Conversion and Casting

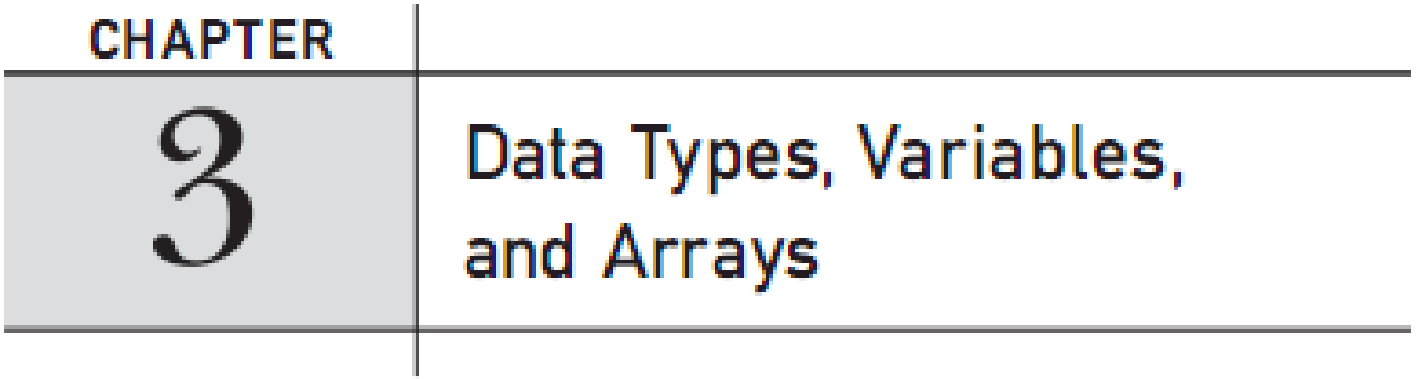
Automatic type conversion
Compatible and Incompatible
Type Casting
Widening & Narrowing

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

It is not enough just to write code that works. It is as important-perhaps more important to write code well; not merely code that works, but code that is legible, maintainable, reusable, fast, and efficient.

CHAPTER -3

Data Types, Variables, and Arrays



Conversion and Casting

- What is Automatic type conversion?
- Compatible and Incompatible Type
- Type Casting
- Widening and
- Narrowing

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

Type Conversion and Casting

- 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.
- There is no automatic 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***.

Type Conversion: Compatible Type

- 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, there are no automatic conversions from the numeric types to 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, long, or char.

Conversion: Widening

```
class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
10.0
```

Type Casting: Incompatible Types

- 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

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.

Casting : Narrowing

```
class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Casting

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.

floating-point to integer

- 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.

Type Cast : Example

```
// 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
```

Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}}
```

Output:

```
20
```

Automatic Type Promotion in Expressions

- Java automatically promotes each ***byte, short, or char operand to int*** when evaluating an expression.

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!
```

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.

Automatic Type Promotion : Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
130
-126
```

Type Promotion : Rules

- Java defines several type promotion rules that apply to expressions. They are as follows:

  - ✓ First, all byte, short, and char 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 are double, the result is double.

Type Promotion : Rules Example

```
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

What is an Array?
One-dimensional
Array declaration in java
Multi-dimensional
Alternative declaration

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

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

CHAPTER -3 : Java Arrays

- **What is an Array?**
- **One-dimensional**
- **Array declaration in java**
- **Multi-dimensional**
- **Alternative declaration**

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

Arrays: One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables

The general form of a one-dimensional array declaration is

type var-name[ ];

Here, type declares the element type (also called the base type) of the array.

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. To link *month_days* with an actual, physical array of integers, you must allocate one using ***new*** and assign it to month_days. **Lets see how?**

Arrays: *The new* operator

***new* is a special operator that allocates memory.**

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 (for numeric types), false (for boolean), or null (for reference types).

Arrays: *example*

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.

Arrays: *Array declaration in java*

Obtaining an array is a two-step process.

- ✓ First, you must declare a variable of the desired array type.

  type var-name[ ]; // int month_days[];

- ✓ Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable.

  array-var = new type [size]; *//month_days = new int[12];*

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: *Accessing Array Elements*

Thus, in Java all arrays are dynamically allocated. 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]);
```

Arrays: *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.")
  }
}
```

Arrays: initialized when declared

Arrays can be initialized when they are declared.

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:

