

Java to C++ conversion tips

Java and C++ look a lot like, which is helpful when learning one after knowing the other. However they have a number of differences, both minor and not so minor, which can be confusing if you are trying to code C++ based on your knowledge of Java. The purpose of this page is to show you some of the similarities and differences.

As a prerequisite you should know Java. You should also have read about C++ in the book of your choice: this page assumes you have seen language features or that you can look them up if you haven't.

Index

- Features of Java with C++ equivalents.
 - [Classes.](#)
 - [Inheritance.](#)
 - [Constructors.](#)
 - [Destructors.](#)
 - [Methods.](#)
 - [Base types.](#)
 - [Variables.](#)
 - [Member.](#)
 - [Local.](#)
 - [Class and global.](#)
 - [Constants](#)
 - [Parameters](#)
 - [Pointers](#)
 - [Arrays](#)
 - [Encapsulation.](#)
 - [Packages & Namespaces.](#)
 - [Strings.](#)
 - [Exceptions.](#)
- [Features of Java with no C++ equivalents.](#)
- [Features of C++ with no Java equivalents.](#)

Features of Java with C++ equivalents

The table below describes a number of Java concepts and shows you their C++ equivalents, as well as notes on how they differ in practice. (A word to the wise: some of them may look similar but they can differ quite a bit.)

Some conventions:

- Normal text is an explanation.
- Code.
- *Fill in with name as appropriate.*

Feature	Java	C++
Classes	<ul style="list-style-type: none"> • Declared and defined: <pre>public class Object { public Object(); private int hashCode; }</pre> • Abstract classes declared and defined: <pre>abstract public class Object { public Object(); private int hashCode; }</pre> 	<ul style="list-style-type: none"> • Declared (As a forward reference -- only pointer variables allowed): <i>No access specifier</i> <pre>class Object;</pre> Declared and defined: <i>No access specifier</i> <pre>class Object { public: Object(); private: int hashCode;</pre>

	<ul style="list-style-type: none"> • Interfaces: <pre>public Interface Implementable { // Constants // Abstract methods }</pre>	<pre>}; <-- Note the semi-colon!</pre> <ul style="list-style-type: none"> • Classes that cannot be instantiated are not explicitly declared as such. (See below about abstract methods.) • Interfaces do not exist <i>per se</i>. A class with only pure virtual methods and no instance variables is effectively an interface.
Inheritance	<ul style="list-style-type: none"> • Only one class can be extended from. <pre>public class Derived extends Base { }</pre> <ul style="list-style-type: none"> • Multiple interfaces can be implemented. <pre>public class Implementor implements Implementable, AlsoImplementable { }</pre>	<ul style="list-style-type: none"> • Inheritance has <code>public</code>, <code>protected</code>, and <code>private</code> forms. The first is the equivalent of Java inheritance. The others have no Java equivalents. <pre>class Derived : public Base { };</pre> <ul style="list-style-type: none"> • Multiple classes can be inherited from. <pre>class Derived : public Base, public AnotherBase { };</pre> <p>Note: Professor Reiss advises against using multiple inheritance at all.</p>
Constructors	<ul style="list-style-type: none"> • Declared and defined: <pre>public Object() { }</pre> <ul style="list-style-type: none"> • Calling a superclass: <pre>public Derived(parameters) { super(parameters); }</pre> <ul style="list-style-type: none"> • Called when an object is initialized with <code>new</code>. 	<ul style="list-style-type: none"> • Declared: <pre>public: Object();</pre> <ul style="list-style-type: none"> • Defined: <pre>Object::Object() { }</pre> <ul style="list-style-type: none"> • Calling a superclass: <pre>Derived::Derived(parameters) : Base(parameters); { }</pre> <ul style="list-style-type: none"> • Called when an object is declared, or is initialized dynamically using operator <code>new</code>.
Destructors	<ul style="list-style-type: none"> • <code>finalize</code> method called by garbage collector to free up resources associated with object. • Cannot predict when it will be called. 	<ul style="list-style-type: none"> • Declared: <pre>public: ~Object();</pre> <ul style="list-style-type: none"> • Defined: <pre>Object::~~Object() { }</pre> <ul style="list-style-type: none"> • Called whenever an object goes out of scope or is dynamically released with operator <code>delete</code>. • Destructors must be declared <code>virtual</code> if the class is going to have any other virtual methods.
Methods	<ul style="list-style-type: none"> • Declared and defined in <code>.java</code> file as: <pre>public Object method (parameters)</pre>	<ul style="list-style-type: none"> • Declared in a class in <code>.H</code> file as: <pre>Object method(parameters);</pre>

