

Note: New Version Available...

A newer edition of *Introduction to Programming Using Java* is available at <http://math.hws.edu/javanotes/> (The new edition requires Java 1.3 or higher, while this version uses Java 1.1.)

Introduction to Programming Using Java Version 3.1, February 2001

(Repackaged with minor corrections June 2004)

Author: [David J. Eck](#) (eck@hws.edu)

WELCOME TO *Introduction to Programming Using Java*, an on-line textbook on introductory programming, which uses Java as the language of instruction. This text has more than enough material for a one-semester course, and it is also suitable for individuals who want to learn programming on their own. This is the third edition of the text. (Version 3.1 is a minor upgrade of Version 3.0, which was released in May, 2000. Version 3.1 was released in February 2001 and incorporated a few changes and corrections. A final packaging of Version 3.1 was released in June 2004, incorporating corrections made since the release of Version 3.1. This version is released under the [Open Publication License](#).) The third edition covers more material and has more examples than the second edition. It also adds end-of-chapter quizzes and solved programming exercises. Previous editions have been used in a course, *Computer Science 124: Introductory Programming*, at [Hobart and William Smith Colleges](#). (The title of the previous editions included a reference to this course.) This textbook covers Java 1.1. Most of the applets that are contained in the text require Java 1.1 or higher.

Links for **downloading** copies of this text can be found at the bottom of this page. To learn more about this on-line text, please read its [preface](#).

Search this Text:

Although this book does not have a conventional index, you can search it for terms that interest you. Note that this searches the version of the text book at its main site, at math.hws.edu.

Search *Introduction to Programming Using Java* for pages...

Containing all of these words:

Short Table of Contents:

- [Full Table of Contents](#)
- [Preface](#)
- [Preface to the Second Edition](#)
- Chapter 1: [Overview: The Mental Landscape](#)
- Chapter 2: [Programming in the Small I: Names and Things](#)
- Chapter 3: [Programming in the Small II: Control](#)
- Chapter 4: [Programming in the Large I: Subroutines](#)
- Chapter 5: [Programming in the Large II: Objects and Classes](#)
- Chapter 6: [Applets, HTML, and GUI's](#)
- Chapter 7: [Advanced GUI Programming](#)
- Chapter 8: [Arrays](#)
- Chapter 9: [Correctness and Robustness](#)
- Chapter 10: [Advanced Input/Output](#)
- Chapter 11: [Linked Data Structures and Recursion](#)
- Appendix 1: [From Java to C++](#)
- Appendix 2: [Some Notes on Java Programming Environments](#)
- Appendix 3: [Source code for all examples in the text](#)
- [News and Errata](#)

This is a free textbook. As of Version 3.1, it is published under the terms of the [Open Publication License](#), Version 1.0. The latest edition is always available, at no charge, for downloading and for on-line use at the Web address <http://math.hws.edu/javanotes/>. This edition, the third, is also permanently archived at the address <http://math.hws.edu/eck/cs124/javanotes3/>.

Downloading Links

Use one of the following direct links to download a compressed archive of this entire textbook. You can use this material on your own computer. You can also re-post it on any Web server: See the [preface](#) for more detailed information about downloading. In the preface, you will also find a link to a PDF file that can be used for printing the textbook.

- <http://math.hws.edu/eck/cs124/downloads/javanotes3.zip> (1.6 MB), for Windows. This can be used directly in Windows XP. On any version of Windows, you can open it using *WinZip* from www.winzip.com or *Aladdin Expander for Windows* from www.aladdinsys.com. (Note: The text files in this archive are in Windows/DOS format.)
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.tar.bz2> (1.0 MB), for Linux/MaxOS. In Linux, you should be able to expand this using the command "bunzip2 javanotes3.tar.bz2" followed by "tar xf javanotes3.tar.bz2"; this will also work in UNIX, if you have the bunzip2 program. For MacOS, your browser will probably expand it automatically when you download it, or it can be opened with *Stuffit Expander* from www.aladdinsys.com.

The following archives in older formats do not include the final corrections of June 2004:

- <http://math.hws.edu/eck/cs124/downloads/javanotes3.sit.hqx> (2.1 MB), for Macintosh.
- <http://math.hws.edu/eck/cs124/downloads/javanotes3.tar.Z> (1.8 MB), for Linux/UNIX.

[David Eck](#) (eck@hws.edu)

Version 3.0, May 2000

Version 3.1, with minor changes, February 2001

Final packaging, with further corrections, June 2004

Appendix 1, Section 2

Pointers and Arrays in C++

A POINTER IS JUST THE ADDRESS OF SOME location in memory. In Java, pointers play an important role behind the scenes in the form of references to objects. A Java variable of object type stores a reference to an object, which is just a pointer giving the address of that object in memory. When you use an object variable in a program, the computer automatically follows the pointer stored in that variable in order to find the actual object in memory. Since all this happens automatically, behind the scenes, you really don't have to worry much about the fact that objects are referenced through pointers.

