# Java versus C++

Barry Cornelius
IT Service, University of Durham, Durham, UK
last updated: 24th April 1997
`mailto:Barry.Cornelius@durham.ac.uk`
`http://www.dur.ac.uk/~dcl0bjc/Java`

## 1 Introduction

This document compares the languages C++ and Java. However, in producing this document I have cheated, because most of its text is extracted from other papers.

In particular, the document reproduces extracts from the following four papers:

- Ian Joyner's *C++?? : A Critique of C++ and Programming and Language Trends of the 1990s* (3rd edition);
- Robert C. Martin's *Java and C++: A Critical Comparison*;
- Markku Sakkinen's *The darker side of C++ revisited*;
- Transframe Technology Corporation's *Transframe, Java & C++: A Critical Comparison.*

Although this document includes a large number of extracts from these papers, you are recommended to read the complete version of each paper. The papers have fuller explanations and include some examples of code, and also make other points which are not included here. More details about where to find the four papers are given at the end of this document.

## 2 Types

### 2.1 The primitive types

---

**Cornelius:** C++ has the following fundamental types: `bool`, `char`, `short int`, `int`, `long int`, `float`, `double`, `long double`, `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `signed char`.

Java has the following primitive types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`.

---

**Cornelius:** In C++, the sizes of the fundamental types vary from one implementation of C++ to the next.

In Java, the sizes of the primitive types are specified by the language. For example, in Java, an `int` is always 32 bits having values from `-2147483648` to `2147483647`.

---

## 2.2 The type `char`

---

**Cornelius:** In C++, the size of the type `char` is usually 8 bits which allows ISO-8859-1 (Latin-1) encoding (which includes ASCII as a subset). There is also a type called `wchar` for a wider character set.

In Java, the size of `char` is 16 bits. The language specification states that the Unicode encoding of characters is to be used both for the type `char` and for the text of Java programs. The use of Unicode means that characters from many languages may be represented. Note that ISO-8859-1 and ASCII are subsets of the Unicode encoding.

Here is an example showing the use of a Greek letter in an identifier:

```
public static final double ß = 4.0*java.lang.Math.atan(1.0);
```

---

**Transframe:** We think that the advantage of the freedom in expression may bring about a problem in releasing, reusing, and exchanging software packages internationally. Identifiers are used for names of objects and classes. If a software package contains classes and methods with names in Japanese characters, this software package will be useless to people who do not speak Japanese.

---

**Cornelius:** In his book *The Design and Evolution of C++*, Stroustrup echoes these views. He says: "...English has an important role as a common language for programmers, and I suspect that it would be unwise to abandon that without serious consideration. To most programmers, a systematic use of Hebrew, Chinese, Korean, etc., would be a significant barrier to comprehension. Even my native Danish could cause some headaches for the average English-speaking programmer."

---

## 2.3 C++'s type `bool` and Java's type `boolean`

---

**Cornelius:** The type `bool` has only recently been added to the language C++. It is a unique signed integral type, where a zero value represents `false` and a non-zero value represents `true`. Existing pieces of C++ software have defined `bool` in different ways, and these differences can cause conflicts and confusion.

---

**Transframe:** ...an existing [C++] application defining a type named `bool` may get errors from the future C++ compiler when the type becomes built-in.

---

**Joyner:** Recently the ANSI/ISO C++ committee has accepted `bool` as a distinct integral type. Before the definition of a boolean type in C/C++ could be any number of definitions which had slightly different semantics. If you were combining libraries that used these slightly different definitions, life could be difficult. This is probably a fundamental reason why libraries have not been as successful in C++ as they should be in an OO environment. Not all compiler implementations have implemented `bool` yet, so you can expect it to be years before this mess is cleaned up.

---

**Cornelius:** In Java, the type `boolean` is a type with two values `false` and `true`. A `boolean` value cannot be cast to or from any other type (including the numeric types).

---

### 2.4 Enumeration types

**Cornelius:** In C++, you can define as many enumeration types as you wish. Each enumeration type defines names for integer values, and each of these types can be given a name:

```
enum Days (mon, tue, wed, thu, fri, sat, sun);
enum (soprano, alto, tenor, bass);
enum direction (left=-1, right=1);
```

A value of an enumeration type can be used in a context expecting an `int`.

**Sakkinen:** ...thus any enumeration value can be automatically (i.e., without an explicit cast) converted to an integer. Arithmetic operators may therefore be freely applied to enumeration values, which does not make much sense.

**Cornelius:** Unfortunately, Java does not have enumeration types.

### 2.5 Uninitialized variables

**Cornelius:** In Java, a local variable must be given a initial value before it is used: any attempt to use an uninitialized local variable is detected at compilation time. Any instance variable automatically has an initial value that depends on its type.

# 3 C++ pointers and Java references

### 3.1 C++ has pointers whereas Java has references

**Cornelius:** In C++, a pointer variable has a value that is the address of some location in memory. This can be used for many different purposes. For example, it can be used with passing arguments to functions, and it can be used for manipulating arrays. Both of these topics will be covered later.

However, in C++, a pointer variable can also be used in association with the `new` operator to allocate space for a value at runtime:

```
Person *composer;
composer = new Person("Mozart", "Wolfgang Amadeus");
```

The variable `composer` is a pointer variable. At runtime, the `new` operator is executed and this creates an object that is of the class `Person`. After the assignment statement has been executed, the pointer variable `composer` will be pointing to this object. Because the object is created at runtime, it is sometimes called a *dynamic variable*.

Java does not have pointers. However, it does have reference types. A variable that is of a reference type can be used to refer to (or to point to) an object (an instance of a class). This is done using the `new` operator:

```
Person composer;
composer = new Person("Mozart", "Wolfgang Amadeus");
```

The variable `composer` is a variable that is of the reference type `Person`. At runtime, the `new` operator is executed and this creates an object that is of the class `Person`. After the assignment statement has been executed, the reference variable `composer` will be pointing to this object.

## 3.2 Pointer arithmetic

---

**Cornelius:** C++ allows arithmetic operations such as `+` and `-` to be performed on pointer values. Stroustrup's book *The C++ Programming Language* gives several examples of pointer arithmetic. In one example, a pointer variable is increased by one so that it now points to the next value in memory. In another example, two pointer variables whose values give the addresses of the start and end of a string are subtracted in order to find the length of the string.

