Object Resources

taligent white papers

by Mark Davis

Introduction

Java has taken the programming world by storm. In our own experience in developing software at Taligent, its turnaround time is several times faster than that of C++, a crucial feature in today's markets. Moreover, it has significant advantages over C++ in terms of portability, power, and simplicity. These advantages are leading many people to consider replacing C++ with Java, not only for web applets, but increasingly for client and server applications as well.

Yet there is a large amount of code already written in C++. For just this reason, many people are considering porting their existing C++ code to Java. This paper is addresses those people, providing a step-by-step approach to porting C++ to Java effectively, with special attention to the following:

- ***Pitfalls.*** In most cases, the differences between the languages are syntactic, and the compiler will discover whether you forget to make a change. However, there are a few instances where the same code in C++ and Java has dangerously different consequences. These pitfalls are marked with the graphic on the right (Unicode character 2620).

- ***Minimal effort.*** We assume that you are not, at this point, interested in completely revamping your code, even though it may be old crufty stuff that you inherited from someone else. As much as possible, therefore, we minimize your work by providing techniques for a one-for-one match between your old code and new.

This paper is not specifically directed at the Java beginner, although it can be useful for those getting started. There are many books available for learning about Java and object-oriented program design; for some of our favorites, see References.

The following sections are covered in this introduction.

If it ain't wrong...

Since to a large degree Java follows C++ conventions, the bulk of your code will remain unchanged: variable names, flow of control, names of primitive types, and so on. As you do in C++, in Java you will write classes, override methods, overload methods, write constructors, instantiate objects, and so on. The following elements of the two languages are very close and generally need little modification.

- Primitives
  - char, int, short, long, float, double, void (but not void*)
  - variable names
- Flow of Control
  - if, else, for, while, do, switch, case, default, break, continue, return, static
- Operators
  - +, -, !, %, ^, &, *, |, ~, /, >, <, (), [], {}, ?:, ., =, ++, --, ==, <=, >=, !=, >>, <<, ||, &&, *=
- Comments
  - /*...*/, //

For example, these snippets of code remain unaltered when ported from C++ to Java.

Bulk of Code Unchanged

C++	Java

`// some sample lines of code`
`int x = 3;`
 	
`for (int i = 0; i < j; ++i) {`
`  x += i * i;`
`}`
 	
`x = y.method1(3,4);`
	`// some sample lines of code`
	`int x = 3;`
	
	`for (int i = 0; i < j; ++i) {`
	`  x += i * i;`
	`}`
	
	`x = y.method1(3,4);`

When porting from C++ to Java, your job is far easier than when porting to Basic, FORTRAN, Lisp, Smalltalk or other radically different languages. Keep that in mind as you go through the remainder of this document: the items I list are the exceptions, not the rules.

The compiler is *your* friend

The Java compiler is much more rigorous than that of C++, so much of the code that needs to be changed will be found by the compiler. With each description of a porting task, examples will show you what you have to do to C++ code to change it to Java code. In these examples, corresponding lines of code in C++ and Java are lined up (although in some older browsers this doesn't work properly). The text is also color-coded, in the following fashion.

Color	Meaning
Red	Pitfall (aka *faux amis*): code that looks the same, but has quite a different meaning!
Blue	Changes between C++ and Java code.
Black	Code that is the same in Java as it is in C++.
Brown	Items with special comments in the notes below.
Green	Comments.

For brevity, code snippets include only enough of the context to be understandable!

In discussing Java programming there is also some terminology that we find useful, especially for discussing how to deal with references and the lack of `const`.

Immutables
: Those classes or objects whose state can't be changed, such as `String`, `Number`, or `Locale`. Typically this means that there are getters (a.k.a. accessors), but no setters (a.k.a. mutators). The entire state of the object is determined by the constructor.

Mutables
: Those classes or objects that are not Immutable. This also excludes primitives (boolean, byte, char, short, int, long, float, double).

Sliced
: An object that has been converted to a superclass object, with loss of data.

Getting down to business

This article is divided up into the web pages below. (It is not split up further, since it is pretty annoying to print very many pages with today's browsers.) It is also organized to be useful when printed, with some caveats.[4]

On each page in this article, there are links to the sections on that page, plus links to the top of the page (that look like🟩). There are also the occasional footnotes, which are indicated in the text with a superscripted number in brackets[3] and found at the very end of the article.

- To contact Taligent about these pages, send mail to cookbook@Taligent.com.
- To find out more about Taligent's work in Java, visit http://www.Taligent.com.

Taligent.

Object Resources

taligent white papers

Basics

The following sections walk you through the main steps necessary to convert your program from C++ to Java.

Placement is everything

Before you start porting, first create the directory structure that you will use. To do this, first figure out what your main package name is. To get it, reverse the fields in your domain name. Thus `xyz.COM` becomes `COM.xyz`. If you have two directories `abc` and `def` for your project, then their packages become `COM.xyz.abc` and `COM.xyz.def`. Now create directories that correspond to this structure, and copy all of your sources to the appropriate directory.

- `COM`
  - `xyz`
    - `abc`
      - `class1.java`
      - `class2.java`
    - `def`
      - `class3.java`
      - `class4.java`

Once you have finished your directory structure, you will have to merge your code, since Java does not distinguish between class interface (declaration) and implementation (definition) as C++ does. To handle this, rename your header file extensions to be `.java`. Then take the implementation of each member and copy it in after the declaration of that member, as if you were doing an `inline` method in C++.

The next step is to take all of the access keywords (`public`, `protected`, `private`), and copy them at the front of each of the succeeding methods and fields that they pertain to. Change the inheritance syntax to use `extends` instead of a colon. If you use multiple inheritance, see Primogenitur entail.

Finally, break apart each class into a separate file, and at the top of the class, put your package name and a list of imports. These imports should be a list of all the other packages that you need to access.[5]

Fixing Basic Structure

C++	inlined C++	Java

// file.h
 	
class Foo : Bar {
 public:
 	
  int square();
 	
  int cube();
 	
 private:
  int x;
}
 	
// file.c (or .cpp)
 	
int Foo::square() {
  return x*x;
}
 	
int Foo::cube() {
  return x*x*x;
}
	// file.h
	
	class Foo : Bar {
	public:
	
	 int square() {
	   return x*x};
	
	 int cube() {
	  return x*x*x;
	 }
	 private:
	 int x;
	}
		// file.java
		package COM.xyz.abc;
		import COM.xyz.abc.*;
		import COM.xyz.def.*;
		
		class Foo extends Bar {
		
		 public int square() {
		   return x*x};
		
		 public int cube() {
		  return x*x*x;
		 }
		
		 private int x;
		}

Notes

- Java does not have the notion of `friend` as does C++. See Java has no friends for more information about how to handle this.
- Java 1.1 does have nested classes, though Java 1.0 does not. If you cannot wait for 1.1, you will have to move your nested classes out to the top. We suggest concatenating the names: for a nested class `Foo` in a class `Bar`, use `Bar_Foo`.
- Imports in Java are not really like interfaces in C++; they are more like namespaces, just allowing you to abbreviate the full name of a class. For example, even if you omit importing `java.util.*`, you can still use the `Hashtable` class by simply writing its full name, `java.util.Hashtable`.

To protect the innocent

Next, you need to change a few names. In one particular case this name change is straightforward .

Simple Name Replacements

C++	Java
`bool x = true;`	`boolean x = true;`

Most name differences, however, depend on the context.

Context-Dependent Name Replacements

C++	Java

`// const field`
`static const x = 3;`
	`// const field`
	`static final x = 3;`

`// const method`
`int doSomething() const;`
	`// const method`
	`int doSomething();`

`// character data`
`char ch = 'b';`
	`// character data`
	`char ch = 'b';`

`// byte data (e.g. short numbers)`
`char b = 31;`
	`// byte data`
	`byte b = 31;`

`// abstract method`
`int someMethod() = 0;`
	`// abstract method`
	`abstract int someMethod();`

`// non-virtual method`
`int someMethod();`
	`// non-virtual method`
	`final int someMethod();`

`// virtual method`
`virtual int someMethod();`
	`// virtual method`
	`int someMethod();`

`// unknown object (not primitive)`
`void* doAnother() {}`
	`// unknown object`
	`Object doAnother() {}`

`// typedef`
`typedef unsigned short UniChar;`
 	
`UniChar ch;`
	`// replace by right type`
	
	`char ch;`

The most important language feature missing from Java is `const`. The simplest approach at the start is to change `const` to `final` for any field, and remove it otherwise. However, I advise you to consult Bullet-proofing for more robust techniques.

Notes

- C and C++ do not distinguish between `char` as a small number or as a piece of character data; in general, though, it usually corresponds to character data and can be left alone. Java characters are

16-bit Unicode characters, but since the first 256 Unicode characters are the same as 8-bit ASCII, you usually don't need to make any changes. This topic is also discussed at more length below.

- The term `final` in Java is roughly equivalent to non-`virtual inline` in C++, while the absence of `final` is roughly equivalent to `virtual` non-`inline` in C++.
    - Put `final` in front of every method that doesn't contain the word `virtual`, then delete all instances of `virtual`.
        - There is one complication: in C++, if a method is marked `virtual` in a superclass, then it is implicitly `virtual` in all subclasses. So, you may need to look at superclasses to see if the method is really `virtual`.
    - Remove the word `inline` everywhere.
    - C++ does allow `inline virtual` methods; you could duplicate the effect of this in Java with separate methods, but it is usually not worth doing. Just leave them non-`final`.
- Remove the word `register` everywhere. This is just a hint to the compiler anyway, and one that is often ignored by modern optimizations.
- Remove `auto`, since it is redundant.
- Remove the `typedef` declarations. All instances of the defined type name need to be replaced by what they represent. Notice that this replacement may not be one-to-one.