In C++, everything about pointers becomes explicit. Some variables can store pointers, but only variables that are of special **pointer types**. When you use a pointer variable, the computer does not follow the pointer automatically -- if you want the computer to follow the pointer, you have to use a special notation to tell it to do so. You can create a new object for a pointer variable to point to, using a "new" operator that is similar to the new operator in Java. But where Java will dispose of such objects automatically when you are done with them, in C++ you are responsible for keeping track of objects created with the new operator and disposing of them when they are no longer needed. Another operator, named `delete` is provided for doing this. All-in-all, working with explicit pointers is fairly difficult and error-prone, and it is generally considered to be one of the advanced aspects of programming. Unfortunately, in C++ there are some pretty basic things that you can't do without using pointers (not unless you use other advanced features of the language or use a class that someone else has written for you).

Suppose that `T` is the name of a type in C++. Then there is a pointer type named `T*` which represents variables that can hold pointers to objects of type `T`. C++ allows pointers to values of any type, not just objects. For example, there is a type `int*` that represents pointers to integers. A variable of type `int*` holds the address of a location in memory; an integer value is stored at that location. Suppose that `T` is some type and that `ptr` is a variable of type `T*`. When you use `ptr` in a program, you are referring to the actual pointer, that is, the memory address that is contained in the variable `ptr`. To refer to the value that is stored at that address, you would use the notation `*ptr`. C++ uses the predefined constant `NULL` -- all upper case -- as the value of a pointer that doesn't point to anything. It is an error to refer to `*ptr` if the value of `ptr` is `NULL`.

The notation for defining a variable, `ptr`, of type `T*` is:

```
T *ptr;
```

Thus, to declare `iptr` as a variable of type pointer-to-int, you would say:

```
int *iptr;
```

The notation "`T *ptr`" is meant to suggest that the value of `*ptr` is of type `T`. But note that the variable that is being declared is `ptr`, not `*ptr`.

The type of a subroutine parameter can be a pointer type. For example, here is a subroutine that will swap the integer values stored in two memory locations. Its parameters are of type pointer-to-int:

```
void swap(int *x, int *y) {
    int temp;
    temp = *x; // copy the value pointed to by x into temp
    *x = *y;   // copy the value pointed to by y into the
               //      location pointed to by x
}
```

```

        *y = temp; // copy the value in temp into the location
                  // pointed to by y
    }

```

At the end of this subroutine, *x* and *y* still point to the same memory locations, but the values in those locations have been swapped.

This becomes more interesting once you know about the **unary address operator**, which is written as **&**. Suppose that *x* is any variable. Then **&x** is a notation that stands for the memory address of *x*. That is, **&x** is a pointer that points to *x*. Consider the following C++ program segment:

```

int x = 7; // x is an integer variable with initial value 7
int *iptr; // iptr is a variable that can hold a pointer to an int
iptr = &x; // iptr now holds a pointer to x, so that now
           // x and *iptr refer to the very same memory location
*iptr += 3; // adds 3 to the value stored in location *iptr
cout << x; // print out the value of x, which is 10

```

Because **iptr* and *x* refer to the same memory location, adding 3 to **iptr* is really the same as adding 3 to the value of *x*. So the output value of *x* in this example is 7 + 3.

Suppose that we combine the address operator with the `swap()` routine defined above. Consider this example:

```

int a,b;
a = 17;
b = 42;
swap(&a, &b);

```

The subroutine call is legal because **&a** and **&b** are pointers to integers. That is, they are of type `int*`, and that is the same type as the formal parameters of the subroutine. The end result of this example is that the original values of *a* and *b* are swapped. So the final value of *a* is 42, and the final value of *b* is 17. Believe it or not, using pointers and addresses like this is a very common way to write subroutines that can alter the values of variables. (In the C programming language, it's the only way. C++ provides a somewhat more slightly alternative, which is discussed below. Note that in Java, it is impossible to write a subroutine that performs the same task as `swap()`, since there is no such thing as a pointer-to-int in Java. Of course, you can do similar things with objects and references to objects, in Java as well as in C++.)

Pointers are probably used most often with classes, but I will put off that discussion to the next section.