Java does not have pointer arithmetic.

---

**Transframe:** Most C++ features omitted in Java are deficiencies in safety. Examples are explicit deallocation and pointer arithmetic.

---

**Joyner:** Pointers can be incremented past the end of the entities they reference, with subsequent updates possibly corrupting other entities, which is a major source of the undetected inconsistencies, which result in obscure failures, . . . .

---

## 3.3 Memory management

---

**Cornelius:** In C++, the `delete` operator is used to indicate that we no longer require the space for a dynamic variable.

In Java, you don't delete objects: instead, Java has *garbage collection.* The garbage collector detects objects no longer in use, and arranges for their space to be reused.

---

**Transframe:** C++ supports one referential type: pointer, which requires explicit deallocations and may result memory leaks and dangling pointers.

---

## 3.4 Bad uses of C++'s `delete` operator

---

**Joyner:** The following example is given on p.63 in the C++ ARM [*Annotated C++ Reference Manual*] as a warning about bad deletions that cannot be caught at compile-time, and probably not immediately at run-time:

```
p = new int[10];
p++;
delete p; // error
p = 0;
delete p; // ok
```

One of the restrictions of the design of C++ is that it must remain compatible with C. This results in examples like the above, that are ill-defined language constructs, that can only be covered by warnings of potential disaster. Removal of such language deficiencies would result in loss of compatibility with C. This might be a good thing if problems such as the above disappear. But then the resultant language might be so far removed from C that C might be best abandoned altogether.

Bad deletions are the kind of problem the Java designers set out to avoid. You do not get bad deletions in either Java or Eiffel for two reasons: firstly, they do not have pointers; secondly, they provide garbage collection so don't delete objects.

---

## 3.5 Garbage collection

---

**Joyner:** One of the hallmarks of high level languages is that programmers declare data without regard to how the data is allocated in memory. In block structured languages, local variables are automatically allocated on the stack, and automatically deallocated when the block exits. This relieves the programmer of the burden of allocating and deallocating memory. Garbage collection provides equivalent relief in languages with dynamic entity allocation.

In C++ the programmer must manually manage storage due to the lack of garbage collection. This is the most difficult bookkeeping task C++ programmers face that leads to two opposite problems: firstly, an object can be deallocated prematurely, while valid references still exist (dangling pointers); secondly, dead objects might not be deallocated leading to memory filling up with dead objects (memory leaks). ... Garbage-collection solves both problems, but has an undeserved bad reputation due to some early garbage-collectors having performance problems, instead of working transparently in the background, as they can and should.

Sun have built garbage collection into Java.

---

**Joyner:** [Concerning the addition of garbage collection to C++,] my question is not *if* or *when*, but *how*? Unless you restrict pointers and pointer operations, garbage collection will be very difficult [in C++], and probably inefficient. By inefficient, I mean either slow, or it won't clean up very well, or even both.

---

**Martin:** C++ is often criticized for its lack of GC. However, many people have added garbage collectors to C++. Some of these are available as third party products, or as shareware on the net. These collectors are far from perfect, but they can be used when convenient.

The corresponding statement cannot be made for Java. There is *no way* that this humble writer could discover to manage memory manually. Apparently, you cannot write your own memory manager and construct objects within the memory that it controls. There is a sound reason for this. Any memory management scheme that allows a program to hold pointers or references to unused space allows certain security violations. ... If manual memory management were allowed, it might be possible for unscrupulous people to put up web pages that contained unsecure applets. ... Once downloaded those applets could then transmit private information back to the author of the web page.

Is the lack of manual memory management in Java a problem? In most cases no. However, [it] makes Java a difficult language to use in applications that have real-time constraints. The problem is that it is very difficult to predict when the garbage collector will run. When it does run, it can use ... significant amounts of CPU time.

---

**Transframe:** [Java's] enforced GC disables the language users to use other memory management protocols.

---

# 4 Arrays

## 4.1 Array type

---

**Cornelius:** The array facility of C++ is the same as that of C. For example, the declaration:

```
float v[3];
```

introduces the three variables `v[0]`, `v[1]` and `v[2]`.
In Java, the declaration:

```
float[] v;
```

introduces the variable `v` as a reference variable that can refer to or point to an array. You can do this by using an *array creation expression*:

```
v = new float[3];
```

The above two lines can be combined as in:

```
float[] v = new float[3];
```

So, `v` is a reference variable that points to an array of 3 floats, with indexes 0, 1, 2. You can access the elements using the usual notation, i.e., `v[0]`, `v[1]` and `v[2]`.

---

**Transframe:** Java's array makes mass data processing inefficient.

---

**Transframe:** C++ array is a hybrid object. Array type is not a class, and arrays are not objects. Java tries to integrate the *array* type into the class hierarchy, but the try is not very successful. In Java, *arrays* are instances of subclasses of class `Object`. If class `A` is a subclass of `B` then `A[]` is a subclass of `B[]`. This introduces the famous covariance problem and hence a hole in the type system. Consider:

```
Animal[] x;
Lamb[] lamb_flock = new Lamb[100];
x = lamb_flock;       // Lamb[] is a subclass Animal[]
x[i] = new Wolf;      // oops, we put a wolf into the lamb flock!
```

The error cannot be captured at compile time. Java reports an error at runtime. As result, the system crashes at the point where an array element type exception is raised.

---

## 4.2 Array bound checking

---

**Joyner:** C arrays provide no run-time bounds checking, not even in test versions of software. . . . An index out of bounds . . . should be treated as severely as divide by zero. But in C this is another significant source of undetected inconsistency, which can result in obscure failures. . . . Java provide bounds checking on arrays.

---

**Transframe:** It is impossible to implement array bounds checking with a full compatibility to the C array.

---

**Sakkinen:** The rules of legal pointer arithmetic, and hence the rules of array subscripting, are such that enforcing them would be prohibitively expensive at run time.

---

**Transframe:** Java still has some unsafe features. Java's array reports type errors at run-time. . . .

---

## 4.3 The relationship between arrays and pointers

**Cornelius:** In both C and C++, pointers and arrays are very closely related. An array is effectively a pointer to the first element of the array.