All lines are busy

Java does not support operator overloading. You will miss this feature for about 5 minutes if you are programming in pure Java, but it is a hassle when converting from C++. First you will need to change all the definitions.

Here is a sample list of operators that could be overloaded, and some typical Java equivalent names (there is no fixed set of replacement names; these are only samples). If you are porting well-behaved C++ code, then the meaning of the operator does not deviate from the core meaning; if not, then you should change the name to correspond to the real meaning (such as `append` for `+`). The brown items have special notes below.

Operator Overload Replacement Names

C++	Java
`+`	`plus`
`-`	`minus`
`!`	`not`
`%`	`remainder`
`*`	`times`
`/`	`dividedBy`
`^`	`bitXor`
`&`	`bitAnd`
`|`	`bitOr`
`~`	`bitNot`
`>>`	`shiftRight`
`<<`	`shiftLeft`
`||`	`or`
`&&`	`and`
`==`	`equals`

C++	Java
`<`	`isLess`
`<=`	`isLessOrEquals`
`!=`	`use !a.equals(b)`
`>`	`use b.isLess(a)`
`>=`	`use b.isLessOrEquals(a)`
`()`	`(see below)`
`[]`	`elementAt, setElementAt`
`=`	`assign`
`++`	`increment`
`--`	`decrement`
`*=...`	`(see context)`
`*`	`getX, setX`
`->`	`getX, setX`
`&a`	`see below`

Notes

- The operator `%` is remainder in Java, not modulo. For negative numbers, this difference is significant. For example, `-3%5 == -3`, while `-3 modulo 5 == 2`. In C++, it is undefined whether `%` is remainder or modulo although most implementations use remainder. So since your C++ code is portable (right?), you never depended on the result with negative numbers, and you don't need to make any changes. If you did depend on negative values being modulo, you will need to change `x%y` to
`(x%y - ((x < 0) ? y : 0))`.
- Don't bother defining an equivalent to `!=`. The value of `(a != b)` should always be the same as if you called `!(a == b)`, so just replace the call sites with `(!a.equals(b))`. You can replace uses of > and >=, by just reversing the order of the operands. By the way, for IEEE floats and doubles the value `(a <= b)` is not the same as `!(b > a)`:
if `a = 1.0`, and `b = 0.0/0.0` , then `(a <= b)` is false, while `!(b > a)` is true!
- The parentheses operator differs so much from case to case that you will have to look at the context to get a good name.
- For the pointer operators `*`, `&` and `->`, Java has no real equivalents. Use getters and setters as appropriate.
- Many homegrown operators will not distinguish between predecrement and postdecrement, or preincrement and postincrement. If you make use of this distinction, use the longer names.
- Assignment (and copy constructors) are more complex than other operators. For more detail, see Difficult assignments.
- For the index operators, define two methods. You will then have to fix the call sites according to the usage, depending on whether you are accessing the value or changing it (see below).
- For the conversion operators, use the pattern below of appending `to` to the name of the type.

Once you have changed all of the definitions, let the compiler find the call sites for you to fix.

Replacing Overloaded Operator Calls

C++	Java

`// defining`
`bool operator==`
` (const Foo& other) const {...`
 	
`operator int () {...`
 	
`operator AType () {...`
 	
`// using`
`if (a == b)...`
 	
`a[3] = 5;`
`x = a[3];`
	`// defining`
	`boolean equals(Foo other) {...`
	
	`int toInt () {...`
	
	`AType toAType() {...`
	
	`// using`
	`if (a.equals(b))...`
	
	`a.setElementAt(3,5);`
	`x = a.elementAt(3);`

Giving pointers

Java is touted as having no pointers. In porting from C++ code, however, you almost want to think of it as the reverse: *all* objects are pointers--there are *no* stack objects or value parameters. The syntax of the language hides this fact from you, but as the following examples show, you have to be careful!

Replacing Pointers

C++	Java
`// initializing`	`// initializing`
`Foo* x = new Foo(3);`	`Foo x = new Foo(3);`
`Foo y(4);`	`Foo y = new Foo(4);`
`Foo z;`	`Foo z = new Foo();`
`// assigning`	`// assigning`
`Foo* a = x;`	`Foo a = x;`
`Foo* c = 0;`	`Foo c = null;`
`Foo* d = NULL;`	`Foo d = null;`
`Foo b = y;`	`Foo b = y.clone();`
`// calling`	`// calling`
`x->doSomething();`	`x.doSomething();`
`y.doSomething();`	`y.doSomething();`
`// comparing`	`// comparing`
`if (x == a);`	`if (x == a);`
`if (y == b);`	`if (y.equals(b));`
`if (&y == &b);`	`if (y == b);`

Important

- Note that assignment of objects with = does *not* assign value; it is the equivalent of pointer assignment. You have to use `clone()` to get a new object.
- Similarly, comparison of objects with == is a *pointer* comparison; you have to use `equals()` to get comparison by value.
- Java does not automatically convert numbers. Use `null` instead of zero for a null object. If you are using pointer arithmetic, see Honest-to-God arrays.

No references necessary

Java also does not have reference parameters in the same way as C++ does. Most C++ programs only use them in passing parameters to a method, or in getting a return value back.

Fixing Parameter References

C++	Java
`// value input parameter`	`// value input parameter`
`int method1(Foo x);`	`int method1(Foo x);`
`// input parameter`	`// input parameter`
`int method2(const Foo& x);`	`int method2(Foo x);`
`// Mutable output parameter`	`// Mutable output parameter`
`int method3(Foo& x);`	`int method3(Foo x);`
`// Immutable output parameter`	`// Immutable output parameter`
`int method4(int& x);`	`int method4(int[] x);`
`// usage`	`// usage`
`Foo x;`	`Foo[] x = new Foo[1];`
`z = y.method4(x);`	`z = y.method4(x);`
`w = x;`	`w = x[1];`

Notes

- Input parameters are those passed by value or as const references. To make your code work, you can simply remove the `const` *(however, the robustness of your code definitely suffers with this simple approach, see Bullet-proofing for better techniques).*
- Output parameters are more complex. Mutable objects (such as `StringBuffer`) are passed in directly. Immutable objects (such as `String`, `Integer`) are more troublesome. You have three choices:
  1. Return the value from the method. This only works if the return was `void`.
  2. The simplest way--though ugly--is to pass in an array as we did in the example above. Since arrays are always Mutable, you can just get/set the first value in the array.
  3. The last way is to create a new class that contains fields corresponding to the output parameters and return value, and return that. If you make that new class Mutable, you can also use it as an output parameter and modify it. This involves more effort, but is somewhat cleaner than the array method.

Alternative Output Parameters

C++	Java
`// output parameter`	`// output parameter`
`int method4(int& x);`	`int method4(IntWrapper x);`
`// usage`	`// usage`
`int x;`	`IntWrapper x = new IntWrapper();`
`z = y.method4(x);`	`z = y.method4(x);`
`w = x;`	`w = x.value;`
	`// new class`
	`class IntWrapper {`
	` public Int value;`
	`}`

Return values can also be references. There are three common idioms for reference returns in C++:

1. Return `*this`. This method allows chaining, as in `x = y = z`. This idiom is used by `StringBuffer` in Java, so that you can write `x.append(y).append(z)`.
2. Return a reference to an input parameter. This allows use of functional returns without requiring memory allocation. Since the principal use of input parameter returns is in handling memory allocation, there is little need for it in Java, but it may make your porting easier to leave it as is.
3. Return a reference to a static.

All of these idioms can be used in Java, though if you try to set a Mutable the compiler will warn you of problems. In that case, you will have to use some of the same techniques as with output parameters.

Fixing References

C++	Java
`// definition`	`// definition`
`Foo& setX();`	`Foo setX();`
`// return of output parameter`	`// return of output parameter`
`Foo& getY(Foo& Y);`	`Foo getY(Foo Y);`
`// return static constant`	`// return static`
`const Foo& getAStatic();`	`Foo getAStatic();`
`// use`	`// use`
`myObject.setX(3).setY(4);`	`myObject.setX(3).setY(4);`
`Foo x;`	`Foo x = new Foo();`
`myObject.doZ(myObject.getY(x));`	`myObject.doZ(myObject.getY(x));`
`z = x * Foo::getAStatic();`	`z = x * Foo.getAStatic();`

Notes

- There are a couple of other cases where references might be used in C++, although they should *not* occur in good C++ code:
    - The method creates a new object on the heap, but then hands back a reference rather than the pointer. This is a prime candidate for memory leaks, since there is no way for the original object to know when to toss the storage. This is a bug in the C++ code and should be corrected.
    - The method returns a reference to an internal element. For example, it could return `&myCharArray[5]`. Although legal, this can be the source of many nasty headaches, since the results are undefined if the enclosing storage is altered or moved.

Honest-to-God arrays

Java arrays are real objects, not just disguised pointers. Generally you replace pointers used to iterate through an array with offsets, and pointers used with the `*` operator with an array access. Most of these cases will be flagged by the compiler.

Fixing Arrays

C++	Java
`// initializing`	`// initializing`
`double x[10];`	`double[] x = new double[10];`
`double* end = x + 10;`	`int end = x.length;`
`double* current = x;`	`int current = 0;`
`// iterating`	`// iterating`
`while (current < end) {`	`while (current < end) {`
` doSomethingTo(*current++);`	` doSomethingTo(x[current++]);`
`}`	`}`
`for (i = 0; i < 10; ++i) {`	`for (i = 0; i < x.length; ++i) {`
` doSomethingTo(*current++);`	` doSomethingTo(x[i]);`
`}`	`}`

Notes

- Both the syntaxes `Foo x[]` and `Foo[] x` work, though the latter is recommended.
- Java arrays can supply you their length, rather than you having to remember it independently. Wherever possible, use the supplied length instead of a hard-coded value.
- As with fields of an object, the items in an array are initialized to zero (for numerical primitives), `false` for `boolean`, and `null` for objects.

