An Introduction to Java

Tom Anderson Andy Collins

January 9, 1998

1 Introduction

Java is an interesting language with several interesting goals. In many ways, it is a reaction to C++, and an attempt to create an object-oriented language with all the useful features of C++ but without all the complexity. Removing features from a language is never a good way to make friends, and for every feature removed you can find at least one vocal person denouncing Java for having removed it.

In my opinion, however, Java is pretty much right on the mark, it represents a nearly ideal subset of C++, and cleans up a whole range of problems that just shouldn't be in a language for programming the vast majority of applications. C++ is way too complicated and way too dangerous for everyday use, and Java is just about right.

This article was adapted from an earlier version on C++. Most of the general information on programming with classes and inheritance has been carried over unchanged, and would apply to most any class-based object-oriented language as well. That article used this space to argue that C++ should be divided into fundamental features, sometimes useful features and features to be avoided at all cost. I think it says a lot about Java that this version no longer has a section on features to be avoided at all costs.

The goal of this paper is to get reasonably experienced C programmers up to speed in Java as quickly as possible. There are lots of examples which I hope will be clear and self-explanatory. However, you will not get very far without some real reference materials. The most useful are available online from `http://www.javasoft.com/`.

The paper is aimed at C and C++ programmers. You can learn Java without knowing either of these, but the syntax will feel unfamiliar. C++ programmers should have an easier time of learning Java, because they will already understand the notion of classes, but beware: Java is not simply (C++)–, and you should read the sections on variables and memory-management before you simply attempt to adapt your C++ knowledge. There are many subtle syntax and functional differences between Java and C++, they are generally for the better, but they are there nonetheless.

1.1 A note on the origins of this paper

This paper was adapted from *A Quick Introduction to C++* by Tom Anderson. That paper was designed to teach the important features of C++ assuming a background in C. The sections

This article is adapted from *A Quick Introduction to C++*, by Tom Anderson, which is based on an earlier version written by Wayne Christopher.

on general object-oriented programming have been carried over pretty much unchanged. This includes section 2 on classes, the beginning of section 4 on constructors and the operator `new`, the "new features of Java and C++" in section 5.1, the discussion of inheritance in section 8.1, and the style guidelines in section 9. If you are familiar with C++ then you can probably just skim these sections to spot the minor syntax differences: the concepts should be the same.

What is new for the Java version is sections 3 and 4, "Java has no pointers" and "Java has no delete," the "New features of Java only" in section 5.2, section 6 on exceptions, section 7 on actually writing programs, and section 8.2 on static members. These are the important parts for C++ programmers to read, because they describe the differences between Java and C++. C++ programmers should be warned: Java looks like C++, but Java is not the same as C++; there are lots of little syntax differences and lots of important changes in memory management. You will benefit from your C++ knowledge, but be careful in directly applying it.

2 Basic Concept: classes

Before giving examples of Java features, we will first go over some of the basic concepts of object-oriented languages. If this discussion at first seems a bit obscure, it will become clearer when we get to some examples.

1. **Classes and objects**. A class (in C++ or Java) is similar to a C *structure*, except that the definition of the data structure, *and* all of the functions that operate on the data structure are grouped together in one place. An *object* is an instance of a class (an instance of the data structure); objects share the same functions with other objects of the same class, but each object (each instance) has its own copy of the data structure. A class thus defines two aspects of the objects: the *data* they contain, and the *behavior* they have.

2. **Member functions**. These are functions which are considered part of the object and are declared in the class definition. They are often referred to as *methods* of the class. In addition to member functions, a class's behavior is also defined by what to do when you create a new object (the **constructor** for that object) – in other words, how you initialize the object's data.

3. **Private vs. public members**. A public member of a class is one that can be read or written by anybody, in the case of a data member, or called by anybody, in the case of a member function. A private member can only be read, written, or called by a member function of that class.

Classes are used for two main reasons: (1) it makes it much easier to organize your programs if you can group together data with the functions that manipulate that data, and (2) the use of private members makes it possible to do *information hiding*, so that you can be more confident about the way information flows in your programs.

In Java there is also a third reason to use classes: you can't not use them. All Java code must appear inside some class. There are no global functions or global variables, although we will see ways to simulate both of these things using `static` members.

2.1 Classes

Java classes are roughly similar to C structures (and C++ classes) in many ways. In the following explanation of how classes work, we will use a stack class as an example.

Member functions. Here is a (partial) example of a class with a member function and some data members:

```
class Stack
{
   public void Push(int value)   // Push an integer
   {
      if (top >= 10)
         HandleAnError();        // See section on exceptions

      stack[top++] = value;
   }
   public int top;               // Index of the top of the stack
   public int[] stack;           // The elements of the stack (an array)
}
```

This class has two data members, `top` and `stack`, and one member function, `Push`.

If we have a `Stack` object called `s`, we can access the `top` element as `s.top`, just as in C. However, in Java (as in C++) we can also call the member function using the following syntax:

```
s.Push(17);
```

Inside a member function, one may refer to the members of the class by their names alone (i.e. without the `s.` used above). In other words, the class definition creates a scope that includes the member (function and data) definitions.

