Chapter 2.1

Data Types, Variables and Arrays

Java is strongly typed

- ❖ It is important to state at the outset that Java is a strongly typed language.
- ❖ Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether it is explicit or via parameter passing in method calls, are checked for type compatibility. There is no automatic of conflicting types as in some languages. The Java compiler checks all expressions and parameters.
- ❖ To ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Simple or primate types

- ❖ Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float, double, and Boolean. These can be put in four groups:
  →**Integers:** This group includes byte, short, int and long which are for signed numbers.
  →**Floating Point Numbers:** This group includes float and double, which represent numbers with fractional precision.
  →**Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
  →**Boolean:** This group includes Boolean, which is a special type for representing true/false values.

Integer types

- The width and ranges of these integer types vary widely.

Floating Point types

- Floating Point numbers, also known as *real numbers*, are used when evaluating expressions that require fractional precision.
- For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.

Characters

- In Java, the data type used to store characters is char. However, C/C++ programmers beware char in Java is not the same as char in C or C++. In C/C++, char is an integer type that is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of char is 0 to 65,536. There is no negative.

```
class CharDemo {
public static void main(String[] args) {
            char ch1,ch2;
            ch1=88; //code for X
            ch2='Y';
            System.out.print("ch1 and ch2:");
            System.out.println(ch1+" "+ch2);
    }
}
```

Boolean

- Java has a simple type, called boolean, for logical values. It can have only one of two possible values, true or false.
- This is the type returned by all relational operators, such as a<b.
- Boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

  Example Boolean:

```
class BoolTest {
public static void main(String[] args) {
        boolean b;
        b=false;
        System.out.println("b is" +b);
        b=true;
        System.out.println("b is" +b);
        // a Boolean value can control the if statement
        if(b) System.out.println("This is executed");
        b=false;
        if(b) System.out.println("This is not executed");
        // outcome of a relational operator is a Boolean
        System.out.println("10 > 9 is" +(10 > 9));
        }
}
```

A close look at literals

Integer Literals

- Integers are probably the most commonly used type in the typical program. Any whole number value in an integer literal. Example are 1, 2, 3 and 42.
- These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals, octal and hexadecimal.
- Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range.
- A more common base for numbers used by programmers is hexadecimal.

Floating Point Literals

- Floating Point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation.
- Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.
- Floating point literals in Java default to double precision.
- To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d. Example: 1.0F, 2D.

Boolean Literals

- Boolean literals are simple.
- There are only two logical values that a boolean value can have, true and false.
- The values of true and false do not convert into any numerical representation. The true literal in Java does not equal 1, nor does the false literal equal 0.
- In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

Character Literals

- Characters in Java are indices into the Unicode Character Set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators.
- A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as ‘a’ , ‘z’ and ‘@’. For characters that are impossible to enter directly, there are several escape sequences, which allows us to enter the character we need, such as ‘ \n ‘ for new line character.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\’	Single quote
\”	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Tab
\b	Backspace

String Literals

- String literals in Java are specified like they are in most other languages – by enclosing a sequence of characters between a pair of double quotes.
- Examples of string literals are:
  → “Hello World”
  → “two\nlines”
  → “\”This is in quotes \””

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of variable declaration is:
- type identifier [=value] [,identifier [=value] ….] ;

Example:
int a, b , c ; // **Declares three ints a, b and c**
int d = 3, e, f = 5; //**Declares three more ints initializing d and f**
double pi = 3.14159 ; //**Declares an approximation of pi**
char x = ‘x’ ; // **The variable x has the value ‘x’**

Dynamic Initialization

- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

- For example

```
class DynInit {
public static void main(String args[ ]) {
double a = 3.0, b = 4.0 ;
// cis dynamically initialized
double c = Math.sqrt(a * a+b * b) ;
System.out.printIn("Hypotenuse Is " +c);
}
}
```

- However, c is initialized dynamically to the length of the hypotenuse (using thePythagorean theorem), The program uses another of Java's built-in methods, sqrt( ),which is a member of the Math class, to compute the square root of its argument.
- The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

Scope and Lifetime of Variables

- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.
- variables declared inside a scope are not visible(that is, accessible) to code that is defined outside that scope.
- Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.
- Scopes can be nested. For example, each time we create a block of code, we are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.
- This Means that objects declared in the outer scope will be visible to code within the innerscope. However, the reverse is not true. Objects declared within the inner scope willnot be visible outside it.
- Example:

```
Class Scope {
        public static void main(String[] args) {
                int x; //known to all code within main
                x = 10 ;
                if (x = = 10) { //Start new scope
                        int y = 20; // known only to this block
                        // x and y both known here.
                        System.out.println("x and y:" +x +" " + y) ;
                        x = y * 2 ;
                }
                // y = 100; //Error ! y not known here
                // x is still known here.
                System.out.println("x is" +x);
        }
}
```

- variables are created when their scope is entered, and destroyed when their scope is left.
- This means that a variable will not hold tis value once it has gone out of scope. Therefore, variables declared within a methed will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of variable is confined to its scope.

Type Conversion and Casting

- 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 int. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, we must use a cast, which performs an explicit conversion between incompatible types.

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:
  I – The two types are compatible.
  II – The destination type is larger than the source type.
- When these two conditions are met, a widening conversion takes place. For example, the double type is always large enough than int so automatic type conversion takes place.

- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- Numeric types are not compatible with boolean.

Casting Incompatible Types

- Although the automatic type conversions are helpful, they will not fulfill all needs.
- what if you want to assign an float value to a int variable? This conversion will not be performed automatically, because a int is smaller than an float. 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
- Example

```
public class test {
public static void main(String[] args) {
float f = 1.0F ;
int b = 2 ;
int c ;
float d ;
d = b; //automatic conversion (widening)
c = (int) f; //type cast (narrowing)
}
}
```

Automatic Type Promotion in Expressions

- In addition to assignments, there is another place where certain type conversions may occur: in expressions.
- 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;
```

- 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!**

Type Promotion Rules

- First, all byte and short values are promoted to int, as just described.
- Also, if one of the operand is higher order data type than other operands in the expression then whole expression is promoted to the higher order data type; e.g:
  - int a=1;float b=2F,double c=3D;
  - (C*b) is of double type but (b*a) is of float type.

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.

One – Dimensional Arrays

- A one-dimensional array is, essentially, a list of like-typed variables.
- To create an array, we first must create an array variable of the desired type. The general form of a one dimensional array declaration is
  - type var-name[];
- Example:
  - int month_days []; **//int is type of array whose** name is 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, we can allocate one using new and assign it to month_days.
- 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];
  - Example:
  • 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.
- 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;

```
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 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 separates 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.
- Example:

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

Multidimensional Arrays

- In Java, multidimensional arrays are actually arrays of arrays.
- 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];
- Two dimensional arrays can be initialized when declaring like one dimensional array like.

```
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}
};
```

Alternative Array Declaration Syntax

- 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];