Declaring an array does ***not*** *create the objects to fill the array!* This is another place where objects behave like pointers, not values. Since arrays of objects are--under the covers--arrays of pointers, they are initialized to `null`, *not* to a list of default-constructed objects. If you want them to be default-constructed objects, you *must* set them yourself!

Fixing Arrays

C++	Java
`// initializing`	`// initializing`
`Foo x[10];`	`Foo[] x = new Foo[10];`
	`for (int i = 0; i < x.length; ++i) {`
	` x[i] = new Foo();`
	`}`
`// initializing`	`// initializing`
`static const int x[] =`	`static const int x[] =`
` {1,2,3,4,5,6,7,8,...`	` {1,2,3,4,5,6,7,8,...`

If the `Foo[]` array were a static field, you could still do the initialization, by putting the `for`-loop inside a static block.

One other point: although the allocation of static arrays in Java looks similar to the allocation of static arrays in C++, be aware that there are *far* different footprint and performance implications. Unlike the C++ compiler, which builds a static table that is loaded in at runtime, the Java compiler actually generates code to place *every* single entry in the array, so a very large table could result in a performance and footprint hit.

Only stringing you along

The majority of C++ programs either still use `char*` to represent strings, or use their own home-grown string class. For the former case, here is how to translate some of the common lines of code.

Replacing char*

C++	Java
`// Strings`	`// Strings`
`char s[100] = "abc";`	`String s = "abc";`
`strcat(s,"def");`	`s += "def";`
`char ch = s[5];`	`char ch = s.charAt(5);`

`s[3] = 'a';`	`s = s.substring(0,3)`
	`+ 'a' + s.substring(4,s.length());`

`// Strings`	`// StringBuffer`

`char s[100] = "abc";`	`StringBuffer sb`
	`= new StringBuffer("abc");`

`strcat(s,"def");`	`sb.append("def");`
`char ch = s[5];`	`char ch = sb.charAt(5);`
`s[3] = 'a';`	`sb.setCharAt(3,'a');`
`// character properties`	`// character properties`
`if (islower(ch)) ...`	`if (Character.isLowerCase(ch)) ...`

Notes

- Note that you have substantially different code depending on whether you use `String` (an Immutable class) or `StringBuffer` (a Mutable class). `String` is generally simpler to use; `StringBuffer` should be used for efficiency in successive modifications. When you are using `String` and doing modifications, remember that you have to reset the value of your variable to the result; methods named as if they modify the value (such as `replace`) really create new `Strings` that you have to assign back.
- The properties in `Character` are much broader than in C++, and are going to be augmented further in Java 1.1 to include all the Unicode 2.0 properties. This is part of a larger addition of code to more fully support Unicode, and enable Java's support of the many languages supported by the tens of thousands of characters in the Unicode Standard. If you are interested in the Java 1.1 international support provided by Taligent, visit http://www.Taligent.com. You should also consider getting a copy of the Unicode Standard, Version 2.0.

Taligent.

Object Resources

taligent white papers

Next Steps

The following sections take you further through the steps necessary to convert your program from C++ to Java.

Who owns what?

In C++, you have to be extremely careful about who *owns* a pointer--that is, whether the object or a caller has the right, *and the responsibility (or blame)*, for deleting the pointer. There are three general cases in C++:

Adoption
: The caller hands over ownership to the object. The caller should relinquish any pointers to the object, so that it does not mistakenly delete or alter the contents of the pointer.

Aliasing
: The caller keeps ownership, and just passes in a pointer. The cleanest case is when the pointer is `const`, since it is clear that the object cannot own the pointer. (Unfortunately, there is no way of indicating in C++ that the object can make changes to the pointer but cannot delete it.)

Assignment
: The caller keeps ownership of what it passes in, and the object makes its own copy. This is the safest mechanism, but must often be avoided because of performance considerations.
  - Note that if the pointer can be a subclass, you *had to* use some extension of RTTI in C++ instead of `new`: otherwise your object will be sliced. Since even standard RTTI does not support this, most people end up having a base class member function called `copy` or `clone` that all derived classes override.
  - To make our examples simpler, we will write the C++ code as if you had two global template functions:

    - `::Copy(x,y)`, which creates a polymorphic copy of y.
    - `::ReplaceByCopy(x,y)`, which deletes `x`, sets it to `NULL`, then sets it to a copy of `y`.
      (After you delete a pointer field in C++, you *must* null it out before assigning it with a function call (including `new`). This is a subtle point, but unless you do this the object's destructor will do a double deletion if the function call throws an exception!)

Java is considerably simpler, as you will see.

Garbage in...

Java has built-in garbage collection, which relieves you from much of the grunt work of memory access. (Not all of it--even if you don't have to worry about who can delete objects, you still have to worry about who can *change* objects: see Bullet-proofing). In general, you will just remove all destructors and deletions, and replace copy constructors by `clone`.

Removing Destructors and Deletions

C++	Java

`// destructor`
 	
` ~MyObject() {`
 	
`   delete field1;`
 	
`   delete field3;`
 	
` }`
	`// no destructor`

`// pointer field adoption`
 	
` void setField1 (Foo* newValue) {`
 	
`  delete field1;`
 	
`  field1 = newValue;`
 	
` }`
	`// field assignment`
	
	`void setField1 (Foo newValue) {`
	
	`  field1 = newValue;`
	
	` }`

`// pointer field aliasing`	`// pointer field aliasing`
`void setField2 (Foo* newValue) {`	`void setField2 (Foo newValue) {`
`field2 = newValue;`	`field2 = newValue;`
`}`	`}`
`// pointer field assignment`	`// field assignment`
`void setField3 (const Foo& newValue) {`	`void setField2 (Foo newValue) {`
`delete field3;`	
`field3 = NULL;`	
`aValue = new Foo(newValue);`	`aValue = newValue.clone();`
`}`	`}`

If the object is not going out of scope or going to be reset soon, then you should replace a deletion by setting to `null`. That allows the garbage collector to get rid of the object without waiting for it to go out of scope. For example:

Enabling Garbage Collection

C++	Java
`// big block with lots of stuff`	`// no destructor`
` {`	` {`
`   ...`	`   ...`
`   Foo x = new Foo();`	`   Foo x = new Foo();`
`   ...`	`   ...`
`   delete x;`	`   x = null;`
`   ...`	`   ...`
` }`	` }`

The only case you have to worry about is where your destructor must also release system resources (such as open files), or perform some other global action (such as removing a corresponding object from a global list). In that case, you may need to put some of the guts of your destructor into a `finalize` method. Unfortunately, you can't control when this method gets called very well, so you may instead have to add an explicit `release` method, and call it in all the places where your object was destroyed in C++.

Difficult assignments

In C++, you generally define a copy constructor and an assignment operator. Both of these should be closely linked in the way they work. In Java, you could replace them both by the use of `clone`. However, to minimize the changes to your C++ code on the calling side (especially for output parameters), it is often easier to go ahead and write an `assign` method.

An assign method may also be faster, since it avoids the cost of making a new object by a `clone`.

> *You must be careful when writing correct clone, equals, and hashCode operators--see Well-Mannered Objects for more information.*

Fixing Assignment

C++	Java
`// defining`	`// defining`

`Foo(const Foo& other) {`	`public clone (Foo other) {`
	` Foo result = super.clone();`
` field1 = other.field1;`	
` field2 = ::Copy(other.field2);`	` field2 = other.field2.clone();`
	` return result;`
`}`	`}`
`Foo& operator= (const Foo& other) {`	`public Foo assign (const Foo& other) {`
` if (&other != this) {`	` if (other != this) {`
`  SuperOfFoo::operator=(other);`	`  super.assign(other);`
`  field1 = other.field1;`	`  field1 = value.field1;`
`  ::ReplaceByCopy`	`  field2 = other.field2.clone();`
`   (field2,other.field2);`	
` }`	` }`
` return *this;`	` return this;`
`}`	`}`
`// using`	`// using`
`Foo a = Foo(c);`	`Foo a = c.clone();`
`a = b;`	`a = b.clone();`

void getStuff(Foo& foo, Bar& bar) {
 	
 foo = otherFoo();
 	
 bar = otherBar();
 	
}
	public void getStuff(Foo foo, Bar bar) {
	
	 foo.assign(otherFoo());
	
	 bar.assign(otherBar());
	
	}

Decolonization

Double-colons occur in two places in C++: with statics and with direct base class methods. In both cases, Java has a different syntax, but there are few opportunities for error since the compiler will catch most mistakes.

All statics must be defined in a class, so you have to move your "unclassy" static data or global functions into an appropriate class, or make up a new class such as `Globals` .

Statics & Base Class Methods

C++	Java

// declaring
 	
class Foo {
 	
 static Foo x;
 	
 void someMethod();
 	
}
 	
int myGlobalFunction() {...
 	
static Foo y;
	// declaring
	
	class Foo {
	
	 static Foo x;
	
	 void someMethod();
	
	}
	
	class Globals {
	
	 static int myGlobalFunction() {...
	
	 static Foo y;

	`}`

`// using`
`a = Foo::x;`
`b = y;`
`c = myGlobalFunction();`
	`// using`
	`a = Foo.x;`
	`b = Globals.y;`
	`c = Globals.myGlobalFunction();`

`// declaring`
`class Fii : Foo {`
`  void someMethod() {`
`    Foo::someMethod();`
`  }`
`}`
	`// declaring`
	`class Fii extends Foo {`
	`  void someMethod() {`
	`    super.someMethod();`
	`  }`
	`}`

In Java, above the immediate superclass, you can't call base class methods directly. Luckily, calling higher base classes is rarely done in C++, so you should have few instances of it. If you do run into a case like this, then you will have to introduce some artificial methods of the class you want to call.