Note that if you are inside a member function, you can refer to the object you were called on by using the variable `this`. This is similar to the C++ pointer `this`, except you use it with a dot (`.`) rather than the C++ arrow (`->`); see the discussion "Java has no pointers" for more explanation, and for now don't worry about it. If you want to call another member function on the same object, you do not need to use the `this` variable. Let's extend the Stack example to illustrate this by adding a `Full()` function.

```
class Stack
{
   public void Push(int value)
   {
      if (Full())                // Use the Full function,
         HandleAnError();        // See section on exceptions

      stack[top++] = value;
   }

   public boolean Full()        // Returns true if the stack is full,
   {                            // false otherwise.
      return (top == 10);       // space for 10 elements
   }

   public int top;
   public int[] stack;
};
```

We could also have written the error check as:

```
    if (this.Full())
       HandleAnError();         // See section on exceptions
```

but in a member function, the `this.` is implicit.

The purpose of member functions is to encapsulate the functionality of a type of object along with the data that the object contains. A member function does not take up space in an object of the class.

Private members. One can declare some members of a class to be *private*, which are hidden to all but the member functions of that class, and some to be *public*, which are visible and accessible to everybody. Both data and function members can be either public or private.

In our stack example, note that once we have the `Full()` function, we really don't need to look at the `top` or `stack` members outside of the class – in fact, we'd rather that users of the Stack abstraction *not* know about its internal implementation, in case we change it. Thus we can rewrite the class as follows:

```
class Stack
{
   public void Push(int value)
   {
      if (Full())
         HandleAnError();         // See section on exceptions

      stack[top++] = value;
   }

   public boolean Full()
   {
      return (top == 10);
   }

   private int top;
   private int[] stack;
};
```

Before, given a `Stack` object, say `s`, any part of the program could access `s.top`, in potentially bad ways. Now, since the `top` member is private, only a member function, such as `Full()`, can access it. If any other part of the program attempts to use `s.top` the compiler will report an error.

Unlike C++, the `public` and `private` keywords must be applied to each member. You cannot declare whole groups of members to be public or private at once. It is also legal to have members be neither public nor private by not specifying either keyword. These members will be accessible to any code in the same *package*. Packages are an advanced feature that we will discuss briefly much later. For now, specify everything to be either public or private explicitly.

One important deisgn prinicple you should adhere to: **all data members of a class should be private.** All operations on data should be via that class' member functions. Keeping data private adds to the modularity of the system, since you can redefine how the data members are stored without changing how you access them.

3 "Java has no pointers"

To many people, this one statement is the most crucial defining aspect of Java. This is unfortunate, because in my opinion this statement is also false.

What is true, however, is that Java has no pointer *syntax*, and that in Java it is the *language*, rather than the programmer, that controls what is and what is not a pointer. Again in my opinion this is one of the dirty little secrets of Java and it seems that that most books on the subject attempt to sweep it under the rug, apparently because they try to promote Java as a language for non-programmers, and non-programmers don't want to think about pointers.

I on the other hand think that it *is* important to think about pointers and pointer/non-pointer semantics when programming. Let's consider the following piece of code. This is legal Java code, except that it would have to appear inside some class declaration, since everything goes inside some class declaration. `Stack` is the stack class that we have defined.

```
void Func1(int a, Stack b, int[] c)
{
   a = 4;
   b.Push(5);         // change b
   c[4] = 6;          // c is an array (Java syntax is a bit different)
}

void Func2()
{
   int x = 5;
   Stack y = someStackInstance;   // don't worry about this yet
   int[] z = someArrayInstance;

   Func1(x, y, z);
}
```

This is a standard example to demonstrate pass by reference versus pass by value semantics, and the question is, "Are the values of `x`, `y` and `z` changed by the call to `Func1`?"

In C or C++ (assuming the minor syntax worked), the answer would be that `x` and `y` are not changed, because they are passed by value (since there is no pointer or reference syntax associated with them), and that `z` is changed, because it is an array and arrays are special.

In Java, on the other hand, the answer is that `x` is not changed, `z` is changed, and `y`, the instance of type `Stack`, *is* changed. The reason that `x` and `y`, behave differently here, even though both were subjected to roughly the same syntax, is that `x` is an instance of a *basic* type, while `y` is a user-defined type. In Java all basic types are handled and passed by value, as in C or C++, but all user-defined types (including many defined by Sun for you to use) are handed and passed using *hidden* pointers. The basic types are defined by the core of the language. They are:

name	**width** (bits)	**format**
`int`	32	signed integer
`long`	64	signed integer
`short`	16	signed integer
`byte`	8	signed integer
`float`	32	IEEE floating point
`double`	64	IEEE floating point
`char`	8	ASCII character
`boolean`	-	either `true` or `false`

Everything else is a "user-defined" type, including arrays, system library classes and actual user-defined classes. All variables of user-defined types are actually pointers, and their behavior is roughly consisent with that of pointers in C or C++. Consider the code:

```
Stack a;          // a 'points to' null (like NULL)
a = new Stack();  // allocate an object on the heap
Stack b = a;      // a and b point to the same object
a.Push(5);        // changes that object, visible through a and b
a = someOtherObj; // a points somewhere else, b unchanged
```

This illustrates most of the important things to pay attention to. Since `Stack` is a user-defined type, it will behave like a pointer. Since Java abhors undefined variables, when `a` is declared it is automatically set to `null` (the Java equivalent of `NULL` in C or C++). Objects of any user-defined type may be compared with and assigned to `null`. This is important to remember, since the statement `a.Push(5)` would be an error if `a` were `null`.

The second line actually allocates an object of type `Stack`, and makes `a` point to it. Memory allocation is described more fully in a moment.

