# Chapter Two

## Basics in Java Programming

### 2.1 Basic Language Elements

#### Lexical Tokens:

The low-level language elements are called lexical tokens (or just tokens for short) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

#### Identifiers:

The name of a variable (or other item like classes, methods and labels) is called an identifier. A Java identifier must not start with a digit and all the characters must be letters, digits, or the underscore (\_) symbol and any currency symbol (such as $, ¢, ¥, or £).

Examples of Legal Identifiers: number, Number, sum\_$, bingo, $$\_100, mal, gruß

Examples of Illegal Identifiers: 48chevy, all@hands, grand-sum, myfirst.java, %change

Java is a **case-sensitive language**; that is, it distinguishes between upper and lowercase letters in the spelling of identifiers. Hence, the following are three distinct identifiers and could be used to name three distinct variables: rate RATE Rate

A Java identifier can theoretically be of any length, and the compiler will accept even unreasonably long identifiers.

#### Keywords or reserved words:

Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. All the keywords are in lowercase and incorrect usage results in compilation errors. All these reserved words cannot be used as identifiers.

##### Keywords in Java

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | default | Implements | protected | throw |
| assert | do | Import | public | throws |
| boolean | double | Instanceof | return | transient |
| break | else | Int | short | try |
| byte | extends | Interface | static | void |
| case | final | Long | strictfp | volatile |
| catch | finally | Native | super | while |
| char | float | New | switch |   |
| class | for | Package | synchronized |   |
| continue | if | Private | this |  |

 |

##### Reserved Literals in Java

|  |  |  |  |
| --- | --- | --- | --- |
|

|  |  |  |
| --- | --- | --- |
| null | True | false |

 |

##### Reserved Keywords not Currently in Use

|  |  |  |
| --- | --- | --- |
|

|  |  |
| --- | --- |
| Const | goto |

 |

#### Literals or constants:

A literal denotes a constant value, that is, the value a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, Boolean or string values. In addition, there is the literal null that represents the null reference.

##### Integer Literals:

Integer data types are comprised of the following primitive data types: int, long, byte, and short.

The default data type of an integer literal is always **int**, but it can be specified as long by appending the suffix L (or l) to the integer value. Without the suffix, the long literals 2000L and 0l will be interpreted as int literals. There is no direct way to specify a short or a byte literal.

In addition to the decimal number system, integer literals can also be specified in octal (base 8) and hexadecimal (base 16) number systems. Octal and hexadecimal numbers are specified with 0 and 0x (or 0X) prefix respectively.

| **Decimal** | **Octal** | **Hexadecimal** |
| --- | --- | --- |
| 8 | 010 | 0x8 |
| 10L | 012L | 0XaL |
| -16 | -020 | -0x10 |
| 27 | 033 | 0x1B |
| 2147483647 (i.e., 231-1) | 017777777777 | 0x7fffffff |

#### Floating-point Literals:

Floating-point data types come in two flavors: float or double. The default data type of a floating-point literal is double, but it can be explicitly designated by appending the suffix D (or d) to the value. A floating-point literal can also be specified to be a float by appending the suffix F (or f).

Examples of double and float Literals: 49.0 49. 49D 0.49F .49F 49.0F 49.F 49F

##### Boolean Literals

The primitive data type boolean represents the truth-values true or false that are denoted by the reserved literals true or false, respectively.

##### Character and string Literals:

A character literal is quoted in single-quotes ('). All character literals have the primitive data type char. A string literal is a sequence of characters, which must be quoted in quotation marks and which must occur on a single line. All string literal are objects of the class String.

Example of character literals:**’ ’ , ’0’ , ’3’ , ’A’ , ’b’ ,** **'\u0020' ,** **'\u0041'**

Example of string literals: “this is java class “, "\"String literals are double-quoted.\""

#### Comments:

A program can be documented by inserting comments at relevant places. These comments are for documentation purposes and are ignored by the compiler.

Java provides three types of comments to document a program:

##### Single-line Comment: All characters after the comment-start sequence // through to the end of the line constitute a single-line comment. The symbols // are two slashes (without a space between them). Comments indicated with // are often called single line comments.

 Eg. // this comment ends at the end of this line.

 int a; //this is a variable

##### Multiple-line Comment: A multiple-line comment, as the name suggests, can span several lines. Such a comment starts with /\* and ends with \*/.