It's all conditional

Conditionals look very similar, except that Java enforces the type `boolean`. If the condition is flagged as an error by the compiler, then put it in parentheses, and add `!= 0`.

Fixing Conditionals

C++	Java
`if (x == 3) {}`	`if (x == 3) {}`
`if (x++) {}`	`if ((x++) != 0) {}`
`if (x = y) {}`	`if ((x = y) != 0) {}`

Notes

- The Java restriction has the extra benefit of ferreting out the unintended use of = instead of ==. One only wonders how many millions of dollars in time that little gem in C and C++ has cost overall; it is surprising that no one has yet filed a class-action suit against K & R!

Java has no `#if` or `#ifdef`. In many cases, these conditionals are not required since they are often used for marking machine-specific code, which is not a problem for Java. Generally, these macro conditionals can be replaced by use of a simple conditional, since Java optimizes away conditionals that evaluate to `false` at compile time.

Replacing `#ifdef`

C++	Java

`#define DEBUG false`
 	
`#if DEBUG`
` ...`
`#endif`
	`class Globals {`
	` static final bool DEBUG = false;`
	`}`
	
	`if (DEBUG) {`
	` ...`
	`}`

However, where you have commented out parts of a block or more than one method, there is just no good substitute for `#ifdef` in Java . Occasionally, `/*...*/` will substitute; but you have to be careful of premature termination since these marks do not nest, and people often have these comment blocks at the front of each method. The last resort is to copy the commented-out material to another file to preserve it, and then to put a comment in pointing to that file.

As a side issue, there is one slight change you might have to make to `for` statements, since declarations inside a `for` statement are scoped slightly differently for older C++ compilers. If there are outside dependencies you might have to pull the declarations out to a higher level, as shown below. The compiler will warn you of these.

Fixing `for`-Loop Declarations

C++	Java

`for (int i = 0; i < j; ++i) {`
`  x += i * i;`
`}`
`z = i;`
	`int i;`
	`for (i = 0; i < j; ++i) {`
	`  x += i * i;`
	`}`
	`z = i;`

A sign from above

The Java primitives do not have signed and unsigned variants. The `char` type is always `unsigned`, while the others are `signed`. First remove all `signed` keywords. (Since the `char` type in Java is larger than `char` in C++, removal of `signed` doesn't make a difference. This discussion presumes that you have already converted non-character C++ `char` to be `byte`, as in To protect the innocent).

Once you have removed `signed`, take a look at the `unsigned` types. If you really need the range they provide, then you will have to change them to the next higher type.

> If your C++ code was portable, you made few assumptions about the sizes of `int` since it could be 16, 32, or even 64 bits wide in C++, and the only one to watch for is `unsigned short`.
>
> The C and C++ languages officially say that bitwise operations on signed integers are not portable. People do it anyway, assuming that all machines are now two's-complement. Thankfully, Java officially defines signed integers to be two's-complement, and bitwise operations on them are reliably portable.

Once you are done, drop the `unsigned` keywords.

Unsigned

C++	Java
`// can be > 2,147,483,648`	
`unsigned int x;`	`long x;`
`// otherwise`	
`unsigned int x;`	`int x;`
`// can be > 32,767`	
`unsigned short x;`	`int x;`
`// otherwise`	
`unsigned short x;`	`short x;`
`signed int y;`	`int y;`
`z = x >> 1;`	`z = x >>> 1;`
`z = y >> 1;`	`z = y >> 1;`

If you are right-shifting an unsigned value, you will need to change to use >>>. Search for all instances of >>, and check the type of the arguments. Luckily, most code doesn't use this construct very much.

Also, watch out for number-wrapping assumptions. For example, with a 16-bit C++ `int`, `(x >> 8)` gives the high byte. With a 32-bit Java `int`, you need to mask off possible garbage in the top bits with `((x >> 8) & 0xFF)` to get the same result.

Defaults

Java does not have default parameters. If you really want them, you have to use overloaded methods, one for each defaulted parameter. (You can make them `final`, which with a good compiler will remove the overhead of overload.) You may find it simpler to replace the call sites instead, depending on your code.

Default Parameters

C++	Java

`int method(int x = 3,`
 	
` char c = 'a');`
	`public int method(int x, char c) {`
	
	`...`
	
	`}`
	
	`public final int method(int x) {`
	
	` return method(x,'a');`
	
	`}`
	
	`public final int method() {`
	
	` return method(3,'a');`
	
	`}`

Exceptional situations

The exception mechanism works pretty much the same in C++ and Java. The main differences are that--

1. As usual, you will need to create the exception with `new`.
2. The "catch everything" clause `(...)` is replaced by `Exception`, which is the base class for Java exceptions. (More precisely, `Throwable` is, but you generally don't need to worry about that.)
3. For the most part, you must declare which exceptions could be thrown by your class. These could be also thrown by Java class methods that you call, such as `System.in.readline()`. The compiler will let you know this, so you can generally just declare the ones it tells you to.

Exceptions

C++	Java
`void someMethod() {`	`void someMethod() {`
` try {`	` try {`
`  ...`	`  ...`
`  throw RangeException();`	`  throw new RangeException();`
`  ...`	`  ...`
` } catch (const RangeException& e) {`	` } catch (RangeException e) {`
`  ...`	`  ...`
` } catch (...) {`	` } catch (Exception e) {`
`  ...`	`  ...`
` }`	` }`
`}`	`}`
`void otherMethod() {`	`void otherMethod()`
	`   throws BadNewsException {`
` ...`	` ...`
` throw BadNewsException();`	` throw BadNewsException();`
` ...`	` ...`
`}`	`}`

Notes

- C++ also lets you declare thrown exceptions, but it doesn't force you to.
- More precisely, you must declare exceptions, except for those exception classes that descend from `RuntimeException` or `Error`. These latter exceptions are for situations which could occur in essentially all code, such as for out-of-memory or file-system-full. Java doesn't require declaring them, since it would be cumbersome and pointless to declare them in essentially all code.

Checking it twice

Unfortunately, Java has no `enums`. You will have to replace all of your `enums` by constants, and you will get no type-checking, and no overloading of methods based on the difference in types. Since you have no type-checking, callers are not prevented from mistakenly passing in some random integer instead of an `enum` value.

Enums

C++	Java
`class Button {`	`class Button {`
` enum ButtonState {inactive, active,`	` // ButtonStates`
`                  mixed, inherited};`	` public static final byte INACTIVE = 0;`
	` public static final byte ACTIVE = 1;`
	` public static final byte MIXED = 2;`
	` public static final byte INHERITED = 3;`
` ...`	`...`
`void method1(ButtonState newState) {`	`void method1(byte newState) {...`
` if (newState == inactive) {...`	` if (newState == INACTIVE) {...`
`}`	`}`
`// usage`	`// usage`
`x.setState(Button::inactive);`	`x.setState(Button.INACTIVE);`

Notes

- Instead of constant numbers, you could make each item relative to the previous. This will save time renumbering if you often insert items in the middle of the list. For example, `active = inactive + 1`.
- If you use `byte` instead of `int`, you recover a little bit of safety, since you can't just pass any number in. Most `enums` are small numbers, which enables this trick.
- Java coding conventions require you to uppercase constants. This may be a pain to do, since you have to change all the call sites as well as all the definitions. So, you may want to just leave them lowercased to minimize the work.
- If you really, really wanted your `enums` to be safe, you could encode them as a class. However, you probably will not find it worth the effort, since you have to make considerable changes to your method definitions and call sites, as you see below:

Replacing enum with a Class

C++	Java

`class Button {`
 	
` enum ButtonState {inactive, active,`
 	
`                   mixed, inherited};`
 	
` void method1(ButtonState newState) {`
 	
`  if (newState == inactive) {...`
 	
`}`
	`class Button {...`
	
	` void method1(ButtonState newState) {...`
	
	`  if (newState == ButtonState.INACTIVE) {...`
	
	`}`
	
	`final class ButtonState {`
	
	` public static final ButtonState INACTIVE`
	
	`  = new ButtonState(0);`
	
	` public static final ButtonState ACTIVE`
	
	`  = new ButtonState(1);`
	
	` public static final ButtonState MIXED`
	
	`  = new ButtonState(2);`
	
	` public static final ButtonState INHERITED`
	
	`  = new ButtonState(3);`

	` public int toInt() {`
	`  return state;`
	` }`
	` private ButtonState(int state) {`
	`  this.state = (byte) state;`
	` }`
	` private byte state;`
	`}`

Notes

- `ButtonState` is one of the few classes where == is the same as `equals`.
- If you use this technique, you have to change your `switch` statements to chains of `if` statements, since the compiler won't determine that `ButtonState.INACTIVE.toInt()` is a constant integer expression.

Not gooey at all

For simple text-only applications, or for debugging, you will want to know how to deal with arguments and how to print from the console.