	<pre>{ }</pre> <ul style="list-style-type: none"> By default methods can be redefined by subclasses. Methods that cannot be redefined are declared <code>final</code>. Methods declared abstract must be redefined by subclasses: <code>abstract int method(parameters);</code> Methods on objects are called as: <code>object.method(parameters);</code> Methods can be defined to work on classes instead of objects by using <code>static</code>. <code>static</code> methods are called as: <code>Class.method();</code> 	<p>Defined in a .C file as:</p> <pre>Object Class::method(parameters) { } </pre> <p>Note that this is done outside the class declaration.</p> <ul style="list-style-type: none"> By default methods cannot be redefined by subclasses. Methods that can be defined are declared <code>virtual</code>. Methods which are <i>pure virtual</i> must be redefined by subclasses: <code>virtual int method(parameters) = 0;</code> Methods on objects are called as: <code>object.method();</code> <p>Methods on objects accessed through pointers are called as:</p> <pre>pointer->method();</pre> <ul style="list-style-type: none"> Methods declared <code>static</code> are called as: <code>Class::method();</code>
Base types	<pre>boolean (1 byte), byte (1 byte), char (2 bytes), short (2 bytes), int (4 bytes), long (8 bytes), float (4 bytes), double (8 bytes)</pre> <p>All numeric types are signed numbers.</p>	<pre>bool (varies), (unsigned char) (1 byte), char (1 byte), short (<= int), int (>= short and <= short), long (>= int), float (<= double), double (>= float), long double (>= double)</pre> <p>All integral types (including <code>char</code>) have both signed and unsigned forms. The default is usually signed but this varies from compiler to compiler.</p>
Variables (general) This is important	<ul style="list-style-type: none"> Instances of base types are allocated in automatic memory (the stack). Instances of user defined types are dynamically allocated (put on the heap). Instances of any type can be placed in static memory. All base types are set to 0 before being initialized. All object references are set to <code>null</code>. Constructors for objects are called by using <code>new</code>. They are <i>not called otherwise</i>. 	<ul style="list-style-type: none"> All instances of both base types and user defined types can be allocated in automatic, dynamic memory, or static memory. All initialization must be done by the programmer. The value of an uninitialized object is undefined. (Initialization should be done in constructors to prevent objects from being uninitialized.) Constructors for objects that are <i>not</i> created with <code>new</code> are called <i>automatically</i> as noted below. Constructors for objects that <i>are</i> created with <code>new</code> (in other words ones declared as pointers) are called when <code>new</code> is used. They are <i>not called otherwise</i>. Destructors for objects that are <i>not</i> created with <code>new</code> are called <i>automatically</i> as noted below. Destructors for objects that <i>are</i> created with <code>new</code> (in other words ones declared as pointers) are called when <code>delete</code> is used. They are <i>not called otherwise</i>.