```
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.");
  }
}
```

Arrays: 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.

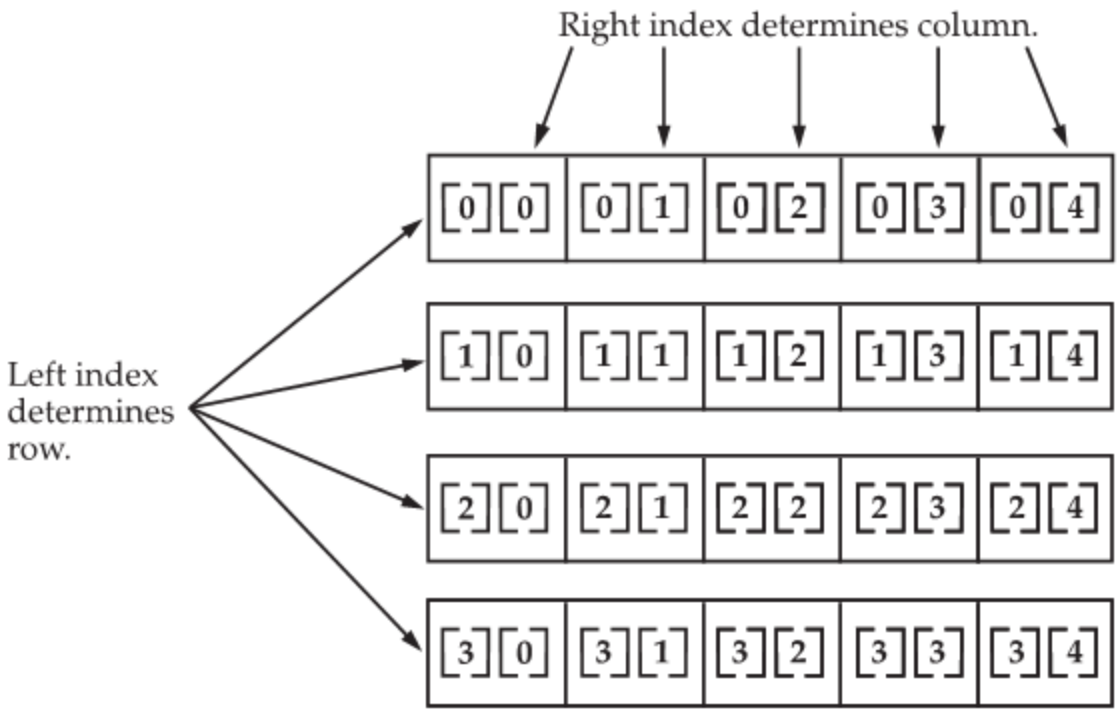
For example, the following declares a two-dimensional array variable called twoD:

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

This allocates a 4 by 5 array and assigns it to twoD.

Arrays: Multidimensional Arrays

Internally, this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in Figure



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

A conceptual view of a 4 by 5, two-dimensional array

Arrays: Demonstrate a 2D 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
```

Arrays: allocate memory for a multidimensional array

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.

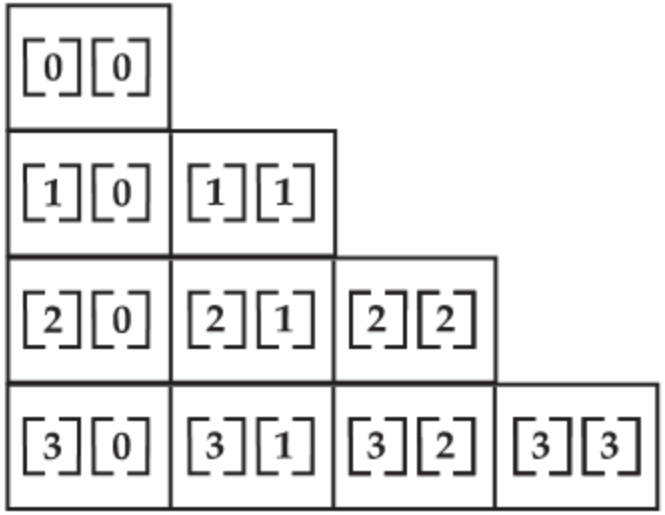
```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Arrays: Uneven second dimension

Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.

For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

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



Arrays: initialize multidimensional arrays

- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.
- Also notice that you can use **expressions as well as literal** values inside of array initializers.

Arrays: Initialize a two-dimensional array.

