Porting C++ to Java

by Mark Davis

Last Update: March 25, 1997

Acrobat Version

Java Cookbook News Group

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.

Basics

The following sections take 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. First, figure out what your main package name is. To get it, take your domain name and reverse the fields. 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

The first thing you will notice is that Java does not distinguish between class interface (declaration) and implementation (definition) as C++ does. 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 second 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.

Fixing Basic Structure

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

Notes

- Java does not have the notion of friend as in C++. See Java has no friends for more information about how to handle this.
- Java 1.1 does have nested classes, though Java 1.1 does not. If you cannot wait for 1.1, you will have to move your nested classes out to the top. We suggest using concatenating the names: for a nested class Foo in a class Bar, use Bar_Foo.

To protect the innocent

Next, you need to change a few names. There are a few cases where there is a relatively straightforward name change.

Simple Name Replacements

Most name differences, however, depend on the context.

Context-Dependent Name Replacements

C++	Java
// const field static const int x = 3;	// const field static final int 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 (no primative) void* doAnother() {}	// unknown object Object doAnother() {}

Notes

- Const, in particular, requires very special handling, and is discussed in detail in Bullet-proofing. The simplest approach at the start is to change it to final for any field, and remove it otherwise.
- 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. It is also discussed at more length below.
- 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. Note that these methods are final, and will be faster to call than non-final (virtual) methods.
- Remove the word register everywhere. This is just a hint to the compiler anyway, and one that is often ignored by modern optimizations.

Java does not support operator overloading. You will miss this 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 good C++, 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 yellow items have special notes.

Operator Overload Replacement Names

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

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

Notes

- % is remainder in Java, not modulo. That is, -3%5 == -3, and not 2. In C++, it is undefined whether it is remainder or modulo. 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. Otherwise, you will need to change `x%y` to `(x%y - ((x < 0) ? y : 0))`
- Don't bother defining an equivalent to !=. The value should always be the same as if you called !(a == b), so just replace the call sites by (!a.equals(b)). You can also do the same for > and >=.
- 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

- Assignment (and copy constructors) are more complex than other operators. For more detail, see Difficult Assignments.
- For the index operators, define 2 methods. You will then have to fix the call sites according to the usage.

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
`// declaring`	`// declaring`
`bool operator==`	`public boolean equals(Foo other) {`
` (const Foo& other) const;`	` /*...*/`
	`}`
`// using`	`// using`
`if (a == b)`	`if (a.equals(b))`
`a[3] = 5;`	`a.setElementAt(3,5);`
`x = a[3];`	`x = 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 you have to be careful, as the following examples show.

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

No references necessary

Java also does not have references in the same way as C++, although they use the term references for normal objects (much like the conflation of objects and pointer to objects). Most C++ programs only use them in passing parameters to a method, or

C++	Java

// input parameter
int method1(const Foo& x);
 	
// Mutable output parameter
int method2(Foo x);
 	
// Immutable output parameter
int method2(int& x);
 	
// usage
Foo x;
z = y.method2(x);
w = x;
	// input parameter
	int method1(Foo x);
	
	// Mutable output parameter
	int method2(Foo x);
	
	// Immutable output parameter
	int method2(int[] x);
	
	// usage
	Foo[] x = new Foo[1];
	z = y.method2(x);
	w = x[1];

Notes

- Input parameters are simple; just remove the const (*however, there is a definite cost to doing this in terms of robustness of your code, see Bullet-proofing for better approaches*).
- Output parameters are more complex. Mutable objects (such as StringBuffer) can be passed in directly. Immutable objects (such as String, Integer) are more troublesome. You have three choices.
    1. Return the value from the method. This probably involves more work in porting, since presumably the reason you had an output parameter was that you were already using the return.
    2. The simplest way--though ugly--is to pass in an array as we did in the example. 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 & modify it. This involves more effort, but is somewhat cleaner than the array method.

Alternative Output Parameters

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

1. Return *this. This method allows chaining, as in x = y = z;
2. Return a reference to an input parameter. This allows use of use of functional returns without requiring memory allocation. Below is an example where method2 fills in x, then returns it for further use. Since the principal use of this 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 the compiler may warn you of problems if you are trying to set a Mutable. In that case, you will have to supply some of the same techniques as with output parameters.

Fixing References

C++	Java

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

Notes

Honest-to-God Arrays

Java arrays are real objects, not just disguised pointers. Generally you replace pointers used to iterate through an array by offsets, and the * operator by 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++]);
}	}

Notes

- Both the syntaxes `Foo x[]` and `Foo[] x` work, though the latter is more Java-like.
- Java arrays can supply you their length, rather than your having to remember it independently. Wherever possible, use this instead of a hard-coded length.
- 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.