**Joyner:** There are many ways you can undermine arrays in C and C++, as an array declaration is really just equivalent to a pointer. The following example comes from [Garfinkel et al 94]:

```
char *str = "bugy";
```

then the following are true:

```
0[str]   == 'b';
*(str+1) == 'u';
*(2+str) == 'g';
str[3]   == 'y';
```

This is amazingly flexible syntax for something as inflexible as C arrays, . . . .

# 5 C++ functions and Java methods

## 5.1 Argument passing in C++

---

**Cornelius:** In both C and C++, a parameter of a function behaves like a local variable of the function which automatically gets its initial value from the argument passed in the call. This mechanism for passing values to functions is sometimes called *call-by-value*. And if you want a function to alter one of your variables, you can pass an argument that is the address of the variable and use a parameter that is a pointer.

Here is an example of a call:

```
assign(27, &x);
```

where the silly function `assign` is declared as:

```
void assign(int val, int *var) {
   *var = val;
}
```

---

**Joyner:** The programmer must also be concerned with correct dereferencing of pointers to access referenced entities. Use of pointers to emulate by reference function parameters are an example. The programmer has to worry about the correct use of `&`s and `*`s. . . . Pointer arithmetic is error prone.

---

## 5.2 C++'s reference types

---

**Cornelius:** Earlier, we saw that Java has *reference*s. This term is also used in C++ (but not in C), but for a different purpose. In C++, a *reference* variable is a variable that is an alias for another variable. When used in a parameter list, a reference parameter makes it easier to pass a value back through an argument.

For example, the call of `assign` given earlier can be simplified to:

```
assign(27, x);
```

when the function `assign` is declared as:

```
void assign(int val, int &var) {
   var = val;
}
```

---

**Sakkinen:** The prime reason for the introduction [of references types] must have been the requirement of overloaded operators. It may also be difficult for programmers to remember the cases when they should pass the *address* of a variable as an actual argument to a function, instead of its *value*; references can help here. . . . Reference types are clearly second-rate datatypes. For instance, structures with reference components are allowed, but arrays of references are not. . . . It would have been better to introduce references just as an argument-passing (and result-passing) mode as in most other languages.

---

## 5.3 Argument passing in Java

---

**Joyner:** In Java arguments can only be passed by-value (as in C). However, there are no pointers, so passing by-reference cannot be simulated.

---

**Transframe:** When more than one object references must be returned from a routine, there is no direct way for a Java method to provide such interface. Let's consider an example: given a URL resource address, we need a method to decompose the resource address into three strings: the path, the resource file name, and the resource type name. That is, for an address:

```
"http://www.foo.com/images/blue_sky.jpg"
```

we would like to use this method to break the address into the following three strings:

```
"http://www.foo.com/images/"
"blue_sky"
"jpg"
```

In C++, we can write a function with the following interface:

```
void decompose ( const char *url_address,
                              char *&url_path,
                              char *&resource_name;
                              char *&resource_type:
                            );
```

The C++ interface declares the three outputs in terms of references of pointers. It is explicit but difficult to understand. What does `*&` mean? Should I write `*&` or `&*`? Java does not have a much simpler way to describe this interface, either array or extra class should be introduced as the following example:

```
void decompose ( String url_address,
                             String[2] decompose_result
                           );
```

The Java method interface does not have any improvement in readability. People have to guess the meaning of expressions like `decompose_result[1]`. The worse, if elements of the array must be in different types, the type of each elements must also be guessed.

---

## 5.4 Optional parameters

---

**Joyner:** Optional parameters that assume a default value according to the routines declaration are supposed to provide a shorthand notation. ... In large scale software production, however, precision is mandatory, and defaults can lead to ambiguities and mistakes. With optional parameters the programmer could assume the wrong default for a parameter. ... Optional parameters mean that C++ is not type safe, and that the compiler cannot check that the parameters in the call exactly match the function signature.

Neither Java nor Eiffel have optional parameters.

---

### 5.5 Result type

---

**Joyner:** The default return type of a function is `int`. A typeless routine returning nothing should be the default, but this must be specified by void. Syntactically no <type> suggests nothing to return. . . . Stroustrup 94 also voices the opinion that default `int` is bad. He had tried to make the type specifier explicit, but was forced to withdraw by users: "I backed out the change. I don't think I had a choice. Allowing that implicit `int` is the source of many of the annoying problems with C++ grammar today. Note the pressure came from users, not management or arm-chair language experts. Finally, ten years later, the C++ ANSI/ISO standard committee has decided to deprecate implicit `int`."

One improvement in Java is that the result type of the method is not optional. That is you don't get `int` by default.

---

### 5.6 Are procedures and functions distinguished?

---

**Joyner:** In order to specify a procedure rather than function, Java still requires the void specifier. Java does discard the C term function (which was wrongly used anyway), but makes the situation no better by calling both procedures and functions methods. Thus there is no clear distinction between procedure and function. Java also allows you to ignore returned values.

---

### 5.7 Type-safe linkage

---

**Joyner:** C++ type safe linkage is a huge improvement over C, where the linker will link a function `f(p1, ...)` with parameters to any function `f()`, maybe one with no or different parameters. This results in failure at run time. However, since C++ type safe linkage is a linker trick, it does not deal with all inconsistencies like this. Java uses a different dynamic linking mechanism, which is well defined and does not use the Unix linker.

---

## 6 Expressions

### 6.1 The null pointer

---

**Joyner:** Stroustrup 94 adds that "nothing seems to create more heat than a discussion of the proper way to express a pointer that doesn't point to an object, the null pointer." And, "The ARM further warns 'Note that the null pointer need not be represented by the same bit pattern as the integer 0.' " Continuing on: "The warning reflects the common misapprehension that if `p=0` assigns the null pointer to the pointer `p`, then the representation of the null pointer must be the same as the integer zero, that is, a bit pattern of all zeros. This is not so. C++ is sufficiently strongly typed that concept such as the null pointer can be represented in whichever way the implementation chooses, independently of how that concept is represented in the source text." No wonder people are confused, and there is much heated debate.

In Java `null` is a reserved word.

---

**Cornelius:** Even though it looks as if, in Java, `null` is a keyword, it is technically the *null literal.*

---

### 6.2 The use of a null terminator for strings

---

**Joyner:** In C strings, metadata about where strings terminate is stored in the string data as a terminating null byte. This means that the distinction between data and metadata is lost. The value chosen as the terminator cannot occur in the data itself. Since inserting a null is often the responsibility of the programmer, not the run-time environment, there is the potential for more undetected inconsistencies resulting in obscure failures. . . . C's null terminator makes the implementation [of strings] visible to the programmer.

