

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, or **long**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

`(target-type) value`

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger

than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a;
byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
    public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);

        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);

        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

323 - 256

Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

257 % 256 (The Range of byte)

The result of the intermediate term $a * b$ easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte or short operand to int when evaluating an expression. This means that the subexpression $a * b$ is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, $50 * 40$, is legal even though a and b are both specified as type byte.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```


The code is attempting to store $50 * 2$, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

In addition to the elevation of **bytes** and **shorts** to **int**, Java defines several *type promotion rules* that apply to expressions. They are as follows. First, all **byte** and **short** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
    public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}
```

Let's look closely at the type promotions that occur in this line from the program:


```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, $f * b$, b is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression i / c , c is promoted to **int**, and the result is of type **int**. Then, in $d * s$, the value of s is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

Note

If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-name[ ];
```

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

```
int month_days[ ];
```

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers,

you must allocate one using **new** and assign it to `month_days`. **new** is a special operator that allocates memory.

You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

```
array-var = new type[size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to `month_days`.

```
month_days = new int[12];
```

After this statement executes, `month_days` will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don't worry. It will be described at length later in this book.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of `month_days`.

```
month_days[1] = 28;
```

The next line displays the value stored at index 3.

```
System.out.println(month_days[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of days in each month.

```
// Demonstrate a one-dimensional array.
class Array {
```



```
public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is `month_days[3]` or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int month_days[] = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use `new`. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
    public static void main(String args[]) {
```



```

int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                    30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}

```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. (In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks.) For example, the run-time system will check the value of each index into `month_days` to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```

// Average an array of values.
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;

        for(i=0; i<5; i++)
            result = result + nums[i];

        System.out.println("Average is " + result / 5);
    }
}

```

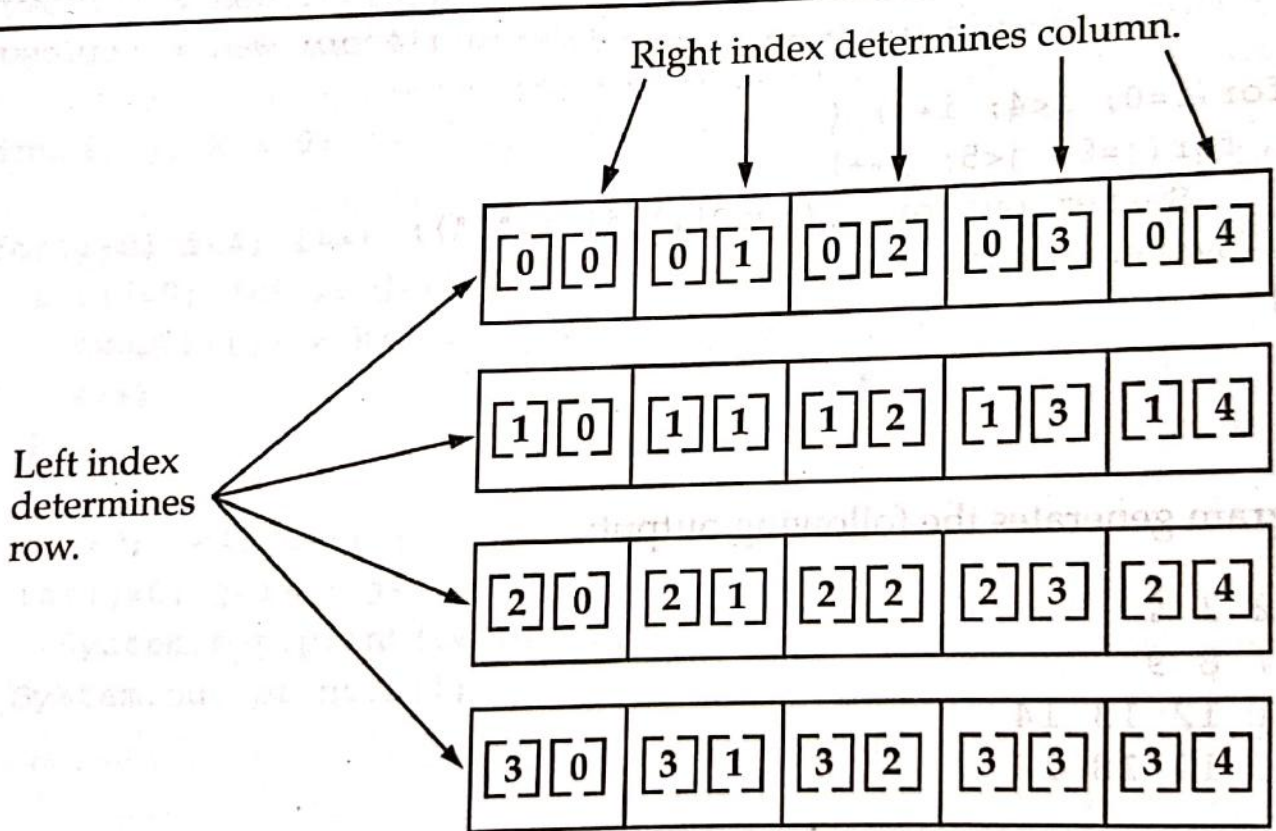
Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see,

there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array of arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1.



```
Given: int twoD [] [] = new int [4] [5] ;
```

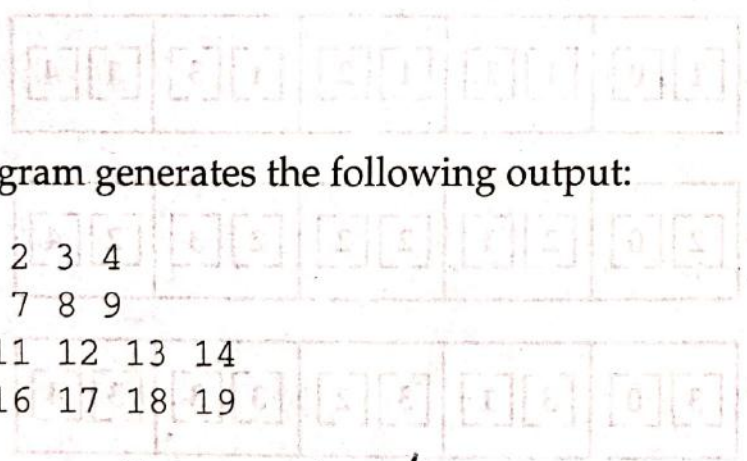
Figure 3-1. A conceptual view of a 4 by 5, two-dimensional array

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;

        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }

        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```



This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```