Pitfalls

*Declaring an array does **not** create object to fill an 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

C++	Java

// initializing
Foo x[10];
	// initializing
	Foo[] x = new Foo[10];
	for (int i = 0; i < x.length; ++i) {
	 x[i] = new Foo();
	}

// initializing static const int x[] = {1,2,3,4,5,6,7,8,...	// initializing static const int x[] = {1,2,3,4,5,6,7,8,...

Next Steps

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

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.

C++	Java

// destructor
~MyObject() {
delete field1;
delete field3;
}
 	
// pointer field adoption
void setField1 (Foo* newValue) {
delete field1;
field1 = newValue;
}
 	
// pointer field aliasing
void setField2 (Foo* newValue) {
field2 = newValue;
}
 	
// pointer field assignment
void setField3 (const Foo& newValue) {
delete field3;
field3 = NULL;
aValue = new Foo(newValue);
}
	// no destructor
	
	// field assignment
	void setField1 (Foo newValue) {
	
	field1 = newValue;
	}
	
	// pointer field aliasing
	void setField2 (Foo newValue) {
	field2 = newValue;
	}
	
	// field assignment
	void setField2 (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
{
...
Foo x = new Foo();
...
	// no destructor
	{
	...
	Foo x = new Foo();
	...

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.

> *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 Object clone (Object other) {
	Foo result = (Foo) super.clone();
field1 = other.field1;	
field2 = ::Copy(other.field2);	field2 = other.field2.clone();
	return result;
}	}
Foo& operator= (const Foo& other) {	public Foo assign (Foo other) {
if (&other != this) {	if (other != this) {
SuperOfFoo::operator=(other);	super.assign(other);
field1 = other.field1;	field1 = other.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) {	public void getStuff(Foo foo, Bar bar) {

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;
 	
// using
a = Foo::x;
b = y;
c = myGlobalFunction();
 	
// declaring
class Fii : Foo {
 void someMethod() {
  Foo::someMethod();
 }
}
	// declaring
	class Foo {
	 static Foo x;
	 void someMethod();
	}
	
	class Globals {
	 static int myGlobalFunction() {...
	 static Foo y;
	}
	
	// using
	a = Foo.x;
	b = Globals.y;
	c = Globals.myGlobalFunction();
	
	// 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

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 boolean 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.

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;`

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.)

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.

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 {	class Button {...
enum ButtonState {inactive, active,	
mixed, inherited};	
void method1(ButtonState newState) {	void method1(ButtonState newState) {...
if (newState == inactive) {...	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.

C++	Java

// fetching command-line arguments
 	
int main(int argc, char *args[]) {
 for (i = 1; i < argc; ++i) {
  doSomething(args[i]);
 }
...
	// fetching command-line arguments
	public class MyApplication {
	 public static void main(String args[]) {
	  for (i = 0; i < args.length; ++i) {
	   doSomething(args[i]);
	  }
	...

// C-style simple output
printf("%s%i", "abc", 3);
	// simple output
	System.out.print("abc" + 3);

// C-style file output
 	
FILE* output = fopen("aFile","r");
if (output == NULL) {
 handleProblem();
}
fprintf(output, "%s%i", "abc", 3);
fclose(output);
	// file output
	try {
	 PrintStream output = new PrintStream(
	  new FileOutputStream("aFile"));
	
	 output.print("abc" + 3);
	 output.close();
	} catch (java.io.IOException e) {
	 handleProblem();
	}

// C++-style simple output
cout << "abc" << 3;
	// simple output
	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

using String concatenation, since the equivalent result += array[i].toString() constantly creates new objects.

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

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

Really Safe

C++	Java

// 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
	// 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 SafeFoo(Foo value) {
	  foo = value;
	 }
	 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.

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 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).

```
// 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!");
}

// Second example
// Goal: try to avoid relayout when size remains the same
Dimension mySize = size();
if (!mySize.equals(oldSize)) { // ALWAYS TRUE, since Dimension fails to override!
 oldSize = mySize;
 relayout();
}

// Work-around: do it yourself
Dimension mySize = size();
if (mySize.width != oldSize.width || mySize.height != oldSize.height) {
 oldSize = mySize;
 relayout();
}
```

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

Implementing `equals`

```
public boolean equals(Object obj) {
  // if top of heirarchy, use code:
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
  // if NOT top of heirarchy, use code:
    if (!super.equals(obj)) return false; // super checks class
  Sample other = (Sample)obj;
  if (myPrimitive != other.myPrimitive) return false;
```

2-6.
- Use lines (3,4) if this class *is* the top of your hierarchy.
- Use line (6) if this class *is not* the top of your hierarchy.