Java's strings are first class objects. You can't determine the length of a string by scanning for a null. You use the `string.length` method (function).

---

### 6.3 Operator overloading

---

**Cornelius:** In C++, it is possible to give new meanings to most of the operators: this is called *operator overloading*. It is not possible to do this in Java.

---

**Martin:** While writing code in Java I have to say that I miss being able to overload operators as in C++. This is not a critical issue, but I am disappointed.

---

### 6.4 Classes that parallel the primitive types

---

**Transframe:** In order to use numbers and other primitive values as objects, Java introduced a parallel class hierarchy in the `java.lang` package. They are `Boolean`, `Character`, `Number`, `Integer`, `Long`, `Float`, and `Double`. This unnecessarily complicated the usage of primitive objects. Since they are not primitive values, obligatory conversions must be used frequently. For example:

```
Double x, y, z;
x = new Double(3.0);
y = new Double(4.0);
z = new Double( x.doubleValue() + y.doubleValue());
```

The worse, Java does not support operator for class methods. Therefore, an instance of `Integer` cannot use operators such as `+`, `-`, `*`, and `/` as we usually use them for primitive types. As result, mathematical expressions using Java number classes will be extremely complicated.

---

### 6.5 The method `toString`

---

**Cornelius:** All classes of Java provide a method called `toString`. When `toString` is applied to an object, the method returns a string that "textually represents" the value of the object.

---

**Transframe:** The language has included many system-dependent features and a number of ad hoc features (e.g., string conversion method) in the built-in classes, which makes it hard to keep built-in classes unchanged in future.

---

**Martin:** This automatic use of `toString()` seems to be an immature version of the automatic conversion system of C++. This feature (among others) makes the `String` class something more special than any other class. I think it would be wise for the Java designers to work on a generic conversion system (e.g., a `to`<*xxx*> method template).

---

# 7 Statements

## 7.1 A boolean expression delivers a `boolean`

---

**Joyner:** Java also uses the `=` symbol to mean assignment, so this has not improved. However, the `=` versus `==` confusion has been improved as in the syntax:

```
if ( Expression ) Statement
```

the Expression must have type boolean, or a compile-time error occurs.

---

## 7.2 `goto`, labelled `breaks` and `continues`

---

**Cornelius:** Although C++ has a `goto` statement, Java does not. However, `goto` is a keyword of Java: some say it's there so as to enable better compilation error messages to be generated.

---

**Sakkinen:** A defect [in C++] in both `break` and `continue` is that they allow the programmer to exit or continue only the smallest enclosing loop or (`break` only) `switch` construct.

---

**Joyner:** Java provides an extension to control structures which allows control structures to be named, and multi-level `break` and `continue` statements can be used to jump to an outer level conditional or loop.

---

**Martin:** The [Java] language designers pride themselves on creating a language that does not have a `goto` statement. Yet, wonder of wonders, they added *labeled* `break` and `continue`. This is a sore point of mine. ... The use of `break` and `continue` in C, C++, or Java constitute a minor violation of [the single-entry/single-exit paradigm]. We use them to transfer control out of the middle of a loop. The use of *labeled* `break` and *labeled* `continue` are a much more serious violation. These can be used to exit deeply nested blocks from their middles. Indeed, some of the enclosing blocks may not know that they are being exited and may be written to assume that they are not. This can lead to errors that are very hard to identify.

---

### 7.3 Exception handling

---

**Cornelius:** A method often detects situations which it knows it cannot handle. Some programming languages (e.g., Ada, Clu, C++, Java) allow the code of the method to signify that an *exception* has occurred, and this is then handled by some other code elsewhere in the program. In both C++ and Java, a `throw` statement is used to signify that an exception has occurred, and a `try` statement with one or more `catch` clauses is used to indicate that a piece of code wants to handle exceptions.

There are two main ways in which Java's exceptions differ from those of C++:

- In Java, a method that throws an exception (or calls a method that throws an exception) must either have a `try` statement that handles the exception or the method must have a `throws` clause in its heading: this signifies that it does not handle the exception.
- In Java, a `try` statement can have a `finally` clause. This will be executed either after the statements of a `catch` clause have been executed (if an exception occurred) or after the `try` statement has been executed (if no exception occurred).

---

**Martin:** I am very pleased with the exception mechanism in Java. Although modeled after the C++ mechanism, it avoids some of C++s more severe problems by using the `finally` clause.

In C++, when an exception leaves the scope of a function, all objects that are allocated on the stack are reclaimed, and their destructors are called. Thus, if you want to free a resource or otherwise clean something up when an exception passes by, you must put that code in the destructor of an object that was allocated on the stack. ... This is artificial, error prone, and inconvenient. ... In many ways, [the Java] scheme seems superior to the C++ mechanism. Cleanup code can be directly specified in the `finally` clause rather than artificially put into some destructor. Also the cleanup code can be kept in the same scope as the variables being cleaned up. In my opinion, this often makes Java exceptions easier to use than C++ exceptions. The main downside to the Java approach is that it forces the application programmer to (1) know about the release protocol for every resource allocated in a block and to (2) explicitly handle all cleanup operations in the `finally` block. Nevertheless, I think the C++ community ought to take a good hard look at the Java solution.

---

## 8 Data abstraction

### 8.1 `struct` versus `class`

---

**Transframe:** The `struct` type constructor in C is a redundant feature when the `class` concept is introduced, and the worse, C++ sets up different accessibility rule for structures and classes.

---

**Edelson:** Having both `struct` and `class` is redundant. A `struct` is the same as a `class` that has by default `public` components. The `struct` keyword could not be eliminated without losing compatibility, but the `class` keyword is unnecessary.

---

**Joyner:** Struct is only in C++ as a compatibility mechanism to C. When you have classes you don't need structs. Again, C++ is unnecessarily complicated with unneeded features. Sun 95 says: "The Java language has no structures or unions as complex data types. You don't need structures and unions when you have classes - you can achieve the same effect simply by using instance variables of a class."

---

## 8.2 `structs` and `typedefs`

---

**Joyner:** Typedef is yet another mechanism not needed. Java, Eiffel and Smalltalk all build their type mechanisms around classes.