The third line shows that assignment of user-defined types leads to shared memory. `a` and `b` now point to the same object, so when that object is changed in the fourth, both `a` and `b` reflect that change.

The fifth line serves again to demonstrate that assigning to the variable itself causes the variable to point somewhere else, and does not change the object that the variable used to point to.

Since user-defined types are handled as pointers, and objects of these types are created on the heap, we should now consider the Java facilities for managing heap memory. These will look somewhat familiar to C++ programmers, but memory management is another area where Java differes substantially from C++ in actual behavior.

4 "Java has no delete:" memory management in Java

Constructors and the operator new. In C, in order to create a new object of type `Stack`, one might write:

```
struct Stack *s = (struct Stack *) malloc(sizeof (struct Stack));
InitStack(s, 17);
```

The `InitStack()` function might take the second argument as the size of the stack to create, and use `malloc()` again to get an array of 17 integers.

The way this is done in Java is as follows:

```
Stack s = new Stack(17);
```

The `new` function takes the place of `malloc()`. To specify how the object should be initialized, one declares a *constructor* function as a member of the class, with the name of the function being the same as the class name:

```
class Stack
{
   public Stack(int sz)
   {
      size = sz;
      top = 0;
      stack = new int[size];     // allocate a new array
   }

   public void Push(int value)
   {
      if (Full())
         HandleAnError();        // See section on exceptions

      stack[top++] = value;
   }

   public boolean Full()
   {
      return (top == size);
   }

   private int size;
   private int top;
   private int[] stack;
};
```

There are a few things going on here, so we will describe them one at a time.

The `new` operator automatically creates (i.e. allocates) the object and then calls the constructor function for the new object.

Since all class instances are managed like pointers and allocated with `new`, the objects are stored on the heap and will remain after the function that allocated it returns. Unlike C++, in Java you do not need to explicitly free this memory: the system implements a *garbage collector*, described below, which will take care of it for you.

The `new` operator can also be used to allocate arrays, illustrated above in allocating an array of `int`s, of dimension `size`:

```
stack = new int[size];
```

Note that you *cannot* use new with basic types, the statement:

```
int i = new int;
```

is illegal, because basic types are not managed like pointers.

Cleaning up: the garbage collector. One way in which Java differes from C and C++ is it's attitude towards freeing memory. In C and C++, the programmer must explictly specify (using `free()` or `delete`) when a piece of heap memory will no longer be used, so that the system may reclaim it for reuse. As any C or C++ programmer knows, this is painful and error-prone. Java eliminates this entire class of problems by using a *garbage collector*.

In Java there is *no* explicit way to tell the compiler to reclaim a piece of memory (in fact there cannot be, because Java strives to be a type-safe language, and explicit deletes are a type-safety loophole). Instead, the programmer simply allocates memory as he or she pleases, and doesn't worry at all (or at least not much) about the possibility of running out. When you are done with an object, you can simply point the variable that you were using somewhere else and forget about the object. In C or C++ this would constitute a memory leak (because the object is still allocated even though the program has no pointers to it). The crucial thing to notice, however, is that if your program has no pointers to a piece of memory (either directly or indirectly, i.e. a pointer to an object which in turn has a pointer to the memory in question) then, by definition, it no longer depends on what happens to that memory, so the Java runtime system can reclaim it automatically whenever it needs more memory. Conceptually this is done by stopping the program momentarily, starting with the set of active variables that the program has access to, following them and marking what they point to as "visible," and then following any pointers in the newly "visible" memory and marking what *they* point to as "visible" and so on. When this process terminates it must be the case that any memory which has not been marked "visible" cannot, through any possible sequence of actions, ever be accessed by the program, and therefore the system can reclaim it. Once the garbage collection is finished, the program is restarted and continues working.

More than any other aspect of Java, garbage collection is a nearly religious issue. Basically this comes from two fronts:

- Those people who have used older garbage-collected systems which tended to be slow and have a "stop-the-world" behavior, where the program would pause unexpectedly while garbage collection happened, and the system would appear to be locked-up.
- C/C++/assembly hacks who want to control every aspect of their program's execution.

The arguments in favor of garbage collection generally look like:

- Lots of research has lead to much better garbage-collectors which almost never exhibit really awful pauses.
- For the vast majority of applications, really precise control just isn't a problem, and
- Computers are so fast that it is more important to worry about programmer time (where garbage-collection is a huge win since it eliminates so many nasty errors) than about execution time.

The problem with religious debates, of course, is that they drown out the relevant facts and any attempt at rational discussion. There is a place for garbage-collection and there is a place for explicit memory management. In my opinion the place for garbage-collection encompasses the lion's share of current large-scale software development, and the place for explicit management is low-level systems and real-time environments. Most applications are neither low-level nor real-time.

`public void finalize()` One final note on the subject of memory deallocation is the `finalize` function. In Java this serves the remaining role of the C++ destructor. Note that virtually nobody actually needs to or does use `finalize`, but it is there. The idea is that, although the garbage collector will free up any memory when needed, sometimes there is additional stuff that you want to do when memory objects go away. As an example, consider a class which tried to keep track of how many instances existed. It is easy enough to have the constructor increment some counter somewhere in the program, but how can we make sure that that count gets decremented when the object goes away? If you put a function named `finalize` in your class, then the Java runtime system will call that function just before the object is garbage collected, so this would provide a means of solving this problem. It is still a bit weird, because the object might not get garbage collected until a long time after it became garbage (and the program might even end before that happens), so often it is best to simulate this with a "normal" member function that you explicitly call when you are done with the object (and then get rid of all your references to the object so the system can gabage-collect it).