```
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
      };
      int i, j;

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

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

Arrays: Demonstrate a three-dimensional array.

```
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Arrays: *Alternative Array Declaration*

There is a second form that may be used to declare an array:

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

Arrays: *Alternative Array Declaration*

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays

creates three array variables of type int.

It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used in this book.

String & Pointer

Java String: Object of String Class
Java Pointer : No Direct Pointer

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

NOTE If you are familiar with C/C++, be careful. String in Java work differently than they do in those languages also java have no explicit pointer

CHAPTER -3 : String & Pointer

- **Few words on String**
- **Note to C/C++ Programmer about Pointer in java**

NOTE *If you are familiar with C/C++, be careful. String in Java work differently than they do in those languages also java have no explicit pointer*

A Few Words About Strings

- ✓ **It is just that Java's string type, called String, is not a primitive type.**
- ✓ **Nor is it simply an array of characters.**
- ✓ **Rather, String defines an object**

- The **String** type is used to declare string variables.
- You can also declare arrays of strings.
- A quoted string constant can be assigned to a String variable.

A Few Words About Strings

A variable of type String can be assigned to another variable of type String. You can use an object of type String as an argument to println().

For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```

Here, str is an object of type String. It is assigned the string "this is a test". This string is displayed by the println() statement.

As you will see later, String objects have many special features and attributes that make them quite powerful and easy to use.

A Note to C/C++ Programmers About Pointers

Java does not support or allow pointers.

(Or more properly, Java does not support pointers that can be accessed and/or modified by the programmer.)

Java cannot allow pointers, because doing so would allow Java programs to **breach the firewall between the Java execution environment and the host computer**.

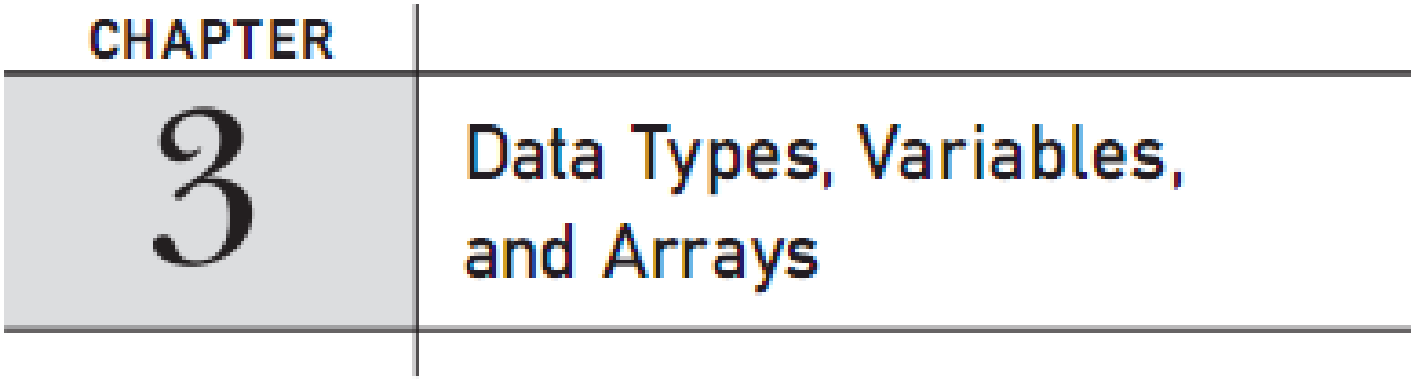
(Remember, a pointer can be given any address in memory—even addresses that might be outside the Java run-time system.)

A Note to C/C++ Programmers About Pointers

- Since C/C++ make extensive use of pointers, you might be thinking that their **loss is a significant disadvantage** to Java.
- However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, **you will never need to use a pointer, nor would there be any benefit in using one.**

Summery: CHAPTER -3

Data Types, Variables, and Arrays



Summery

- **Data Type:**
  - **Primitive: 8 Type**
    - **byte, short, int, long, float, double, char boolean**
  - **Non-Primitive: Derived & User Define**
    - **Derived: Array, String etc**
    - **User Define: Class**
- **Variable: 3 Type**
  - **Instance, static and local**
- **Array : different from C/C++**
  - **Two step Process**
    - **Declare a ref/var and allocation through new**

Students Responses : Class Assessment

Questions ???

&&

Answer