but you don't have much control over this. You may suggest inlining in Java by using the final keyword for a method. However, inline functions are only suggestions to the C++ compiler, as well.

31. Inheritance in Java has the same effect as in C++, but the syntax is different. Java uses the extends keyword to indicate inheritance from a base class, and the super keyword to specify methods to be called in the base class that have the same name as the method you're in (however, the super keyword in Java allows you to access methods only in the parent class, one level up in the hierarchy. Base-class scoping in C++ allows you to access methods that are deeper in the hierarchy). The base-class constructor is also called using the super keyword. As mentioned before, all classes are ultimately, automatically inherited from Object. There's no explicit constructor initializer list like in C++ but the compiler forces you to perform all base-class initialization at the beginning of the constructor body and it won't let you perform these later in the body. Member initialization is guaranteed through a combination of automatic initialization and exceptions for uninitialized object handles.

```
public class Foo extends Bar {

    public Foo(String msg) {

        super(msg); // Calls base constructor }

    public baz(int i) { // Override

        super.baz(i); // Calls base method }

        }
```

32. Inheritance in Java doesn't change the protection level of the members in the base class. You cannot specify public, private or protected inheritance in Java as you can in C++. Also, overridden methods in a derived class cannot reduce the access of the method in the base class. For example, if a method is public in

the base class and you override it, your overridden method must also be public (the compiler ensures this).

33. Java provides the interface keyword which creates the equivalent of an abstract base class filled with abstract methods and with no data members. This makes a clear distinction between something designed to be just an interface versus an extension of existing functionality using the extends keyword. It's worth noting that the abstract keyword produces a similar effect, in that you can't create an object of that class. However, an abstract class may contain abstract methods but it can also contain implementations, so it is restricted to single inheritance. Together with interfaces, this scheme prevents the need for some mechanism like virtual base classes in C++. To create a version of the interface that can be instantiated, you use the implements keyword, whose syntax looks like inheritance:

```
public interface Face {

    public void smile(); }

public class Baz extends Bar implements Face {

    public void smile( ) {

    System.out.println("a warm smile"); } }
```

34. There's no virtual keyword in Java because all non-static methods always use dynamic binding. In Java, the programmer doesn't have to decide whether or not to use dynamic binding. The reason virtual exists in C++ is so you can leave it off for a slight increase in efficiency when you're tuning for performance (or, put another way, "if you don't use it you don't pay for it"), but this often results in confusion and unpleasant surprises. The final keyword provides some latitude for efficiency tuning – it tells the compiler that this method may not be overridden, and thus that it can be statically bound (and made inline, thus using the equivalent of a C++ non-virtual call). These optimizations are up to the compiler.

35. Java doesn't provide multiple inheritance (MI), at least not in the same sense C++ does. Like protected, MI seems like a good idea but you know you need it only when you are face to face with the right design problem. Since Java uses a singly-rooted hierarchy, you'll probably run into fewer situations where MI is necessary. The aforementioned interface keyword takes care of combining multiple interfaces.

36. Run-time type identification functionality is quite similar to C++. To get information about handle X you can say, for example: X.getClass().getName(); To perform a type-safe downcast you say: derived d = (derived)base; just like an old-style C cast. The compiler automatically invokes the dynamic casting mechanism without requiring extra syntax. Although this doesn't have the benefit of easy location of casts as in C++ "new casts," Java checks usage and throws exceptions so it doesn't allow bad casts like C++ does.

37. Exception handling in Java is different because there are no destructors. A finally clause can be added to force execution of statements that perform necessary cleanup. All exceptions in Java are inherited from the base class Throwable, so you're guaranteed a common interface.

```
public void f(Obj b) throws IOException {

    myresource mr = b.createResource();

    try { mr.UseResource(); } catch (MyException e) { // handle my exception }

    catch (Throwable e) { // handle all other exceptions }
```

```
finally { mr.dispose(); // special cleanup }

}
```

38. Exception specifications in Java are vastly superior to those in C++. Instead of the C++ approach of calling a function at run-time when the wrong exception is thrown, Java exception specifications are checked and enforced at compile-time. In addition, overridden methods must conform to the exception specification of the base-class version of that method: they can throw the specified exceptions, or exceptions derived from those. This provides much more robust exception-handling code.

39. There is method overloading, but no operator overloading in Java. The String class does use the + and += operators to concatenate strings and String expressions use automatic type conversion, but that's a special built-in case.

40. The const issues in C++ are avoided in Java by convention. You pass only handles to objects, and local copies are never made for you automatically. If you want the equivalent of C++'s pass-by-value, you call clone( ) to produce a local copy of the argument (although the clone( ) mechanism is somewhat poorly designed). There's no copy-constructor that's automatically called. To create a compile-time constant value, you say: static final int SIZE = 255; static final int BSIZE = 8 * SIZE;

41. Because of security issues, programming an "application" is quite different from programming an "applet." A significant issue is that an applet won't let you write to disk, because that would allow a program downloaded from an unknown machine to trash your disk. This changes somewhat with Java 1.1 digital signing, which allows you to unequivocally know everyone that wrote all the programs that have special access to your system (one of which may have trashed your disk; you still have to figure out which one and what to do about it…).

42. Since Java can be too restrictive in some cases, you may be prevented from doing important tasks like directly accessing hardware. Java solves this with native methods that allow you to call a function written in another language (currently only C/C++ are supported). Thus you can always solve a platform-specific problem (in a relatively non-portable fashion, but then that code is isolated). Applets cannot call native methods, only applications.

43. Java has built-in support for comment documentation, so the source code file can also contain its own documentation, which is stripped out and reformatted into HTML using a separate program. This is a boon for documentation maintenance and use.

44. Java contains standard libraries for solving specific tasks. C++ relies on non-standard third-party libraries. These tasks include (or will soon include): – Networking – Database Connection (via JDBC) – Multithreading – Distributed Objects (via RMI and CORBA) – Compression – Commerce The availability and standard nature of these libraries allow for more rapid application development.

45. Java 1.1 includes the Java Beans standard, which is a way to create components that can be used in visual programming environments. This promotes visual components that can be used under all vendor's development environments. Since you aren't tied to a particular vendor's design for visual components, this should result in greater selection and availability of components. In addition, the design for Java Beans is simpler for programmers to understand; vendor-specific component frameworks tend to involve a steeper learning curve.

46. If the access to a Java handle fails, an exception is thrown. This test doesn't have to occur right before the use of a handle; the Java specification just says that the exception must somehow be thrown. Many C++

runtime systems can also throw exceptions for bad pointers.

47. Generally, Java is more robust, via: – Object handles initialized to null (a keyword) – Handles are always checked and exceptions are thrown for failures – All array accesses are checked for bounds violations – Automatic garbage collection prevents memory leaks – Clean, relatively fool-proof exception handling – Simple language support for multi-threading – Bytecode verification of network applets