Text-only Applications

C++	Java
`// fetching command-line arguments`	`// fetching command-line arguments`
	`public class MyApplication {`
`int main(int argc, char *args[]) {`	` public static void main(String args[]) {`

for (i = 1; i < argc; ++i) {	for (i = 0; i < args.length; ++i) {
doSomething(args[i]);	doSomething(args[i]);
}	}
...	...
// C-style simple output	// simple output
printf("%s%i", "abc", 3);	System.out.print("abc" + 3);
// C-style file output	// file output
	try {
FILE* output = fopen("aFile","r");	PrintStream output = new PrintStream(
if (output == NULL) {	new FileOutputStream("aFile"));
handleProblem();	
}	
fprintf(output, "%s%i", "abc", 3);	output.print("abc" + 3);
fclose(output);	output.close();
	} catch (java.io.IOException e) {
	handleProblem();
	}
// C++-style simple output	// simple output

`cout << "abc" << 3;`	`System.out.print("abc" + 3);`

Notes

- Use *plus signs* instead of *commas* to separate operands in the print statements.
- The file output code is reordered in handling error conditions

Unfortunately, an array doesn't have a meaningful `toString`, despite the fact that it would be easy to iterate over the contents. For debugging it is useful to code a replacement, as shown below:

Printing Arrays

```
// definition

static String arrayToString(Object[] array) {

 StringBuffer result = new StringBuffer("<");

 for (int i = 0; i < array.length; ++i) {

  if (i != 0) result.append(", ");

  result.append(array[i].toString());

 }

 result.append('>');

 return result.toString();

}



// usage

System.out.println(arrayToString(foo2));
```

This example also illustrates a common idiom: allocating a `StringBuffer`, successively appending to it, then returning its conversion to a `String`. This is much, much faster than using `String` concatenation, since the equivalent `result += array[i].toString()` constantly creates new objects.

Introduction, Basics, Next Steps, Well-Mannered Objects, Esoterica, Background, Index

Products Object Resources In the News Company Info Search

Well-Mannered Objects

The following sections describe particular issues that are common to almost all classes, but are often tricky to get right.

Bullet-proofing

The most important language feature missing from Java is `const`. The absence of this feature significantly compromises the robustness of your code.

In Java, you can't determine on an object-by-object basis whether someone can change an object; you can only do it on the *class* level. This significantly complicates your life, if you want to provide the same level of robustness against mistaken modifications as you have in C++.

With the Java paradigm, the only way to have constant objects is to write an Immutable class. There are certainly advantages to this approach:

- Immutables do not need to be cloned.
- Multiple variables and other objects can safely refer to Immutables without fear that some object will modify them behind their backs.
- Immutables are also thread-safe, without taking special provisions.

The downside is that in order to make an object Immutable, you generally have to write two classes, with a fast conversion between them. We see that with `String` and `StringBuffer`. `StringBuffer` provides a mechanism to modify strings, while `String` provides the Immutable counterpart. Behind the scenes, they are designed to share a single character buffer--where possible--so that conversions are not too onerous.

Short of taking this approach, it is difficult to maintain the advantages of `const` when porting your code. Suppose that you are returning a `const` pointer from a getter. Without `const` the integrity of your object can be compromised if someone mistakenly alters the object returned from the getter. Suppose that you are passing your object in as a parameter. Without `const` you have no indication when your object is just an input parameter, and when it could be modified (perhaps mistakenly) behind your back.

Our recommended approach with the current Java language definition is to write an Immutable interface, one that provides API for just the "`const`" methods, such as getters. If you then return (or pass) objects of type Immutable, you get the same degree of safety as in C++. (Note that, just as in C++, the "constness" can be cast away, so it doesn't prevent malicious coders!)

Replacing `const`

C++	Java
`// definition`	`// definition`
`class Foo {`	`class Foo implements ConstFoo {`
` public:`	
`  int getSize() const;`	` public int getSize();`
`  int setSize();`	` public int setSize();`
` private:`	
`  int size;`	` private int size;`
`}`	`}`
`// in another class's definition`	`// in another class's definition`
`const Foo* method1()  {...`	`ConstFoo method1() {...`
`void method2(`	`void method2(`
`  const Foo& input,`	`  ConstFoo input,`
`  Foo& output) {...`	`  Foo output) {...`
`// usage`	`// usage`
`const Foo* y = x.method1();`	`ConstFoo y = x.method1();`
`z = y.getSize();`	`z = y.getSize();`
`y.setSize(3); // compilation error`	`y.setSize(3); // compilation error`
`(*(Foo*)&y).setSize(3); // cast`	`((Foo)y).setSize(3); // cast`
	`// additional interface`
	`interface ConstFoo {`
	` int getSize();`
	`}`

The other safe alternatives are:

- Clone the pointer before returning. This has the advantage of requiring a small amount of programming effort, but may be a performance hit. See Doppelgänger and Don't try this at home.
- Write an Immutable cover class, one that *delegates* all of its calls to the original class. This is a real class, not just an interface, and cannot be cast away. This approach provides complete safety, but at the cost of considerably more work and some performance (an extra method call for every delegated method).

Really Safe

C++

```
// definition
class Foo {
 public:
  int getSize() const;
  int setSize();
 private:
  int size;
}

// in another class's definition
const Foo* method1() {...

void method2(
  const Foo& input,
  Foo& output) {...

// usage
const Foo* y = x.method1();
z = y.getSize();
y.setSize(3); // compilation error
(*(Foo*)&y).setSize(3); // cast
```

Java

```
// definition
class Foo {

 public int getSize();
 public int setSize();

 private int size;
}

// in another class's definition
SafeFoo method1() {...

void method2(
  SafeFoo input,
  Foo output) {...

// usage
SafeFoo y = x.method1();
z = y.getSize();
y.setSize(3); // compilation error
((Foo)y).setSize(3); // comp. error

// additional class
final class SafeFoo {
 public int getSize(); {
  return foo.getSize();
 }
 private Foo foo;
}
```

Be careful of static final data fields; unless they are Immutable, they are ***not*** safe. You have to use the same techniques as shown above to make them so.

Unexpected Damage

```
// declaration
import java.awt.Point;
class Foo {
 public static final Point ORIGIN = new Point(0,0);
}

// usage
Point y = Foo.ORIGIN;
y.translate(3,5);

// Danger, Will Robinson!
// ORIGIN has been changed to be <3,5> at this point!
```

On pins and needles

Thread-safety is a new concept for many C++ programmers. The C++ language provides no standard assistance for multithreaded programs, so all of the C++ synchronization (if any) is dependent on external libraries. Since it appears explicitly, you should be able to translate it according to the semantics of that library into explicit synchronization calls. However, you will need to understand both how the particular C++ synchronization and how Java's synchronization work.

Java offers powerful, built-in support for threads, but you will need to design your classes for thread-safety to ensure that they work properly. In general, your classes will fall under three cases.

No thread-safety

If your class will *only* ever be used in a single thread, you don't need to do anything.

Minimal thread-safety

Minimal thread-safety allows you to use different instances in different threads, but not references from two threads to the same object. To make your class minimally thread-safe, determine which fields have class data (a.k.a. static data) that can be altered. Synchronize all methods that access or change that static data. (This actually overstates it a bit; you need only synchronize the actual code that accesses that data, not the entire routine. However, it may be simpler in porting to just add the `synchronized` keyword to these methods in your first pass.)

If you don't make your classes minimally thread-safe, you can get into trouble. Imagine what happens if in thread A, `object1` is trying to access static data, while in thread B, a completely different `object1` (but of the same class, or a subclass) is modifying the same static data!

Full thread-safety

With fully thread-safe objects, you don't have to worry how you use them at all. Full thread-safety allows two different threads to have variables referring to the same object, with either one able to make changes to that object without causing problems. As well as making the changes for minimal thread-safety, you have to synchronize *all* methods that either change instance data (a.k.a. object data), or access instance data that could be changed after the construction of the object.

There is a price for full-thread safety: access to your object is always slower, even if the object is not being used in a multithreaded environment. Full thread-safety is not generally necessary for all objects.

Immutables don't need to be synchronized in order to be fully thread-safe, except for those methods that change hidden caches. For example, Locale is Immutable, but there is a `hashCode` method that changes a hidden data field. That method then has to be synchronized.

Even if you follow the above guidelines, you need to make sure that the objects are left in a consistent state whenever any method returns. Unless additional synchronization mechanisms are set up, client code of your class can't do any transaction-like operations that span multiple calls. For example, if two threads are both iterating through a `Vector` and reversing the order of the elements at the same time, even if all of the methods are synchronized the results can be undefined. Complete guidelines to thread-safety are beyond the scope of this article.

If an object has only minimal thread-safety, callers have to do their own synchronization for that object if it can be referenced by multiple threads; e.g., by protecting all the code that accesses that object.

Liberté, Égalité, Fraternité

The way Java is set up, classes should implement `hashCode` and `equals`[1]. However, it is easy to get these wrong, and the failures may be difficult to debug. Although Java memory management saves some complications, there are other problems similar to those of C++. Unless you are aware of these problems, you will get non-robust *(fragile)* code. So here is a fairly complete example of how to write `equals`.

As discussed under **Basics**, there is quite a difference between == and `equals()`. The operator == represents pointer identity, while `equals` represents value or semantic equality. To correctly define `equals`, you must make sure that the following principles are observed.

Semantic Equality
: If you use the same steps to create `x` as you do to create `y`, then `x.equals(y)`.

Symmetry
: If `x.equals(y)`, then `y.equals(x)`.

Transitivity
: If `x.equals(y)`, and `y.equals(z)`, then `x.equals(z)`

If you don't maintain these invariants, then users of your code (a.k.a. clients) will become rather annoyed when your class doesn't work as expected, or--worst yet--data structures can become corrupt (see Making a hash of it).

Note that if you depend on inheriting the default implementation of `equals` from `Object`, you will get the wrong answer! The default implementation, as we see below with `StringBuffer`, does *not* preserve semantic identity.

Bad `equals` in Action

```
// use same steps to create x and y
StringBuffer x = new StringBuffer("abc");
StringBuffer y = new StringBuffer("abc");

// failing code
if (x.equals(y)) {
 System.out.println("Correct!"); // never reached
}

// work-around, for this case
if (x.toString().equals(y.toString())) {
 System.out.println("Correct!");
}
```

Here is an example of how to correctly implement `equals`, with the different cases that you may be faced with annotated.