---

## 8.3 Java has no `union` type

---

**Joyner:** Union is another construct that is superfluous in OOP. Similar constructs in other languages are recognised as problematic: for example, FORTRAN's `EQUIVALENCE`s, COBOL's `REDEFINES`, and Pascal's variant records. When used to overload memory space these force the programmer to think about memory allocation. . . . Union is also not needed to provide the equivalent to COBOL `REDEFINES` or Pascal's variants. Inheritance and polymorphism provide this in OOP.

Sun recognises that the union construct is unnecessary, and has removed it from Java.

---

**Transframe:** Union type is practically useful but is not available in Java.

---

## 8.4 Accessing a component of an object

---

**Cornelius:** In C++, the operators `.` and `->` are both used to access a component (or member) of an object.

---

**Joyner:** [This] member access syntax came from C structures, and illustrates where the C base adversely affects flexibility. Semantically both access a member of an object. They are, however, operationally defined in terms of how they work. The dot (`.`) syntax accesses a member in an object directly: `x.y` means access the member `y` in the object `x`. . . . The `->` syntax means access a member in an object referenced by a pointer: `x->y` (or the equivalent `*(x).y`) means access the member `y` in the object pointed to by `x`. . . . In these examples, *what* is to be computed is *access the element y of object x.*. . . . In C++, however, the programmer must specify for every access the detail of *how* this is done. That is the access mechanism to the member is made visible to the programmer, which is an implementation detail. Thus the distinction between `.` and `->` compromises implementation hiding, and very seriously the benefit of encapsulation. . . . The compiler could easily restore implementation hiding by providing uniform access and remove this burden from the programmer, as in fact most languages do.

For example, if the `OBJ x` declaration is changed to `OBJ x`, the effect is widespread as all occurrences of `x.y` must be changed to `x->y`. Since the compiler gives a syntax error if the wrong access mechanism is used, this shows that the compiler already knows what access code is required and can generate it automatically. Good programming centralises decisions: the decision to access the object directly or via a pointer should be centralised in the declaration. So again, C++ uses low level operators, rather than the high level declarative approach of letting the compiler hide the implementation and take care of the detail for us.

Java only supports the dot form of access. The `->` form is superfluous. Java objects are only accessed by reference; there are no embedded objects.

---

## 8.5 The `inline` directive

---

**Joyner:** Requiring a programmer to specify `inline` is a manual bookkeeping task. It is not hard for a compiler or optimiser to work out that `C::get_di(){return di;}` or even more complex routines could be inlined. This is exactly the kind of optimisation that Eiffel and other sophisticated languages perform.

[Flanagan 96] says: "A good Java compiler should automatically be able to inline short Java methods where appropriate." An article in *Byte* of September 1996 suggests that to optimise Java method calls "you should make liberal use of the `final` keyword". In this respect, Eiffel again proves itself superior. Eiffel automatically determines that a routine is `final`, or in C++'s terminology, that a routine is not `virtual`. Also Eiffel automatically inlines.

---

## 8.6 Friends

---

**Cornelius:** In C++, it is possible for any function to access a private member (or a protected member) of a class if the class declares the function to be a *friend*. Also, in C++, if class A is declared to be a friend of class B, then its member functions are allowed to access the private members and protected members of class B.

---

**Joyner:** ... Friends are useful, and a case can be made for shades of grey between public, protected and private members.

... One reason given for friends is they allow more efficient access to data members than a member function call. The way C++ is often used is that data members are not put in the public section, because this breaks the data hiding principle. ... since there are inlines, is there a need for the similar mechanism of friends? If you mark a function inline, it is going to expand inline, and avoid the function call overhead. So in this case, friend is a superfluous mechanism.

In Java, classes in the same package can access instance variables from other classes in a *friendly* fashion. This is contrary to good programming practice and OO design, as it means you can access things without going through the published interface of a class. However, in Java, explicit friends are gone.

---

## 8.7 Using modifiers to restrict the access to fields

---

**Joyner:** As noted in the section on friends, there is a case for finer grained control of exports than `public`, `private` and `protected`. Except for friends, Java uses the same mechanism as C++, but adds two more categories, default and `private protected`. This complicates the mechanism, and it is difficult to remember exactly what each category does.

... A further complication in C++ is that `public`, `private`, `protected` can be specified when inheriting a base class. ... Java has no equivalent.

---

## 8.8 Template classes

---

**Joyner:** Java, alas has no genericity mechanism. The Java recommendation is to use type casts when ever retrieving an object from a container class.

---

**Cornelius:** The class `java.util.Stack` provides the methods:

```
public Object push(Object item);
public Object pop() throws EmptyStackException;
```

So if you are using an object of this class in order to store a stack of books, then to retrieve a book from the stack you can use:

```
Stack pile = new Stack();
Book nextBook;
...
nextBook = (Book) pile.pop();
```

The above assignment statement includes a type-cast.

If the stack contains both books and magazines, then you may want something like:

```
Stack pile = new Stack();
Object nextObject;  Book nextBook;  Magazine nextMagazine;
...
nextObject = pile.pop();
if ( nextObject instanceOf Book )
   nextBook = (Book) nextObject;
else
   nextMagazine = (Magazine) nextObject;
```

---

**Transframe:** Without a support of parameterizations, many interface cannot describe the type dependencies among inputs; hence the code relies more on run-time type casting and possibly produces more run-time type exceptions.

---

**Transframe:** Many people view parameterization as an alternative means for reuse and polymorphism. But there is another important usage that cannot be replaced by other features such as inheritance and polymorphic variables. It is the specification of type dependency.

Consider a polymorphic C++ function interface:

```
Animal* propagate (Animal *x, Animal *y);
```

where `Animal` is a class. The function can take two references that refers to objects of two different types (as long as they are `Animal`'s subclasses.

But if the function requires that the two inputs are in the same type and the output type must be the same type of the input, the above function interface is at least not precise, if not wrong, which would make wrong type combination possible, for example `propagate(tiger,cow)`, and hence raise a run-time type error.

Parameterizing the function interface with a class parameter will give a more precise interface:

```
template < class AnimalType >
AnimalType* propagate (AnimalType *x, AnimalType *y);
```

With this interface, the function call `propagate(tiger,cow)` will be invalid.

…Parameterization is a missing part in Java, which is the major imperfection in language design.