<p>Member (instance) variables</p>	<ul style="list-style-type: none"> Declared inside classes in .java files as: <pre>private Object variable;</pre> <p>Defined in constructors:</p> <pre>public Object() { variable = new Object(); }</pre>	<ul style="list-style-type: none"> Declared inside classes in .H files as <pre>Object object; Object * pointer;</pre> <p>Defined in the .C file in the constructor of the class they are in:</p> <pre>// Initialization list Object::Object() : object(), pointer(new Object()) { // Normal initialization. // object is automatically constructed; pointer = new Object(); }</pre> <ul style="list-style-type: none"> Constructors for members are called automatically by the constructors of their containing class. <p>Destructors for members are called automatically by the destructors of their containing class.</p>
<p>Local variables</p>	<ul style="list-style-type: none"> Local variables are declared inside methods in .java files as: <pre>Object variable;</pre> <p>They are defined at any point in the method.</p>	<ul style="list-style-type: none"> Local variables are declared inside methods in .C files: <pre>Object variable;</pre> <p>Constructors for locals are called automatically when the object is declared.</p> <p>Destructors for locals are called automatically when the object goes out of scope.</p>
<p>Class and global variables</p>	<ul style="list-style-type: none"> Class variables are declared inside classes in .java files: <pre>private static Object variable = value;</pre> <p>They are defined where they are declared.</p> <pre>public static Object object = new Object();</pre> <ul style="list-style-type: none"> Global variables do not exist but public class variables are effectively global. 	<ul style="list-style-type: none"> Declared inside classes in .H files: <pre>static Object variable;</pre> <p>Defined <i>outside</i> classes in .C files, prefixed with the class name, and without the <code>static</code> keyword:</p> <pre>Object Class::variable = value;</pre> <ul style="list-style-type: none"> Global variables are declared outside any class or method. They are declared and defined just like class variables except the class name is not necessary in the definition. <p>Constructors for globals and statics are called when object is defined, before <code>main()</code> is called. <i>The order in which static objects are actually constructed is undefined.</i></p> <p>Destructors for globals are called after <code>main()</code> exits. The order of those calls is undefined.</p>
<p>Constants</p>		<ul style="list-style-type: none"> Declared using the <code>const</code> keyword in a .H file. (The <code>extern</code> means the definition is in a .C file):

- Declared using the final keyword.

```
public final int
CHARS_PER_LINE = 80;
public final String FILENAME =
"/tmp/foo";
public final Color RED = new
Color(255, 0, 0);
```

- Are always declared inside a class either per object or per class (using the keyword `static`).

```
// Object
public final int
CHARS_PER_LINE = 80;
// Class
public static final int
CHARS_PER_LINE = 80;
```

- Most constants are declared `static` so they can be accessed outside the class without needing an object of that class.

```
public class Object
{
    public static final int
    CHARS_PER_LINE = 80;
}
```

```
int cpl =
Object.CHARS_PER_LINE;
```

- A final base type means the constant's value cannot change.

A final object **can** have it's value changed, but the reference cannot attach to a different object.

```
// Illegal
```

```
extern const int CHARS_PER_LINE;
extern const char * FILENAME;
extern const Color RED;
```

Can be **defined** in a .H file, but only safe for integers or simple base types:

```
const int CHARS_PER_LINE = 80;
const int PI = 3.14;
```

Compound or user defined types (and this include `char *` and `string` should be defined in a .C file (the reason has to do with memory allocation and linking). This looks just like defining in the .H file:

```
const char * FILENAME = "/tmp/foo";
const Color Red = Color(255, 0, 0);
```

- Can be declared inside a class either per object or per class (using the keyword `static`):

```
class Object
{
    const int CHARS_PER_LINE;
    static const int CHARS_PER_LINE;
};
```

Constants in a class can be initialized either in a .H or in a .C. In either case this must occur in *every* constructor's initialization list (this is a pain...).

```
// .H
class Object
{
    const int CHARS_PER_LINE;
    Object() : CHARS_PER_LINE(80)
    {
    }
};

// .C
Object::Object()
    : CHARS_PER_LINE(80)
{
}
```

`static` constants can be initialized in a .H or the .C. In the latter case they are initialized outside a method body and are prefixed with a class name.

```
// .H
class Object
{
    static const int CHARS_PER_LINE = 80;
};

// .C
const int Object::CHARS_PER_LINE = 80;
```

- Most constants are declared `static` so they can be accessed outside the class without needing an object of