☠ Never call super.equals at the top of your hierarchy; Object.equals will give you the wrong result!

This way, each subclass depends on its superclasses to check their fields; the top class is the only one that needs to to check that the classes are the same.

Example:

```
class A {
 public boolean equals(Object obj) {
  if (this == obj) return true;
  if (getClass() != obj.getClass()) return false;
 ...
class B extends A {
 public boolean equals(Object obj) {
  if (!super.equals(obj)) return false; // super checks class
 ...
class C extends B {
 public boolean equals(Object obj) {
  if (!super.equals(obj)) return false; // super checks class
 ...
```

If you have a special hierarchy (such as Number) where you want equality checks to work across different classes, 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!

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

```
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!**

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

```
public int hashCode() {
  int result = 0;
  // result ^= super.hashCode();
  result = 37*result + myNumericalPrimitive;
  result = 37*result + (myBoolean ? 1 : 0);
  result = 37*result + myObject.hashCode();
  result = 37*result +
   (myPossNull != null ? myPossNull.hashCode() : 0);
  // if (!myTransient.equals(other.myTransient)) return false;
  return result;
}
```

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

```
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.

Independence and *Equality*.

Be careful though; if a class is not final, any subclass could fail to uphold its immutability (since there is no language support for it). So if you want to be safe, don't depend on immutablity for features such as this *unless* the class is final. For example:

```
class SupposedlyImmutable implements Cloneable {
 SupposedlyImmutable(int x) {
  value = x;
 }
 int getValue() {
  return value;
 }
 public Object clone() {
  return this; // shouldn't do since class isn't final
 }
 private int value;
}

class BreaksImmutable extends SupposedlyImmutable {
 BreaksImmutable(int x, int y) {
  super(x);
  value2 = y;
 }
 void setExtra(int y) {
  value2 = y;
 }
 int getExtra() {
  return value2;
 }
 public Object clone() {
  return super.clone(); // fails Independence
 }
 private int value2;
}
```

Don't try this at home

```
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 the caller may make to objects passed in or returned, and depend on your callers not to make a mistake!

Allegro ma non troppo

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;
```

- 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()
```

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. First, pick the base class that is most central to the function of the class in question. For each of the

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

Generally you can just dispense with the notation. If you *really*, *really* need bitfields to save storage, see Shave and a Haircut.

Shave and a Haircut

If you *really*, *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 (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++	Java

```
// declaring
struct Foo {
 // ...
 unsigned int z:3;
}

// using
a = myFoo.z;
z = myFoo.b;
```
	```
	// declaring
	class Foo {
	 // ...
	 public byte getZ() {
	  return (xyz & Z_MASK) >>> Z_SHIFT;
	 }
	 public void setZ() {
	  xyz = (b << Z_SHIFT) & Z_MASK
	        | xyz & ~Z_MASK;
	 }
	 private int xyz;
	 private final static Z_SHIFT = 17;
	 private final static Z_SHIFT = 0x7;
	}

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

Where they are being used for storage savings, you can sometimes get the same effect by using Object. The only disadvantage is that you are substituting runtime type-checking for compile-time checking.

Where unions are being used for scurrilous casting, you will have to work around it. For example, where such castings are used for hiddent bit-manipulations, you'll have to use the appropriate arithmetic operations, as below. You will have the advantage of having much more portable code in the end, though, 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;
	// declaring
	int i;
	
	// using
	x = 1066;
	z = x & 0xFF; // if C++ was BE
	z = x & 0xFF00; // if C++ was LE

Directly to Jail

Well-written C++ code should have very few gotos, if any. There are, however, times where 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 construct that replaces their use, and in a much cleaner and less dangerous way. You can name a loop with a label, then use break or continue with that label to escape from an inner block

If your gotos don't follow this pattern, then it is still fairly easy to convert if the gotos don't cross blocks. This is a bit kludgy, but saves your having to go in and figure out this snarled code.

Goto higher levels

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

If your gotos 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 privates by any other class in your package by making the access *package-private*. You do this by omitting the keyword private from your methods or data fields, and ensuring that the former friends are in the same package.

Replacing Friends

C++	Java

class Foo {
 private int foe;
 protected int fii;
 friend class Bar;
}
 	
class Bar {
 private Foo foo;
 public method() {
  ...
  y = foo;
  z = fii;
  ...
 }
}
	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 allow access to protected fields or methods, then you have to write cover methods that allow package-private access.
- 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
  - copy the class into both packages (this works for small classes)

Background Information

The following sections provide background

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/lists/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

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.

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.

Updates

Java Education   **Java Home**

IBM HOME   ORDER   EMPLOYMENT   CONTACT IBM   LEGAL