---

**Martin:** Templates are a wonderful feature of C++. The fact that Java does not have them is of some concern to me. In Java, one cannot create a type-safe container. All containers in Java can hold any kind of object. This can lead to some ugly problems. Mitigating these problems is the fact that all casts in Java *are* type safe. That is, Java casts are roughly equivalent to `dynamic_cast` of references in C++. All incorrect cast results in an exception being thrown. Since all objects coming out of a Java container must be downcast, and since such casts are relatively safe, the need for type safe containers is somewhat lessened.

## 8.9 Template functions

**Cornelius:** As well as template classes, C++ also has *template functions*. Java doesn't have either. Here is an example of a C++ template function:

```
template <class ArgType> ArgType min(ArgType a, ArgType b) {
   return a<b?a:b;
}
```

**Martin:** Templates in C++ are a very nice way of achieving static polymorphism. . . . Here we see a C++ template function that employs static polymorphism. . . . Although it is more typical in both C++ and Java to gain this kind of polymorphism using abstract base classes, there are some distinct advantages to using templates. . . .

## 8.10 Assignment and copying

**Sakkinen:** If we compare C++ assignments to languages with reference semantics, we should note that those languages typically offer no operations similar to *object* assignment. A *copy* operation yields a *new* object, and is therefore the equivalent of a copy constructor in C++. Further, the automatically available operations are *shallow copy* and *deep copy.* Unless the class is very simple, neither of these will probably be sensible: the former is too shallow, the latter too deep.

The default functions in C++ have a much better chance of being meaningful, so that the class designer need not always write his/her own. However, there is a conceptual problem that tends to be overlooked. When we come to large application objects, copying them in *any* way may no more make sense. . . . It would be nice if the class designer could specify that copying is not applicable to a certain class. The closest thing that can be done in C++ is to declare the copy constructor and assignment operator as `private`.

# 9 Inheritance and dynamic binding

## 9.1 Virtual functions

---

**Joyner:** The problem in C++ is that if a parent class designer does not foresee that a descendant class might want to redefine a function, then the descendant class cannot make the function polymorphic. This is a most serious flaw in C++ because it reduces the flexibility of software components and therefore the ability to write reusable and extensible libraries.

---

## 9.2 Overriding

---

**Joyner:** Virtual, however, is the wrong mechanism for the programmer to deal with. A compiler can detect polymorphism, and generate the underlying virtual code, where and only where necessary. Having to specify `virtual` burdens the programmer with another bookkeeping task. ...Polymorphism is the *what*, and `virtual` is the *how*. ...This is the main reason why C++ is a weak object-oriented language as the programmer must constantly be concerned with low-level details, which should be automatically handled by the compiler. In Java, there is no `virtual` keyword: all methods are potentially polymorphic. Java uses direct call instead of dynamic method lookup when the method is `static`, `private` or `final`. This means that there will be non-polymorphic routines that must be called dynamically, but the dynamic nature of Java means further optimisation is not possible.

---

**Joyner:** C++ does not cater for the prohibition of overriding a routine in a descendant class. Even `private virtual` routines can be overridden. ...Neither Eiffel nor Java have these problems. Their mechanisms are clearer and simpler, and don't lead to the surprises of C++. In Java, everything is `virtual`, and to gain the effect where a method must not be overridden, the method may be defined with the qualifier `final`.

---

**Transframe:** C++ has the essential difference to Java and Transframe that member functions are by default not to be overridden. By C++, only member functions declared as virtual can be overridden in a derived class. In Java and Transframe, member functions are by default to be overridable. To prevent a member function being overridden in a derived class, the member function must be declared as final.

---

## 9.3 Pure virtual functions

---

**Transframe:** A pure virtual function in C++ is an member function that has no implementation. It is equivalent to Java's abstract method ....

---

**Cornelius:** Here is an example of a class in C++ that only has pure virtual functions:

```
class APop {
   public:
      virtual ~APop() {};
      virtual void addPerson(const Person &rPerson) = 0;
      virtual int removePerson(const char *vName) = 0;
      virtual Person & obtainPerson(const char *vName) const = 0;
} ;
```

In Java, this might be coded as:

```
interface APop {
   void addPerson(Person vPerson);
   int removePerson(String vName);
   Person obtainPerson(String vName);
}
```

---

**Joyner:** Pure virtual functions provide a means of leaving a function undefined and abstract. ... A class that has such an abstract function cannot be directly instantiated. A non-abstract descendant class must define the function. The C++ pure virtual syntax is:

```
virtual void fn () = 0;
```

This leaves the reader new to C++ to guess its meaning, even those well versed in object-oriented concepts. [The syntax] =0 might make sense for the compiler writer, as the implementation is to put a zero entry in the virtual table. This shows how implementation details which should not concern the programmer are visible in C++. ... The mathematical notation used in C++ suggests that values other than zero could be used. What if the function is equated (or is that assigned) to 13?

```
virtual void fn () = 13;
```

Stroustrup 94 gives the curious tale about the *=0* syntax: "The curious `=0` syntax was chosen over the obvious alternative of introducing a keyword `pure` or `abstract` because at the time I saw no chance of getting a new keyword accepted. Had I suggested `pure`, Release 2.0 would have shipped without abstract classes. Rather than risking delay and incurring the certain fights over `pure`, I used the traditional C and C++ convention of using 0 to represent *not there*."

Java and Eiffel use much clearer syntax. Java simply uses:

```
abstract void fn ();
```

---

## 9.4 The method `finalize`

---

**Martin:** The `finalize` function in Java roughly corresponds to the destructor in C++. When an object is collected by the garbage collector, its `finalize` method is called. This allows objects to clean up after themselves. However, it should be noted that in most cases `finalize` is not a good place to release resources held by the object. It may be a very long time before such objects get collected by the garbage collector. Thus, any resources they release in `finalize` may be held for a very long time.

However, . . . the `finalize` of the derived class must *explictly* call the `finalize` of the base class. If you forget to do this, then base class `finalize` functions simply don't get called. Arnold and Gosling 96 recommend: "Train your fingers so that you always do so in any `finalize` method you write." Unfortunately, this is very error prone. In my opinion such "training" belongs to the compiler. The calling of base class `finalize` methods should have been taken care of by the compiler in the manner of destructors in C++.