5 Other Basic Java Features

Here are a few other Java features that are useful to know. Some of these are common with C++ and some are unique to Java.

5.1 New features in Java and C++

These features were added in C++, and are carried over in Java.

1. As we have seen, when you define a `class Stack`, the name `Stack` becomes usable as a type name. In C, if you wanted to do the same thing you needed to use a `typedef`. Java has no keyword `typedef`.

2. As in C++, inside a function body, you can declare some variables, execute some statements, and then declare more variables. This can make code a lot more readable. In fact, you can even write things like:

   ```
   for (int i = 0; i < 10; i++) ;
   ```

3. Like C++, comments can begin with the characters `//` and extend to the end of the line. These are usually more handy than the `/* */` style of comments.

4. Java and C++ provide some new opportunities to use the `const` keyword from ANSI C. The basic idea of `const` is to provide extra information to the compiler about how a variable or function is used, to allow it to flag an error if it is being used improperly. You should always look for ways to get the compiler to catch bugs for you. After all, which takes less time? Fixing a compiler-flagged error, or chasing down the same bug using the debugger?

   For example, you can declare that a member function only reads the member data, and never modifies the object:

   ```
   class Stack
   {
      ...
   ```

```
    boolean Full() const;  // Full() never modifies member data
    ...
};
```

As in C and C++, you can use `const` to declare that a variable is never modified:

```
const int InitialHashTableSize = 8;
```

5.2 New features in Java only

These features are unique to Java, and set it apart from C++.

1. Java has no preprocessor. All the C and C++ "statements" that began with a # sign no longer exist. Hopefully this means only `#include` for header files and `#define` for constants. The `const` syntax above replaces defined constants, and the `import` command is used in place of `#include` and *packages* are used in place of `.h/.cpp` file pairs.

2. Packages are Java's way of grouping classes into libraries. You won't need to create your own packages for this class, but you will need to import and use several library packages. The packages which come standard with every Java implementation are:

	name	purpose
	java.applet	Stuff for running Java programs inside web pages
	java.awt	Graphical interface library
	java.awt.image	More GUI stuff
	java.awt.peer	Even more implementation-related GUI stuff
⋆	java.io	Input and output classes
⋆	java.lang	Almost part of the language
	java.net	Networking classes
⋆	java.util	Lots of standard data structures

   The packages marked with a star (⋆) are those which you will have to use for this class. Note that the package java.lang is automatically imported into all Java programs.

   To use classes from one of these packages, use the import statement. The statement:

   ```
   import java.io;
   ```

   will allow you to use all the classes in the java.io package. The `import` statement appears on its own, not inside a class declaration. See the documentation for a list of all the classes in these packages and information on what they can do.

3. Java has a completely different scheme for input and output than either C or C++. The fundamental classes are `InputStream` and `OutputStream` in the java.io package. These are related in concept to the C++ `istream` and `ostream`, but, since Java does not allow overloaded operators, the usage is rather different.

   To print out a message in a Java program, use the following syntax:

   ```
   System.out.println("Hello world.");
   ```

This statement is broken down as follows:

- `System` is a class in the java.lang package. It provides an interface to a bunch of things that the computer platform provides for your program, including input and output. This class is rather weird, because there are *no* instances of it: all the members are static, and as a result you can use the seemingly broken syntax of requesting a member from a class. The dot (.) in `System.out` is actually like the `::` operator in C++, since it is being applied to a class rather than an object. If you don't understand this, don't worry about it, it isn't important. What is important is that
- `System.out` is an object of type `PrintStream`. `PrintStreams` are a form of output stream (the relationship is set up with inheritance, which is described in the advanced features section). This particular instance is set up for you so that anything you output here goes to standard output (the same place that `cout` does in C++ or `stdout` does in C, generally the screen).
- Since `System.out` is a `PrintStream`, it has a member function named `println` which takes a string and outputs that string followed by a newline (there is also a `print` function which does not output a newline). So the big thing going on in the statement is that this function is getting called, on a stream connected to standard output, and with the string `"Hello world."` as an argument.

One other output-related trick is that you can print variables other than strings through a cute string-concatenation trick. This works because strings in Java, unlike strings in C or C++, are implemented using a real class with a proper interface (the class `String` in the package java.lang, take a look in the documentation to see all the string routines that are available). One thing that strings allow is concatenation by using the plus (+) operator, so you could do the following:

```
System.out.println("Hello" + " " + "world.");
```

with exactly the same effect. Even cuter, if you try to "add" a string and some other basic data type, the basic type gets converted to a string form first, so you could do:

```
int i = 37;
double n = 42.3;
char s = ' ';
System.out.println ("The integer is " + i + ", and the double is" + s + n);
```

and this would output

```
The integer is 37, and the double is 42.3
```

A common trick you will see is to simply print out the value of a basic type:

```
int i = 37;
System.out.println ("" + i);
```

The argument to `println` must be a string, so you can't just pass `i`, but `"" + i` forces conversion to a string.