 Eg. /\*This is a multi-line comment.

Note that there is no comment symbol

of any kind on the second line.\*/

##### Documentation Comment(javadoc):

A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the javadoc tool to generate HTML documentation for the program. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /\*\* and ends with \*/

Eg. /\*\*

 \* This class implements a gizmo.

 \* @author K.A.M.

 \* @version 2.0

 \*/

### 2.2 Primitive Data Types

Java has basic types for characters, different kinds of integers, and different kinds of floating-point numbers (numbers with a decimal point), as well as a type for the values true and false. These basic types are known as ***primitive types.***



Primitive data types in Java can be divided into three main categories:

#### Integer Types

Integer data types are byte, short, int, and long.

| **Data Type** | **Width (bits)** | **Minimum value MIN\_VALUE** | **Maximum value MAX\_VALUE** |
| --- | --- | --- | --- |
| byte | 8 | -27 (-128) | 27-1 (+127) |
| short | 16 | -215 (-32768) | 215-1 (+32767) |
| int | 32 | -231 (-2147483648) | 231-1 (+2147483647) |
| long | 64 | -263 (-9223372036854775808L) | 263-1 (+9223372036854775807L) |

#### Character Type

Characters are represented by the data type **char.** Their values are unsigned integers that denote all the 65536 (216) characters in the 16-bit Unicode character set. Their values are unsigned integers that denote all the 65536 (216) characters in the 16-bit Unicode character set.

| **Data Type** | **Width (bits)** | **Minimum Unicode value** | **Maximum Unicode value** |
| --- | --- | --- | --- |
| char | 16 | 0x0 (\u0000) | 0xffff (\uffff) |

#### Floating-point Types

Floating-point numbers are represented by the float and double data types.

| **Data Type** | **Width (bits)** | **Minimum Positive and max negative Value**  | **Maximum Positive Value minimum negative MAX\_VALUE** |
| --- | --- | --- | --- |
| float | 32 | +1.401298464324817E-45f | +3.402823476638528860e+38f |
| double | 64 | +4.94065645841246544e-324 | +1.79769313486231570e+308 |

#### Boolean Type

The data type boolean represents the two logical values denoted by the literals true and false.

### 2.3 Variable

A variable stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can only store values **of primitive data types and references to objects.** Variables that store references to objects are called **reference variables.**

#### Variable Declarations for primitive data type

When you declare a variable, you are telling the compiler and, ultimately, the computer what kind of data you will be storing in the variable.

In Java, a variable must be declared before it is used. Variables are declared as described here.

SYNTAX

Type Variable\_1 ,Variable\_2,. . .;

EXAMPLES

int count, numberOfDragons, numberOfTrolls;

char answer;

double speed, distance;

A declaration can also include initialization code to specify an appropriate initial value for the variable

 int i = 10, // i is an int variable with initial value 10.

 j = 101; // j is an int variable with initial value 101.

#### Variable Declarations for object reference

A variable declaration that specifies a reference type (i.e., a class, an array, or an interface name) declares an object reference variable.

 Student stud1; Variable stud1 can reference objects of class Student;

A declaration can also include an initializer to create an object whose reference can be assigned to the reference variable: Student stud1=new Student(“001”,”abebe”);

The keyword new, together with the constructor call Student("001",”abebe”), creates an object of class Student.

#### Lifetime of Variables

Lifetime of a variable, that is, the time a variable is accessible during execution, is determined by the context in which it is declared. We distinguish between lifetime of variables in three contexts:

1. **Instance variables:** members of a class and created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the state of the object. Instance variables exist as long as the object they belong to exists.
2. **Static variables**: also members of a class, but not created for any object of the class and, therefore, belong only to the class. They are created when the class is loaded at runtime, and exist as long as the class exists.
3. **Local variables:** declared in methods and in blocks and created for each execution of the method or block. After the execution of the method or block completes, local (non-final) variables are no longer accessible.

#### Initial Values for Variables:

Default values for fields of primitive data types and reference types are listed in table below. The value assigned depends on the type of the field.

| **Data Type** | **Default Value** |
| --- | --- |
| boolean | False |
| char | '\u0000' |
| Integer (byte, short, int, long) | 0L for long, 0 for others |
| Floating-point (float, double) | 0.0F or 0.0D |
| Reference types | Null |

If no initialization is provided for a static variable either in the declaration or in a static initializer block it is initialized with the default value of its type when the class is loaded.