References and Passing Parameters by Reference

In C++, it is possible to write a subroutine that can change the value stored in a variable, when that variable is passed to the subroutine as an actual parameter. Here is a variation of the `swap` routine that does this:

```

void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}

```

The type `int&` which is specified in this subroutine for the formal parameters *x* and *y* is called a **reference type**. If *T* is any type in C++, then you can form a reference type, *T&*. You can actually declare variables using reference types, but I have never seen any practical use for them except for

formal parameters and occasionally as the return type of a function. When a formal parameter is declared using a reference type, the parameter is said to be **passed by reference**. The effect of passing a parameter by reference is to allow the subroutine to change the value of the actual parameter. Thus, if the subroutine `swap()` is defined as above, using pass-by-reference, then the subroutine call statement `swap(a,b)` will interchange the values stored in the variables `a` and `b`. On the other hand, consider the subroutine

```
void failsToSwap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

In this case, a subroutine call statement `failsToSwap(a,b)` will have no effect at all on the values stored in `a` and `b`. The values of the actual parameters `a` and `b` are **copied** into the formal parameters `x` and `y` when the subroutine is called. The subroutine interchanges the values of `x` and `y`, but this has no effect on what's stored in `a` and `b`. In the `failsToSwap()` subroutine, the parameters are said to be **passed by value**.

When a parameter is passed by reference, then what's passed to the subroutine is really a reference, or pointer, back to the actual parameter. The effect is similar to what was done in the first version of `swap()`, which passed pointers explicitly. But in pass-by-reference, the pointers are handled implicitly.

In Java, all parameters are passed by value, and so there is no way to write a subroutine that will swap the values stored in two integer variables. More generally, in Java, no subroutine can ever change the value stored in a variable that is passed to the subroutine as an actual parameter. This is kind of a shock to programmers who are used to other languages. (Be careful: when an **object** variable is passed as a parameter in Java, the value stored in the variable is a reference to an object. The values of instance variables in that object **can** be changed by the subroutine, even though the reference stored in the variable cannot be changed. When the subroutine ends, the variable will still refer to the same location in memory, but the values in the object stored at that location might have changed. In a sense, then, **objects** are passed by reference in Java, even though **variables** are not.)

Arrays

Arrays in C++ are superficially similar to arrays in Java. An array is just an indexed list of values, where the index goes from 0 up to some maximum. But in C++, an array variable is considered to be nothing more than a pointer to the first element in the array. In fact, array variables and pointer variables can be used interchangeably. For example, if `list` is an array variable, then you can use either `list[0]` or `*list` to refer to the first item in the array. And you can use an assignment statement to copy the value of an array variable into a pointer variable.

Arrays can be created in C++ using the `new` operator, just as in Java. For example, to create an array of 100 ints, you could say either

```
int list[] = new int[100];
```

or

```
int *list = new int[100];
```

(Note, however, that the notation for declaring an array variable is `"int list[]"`, not `"int[] list"`. In Java, either notation would be allowed, but I prefer the latter.)

It is also possible to create arrays without using the new operator:

```
int list[100]; // declares an array of 100 ints
```

The main difference is that arrays created with the new operator should eventually be disposed of with the delete operator, in order to avoid memory leaks. Arrays created with the second notation do not have to be -- and in fact cannot be -- deleted. Most beginning C++ programmers would probably use arrays for quite a while before they even learn about the new operator.

Although an array in C++ has some fixed length, the computer does not keep track of that length in any way, and there is nothing to stop you from trying to refer to array locations that don't even exist. Thus, arrays -- like pointers in general -- are a common source of bugs in C++ programs.

Strings as Arrays of Characters

Strings in C++ are considered to be arrays of characters. A string, therefore, has type `char[]`. Since array types and pointer types are considered to be equivalent, we could also consider a string to be of type `char*`, that is pointer to `char`.

Since a string in C++ is just as array of characters, strings can be manipulated using array notation. For example, if `str` is a string variable, then `str[i]` refers to the *i*-th character in the string. You can change one of the characters in a string by using an assignment statement to the appropriate array element. For example, if `str` is the string "bit", then the assignment statement `str[1]='e'` changes the value of `str` to "bet". (In Java, the *i*-th character of a string `str` is referred to as `str.charAt(i)`, and there is no way to change the characters in a string.)

In C++, there is no `String` class, and strings are not objects. So there are no class methods for operating on strings. Instead, a large number of subroutines for working with strings are available through a standard header file, `string.h`.

Since strings are arrays and since the computer does not keep track of the lengths of arrays, the computer does not know how long a string is. However, by convention, strings in C++ are assumed to end with a null character (the character with ASCII code zero). The null character is not really considered to be part of the string. It's there just as an end-of-string marker. This convention is used when a string variable is initialized. For example, if you say

```
char *greeting = "Hello World";
```

the computer creates an array of 12 characters, not just 11, to hold the string. The characters in `greeting[0]` through `greeting[10]` are the characters 'H' through 'd' in the string that you've specified. The computer adds a null character in the last location, `greeting[11]`, to mark the end of the string.

[[Next Section](#) | [Previous Section](#) | [Appendix Index](#) | [Main Index](#)]