Java's input mechanisms are a bit more complicated. Similar to output, you can use `System.in` to read from standard input. `System.in` is an `InputStream`. This is a very simple interface to an input device. You can read a single character by:

```
int i = System.in.read();
char c = (char) i;
```

The best way to do more powerful input is to use the other classes in the java.io package. The most powerful is the `StreamTokenizer`. You can create a tokenizer connected to standard input with

```
StreamTokenizer t = new StreamTokenizer(System.in);
```

The `StreamTokenizer` constructor takes an `InputStream` instance as a parameter. Actually using it is more complicated than I want to describe here, so you'll need to look at the documentation.

6 Handing Errors: Exceptions

Exceptions are a complicated, flexible and powerful way of dealing with errors while the program is running. They allow you to separate out the "normal case" processing from the "exceptional case" processing to reduce reader confusion, and they make it possible to ensure that errors are dealt with, even in the face of client programmers who don't want to deal with errors.

Exceptions in Java are very similar to exceptions in C++, with one important difference: in Java you cannot *not* use exceptions. We will demonstrate exceptions by converting our stack example to properly handle overflow errors:

```
class OverflowException extends Exception
{
   public OverflowException (int mx)
   {
      max = mx;
   }

   public int MaxSize ()
   {
      return max;
   }

   private int max;
}

class Stack
{
   public Stack(int sz)
   {
      size = sz;
      top = 0;
      stack = new int[size];     // allocate a new array
   }

   public void Push(int value) throws OverflowException
```

```
   {
      if (Full())
         throw new OverflowException (size)

      stack[top++] = value;
   }

   public boolean Full()
   {
      return (top >= size);
   }

   private int size;
   private int top;
   private int[] stack;
};
```

There are three new things going on here:

1. There is a new class `OverflowException`, which will be used to pass around information about the error that occurred. This class inherits from `Exception`. You don't need to worry about this (it is explained later), but all classes used to pass exception information should extend `Exception` in this way.

2. The `Push` function has been declared to "throw `OverflowException`." This tells clients of `Push` that they need to expect the possibility of the error. The name of the class `OverflowException` is synonymous with the error.

3. When `Push` discovers an error, it creates a new instance of `OverflowException` and `throws` it. The `OverflowException` instance contains the information that will be needed when we find someone to take care of the problem, and the `throw` statement stops what we are doing and begins the search for someone to handle the error.

The thing that makes exceptions so pervasive in Java is the following rule:

All functions must either handle or pass on any exceptions they might receive.

This seems like a good enough rule, but what it means is that the following is now illegal:

```
void MyFunction()
{
   Stack s = new Stack(10);
   s.Push(1);
}
```

because it is ignoring the possibility that the call to `Push` might result in an exception. There are two things we can do about it:

1. Pass it on. If we change the function to say

   ```
   void MyFunction() throws OverflowException
      ...
   ```

then the compiler will be happy, since we have declared that we will handle any `OverflowExeceptions` that occur by passing them on to whoever called us. Unfortunately this really hasn't solved the problem, since now someone else has to deal with it. The other option is

2. Handle the exception. If we change the function to say

```
void MyFunction()
{
   Stack s = new Stack(10);

   try
      {
      s.Push(1);
      System.out.println("Push completed successfully.");
      }
   catch (OverflowException e)
      {
      System.out.println("Push failed, max size was " + e.MaxSize());
      }
   finally    // the finally block is optional
      {
      System.out.println("This gets printed in either case");
      }
}
```

then the exception, if it occurs, will be handled, and there is no need to pass it on. The `try` construct can be used to group a bunch of statements that might throw exceptions. It is followed by (potentially many) `catch` blocks which will be invoked if an exception of the specified type is thrown, and an optional `finally` block which will always be executed after either the `try` block succeeds or after the exception is processed. This is often used for cleanup operations.

7 Actually Writing Programs

At this point we have discussed most all of the features you will need to write programs in Java. Now let's put some of these things together and actually write and compile a complete program. The following is a complete implementation of the standard "hello world" program:

```
// This is the file HelloWorld.java

public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello world.");
   }
}
```

There are a few new things here:

- The filename for this file must be `HelloWorld.java` (including the capitalization if your system is case-sensitive). In Java there is only one kind of source code file, with the extension `.java`, in contrast to separate `.cpp` and `.h` files in C or C++.

  This does not mean that Java does not do separate compilation. The standard mode of operation for Java development is to keep all your `.java` files in a single directory (or spread them out and use compiler arguments to tell the compiler where you've put them). `.java` files are compiled into `.class` files (this is a one-to-many relationship, since a single `.java` file can define multiple classes, but a `.class` file contains a single class. `.class` files are "executable" bytecodes, but they also contain enough information for the compiler to connect other classes to the methods they export. In a sense then, the `.class` file takes the role of the `.h` file, because it allows other classes to use the provided interface without seeing the original source. What the `.class` file doesn't provide is user-readable documentation as to what the class does. This you have to do separately.

- The class `HelloWorld` is a `public` class. Each `.java` file may contain many classes, but it may contain at most one `public` class, whose name must exactly match the name of the file. Public classes are declared entry points into files.

- The main entry point to the program is the function member

  ```
  public static void main(String[] args)
  ```

  in the public class. Static members are described later on, but are not really important for the moment. This is equivilent to `void main()` in a C or C++ program: it is the place where the compiler knows to start running your program.

To compile and run this program from the command line, we need to issue two commands:

```
% javac HelloWorld.java
% java HelloWorld
```

Case is generally important here, as is the absence of a `.java` extension on the second line. What these commands say is

1. Compile all the classes in `HelloWorld.java` into `.class` files, and

2. Find the file `HelloWorld.class`, look for the public class `HelloWorld` in it, find the public static function `main` in that class, and call it.