Implementing `equals`

```
 1  public boolean equals(Object obj) {
 2    if (this == obj) return true;
 3    // if (!super.equals(obj)) return false;
 4    if (getClass() != obj.getClass()) return false;
 5    Sample other = (Sample)obj;
 6    if (myPrimitive != other.myPrimitive) return false;
 7    if (!myObject.equals(other.myObject)) return false;
 8    if ((myPossNull == null) {
 9      if (other.myPossNull != null) return false;
10    }
11    if (!myPossNull.equals(other.myPossNull)) return false;
12    // if (!myTransient.equals(other.myTransient)) return false;
13    if (myBad.getSize() != other.myBad.getSize()
14       || myBad.getColor().equals(other.myBad.getColor()) return false;
15    return true;
16  }
```

Notes

Line Comment

2. This quick check is worth it.

3. Uncomment this line *if-and-only-if* the immediate superclass is *not* `Object`; otherwise you will get the wrong result!

4. So why don't we write the following?

   ```
   if (!(obj instanceof Sample)) return false;
   ```

   Here is why. Suppose `A` is a superclass of `B`, and we are comparing two objects of those classes, `a` and `b`.

   - In the code for `a.equal(b)`, `(b instanceof A)` is **true**.
   - But in the code for `b.equals(a)`, `(a instanceof B)` is **false**.

   Using `(getClass() != obj.getClass())` instead solves this problem. If you have a special hierarchy (such as Number) where you want equality checks to work across some different classes (but not others), then you will need to use special code. You can do it, but be forewarned that such cases get very tricky unless you have a closed set of classes, with no outside subclassing!

8. You need this more complicated code if a field could be `null`.

12. Transient fields, such as caches, are irrelevant to the equality of the object, and must be ignored.

13. If one of your fields does not implement `equals` correctly, then you have to do your own comparison.

   (I have seen some people use `toString()` to work around bad `equals`. Don't do it except with `StringBuffer`. The `toString` method is relatively expensive and not guaranteed to contain the complete state of the object. In practice, if objects can be reasonably converted to a string, `toString` is used for the name of that method. If objects cannot be, then `toString` spews whatever debugging information the class designer thought worthwhile.)

Making a hash of it

The way Java is set up, most classes should implement `hashCode` and `equals`. However, it is easy to get these wrong, and the failures may be difficult to debug. So here is a fairly complete example of how to write `hashCode`.

Writing `hashCode` is much simpler than writing `equals`. The only strict principle that you absolutely must follow is:

Agreement with Equality
    If `x.equals(y)`, then `x.hashCode() == y.hashCode()`.

If you don't maintain this invariant, then HashTable data structures get corrupt! Here is an example of how to correctly implement `hashCode`, with the different cases that you may be faced with. You will see that this corresponds closely with the code for `equals`.

*Unlike `equals` , `hashCode` does **not** need to use all the nontransient fields of an object; just enough of them to get a reasonable distribution from `0` to `Integer.MAX_VALUE` .*

Implementing HashCode

```
 1  public int hashCode() {
 2    int result = 0;
 3    // result ^= super.hashCode();
 4    result = 37*result + myNumericalPrimitive;
 5    result = 37*result + (myBoolean ? 1 : 0);
 6    result = 37*result + myObject.hashCode();
 7    result = 37*result +
 8     (myPossNull != null ? myPossNull.hashCode() : 0);
 9    // if (!myTransient.equals(other.myTransient)) return false;
10    return result;
11  }
```

Notes

Line	Comment
...	Why 37, you might ask? Actually, any reasonably sized prime number works pretty well.
3.	Uncomment this line *if-and-only-if* the immediate superclass is *not* `Object`; otherwise you will get the wrong result!
7.	You need this slightly more complicated code if a field could be `null`.
9.	Transient fields, such as caches, are irrelevant to the equality of the object, and should be ignored. You *must not* include any fields in your `hashCode` that are not included in your `equals` code.

If your keys in a `Hashtable` are not Immutable, be careful; if you change the value of the key you must *first* remove the key-value pair from the table, and then re-enter the pair after you change the value of the key. Otherwise your `Hashtable` becomes corrupt!

Doppelgänger

Implementing `clone` allows other programmers to use your objects as fields and to safely implement getters, setters, and `clone` themselves. You should provide a `clone` operator for all of your classes.

However, suppose you are feeling lazy, and want to get away with the absolute minimum. You do not need to provide a `clone` method if your superclass does not implement a public `clone` method, and your object falls under one of the following cases:

- It is `Immutable`, or
- It would never be a field in another object that itself will need to implement `clone`, or
- It is `final`, and can be duplicated with public getters and setters. (That is, your object can be duplicated by getting all of the state of your object with public getters, then creating a new object with the identical state.)

The only strict principles that you must follow for `clone` are:

Clone Equality
: If `y == clone(x)`, then `x.equals(y)`.

Clone Independence
: If `y == clone(x)`, then no setter on `y` can cause the value of `x` to be modified.

This is what is known as a *deep clone* . There are cases where it may make sense to provide a *shallow clone* , especially with collection classes. Such a shallow clone only clones the top-level structure of the object, not the lower levels. A shallow clone is useful in many circumstances so long as programmers can somehow still implement a deep clone on top of those objects. Ideally, the class would implement both, with a separate method called `shallowClone`.

Here is an example of how to correctly implement `clone`, with the different cases that you may be faced with.

Implementing Clone

```
 1  protected Object clone() throws CloneNotSupportedException {
 2   Sample result = (Sample) super.clone();
 3   result.myGood = (Good) myGood.clone();
 4   result.myTransient = null;
 5   result.myVector = (Vector) myVector.clone();
 6   for (int i = 0; i < myVector.size(); ++i) {
 7    result.myVector.setElementAt(
 8      ((Cloneable) myVector.elementAt()).clone(), i);
 9   }
10   result.myBad = new Bad(myBad.getSize(), myBad.getColor());
11   result.myBad.setActiveStatus(Bad.INACTIVE);
12   return result;
13  }
```

Notes

Line	Comment
2.	This copies the superclasses fields, and makes bitwise copies of your fields. *You do not have to copy any primitives or Immutables (such as `String`) in the rest of your code.*
3.	You should set your transient fields to an invalid state, to signal that they need to be rebuilt. Do this if the field is Mutable and not shared between objects.
6.	If the members on the `Vector` are Immutable, then you don't have to clone them, as in lines 6-9. Use the same style for arrays: for example, you can just call `foo = (int[])other.foo.clone();`
8.	Unfortunately, this method of deep-cloning a `Vector` (or array, or `Dictionary`) actually will not work, because of an annoying flaw in the `Cloneable` interface; surprisingly, it does not have `clone()` as a method! (And `Object`'s `clone` is `protected`, not `public`.) This is despite the statement in JPL (page 68) that "The `clone` method in the `Cloneable` interface is declared `public`..." The result is, you cannot polymorphically implement `clone` in many cases; you have to have preknowledge of the precise type (or an overall superclass) of the objects in the collection, and cast them to that type to call their `clone` operator. *Keep your fingers crossed that flaw is fixed in JDK 1.1!*
10.	If the author of the `Bad` class was a bit lazy, and did not supply you with a `clone` operator, you will do it yourself with a constructor and setters as necessary. If the object is of a subclass of `Bad` which you are not aware of, then despite your best efforts the object will be sliced, and data will be lost.

In implementing clone, getters, setters, and thread-safety, Immutable would actually be a very useful Java interface. Although it is not in the Java class libraries, you may find it useful to define it as an interface in your own code.

Don't try this at home

Getters and setters seem trivial, but incorrect construction can leave your object open to pernicious bugs. For example, look at the following:

Dangerous Getter/Setters

```
1   // definition
2   public Foo[] getFooArray() {
3    return fooArray;
4   }
5
6   public setFooArray(Foo[] newValue) {
7    fooArray = newValue;
8   }
9
10  private Foo[] fooArray;
11  ...
12  // usage
13  Foo[] y = x.getFooArray();
14  y[3].changeSomething();
15
16  x.setFooArray(z);
17  z[3].changeSomething();
```

With these setters and getters, lines 14 and 17 change the state of your object behind your back. If you had other state in your object that needed to be in sync with `fooArray`, you are now in an inconsistent state. Moreover, even if you didn't have such state, if any of your potential subclasses had such state, they would now be corrupted.You might just as well have made `fooArray` public!

If your field is Immutable or a primitive, then you can just use the simple code with perfect safety. If not, then you need to consider the use of your field. Your choices are:

- For complete safety, clone the field in getters and setters of Mutables. The downside of this approach is that you take a certain performance hit, sometimes an unacceptable one.
- For pretty good safety, use a read-only interface on your getter, as in Bullet-proofing. This prevents most accidents from happening. For full safety, you still would need to clone incoming Mutable parameters in your setter.
- Bite the bullet, document what changes are acceptable to make to objects that are gotten or set, and depend on your callers not to make a mistake.

Allegro ma non troppo

There is a technique for speeding up `equals` and `hashCode`. It is worth implementing under the following circumstances:

- You are doing a lot of equality comparisons or `hashCode` calls.
- Your objects don't change often.

While it speeds up hashing and comparison dramatically, if your objects are not compared or hashed very often, don't bother using this technique.

This technique provides some very fast checks for equality by adding a version count and a hash cache. To use it, add the following code marked in blue to your class definition. Then, in any of your methods where you alter any of the nontransient fields of the object, call `changeValue`.