	<pre> CHARS_PER_LINE = 79; // Illegal RED = new Color(0, 255, 0); // Legal! RED.setColor(0, 255, 0); </pre>	<p>the class.</p> <pre> class Object { static const int CHARS_PER_LINE = 80; }; int cpl = Object::CHARS_PER_LINE; </pre> <ul style="list-style-type: none"> • A <code>const</code> base type means the constant's value cannot change. A <code>const</code> object cannot have <i>any</i> of its members change. A <code>const</code> pointer cannot change what it is pointing to, but the <i>value</i> of what it points to can change unless what it points to is <i>also</i> <code>const</code>. Control over <code>const</code>-ness is fine but complex to understand. • It is a good idea to put constants in a class and/or namespace to prevent name conflicts: <pre> class FileManager { static const int CHARS_PER_LINE; static const char * FILENAME; }; namespace project { extern const int CHARS_PER_LINE; extern const char * FILENAME; } </pre>
Parameters	<ul style="list-style-type: none"> • All base types are passed by value (a copy is made in the called function). <pre> void dont_change_value(int value) { value = local; } </pre> <ul style="list-style-type: none"> • All user defined types are passed by references (the original object is passed to the function). <pre> void change_value(Object value) { value = local; } </pre> <ul style="list-style-type: none"> • Arrays are treated as objects (passed by reference). 	<ul style="list-style-type: none"> • All types (base or user-defined) are passed by <i>value</i> by default. This includes pointers. <p>This means that in function <code>some_array</code> is a pointer to <code>buffer</code>. Size information is not available. It must be passed separately.</p> <ul style="list-style-type: none"> • Any object can be passed by reference: <pre> void dont_change_value(Object object) { object.method(); object = local_object; } void change_value(Object& reference) <-- Note the ampersand! { reference.method(); reference = local_object; } </pre> <p>The syntax for using <code>object</code> and using <code>reference</code> is identical.</p> <ul style="list-style-type: none"> • Arrays are passed <i>as pointers</i>: <pre> Object buffer[1024]; function(buffer); </pre>

		<pre>void function(Object * some_array, int size) { some_array[0] = local_object; }</pre>
Pointers	<ul style="list-style-type: none"> All references to objects are pointers constrained to point actual objects. null signifies no object. 	<ul style="list-style-type: none"> Pointers are used to refer to dynamically allocated objects. Pointers can also point to base types (and in fact to any random address). A "null" pointer is signified by NULL (which is 0). How to read pointers: Object o; -- Object Object * p; -- Pointer Object a[SIZE]; -- Array of objects <pre>o; // Object &o; // Address of/pointer to object p; // Pointer to object *p; // Object &p; // Address of/pointer to pointer to object a[0]; // Object &a[0]; // Pointer to object a; // Array (pointer). Equivalent to previous.</pre>
Arrays	<ul style="list-style-type: none"> Arrays are objects. Declared and initialized: <pre>Object buffer; buffer = new Object[size];</pre> <ul style="list-style-type: none"> Multi-dimensional arrays: <pre>Object [][] array; array = new Object[rowsize][columnsize];</pre>	<ul style="list-style-type: none"> Arrays are objects, but array <i>variables</i> are actually pointers. Locally declared and initialized: <pre>Object buffer[size];</pre> <p>Dynamically declared and initialized:</p> <pre>Object * buffer; buffer = new Object[size];</pre> <ul style="list-style-type: none"> Dynamically destroyed: <pre>delete[] buffer;</pre> <ul style="list-style-type: none"> Multi-dimensional arrays: <pre>Object **array; array = new Object* [rowsize]; <-- Note the asterisk! That's a pointer not a multiply. for(int i = 0; i < rowsize; i++) array[i] = new Object[columnsize];</pre>
Encapsulation	<ul style="list-style-type: none"> Four levels: <pre>public: Accessible to all classes in the program. protected: Accessible only by subclasses and other classes in the package. private: Accessible only by the class.</pre>	<ul style="list-style-type: none"> Three levels: <pre>public: Accessible anywhere in the program. protected: Accessible only in a subclass. private: Accessible only inside the class.</pre>