8 Advanced Concepts in Java: Dangerous but Occasionally Useful

There are a few Java features, namely (single) inheritance and static members, which are easily abused, but can dramatically simplify an implementation if used properly. I describe the basic idea behind these "dangerous but useful" features here, in case you run across them. Feel free to skip this section – it's long, complex, and you can understand 99% of the code for this class without reading this section.

Up to this point, there really hasn't been any fundamental difference between programming in C and in Java. In fact, most experienced C programmers organize their functions into modules that relate to a single data structure (a "class"), and often even use a naming convention which mimics Java, for example, naming routines `StackFull()` and `StackPush()`. However, the features I'm about to describe *do* require a paradigm shift – there is no simple translation from them into a normal C program. The benefit will be that, in some circumstances, you will be able to write generic code that works with multiple kinds of objects.

Nevertheless, I would advise a beginning Java programmer against trying to use these features, because you will almost certainly misuse them. It's possible (even easy!) to write completely inscrutable code using inheritance and/or templates. Although you might find it amusing to write code that is impossible for your graders to understand, I assure you they won't find it amusing at all, and will return the favor when they assign grades. In industry, a high premium is placed on keeping code simple and readable. It's easy to write new code, but the real cost comes when you try to keep it working, even as you add new features to it.

8.1 Inheritance

Inheritance captures the idea that certain classes of objects are related to each other in useful ways. For example, lists and sorted lists have quite similar behavior – they both allow the user to insert, delete, and find elements that are on the list. There are two benefits to using inheritance:

1. You can write generic code that doesn't care exactly which kind of object it is manipulating. For example, inheritance is widely used in windowing systems. Everything on the screen (windows, scroll bars, titles, icons) is its own object, but they all share a set of member functions in common, such as a routine `Repaint` to redraw the object onto the screen. This way, the code to repaint the entire screen can simply call the `Repaint` function on every object on the screen. The code that calls `Repaint` doesn't need to know which kinds of objects are on the screen, as long as each implements `Repaint`.

2. You can share pieces of an implementation between two objects. For example, if you were to implement both lists and sorted lists in C, you'd probably find yourself repeating code in both places – in fact, you might be really tempted to only implement sorted lists, so that you only had to debug one version. Inheritance provides a way to re-use code between nearly similar classes. For example, given an implementation of a list class, in Java you can implement sorted lists by replacing the insert member function – the other functions, delete, isFull, print, all remain the same.

It is worth pointing out that inheritance is basically the same in Java and C++, except that java does not allow multiple inheritance, and the syntax for specifying the inherited-from class uses the Java keyword `extends` instead of the C++ `: public`. There is also no private or protected inheritance in Java (which is a *very* good thing, because private and protected inheritance in C++ is just wrong).

8.1.1 Shared Behavior

Let me use our Stack example to illustrate the first of these. Our Stack implementation above could have been implemented with linked lists, instead of an array. Any code using a Stack shouldn't care which implementation is being used, except that the linked list implementation can't overflow. (In fact, we could also change the array implementation to handle overflow by automatically resizing the array as items are pushed on the stack.)

To allow the two implementations to coexist, we first define an *abstract* Stack, containing just the public member functions, but no data.

```
class Stack
{
   public void Push(int value)
   {
      // no implementation
   }
   public boolean Full()
   {
      // no implementation
      return true;
   }
}
```

The Stack definition is called a *base class* or sometimes a *superclass*. We can then define two different *derived classes*, sometimes called *subclasses* which inherit behavior from the base class. (Of course, inheritance is recursive – a derived class can in turn be a base class for yet another derived class, and so on.)

Here's how we could declare the array-based and list-based implementations of Stack. The syntax `: public Stack` signifies that both ArrayStack and ListStack are kinds of Stacks, and share the same behavior as the base class.

```
class ArrayStack extends Stack
{
   public ArrayStack(int sz);   // Constructor:  init variables, allocate space.
   public void Push(int value);
   public boolean Full();

   private int size;        // The maximum capacity of the stack.
   private int top;         // Index of the lowest unused position.
   private int[] stack;     // A pointer to an array that holds the contents.
};

class ListStack extends Stack
{
   public ListStack()
   {
      list new new List;
   }

   public void Push(int value)
   {
      list.Prepend(value);
   }

   public boolean Full()
   {
      return false;       // this stack never overflows!
   }
```

```
   private List list;          // list of items pushed on the stack
}
```

The neat concept here is that I can assign a variable of type `Stack` to point to instances of type `ListStack` or `ArrayStack` and then use them as if they were of the base type.

```
   Stack s1 = new ListStack();
   Stack s2 = new ArrayStack(17);

   if (!stack.Full())
      s1.Push(5);
   if (!s2.Full())
      s2.Push(6);
```

The compiler automatically invokes `ListStack` operations for `s1`, and `ArrayStack` operations for `s2`; this is done by creating a procedure table for each object, where derived objects override the default entries in the table defined by the base class. To the code above, it invokes the operations `Full` and `Push` by indirection through the procedure table, so that the calling code doesn't need to know which kind of object it is.

In this example, since I never create an instance of the abstract class `Stack`, I do not need to *implement* its functions. This might seem a bit strange, but remember that the derived classes are the various implementations of Stack, and Stack serves only to reflect the shared behavior between the different implementations.

