JAVA : Part#1

Learning Objective

- To understand the data types
- To understand variables
- To understand arrays

Data Types

Java is a strongly typed language

The Simple 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:

Data type	Description
Integers	This group includes **byte**, **short**, **int**, and **long**, which are for whole valued 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.

Data Types

Integers: Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values.

byte: The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Byte variables are declared by use of the **byte** keyword. For example, the ollowing declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short: **short** is a signed 16-bit type. It has a range from –32,768 to 32,767. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int: The most commonly used integer type is **int**. It is a signed 32-bit type that has a range rom –2,147,483,648 to 2,147,483,647.

```
int x, y;
```

long: **long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value.

```
long total;
```

Data Types

Floating-Point Types: Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, **float** and **double**, which represent single and double-precision numbers, respectively

float: The type **float** specifies a *single-precision* value that uses 32 bits of storage. Here are some example **float** variable declarations:

float hightemp, lowtemp;

double: Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Here are some example **double** variable declarations:

double pi, radius;

Data Types

Characters: In Java, the data type used to store characters is **char**. Java uses **Unicode** to represent characters. In Java, **char** is a 16-bit type. This is how you declare a character variable:

char ch1, ch2;

Booleans: 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**.

boolean b;

Variables

VARIABLE

A *variable* is a named memory location that can be assigned a value. It is a placeholder. You can store a number like 5.95 into a variable. After you've placed a number in the variable, you can change your mind and put a different number, like 30.95, into the variable.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

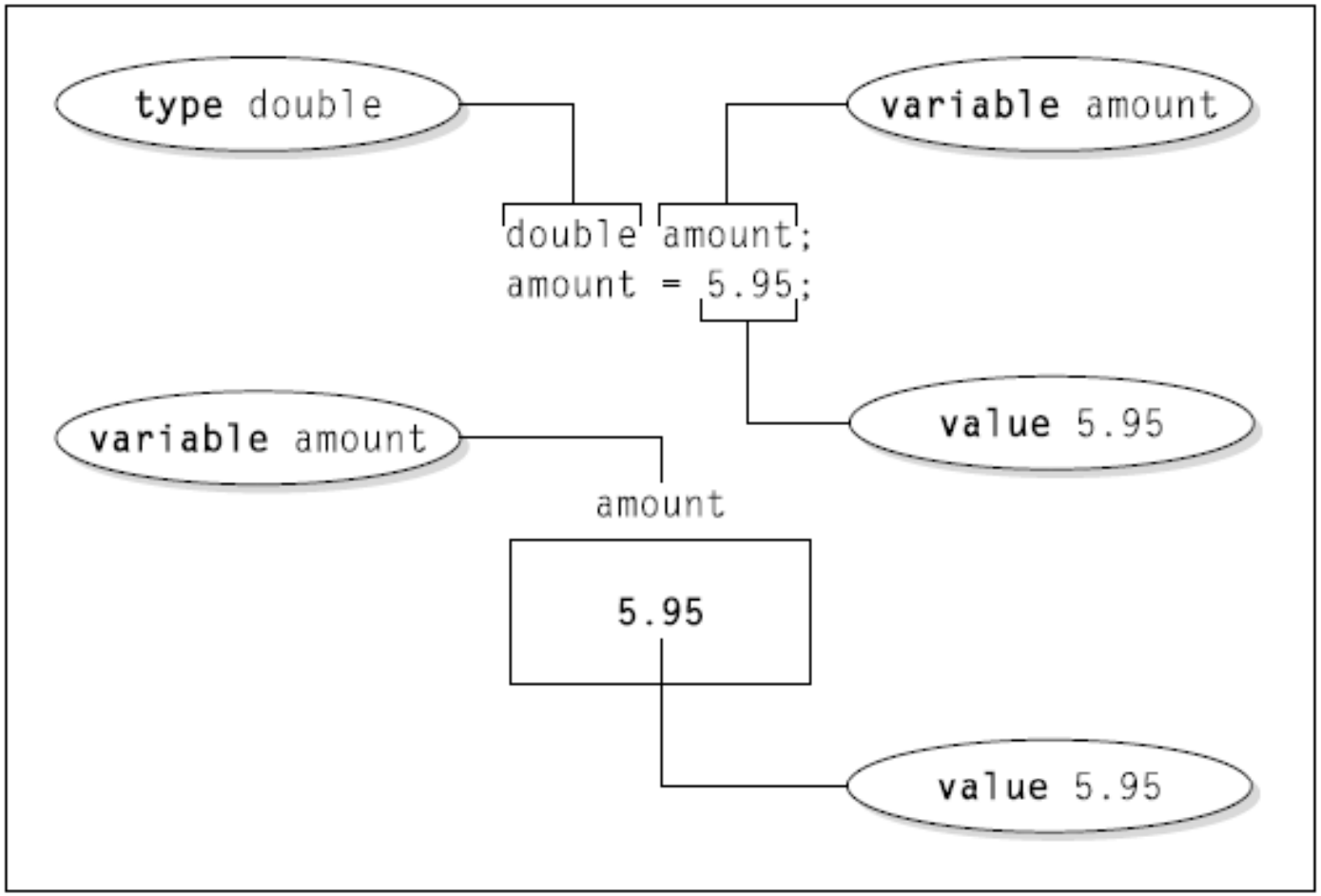
***type identifier* [= *value*][, *identifier* [= *value*] ...] ;**

The *type* is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. To declare more than one variable of the specified type, use a comma-separated list.