	<p>package: Accessible only inside the defining package.</p> <p>Each member function or variable must be preceded by an accessor keyword (the default is package):</p> <pre>public Object() { ... } private Object o; private int i;</pre>	<p>Access is determined for sections of the class:</p> <pre>public: Object(); private: Object o; Object * op; int i;</pre>
<p>Packages and Namespaces</p>	<ul style="list-style-type: none"> Namespaces are managed on per-file basis by using the <code>package</code> keyword. There is only one package per file. <pre>package example; // ...</pre> <ul style="list-style-type: none"> Names are brought into the current package by using the <code>import</code> keyword. Entire packages can be imported or individual names (classes) declared inside them. Names can also be accessed by prefixing them with their package. <pre>// all names in the example package import example.*; // The class Class in the example package import example.Class; // Using a class with it's full name example.Class object;</pre> <ul style="list-style-type: none"> Packages can be nested. The nesting follows the directory structure containing the package binaries and source. <pre>package outer.inner; // ...</pre> <ul style="list-style-type: none"> Standard library is located inside the <code>java</code> package and its sub-packages. Importing accesses a package and makes the classes and methods inside them accessible. <pre>// import java.lang import java.lang.*; // and access a class in it private String a_string;</pre>	<ul style="list-style-type: none"> Namespaces are managed in blocks (starting with <code>{</code> and ending with <code>}</code>) by using the <code>namespace</code> keyword. There may be more than one namespace declared in a file. <pre>namespace example { // ... }</pre> <ul style="list-style-type: none"> Names are brought into the current namespace by using the <code>using</code> keyword. Entire namespaces can be imported or individual names (types, variables, functions) declared inside them. <pre>// all names in the example namespace using namespace example; // the class Class in the example namespace using example::Class; // using a class with it's full name example::Class object;</pre> <ul style="list-style-type: none"> Namespaces can be nested. This nesting can follow the directory structure containing the code but does not have to. <pre>namespace outer { namespace inner { // ... } }</pre> <ul style="list-style-type: none"> Standard library is located inside the <code>std</code> namespace. Using a namespace allows access to objects declared in that namespace without requiring a prefix. It does NOT allow access to all the objects in that namespace. You still need to declare them or get access to standard declarations by <code>#include</code>'ing headers. <pre>// use std using namespace std; // doesn't mean you can use a class in it // you still need to #include <string> string a_string;</pre>
<p>Strings</p>		<p>No built-in string support, but a class <code>string</code> is in the</p>

	Classes <code>String</code> and <code>StringBuffer</code> . Language understands strings and treats them as primary types.	standard library. It's fairly new: much older code, particularly that derived or using C code has the convention of passing strings around as arrays of characters (<code>char *</code>).
Exceptions	<ul style="list-style-type: none"> All catchable exceptions subclass off of <code>java.lang.Exception</code>, which is part of the standard library. Objects are caught by class name. Throwing exceptions: <code>throw new Exception();</code> Catching exceptions: <pre>try { // possible exception } catch(Exception e) { // handle exception } finally { // handle cleanup }</pre> To catch all exceptions: <code>catch (Exception e)</code> 	<ul style="list-style-type: none"> Any object or base type can be thrown as an exception. The standard library has a class <code>exception</code> but exceptions are not required to subclass off of it. Objects are caught by type name. Note that this can include pointers and references to Objects (you almost always want to catch by reference). Throwing exceptions: <code>throw Exception();</code> Catching exceptions: <pre>try { // possible exception } catch(SomeClass& object) { // code }</pre> No equivalent to <code>finally</code>. To catch all exceptions: <code>catch(...)</code>
Comments	<ul style="list-style-type: none"> <code>// Line comment.</code> <code>/* Block comment. */</code> <code>/** Documentation comment -- read by javadoc */</code> 	<ul style="list-style-type: none"> <code>// Line comment</code> <code>/* Block comment. */</code> <i>No particular documentation comments</i>

Features of Java with no C++ equivalents

Not too many of these: Java was designed to be simpler than C++.

- Final classes*
You can't make a class unsubclassable in C++. However the main reason to do that in Java is to allow optimizations that C++ does most of the time as a statically binding language.
- Garbage collector:*
They can be written and used but they are not part of the language.
- Native methods*
All methods are native in C++.
- Threads and synchronized blocks*
Threads are not part of the C++ language but are accessible in libraries. The equivalent of a synchronized block is a lock object of some kind (semaphores and mutexes).

Features of C++ with no Java equivalents

There are a lot of these. They're just listed here: if you want to know what they are you should grab a book. Note that you really don't need to know many of these for CS32. But most books will mention them if you are interested in knowing more.

- Class initialization lists*
- Enumerations*
- Global variables and functions (one outside a class)*

- *Inline methods*
- *Multiple inheritance (of classes, as opposed to interfaces)*
- *Operator overloading*
- *Preprocessor*
- *Private and protected inheritance*
- *Structs and unions*
- *Templates*

Last modified: Fri Jan 28 10:55:29 EST 2000