8.1.2 Shared Implementation

What about sharing code, the other reason for inheritance? In Java, as in C++, it is possible to use member functions of a base class in its derived class. (You can also share data between a base class and derived classes, but this is a bad idea for reasons I'll discuss later.)

Suppose that I wanted to add a new member function, `NumberPushed()`, to both implementations of `Stack`. The `ArrayStack` class already keeps count of the number of items on the stack, so I could duplicate that code in `ListStack`. Ideally, I'd like to be able to use the same code in both places. With inheritance, we can move the counter into the `Stack` class, and then invoke the base class operations from the derived class to update the counter.

```
class Stack
{
   public void Push(int value)       // modify numPushed
   {
      ++numPushed;
      // no actual push implementation
   }

   public boolean Full()
   {
      // no implementation
   }

   public int NumPushed()            // how many are currently on the stack?
```

```
   {
      return numPushed;
   }

   protected Stack()                // initialize numPushed
   {
      numPushed = 0;
   }

   private int numPushed;
}
```

We can then modify both ArrayStack and ListStack to make use the new behavior of Stack. I'll only list one of them here:

```
class ArrayStack extends Stack
{
   public ArrayStack(int sz)
   {
      super();          // Tell Java how to call the base class constructor
      size = sz;
      stack = new int[size];
   }

   public void Push(int value)
   {
      if (Full())
         HandleAnError();
      stack[NumPushed()] = value;
      super.Push();     // invoke the base class to increment numPushed
   }

   public boolean Full()
   {
      return (NumPushed() >= size);
   }

   private int size;        // The maximum capacity of the stack.
   private int[] stack;     // The array that holds the contents.
};
```

There are a few things to note:

1. The constructor for ArrayStack needs to invoke the constructor for Stack, in order to initialize numPushed. It does that by adding super() to the first line in the constructor:

   ```
   super();
   ```

   This statement is not strictly necessary, because the compiler would automatically call the base class constructor if we hadn't made this statement. What this syntax allows you to do is pass arguments to the base class constructor. If present, this *must* be the first line of the constructor.

2. I introduced a new keyword, `protected`, in the new definition of `Stack`. For a base class, `protected` signifies that those member data and functions are accessible to classes derived (recursively) from this class, but inaccessible to other classes. In other words, protected data is `public` to derived classes, and `private` to everyone else. For example, we need `Stack`'s constructor to be callable by `ArrayStack` and `ListStack`, but we don't want anyone else to create instances of `Stack`. Hence, we make `Stack`'s constructor a protected function.

   Note however that I made `Stack`'s data member `private`, not `protected`. Although there is some debate on this point, as a rule of thumb you should never allow one class to see directly access the data in another, even among classes related by inheritance. Otherwise, if you ever change the implementation of the base class, you will have to examine and change all the implementations of the derived classes, violating modularity.

3. The interface for a derived class automatically includes all functions defined for its base class, without having to explicitly list them in the derived class. Although we didn't define `NumPushed()` in `ArrayStack`, we can still call it for those objects:

```
ArrayStack s = new ArrayStack(17);

int n = s.NumPushed();     // should be initialized to 0
```

4. Conversely, even though we have defined a routine `Stack::Push()`, if we invoke `Push()` on an `ArrayStack` object, we will get `ArrayStack`'s version of `Push`:

```
Stack s = new ArrayStack(17);

if (!s.Full())              // ArrayStack::Full
    s.Push(5);              // ArrayStack::Push
```

5. Member functions in a derived class can explicitly invoke public or protected functions in the (immediate) base class, by using the special variable `super`. Super is a slightly weird concept, because what it refers to is the current object (the same object that `this` refers to), but it refers to it as an object of the base class type.

```
public void Push(int value)
{
   ...
   super.Push();           // invoke base class to increment numPushed
}
```

   Of course, if we just called `Push()` here (without prepending `super.`, the compiler would think we were referring to `ArrayStack`'s `Push()`, and so that would recurse, which is not exactly what we had in mind here.

Whew! Inheritance in Java (as in C++) involves lots and lots of details. But it's real downside is that it tends to spread implementation details across multiple files – if you have a deep inheritance tree, it can take some serious digging to figure out what code actually executes when a member function is invoked.

So the question to ask yourself before using inheritance is: what's your goal? Is it to write your programs with the fewest number of characters possible? If so, inheritance is really useful, but so is changing all of your function and variable names to be one letter long – "a", "b", "c" – and once you run out of lower case ones, start using upper case, then two character variable names: "XX XY XZ Ya ..." (I'm joking here.) Needless to say, it is really easy to write unreadable code using inheritance.

So when is it a good idea to use inheritance and when should it be avoided? My rule of thumb is to only use it for representing *shared behavior* between objects, and to never use it for representing *shared implementation*. With Java (or C++), you can use inheritance for both concepts, but only the first will lead to truly simpler implementations.

To illustrate the difference between shared behavior and shared implementation, suppose you had a whole bunch of different kinds of objects that you needed to put on lists. For example, almost everything in an operating system goes on a list of some sort: buffers, threads, users, terminals, etc.

A very common approach to this problem (particularly among people new to object-oriented programming) is to make every object inherit from a single base class *Object*, which contains the forward and backward pointers for the list. But what if some object needs to go on multiple lists? The whole scheme breaks down, and it's because we tried to use inheritance to share implementation (the code for the forward and backward pointers) instead of to share behavior. A much cleaner (although slightly slower) approach would be to define a list implementation that allocated forward/backward pointers for each object that gets put on a list.