---

## 9.5 Multiple inheritance

---

**Joyner:** Both Eiffel and C++ provide multiple inheritance. Java does not, claiming that it results in many problems. Instead Java provides *interfaces*, which are similar to Objective C's protocols. Sun claims interfaces provide all the desirable features of multiple inheritance.

Sun's claim that multiple inheritance results in problems is true particularly in the way that C++ has implemented multiple inheritance. What seems like a simple generalisation of inheriting from multiple classes instead of just one, turns out to be non-trivial. For example, what should be the policy if you inherit an item of the same name from two classes? Are they compatible? If so should they be merged into a single entity? If not, how do you disambiguate them? And so the list goes on.

Java's interface mechanism implements multiple inheritance, with one important difference: the inherited interfaces must be abstract. This does obviate the need to choose between different implementations, as with interfaces there are no implementations. Java allows the declaration of constant fields in an interface. Where these are multiply inherited, they merge to form one entity so that no ambiguity arises, but what happens if the constants have different values?

Since Java does not have multiple inheritance, you cannot do *mixins* as you can in C++ and Eiffel. Mixin is the ability to inherit sets of non-abstract routines from different classes to build a new complex class. For example, you might want to import utility routines from a number of different sources. However, you can achieve the same effect using composition instead of inheritance, so this is probably not a great minus against Java.

---

**Transframe:** With the belief that multiple inheritance generates more problems that benefits, the designers of Java discarded multiple inheritance. The desirable feature of multiple inheritance is provided by multiple interface implementation. A Java interface is a class that has nothing but abstract methods (pure virtual functions) and constants. A subclass can be extended from a single base (super) class as well as implement more than one interfaces.

---

**Martin:** . . . disallowing multiple inheritance of classes, and only allowing the multiple implementation of interfaces, . . . was probably a good trade-off. In all likelihood it has simplified the language appreciably. However, it has left a problem. It prevents us from inheriting implementation from more than one class in cases where the *deadly diamond of death* does not appear. This is unfortunate since it is often the case that we want to inherit from more than one base class which has functions and data. . . .

---

## 9.6 Various forms of inheritance

**Cornelius:** C++ has `public` inheritance, `protected` inheritance and `private` inheritance, whereas Java has one form that is equivalent to C++'s `public` inheritance.

**Sakkinen:** It is worth acknowledging even here that private inheritance is a semantically and conceptually valuable capability, although it also makes the language more complex.

## 10 Some other points

### 10.1 Threads

**Joyner:** It might not be impossible to implement concurrent processing in C++, but it is difficult as in many ways C++ is not suited to concurrent processing. Java provides threads. It also removes C features like globals that are problematic to concurrency.

### 10.2 Libraries

**Cornelius:** During the course of standardising C++, a class library called the STL (*Standard Template Library*) was added to C++. There are many other C++ class libraries.

Java is accompanied by a Core API together with some APIs described as Standard Extensions. Other people are also supplying APIs: these include Netscape's IFC (*Internet Foundation Class*) and Microsoft's AFC (*Application Foundation Class*). Sun, Netscape and IBM also plan to release JFC (*Java Foundation Classes*) with the next release of the JDK.

### 10.3 C as a subset

**Cornelius:** Unlike Java, the language C++ is (on the whole) a superset of C.

**Joyner:** While C programmers can immediately use C++ to write and compile C programs, this does not take advantage of OO. Many see this as a strength, but it is often stated that the C base is C++'s greatest weakness. However, C++ adds its own layers of complexity, like its handling of multiple inheritance, overloading, and others. I am not so sure that C is C++'s greatest weakness. Java has shown that in removing C constructs that do not fit with object-oriented concepts, that C can provide an acceptable, albeit not perfect base.

**Sakkinen:** When someone sets out to enrich an existing language with object-oriented or other *higher-level* features, trying to keep totally upward compatible with the base language can be problematic. ... Although C++ is in many ways a seamless whole, almost all of its higher-level constructs and protections can be corrupted and circumvented at will by low-level manipulations. ... The C language is so unsafe that striving to a total or almost total upward compatibility from C *cannot result in a good general-purpose object-oriented language.* What can be had is an object-oriented language mostly suited for low-level systems programming. ... Previous C users can quite well upgrade *gradually* to programming in C++, in the first step just feeding their existing C code through the C++ translator and checking if some small modifications would be necessary. Many people consider this rather a disadvantage. They claim that an abrupt change of paradigm is almost necessary to make programmers think in an object-oriented fashion. Therefore, it would be better to start with a language that *requires* object-oriented programming (e.g., Smalltalk or Eiffel), instead of one that merely *allows* it (e.g., Simula or C++). Today I think that good object-oriented programming (OOP) is more a matter of restraint and moderation than of very powerful features and extremism. The distinctive properties of OOP seem to be rather tempting to be overused. ... Anybody who has read some amount of recent OOP literature ... must have encountered interesting misuses of inheritance.

**Transframe:** Tailoring, rather than patching, has rid Java of the burden of the C-compatibility and made the language safer and simpler than C++.

## 10.4 Low-level access

---

**Joyner:** One of the stated advantages of C++ is that you can get free and easy access to machine level details. This comes with a down side: if you make a great deal of use of low level coding your programs will not be economically portable. Java has removed all of this from C, and one of Java's great strengths is its portability between systems, even without recompilation.

---

## 10.5 Use of C syntax

---

**Transframe:** Both C++ and Java have the C-style declaration syntax:

```
type name_list;
```

However, pointers and arrays are not the part of type in C++ variable declarations, this would cause a problem of confusion. in the following declaration:

```
int* i,j;
```

the `*` belongs to the variable `i`, not to the type `int*`. Java does not have pointers with `*`, but have arrays with `[]`. The declaration in C++:

```
int a[], b[];
```

can be alternatively written as:

```
int[] a, b;
```

in Java.

---

**Transframe:** Programming in C++ pointers is not only dangerous, but also very difficult in terms of its very complicated syntax with operators like `&`, `*`, and `->`. This is especially true when pointers are combined with templates, array structures, variable number of function inputs, etc. Just consider an example in using a template in C++'s STL [*Standard Template Library*]:

```
Set<T*> y;
for (Iterator x(y); x; x++) (*x)->M():
```

where `M` is a member function name defined in the class `T`. The code in the second line is very tricky. It is almost impossible for an average C++ user to figure out why the operator `*` should be used. Java and Transframe simplifies the usage of object references. Operators like `&`, `*`, `->` are not required. There is no tricky C++ expressions like `&(**x)->foo()`.