Fast equals & hashCode

```
public int hashCode() {
 if (hashCache == -1)
  hashCache = <old hashCode computionation code here>
  if (hashCache == -1) {
   hashCache = 1;
  }
 }
 return hashCache;
}

public boolean equals(Object other) {
 if (other == this) return true;
 if (getClass() != other.getClass()) return false;
 MyType x = (MyType) other;
 if (versionCount == x.versionCount) return true;
 if (hashCache != x.hashCache) return false;
 <rest of old field comparison code here>
 if (versionCount < other.versionCount) {
  versionCount = other.versionCount;
 } else {
  other.versionCount = versionCount;
 }
 return true;
}

public MyType setFoo(ConstFoo newValue) {
 foo = newValue;
 changeValue();
}

// ============= privates =============

private static int masterVersionCount = 0;
private long versionCount = 0;
private int hashCache = -1;

private final void changeValue() {
    hashCache = -1;
    versionCount = ++masterVersionCount;
}
```

Theoretically, you could have a problem with the `versionCount` wrapping back to zero. However, even if you altered your objects once every nanosecond, it would still take over 100 years for a wrap to occur. However, if you *really* want to be safe, instead of incrementing `versionCount` you can use the clever trick of allocating a new `Object` each time. This will be airtight even in the days of Terahertz processors.

Pitfalls

- Suppose that you want to remove characters from a `StringBuffer`. Unfortunately there is no method to do so; you have to resort to the following code to delete from `start` to `end`.

```
a = new StringBuffer(a.toString().substring(0,start)))
        .append(a.toString().substring(end,a.length()));
```

- `StringBuffer` doesn't implement `equals` correctly, as discussed in Liberté, Égalité, Fraternité.
- There is no constructor to make a `String` from a `char`, so use:

```
String foo = new String(ch + "");
```

  Similarly, the following code doesn't do what you expect; since there is no explicit constructor for a `char`, `StringBuffer` casts up to an `int` and allocates a buffer of length `0x61`!

```
StringBuffer result = new StringBuffer('a');
```

- In `String`, the version of `indexOf` and `lastIndexOf` that searches for characters has the `char` typed as an `int`. This makes it easy to make a mistake, as illustrated in the following code, which searchs for `(char)start` in `myString`, starting at offset `(int)myChar`!

```
position = myString.indexOf(start, myChar);
```

- Unfortunately, many objects (`StringBuffer, Vector, Dictionary`...) do not implement a `clone`, or implement only a shallow `clone`. This causes a number of problems: see Doppelgänger.
- The methods `DataInput.readline` and `PrintStream.println` only handle `'\n'` delimited strings properly. If you are reading and writing platform-specific text files (which is the vast majority of the cases!), you will have to work around that. Luckily, you can get the line delimiter from

```
static String eol = System.getProperties()
 .getProperty("line.separator");
```

  So to handle output, just replace `println(x)` with `print(x+Globals.eol)`. Input is a quite a bit more annoying; you will have to write your own input routine that recognizes `eol` instead of just `\n`.

Taligent.

taligent white papers

Object Resources

Esoterica

The following sections deal with somewhat less common constructs in C++.

Primogenitur entail

Java does not support multiple inheritance. It *does* support *interfaces*, which can get you a long way towards replacing multiple inheritance. You can think of interfaces as fully abstract classes, with no data fields and all pure virtual methods. If all but one of the base classes for your class are fully abstract classes, then just turn them into interfaces.

Simple Multiple Inheritance

C++	Java

```
// Bar is fully abstract
class Bar {
 ...
 void someMethod() = 0;
}

// simple multiple inheritance
class Foo : Fii, Bar {...
```
	```
	// Bar is purely abstract
	interface Bar {
	 ...
	 void someMethod();
	}

	// simple multiple inheritance
	class Foo extends Fii
	          implements Bar {...
	```

If the inheritance is not simple, then you will have to do some more work on your target class. First, pick the base class that is most central to the function of target class; you will leave that one alone. Then for each of the *other* base classes:

1. Define a new interface for each one to implement.
2. Declare a field of that type in your target class.
3. Delegate methods in your target class to that field.

General Multiple Inheritance

C++	Java

`// Other base classes`
`class Bar {`
` ...`
` void methodA();`
`}`
 	
`class Foe {`
` ...`
` int methodB();`
`}`
 	
`// simple multiple inheritance`
`class Foo : Fii, Bar, Foe {`
 	
` ...`
 	
`}`
	`// Other base classes`
	`class Bar {`
	` ...`
	` void methodA();`
	`}`
	
	`class Foe {`
	` ...`
	` int methodB();`
	`}`
	
	`// simple multiple inheritance`
	`class Foo extends Fii`
	`          implements BarInterface,`
	`            FoeInterface {...`
	` ...`
	` void methodA() {`
	`  bar.methodA();`
	` }`
	` void methodB() {`
	`  return foe.methodB();`
	` }`
	` private Bar bar = new Bar();`
	` private Foe foe = new Foe();`
	`}`
	
	`// Interfaces`
	`interface BarInterface {`
	` ...`
	` void methodA();`
	`}`
	
	`interface FoeInterface {`
	` ...`
	` void methodB();`
	`}`

Size doesn't matter

In Java, there is no equivalent to the C++ `sizeof` function. However, you generally only use `sizeof` when you are doing unions, bit-field manipulations, or C-style memory management. To see how to port the code, you will need to look carefully at the intent. For example:

Replacing `sizeof`

C++	Java
`x = (Foo*) malloc(sizeof(Foo)*len);`	`x = new Foo[len];`

Shave and a haircut

In Java, there is no equivalent to C++ bitfield notation. Generally you can just dispense with the notation. If you *really* need bitfields to save storage, then you will need to do it yourself, basically by duplicating the code that is behind the use of bitfields in C++. If you are using large numbers of single bits, use `java.util.BitSet` instead (using `Bitset` is safer and easier than managing the masks and shifting yourself, but will not save you storage unless you have a significant number of bits).

Replacing Bitfields

C++

```
// declaring
struct Foo {
 // ...
 unsigned int x:5, y:9, z:3;
}
```

```
// using
a = myFoo.z;
z = myFoo.b;
```

Java

```
// declaring
class Foo {
 // ...
 public int getZ() {
  return zFields.extract(xyz);
 }
 public void setZ(int newValue) {
  xyz = zFields.insert(xyz, newValue);
 }
 private int xyz;
 static BitFields xFields
  = new BitFields(5,0);
 static BitFields yFields
  = new BitFields(9,5);
 static BitFields zFields
  = new BitFields(3,14);
}
```

```
// using
a = myFoo.getZ();
myFoo.setZ(b);

// new class
public class BitFields {

 public BitFields (
   int bitCount,
   int leastSignificantBit) {
  lsb = leastSignificantBit;
  mask = ((1 << bitCount) - 1) << lsb;
 }

 public final int extract
   (int source) {
  return (source & mask) >> lsb;
 }

 public final int insert
   (int source, int value) {
  return (source & ~mask)
   | ((value << lsb) & mask);
 }

 private int mask;
 private int lsb;
}
```

Scabs

In Java, there is no equivalent to C++ `unions` and `structs`. Generally `structs` are easy--just change them to classes. Remember that in C++ the default access control in `structs` is `public` (while in classes it is `private`), so mark members accordingly.

Replacing `structs`

C++	Java

`// declaring`
`struct Foo {`
`  int x;`
`  float y;`
`}`
	`// declaring`
	`class Foo {`
	`  public int x;`
	`  public float y;`
	`}`

If you are using bitfield notation, see Shave and a haircut.

The easiest approach to porting a `union` is also just to make it a class.

Unions

C++	Java

`// declaring`
`union Foo {`
`  bool isFiiVsBarr;`
`  Fii x;`
`  Barr y;`
`}`
	`// declaring`
	`class Foo {`
	`  public bool isFiiVsBarr;`
	`  public Fii x;`
	`  public Barr y;`
	`}`

Where `unions` are being used for storage savings and the fields are not primitives, you can sometimes get the same effect by using `Object`. The only disadvantage is that you will have to cast to the right type being used.

Unions

C++	Java

`// declaring`
`union Foo {`
`  bool isFiiVsBarr;`
`  Fii x;`
`  Barr y;`
`}`
 	
`// using`
`if (isFiiVsBarr) {`
`  z = myFoo.x;`
`}`
	`// declaring`
	`class Foo {`
	`  public bool isFiiVsBarr;`
	`  public Object xy;`
	
	`}`
	
	`// using`
	`if (isFiiVsBarr) {`
	`  z = (Fii)myFoo.xy;`
	`}`

Where `unions` are being used for scurrilous casting, you will have to work around it. For example, where such castings are used for hidden bit-manipulations, you'll have to use the appropriate arithmetic operations, as below. On the plus side, you will have the advantage of much more portable code in the end, without big-endian or little-endian troubles.

Bit Twiddling in Unions

C++	Java

`// declaring`
`union Foo {`
`  int i;`
`  char c;`
`}`
 	
`// using`
`x.i = 99;`
`z = x.c;`
 	
`// traditional, awful floating-point hack`
`union yech { long x; float y; } convert;`
`convert.y = pi;`
`exponent = (convert.x >> 23) & 0xff;`
	`// declaring`
	`int i;`
	
	`// using`
	`x = 1066;`
	`z = x & 0xFF;   // if C++ was BE`
	`z = x & 0xFF00; // if C++ was LE`
	
	`// clean, portable Java solution`
	
	`int exponent =`
	`  (Float.floatToIntBits(pi) >> 23) & 0xff;`

Directly to jail

Well-written C++ code should have very few, if any, `gotos`. However, there are times when a `goto` produces less convoluted code: where you need to escape from an inner loop. Although Java has completely eliminated `gotos`, it has added a clever construct that replaces their use, and in a much cleaner and less dangerous way. You name a loop with a label, then use `break` or `continue` with that label to escape from an inner block.