In sum, if two classes share at least some of the same member function signatures – that is, the same behavior, *and* if there's code that only relies on the shared behavior, then there *may* be a benefit to using inheritance.

8.1.3 Interfaces

Java does not allow multiple inheritance, but it does allow a weakened form known as an *interface*. Interfaces are like inheritance, but you don't inherit the code, only the function signatures, consider:

```
interface Printable
{
   public void Print();
}

class MyClass implements Printable
{
   public void Print()
   {
      System.out.println("42");
   }
}
```

In this example, `MyClass` is promising to uphold the interface set out in `Printable`, in this case it is promising to implement a function named `Print`. Unlike inheritance, there is no way for `MyClass` to get a default implementation of `Print` or any other help from `Printable`. What interfaces do allow, though, is the ability to say:

```
   Printable p = new MyClass();
   p.Print();
```

that is that the compiler now knows that a `MyClass` may be used anywhere that something `Printable` is expected, because `MyClass` is obligated to support the same interface.

A class can implement multiple interfaces by saying

```
class MyClass implements Interface1, Interface2
```

8.2 Static Members

Static members are a feature that Java borrows from C++. If you know how they work in C++ you won't be particularly confused by them in Java. The big difference is that, because all code in Java must reside within a class, you end up having to use static members in a few situations, where in C or C++ you would have used a global function or variable.

The one example of a static member that we have seen so far is the main entry point to a program:

```
class MyProgram
{
   static void main (String[] args)
   {
      ...
   }
}
```

What it means to be a static member is that there is only *one* copy of the member, and it is shared by *all* instances of the class. There are two reasons why this can be crucially useful:

- We want a variable with exactly one copy (i.e. a global), or
- We want to be able to use a function or variable *without* having an instance of the class that contains it.

It is the second point that really forces the issue of static members. Assume we want a global variable of some sort. How could we accomplish that with what we know? The first thing to try would be to make it a data member of some class. We can make it `public` so that everyone can access it:

```
class Globals
{
   public int ourInteger;
}
```

But now, the only way we can access `ourInteger` is if we have an object of type `Globals`. Worse than that, everyone who wants to access `ourInteger` had better be using the same object, so were back to exactly the same problem: we need a place to keep track of the `Globals` object where everyone can access it. The solution is to do:

```
class Globals
{
   public static int ourInteger = 0;
}
```

Since `ourInteger` is `public`, everyone can access it, and since it is `static`, they don't need an instance of `Globals` to do it. Any part of the program can refer to

```
Globals.ourInteger
```

to access the variable. The important thing to notice is the use of a type name instead of an object in front of the dot (.). This is legal because `ourInteger` is a static member. It is also legal to access static members as if they were not static (i.e. through an instance with the dot operator, or directly inside a member function of the same class).

9 Style Guidelines

Even if you follow the approach I've outlined above, it is still as easy to write unreadable and undebuggable code in C++ as it is in C, and perhaps easier, given the more powerful features the language provides. We suggest you adhere to the following guidelines (and tell us if you catch us breaking them):

1. Words in a name are separated SmallTalk-style (i.e., capital letters at the start of each new word). All class names and member function names begin with a capital letter, except for member functions of the form `getSomething()` and `setSomething()`, where `Something` is a data element of the class (i.e., accessor functions). Note that you would want to provide such functions only when the data should be visible to the outside world, but you want to force all accesses to go through one function. This is often a good idea, since you might at some later time decide to compute the data instead of storing it, for example.

2. Minimize the use of global variables (i.e. public static data members). If you find yourself using a lot of them, try and group some together in a class in a natural way or pass them as arguments to the functions that need them if you can.

3. Minimize the use of global functions (i.e. static member functions). If you write a function that operates on some object, consider making it a member function of that object.

4. Write a module test for every module in your program. Many programmers have the notion that testing code means running the entire program on some sample input; if it doesn't crash, that means it's working, right? Wrong. You have no way of knowing how much code was exercised for the test. Let me urge you to be methodical about testing. Before you put a new module into a bigger system, make sure the module works as advertised by testing it standalone. If you do this for every module, then when you put the modules together, instead of *hoping* that everything will work, you will *know* it will work.

   Perhaps more importantly, module tests provide an opportunity to find as many bugs as possible in a localized context. Which is easier: finding a bug in a 100 line program, or in a 10000 line program?

   A common way to implement module test routines is to give the main class in every file its own `public static void main()` function which tests that class, and to make that class a `public` class, so that you can test that file by running the class. In normal operation these added functions will never be called, so they can be left in.

10 Epilogue

Java, at least superficially, is a reduced flavor of C++, with certain features removed. But you're probably thinking I must be leaving something out – if someone put the feature in the language, there must be a good reason, right? I believe that every programmer should strive to write code whose behavior would be immediately obvious to a reader; if you find yourself writing code that would require someone reading the code to thumb through a manual in order to understand it, you are almost certainly being way too subtle. There's probably a much simpler and more obvious way to accomplish the same end. Maybe the code will be a little longer that way, but in the real world, it's whether the code works and how simple it is for someone else to modify, that matters a whole lot more than how many characters you had to type.

A final thought to remember:

> "There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies."
>
> C. A. R. Hoare, "The Emperor's Old Clothes", CACM Feb. 1981