---

## 10.6 C++ is large and complicated

---

**Joyner:** [Ellemtel 92] suggests "C++ is a difficult language in which there may be a very fine line between a feature and a bug. This places a large responsibility upon the programmer." Is this a responsibility or a burden? The *fine line* is a result of an unnecessarily complicated language definition. The C++ standardisation committee warns "C++ is already too large and complicated for our taste".

Sun's Java White Paper says that in designing Java, "the first step was to *eliminate redundancy* from C and C++. In many ways, the C language evolved into a collection of overlapping features, providing too many ways to do the same thing, while in many cases not providing needed features. C++, even in an attempt to add *classes in C* merely added more redundancy while retaining the inherent problems of C."

---

## 10.7 Java is not for compute-intensive applications

---

**Transframe:** Java is not designed for compute-intensive applications that are critical in space and speed, as it is stated in *The Java White Paper*. . . .

---

## 10.8 Many meanings of `static`

---

**Transframe:** The keyword `static` in C++ has multiple meanings:

- a static member that has a class scope: access to the member should use the class name as a qualifier;
- a static local name that has a static duration in the global scope.

The second case is actually unnecessary in object-oriented programming. Therefore, Java removed the second usage and reserved the keyword `static` only for class members.

---

## 10.9 Many meanings of `final`

---

**Transframe:** Java uses `final` for three different meanings and sometimes may cause minor confusions:

- a final method whose implementation cannot be overridden in subclasses;
- a final name that constantly refers to the same object during the lifetime of the name;
- a final class which cannot have further declared subclasses.

---

## 10.10 Two meanings of `abstract`

---

**Transframe:** The term `abstract` used in Java has different meanings that must be distinguished by differences in the context. When it is used with a class, it means non-instantiatable; when it is used with a method, it means a delayed implementation.

---

## 10.11 All Java programs have concurrency and garbage collection

---

**Transframe:** Another fact is that each Java program needs the support of the built-in `java.lang` package, no matter how small and simple your application is. Concurrency is built in every object. You'll get the space penalty even you do not need garbage collection and concurrency. Many small embedded systems would not be able to use Java.

---

## 10.12 Use of header files and the preprocessor

**Sakkinen:** The *preprocessor* facility of C++ is inherited from C, and feels like a relict from the sixties. Preprocessing directives are rather foreign to the language proper, and they operate only on the lexical level. . . . Fortunately, there is less need to use preprocessor directives in C++ than in C (especially pre-ANSI C). . . . The preprocessor directive `#include` is the only means in C++ for defining export-import relationships between modules. It is a poor substitute for the facilities of most other object-oriented languages or e.g., Modula-2.

**Joyner:** In C++ a class interface must be maintained separately from its body . . . programmers must maintain the two sets of information. . . . Replicated information has the well known drawback that in the event of change, both copies must be updated. . . . Tools can automatically extract abstract class descriptions from class implementations, and guarantee consistency.

Neither Java, nor Eiffel need header files or the `#include` mechanism. This means that programmers do not have to maintain headers separately.

**Cornelius:** In the language Modula-2, the text of the interface for a module is placed in a definition module and the text of the code that implements the interface is placed in an implementation module. These two texts are stored in separate files. Similarly, in the language Ada, a package is divided into two parts: a package specification and a package body. You can mimic this in Java: instead of just defining a class, define an interface together with a class that implements that interface. Arnold and Gosling 96 says: "Any major class you expect to be extended, whether abstract or not, should be an implementation of an interface. Although this requires a little more work on your part, it enables a whole category of use that is otherwise precluded."

## 11 References

- Ken Arnold and James Gosling, *The Java Programming Language*, 0-201-63455-4, Addison-Wesley, 1996.
- Dave Dyer, *Top 10 Ways to get screwed by the C programming language*, http://www.andromeda.com/people/ddyer/topten.html.
- Daniel Edelson and Ira Pohl, *C++: Solving C's Shortcomings?*, *Computer Languages*, Vol. 14 No. 3 (September 1989), pp. 137-152.
- Ellemtel Telecommunication Systems Laboratories (Sweden), *Programming in C++: Rules and Recommendations*, 1992.
- M.A.Ellis and B.Stroustrup, *The Annotated C++ Reference Manual*, 0-201-51459-1, Addison-Wesley, 1990.
- David Flanagan, *Java in a Nutshell*, 1-56592-183-6, O'Reilly and Associates, 1996.
- Simon Garfinkel, Daniel Weise and Steven Strassman, *The Unix-Haters Handbook*, 1-56884-203-1, IDG Books, 1994.
- Jesper Jørgensen and Rod Ellis, *A Comparison of the Object Oriented Features of Ada95 and C++*, http://www.ddci.dk/ddci/papers/ada-cpp.html.
- Ian Joyner, *C++??: A Critique of C++ and Programming and Language Trends of the 1990s* (3rd edition), http://www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3.html.
- Robert C. Martin, *Java and C++: A Critical Comparison*, http://www.oma.com/PDF/javacpp.pdf.
- Bertrand Meyer, *Eiffel vs. C++*, http://www.progsoc.uts.edu.au/~geldridg/cpp/bmefcpp.htm.
- Markku Sakkinen, *The darker side of C++ revisited*, http://galen.med.virginia.edu/~sdm7g/LangCrit/shadow/Dark-Cplusplus.ps.gz.
- Sun Microsystems, *The Java Language Environment White Paper*, http://www.javasoft.com/nav/read/whitepapers.html.
- Bjarne Stroustrup, *The C++ Programming Language* (2nd edition), 0-201-53992-6, Addison-Wesley, 1994.
- Bjarne Stroustrup, *The Design and Evolution of C++*, 0-201-54330-3, Addison-Wesley, 1994.
- Jeff Sutherland, *Smalltalk, C++, and OO COBOL: the Good, the Bad and the Ugly*, http://www.tiac.net/users/jsuth/papers/oocobol.html.
- Transframe Technology Corporation, *Transframe, Java & C++: A Critical Comparison*, http://www.transframe.com/research/tf/tf_criti.htm.

---

The text of this comparison is updated from time to time. The latest version is available at: http://www.dur.ac.uk/~dcl0bjc/Java/. This WWW page also contains an overview of Java and a tutorial on how to write Java programs.