Go to Block End

C++	Java

`for (int i = 0; i < n; ++i) {`
` for (int j = 0; j < m; ++j) {`
`  if (f(i,j)) {`
`   ...`
`   goto done;`
`  }`
` }`
`}`
`done:`
	`mainLoop:`
	`for (int i = 0; i < n; ++i) {`
	` for (int j = 0; j < m; ++j) {`
	`  if (f(i,j)) {`
	`   ...`
	`   break mainLoop;`
	`  }`
	` }`
	`}`

If your `goto`s don't follow this pattern, then they are still fairly easy to convert as long as they don't cross blocks. This is a bit kludgy, but saves your having to go in and figure out a mass of snarled code.

Go to Higher Levels

C++	Java

`{...`
` {...`
`  {...`
`   goto done;`
`  ...`
`  }`
` ...`
` }`
`...`
 	
`}`
`done:`
	`kludgeLoop:`
	`while (true) {...`
	` {...`
	`  {...`
	`   break kludgeLoop;`
	`  ...`
	`  }`
	` ...`
	` }`
	`...`
	`break;`
	`}`

If your `goto`s jump into the middle of nested blocks (such as into a `switch` statement), then you will have no choice but to try to untangle the code.

Java has no friends

Java doesn't have the `friend` keyword. You can, however, permit access to your `private`s by any other class in your package by making the class or methods *package-private* . You do this by omitting the keyword `private` from your classes, methods, or data fields, and ensuring that the former `friend`s are in the same package. If you need to allow access to `protected` fields or methods, then you have to write cover methods that allow package-private access.

Replacing `friend`

C++

```
class Foo {
private:
 int foe;
protected:
 int fii;
friend class Bar;

}

class Bar {
 private Foo foo;
 public method() {
  ...
  y = foo;
  z = fii;
  ...
 }
}
```

Java

```
class Foo {

 int foe;

 protected int fii;

 int getFii() {...}
}


class Bar {
 private Foo foo;
 public method() {
  ...
  y = foo;
  z = getFii();
  ...
 }
}
```

Notes

- If you need to have `friend` access from two different packages, then you are out of luck. Your only choices are:
  - To make the methods or fields public, or
  - To copy the class into both packages (this works for small classes)

Liposuction

Java does not have templates, which many people [2] will miss. Some of the principal applications of C++ templates can be replaced by a use of `Objects` or `Cloneables` in Java.

Replacing `template`

C++	Java

`template <AType>`
`class Set {`
` public:`
 	
` void add`
`  (const AType& toAdd) {...`
 	
` bool contains`
`  (const AType& toTest) {...`
` ...`
`}`
	`class Set {`
	
	` public void add`
	`  (Object toAdd) {...`
	
	` public boolean contains`
	`  (Object toTest) {...`
	` ...`
	`}`

Other very useful cases in C++ are memory janitor functions like `autoptr`, which are not necessary in Java. However, if you are looking at replacing more entrenched uses of templates, you will be driven to making multiple copies of the code. For example, we have a `CompactArray` class in C++, for doing maps from Unicode characters to data (typically primitive data). Since the whole purpose of the templatized class is to save space, just using `Objects` instead of the primitives was not an option when we ported to Java.

Mutatis Mutandis

C++	Java

`template <AType>`
`class CompactArray {`
` public:`
` AType elementAt (UniChar ch) {`
` ...`
`}`
	`class CompactArray_Short {`
	
	` public short elementAt (char ch) {...`
	` ...`
	`}`
	
	`class CompactArray_Byte {`
	
	` public byte elementAt (char ch) {...`
	` ...`
	`}`
	
	`...`

Off the charts

Most C++ code doesn't use the newest features of C++, so we won't devote much space to them. These include:

New C++ Features

C++	Java
namespaces	Use packages.
covariant return types	Use the common base class as the return type.
declarations in "if", "while", "switch"	Embed in a {} block, and move the declaration out to the block level. See It's all conditional.

Other features, though not new, shouldn't occur in much C++ code.

Uncommon C++ Features

C++	Java
variable numbers of arguments	Replace by array of highest base class.
placement `new`	Replace with regular `new`.
function (or method) pointers	Use Functor classes.

The last item is perhaps worth an example.

Function Pointers

```
// create an interface for the kind of function you want to call
interface Functor {
 void handle(Object obj);
}

// create a concrete class that implements it,
// perhaps calling a method on your other objects to do the actual work
class AppendFunctor implements Functor {
 public void handle(Object obj) {
  result.append(obj.toString());
 }
 public String toString() {
  return result.toString();
 }
 private StringBuffer result = new StringBuffer();
}
```

Background Information

The following sections provide background information, plus an index to topics.

References

I only mention a few books that I think particularly useful. There is already a huge, and growing, list of introductory Java programming books. If you are interested, there are some pretty good book reviews on the net, such as:

- http://www.cbooks.com/java.html
- http://www.webreference.com/books/programming/java.html

Introductions to Java

David Flanagan	*Java in a Nutshell: A Desktop Quick Reference for Java Programmers (Nutshell Handbook)* O'Reilly & Assoc, 1996 ISBN: 1565921836
Ken Arnold, James Gosling	*The Java Programming Language (Java Series)* Addison-Wesley, 1996 ISBN: 0201634554
James Gosling, Bill Joy, Guy Steele	*The Java Language Specification (Java Series)* Addison-Wesley, 1996 ISBN: 0201634511

Java and C++

Barry Boone	*Java Essentials for C and C++ Programers* Addison-Wesley Developers Press, 1996 ISBN: 020147946X
Michael C. Daconta	*Java for C/C++ Programmers* Wiley Computer Publishing, 1996 ISBN: 0471153249

Object-oriented programming

Taligent (David Goldsmith)	*Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++* Addison-Wesley, 1994 ISBN: 0201408880
Erich Gamma, Richard Helm, Ralph Johnson, John Ulissides	*Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley, 1994 ISBN: 0201633612
Scott Meyers	*Effective C++: 50 Specific Ways to Improve Your Programs and Designs* Addison-Wesley, 1992 ISBN: 0201563649

About the author

Dr. Mark Davis is the director of the Core Technologies department at Taligent, Inc, a wholly owned subsidiary of IBM. Mark co-founded the Unicode effort, and is the president of the Unicode Consortium. He is a principal co-author and editor of the Unicode Standard, Version 1.0 and the new Version 2.0.

Mark has considerable expertise in both management and software development. His department encompasses Operating System Services, Text, International, Web Server Components, and Technical Communications. Technically, he specializes in object-oriented programming and in the architecture and implementation of international and text software: ranging from the years he spent programming in Switzerland, to co-authoring the Macintosh KanjiTalk and the Macintosh Script Manager (which later became WorldScript), to authoring the Arabic and Hebrew Macintosh systems, and most recently to architecting the CommonPoint international frameworks and the bulk of the Java 1.1 international libraries.

Mark has a doctorate from Stanford University, and is an avid reader of Jane Austin and follower of NPR's "Car Talk." This may help to explain the section titles.

Acknowledgments

The material in this article is based on collective years of experience at Taligent with the development of large object-oriented projects, plus more recent experience in porting from C++ to Java--and back again. The article itself was written within a very short time frame; my special thanks to the following individuals (in chronological order) whose efforts made this possible:

- Doug Felt, for his careful reading of the first and second drafts, suggestions of topics to cover, and identification of early errors.
- John Fitzpatrick, for his technique of read-only interfaces to replace `const`.
- M. Srinivasan, Anson Mah, Tony Tseung, and Jay Tobias for catching various errors in the second draft.
- Andy Heninger, for his detailed feedback on thread-safety in On pins and needles and for catching many typos.
- Mike Potel, for his many clarifications of wording.
- Bill Gibbons, for his detailed review of the C++ side, and improvements to the Shave and a haircut examples.
- Guy Steele, for correcting a number of fine points in Java, and especially for improvements to the Allegro ma non troppo section.
- Denise Costello, for her tireless work in managing media and artwork.
- Brian Beck, for some good last-minute catches.
- Odile Tarazi, for her final editing on short notice.

I hasten to add that these contributors have all reviewed different drafts of the document, and that they bear no responsibility for errors in the final version!

We envision continuing to develop articles of this flavor. If you have any criticisms, suggestions, or encouragement, please email cookbook@taligent.com.

Topic index

The following is an alphabetical list of the topics covered in this paper. Although most of the topics are relatively independent, the ones in Basics and Well-Mannered Objects may need to be read the first time in sequence.

Footnotes

[1] Unfortunately, `Object` defines `equals` and `hashCode` to be public. A better solution would have been to have followed the pattern of `Cloneable` by defining:

- A `Comparable` interface that contains `equals` and `hashCode`
- A `MethodNotSupportedException` for classes that don't want to implement them

Well, that's water under the bridge at this point. The only improvement to Java that would not break backward compatibility would be to at least allow `equals` and `hashCode` to throw a `MethodNotSupportedException`.

[2] By the way, I'm not one of them. For us, large-scale introduction of C++ templates were an absolute, unmitigated disaster, costing our project hundreds of person-months to manage the code size and interface problems they introduced. If JavaSoft introduces templates (a.k.a. *generics*), I sincerely hope they don't repeat history!

[3] Brackets are used, since superscripts may not show up on some browsers.

[4] If you do print, be forewarned that certain unnamed version 3.0 browsers often:

- Clip lines at the top and bottom of pages.
- Separate headings from their first paragraphs, captions from their tables, and terms from their definitions.
- Position italics incorrectly next to roman text, as in [*this*].

[5] Some classes do not need to be in their own files. Also, it is better form to import by class name rather than importing a whole package; see JLS for more information on both of these topics.