Variables

Declaring a variable:

double amount



Arrays

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

The general form of a one dimensional array declaration is

***type varName*[ ];**

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 **monthDays** with the type “array of int”:

int monthDays[];

Arrays

Although this declaration establishes the fact that **monthDays** is an array variable, no array actually exists. In fact, the value of **monthDays** is set to **null**, which represents an array with no value. To link **monthDays** with an actual, physical array of integers, we must allocate one using **new** and assign it to **monthDays**. **new** is a special operator.

The general form of **new** as it applies to one-dimensional arrays appears as follows:

***arrayVar* = new *type*[*size*];**

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *arrayVar* 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 **monthDays**.

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

Arrays

Once we have allocated an array, we 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 **monthDays**.

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

The next line displays the value stored at index 3.

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

Arrays can be initialized when they are declared. 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 we 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 line creates an initialized array of integers:

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Arrays

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.* 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.***

When we allocate memory for a multidimensional array, we need only specify the memory for the first (leftmost) dimension. We 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];
```

Alternative Array Declaration Syntax

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

type[ ] varName;

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];
```

JAVA : Part#2

Learning Objective

- To understand arithmetic and relational operators

- To understand the selection statements in JAVA

Operators

Arithmetic operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result	Remarks
+	Addition	
-	Subtraction	The unary form of minus operator negates its single operand
*	Multiplication	
/	Division	When applied to integer, the result will not have fractional component
%	Modulus	Returns the remainder of a division operation. Can be applied to floating point types as well.
++	Increment	Increases its operand by one. (e.g.) a++ ; ➔ a = a + 1;
--	Decrement	Decreases its operand by one. (e.g.) a-- ; ➔ a = a - 1;
+=	Addition Assignment	Combines an arithmetic operation with an assignment. Any statement of the form **var = var op expression;** can be rewritten as **var op= expression;** For example **a = a + 4;** can be rewritten as **a+=4;**
-=	Subtraction Assignment	
*=	Multiplication Assignment	
/=	Division Assignment	
%=	Modulus Assignment	

Operators

Relational operators

The relational operators determine the relationship that one operand has to the other. They determine equality and ordering. The outcome of these operations is a boolean value. Only integer, floating-point and character operands may be compared to see which is greater or lesser than the other.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Operators

The Assignment Operator

The assignment operator is the single equal sign, =. It has this general form:

var = expression;

Here, the type of var must be compatible with the type of expression. It also allows to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement.

Operators

The ? Operator

It is a ternary (three-way) operator that can replace certain types of if-then-else statements. The ? operator has this general form:

expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value.
If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.

Here is an example of the way that the ? is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark. If denom equals zero, then the expression between the questionmark and the colon is evaluated and used as the value of the entire ?expression. If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to ratio.

Control Statements

Control statements:

Control statements makes the flow of execution to advance and branch based on changes to the state of a program.

Java's program control statements can be put into the following categories:

- Selection
- Iteration
- Jump.

Control Statements (Selection)

Selection statements:

Selection statements allow the program to choose different paths of execution based upon the outcome of an expression or the state of a variable

Java supports two selection statements: **if and switch.**

if

It can be used to route program execution through two different paths. Here is the general form of the **if** statement**:**

```
if (condition)
        statement1;
else
        statement2;
```

The **if** works like this: If the ***condition*** is *true***,** then ***statement1*** is executed. Otherwise, *statement2 (if it exists)* is executed. *In no case will both statements be executed.*

Each *statement may be a single statement or a compound statement enclosed in* curly braces (that is, a *block). The condition is any expression that returns a* ***boolean*** value**.** The **else** clause is optional.

Control Statements (Selection)

For example, consider the following:

```
int a, b;
// ...
if(a < b)
   a = 0;
else
    b = 0;
```

Here, if **a** is less than **b,** then **a** is set to zero**.** Otherwise, **b** is set to zero**.** In no case they are both set to zero

Only one statement can appear directly after the **if** or the **else.** To include more statements, create a block, as in this fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
        ProcessData();
        bytesAvailable -= n;
} else
        waitForMoreData();
```

Control Statements (Selection)

if-else-if Ladder

The general form of the if else if ladder is

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
...
else
        statement;
```

The ***if*** statements are executed from the top down. As soon as one of the conditions controlling the ***if*** is ***true***, the statement associated with that ***if*** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final ***else*** statement will be executed.

Control Statements (Selection)

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

```
class IfElse {
        public static void main(String args[]) {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
                season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
                season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
                season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
                season = "Autumn";
        else
                season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
        }
}
```

Control Statements (Selection)

switch

The **switch statement is Java's multiway branch statement.** Often, it is a better alternative than a large series of **if-else-if** statements**.**

The general form of a **switch** statement**:**

```
switch (expression) {
        case value1:
                // statement sequence
        break;
        case value2:
                // statement sequence
        break;
        ...
        case valueN:
                // statement sequence
        break;
        default:
                // default statement sequence
}
```

Control Statements (Selection)

- The expression must be of type **byte, short, int, or char.**
- Each of the values specified in the case statements must be of a type compatible with the expression.
- Each case value must be a unique literal (that is, it must be a constant, not a variable).
- Duplicate case values are not allowed.

The **switch** statement works like this:

- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the default statement is executed. The default statement is optional.
- If no case matches and no default is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

Control Statements (Selection)

```
// A simple example of the switch.
class SampleSwitch {
        public static void main(String args[]) {
            int i=3;
             switch(i) {
                    case 0:
                            System.out.println(“ZERO.");
                    break;
                    case 1:
                            System.out.println(“ONE.");
                    break;
                    case 2:
                            System.out.println(“TWO");
                    break;
                    case 3:
                            System.out.println(“THREE");
                    break;
                    default:
                            System.out.println(“NOT IN THE RANGE (0-3)");
                    }
} }
```