If no initialization is provided for an instance variable either in the declaration or in an instance initializer block, it is initialized with the default value of its type when the class is **instantiated.**

**public class Student{**

 **// Static variable**

 **static int passw; // Default value 0 when class is loaded.**

 **// Instance variables**

 **int age = 23; // Explicitly set to 23.**

 **boolean isPass; // Implicitly set to default value false.**

 **String name; // Implicitly set to default value null.**

 **public static void main(String[] args) {**

 **Student stud1 = new Student();**

 **System.out.println("Static variable passw: " + stud1.passw);**

 **System.out.println("Instance variable age: " + stud1.age);**

 **System.out.println("Instance variable ispass: " +stud1.ispassw);**

 **System.out.println("Instance variable name: " + stud1.name);**

 **return;**

 **}**

**}**

Output:

**Static variable passw:0**

**Instance variable age:23**

**Instance variable ispass:false**

**Instance variable name:null**

### 2.4 Type Conversion/ Casting

Different kinds of type conversions and list the contexts in which these can occur. Some type conversions must be **explicitly** stated in the program, while others are done **implicitly**.

**A type cast** takes a value of one type and produces a value of another type. Where an operator would have incompatible operands for example, assigning a double to an int, Java demands that a cast be used to explicitly indicate the type conversion.

The cast construct has the following syntax: **(<type>) <expression>**

**(type)** is unary cast operator **.**At runtime, a cast results in a new value of <type>, which best represents the value of the <expression> in the old type.

We use the term casting to mean applying the cast operator for **explicit type** conversion.

Casting can be applied to **primitive values** as well as **references**. Casting between primitive data types and reference types is not permitted.

#### Narrowing and Widening Conversions

For the primitive data types, the value of a narrower data type can be converted to a value of a broader data type without loss of information. This is called a **widening primitive conversion**.



Figure Widening Numeric Conversions

The conversions shown are **transitive**. For example, an int can be directly converted to a double without first having to convert it to a long and a float.

Converting from a broader data type to a narrower data type is called **a narrowing primitive conversion**, which can result in loss of magnitude information. Any conversion which is not a widening conversion according to Figure above is a narrowing conversion.

All conversions between **char** and the two integer types **byte** and **short** are **considered narrowing conversions**.

Both narrowing and widening conversions can be either explicit (requiring a cast) or implicit. Widening conversions are usually done implicitly, whereas narrowing conversions typically require a cast. It is not illegal to use a cast for a widening conversion.

Numeric promotion is implicitly applied on the operands to convert them to permissible types. But distinction is made between ***unary and binary numeric promotion***.

Unary numeric promotion (operators + and -, integer bitwise complement operator ~, indexing array elements, shift operators <<, >> and >>>)

**Unary numeric promotion states that:** If the single operand of the operator has a type narrower than int, it is converted to int by an implicit widening primitive conversion; otherwise, it is not converted.

Binary Numeric Promotion (arithmetic operators \*, /, %, +, and - , relational operators <, <=, >, and >=, integer bitwise operators &, ^, and |)

**Binary Numeric Promotion states that:** Given T to be numeric type, If T is broader than int, both operands are converted to T; otherwise, both operands are converted to int.

### 2.5 Operators and Assignments

#### Simple Assignment Operator

The assignment statement has the following syntax:

**<variable> = <expression>**

The destination <variable> and the source <expression> must be type compatible. The destination variable must also have been declared.

**Assigning Primitive Values:** The following examples illustrate assignment of primitive values:

 int j, k;

 j = 10; // j gets the value 10.

 j = 5; // j gets the value 5. Previous value is overwritten.

 k = j; // k gets the value 5.

**Assigning References:**

Copying reference by assignments creates aliases. It merely assigns the reference value to the variable on the right-hand side to the variable on the left-hand side, so that they denote the same object.

**Student stud1=new Student(“001”,”ahmed “);**

 **Student stud2=new Student(“002”,”Helen”);**

 **stud1=stud2;//**

**Multiple Assignments:** multiple assignments have right associativity of the assignment operator.

 Eg1. int k = j = 10; // (k = (j = 10))

 Eg2. int[ ] a = {10, 20, 30, 40, 50}; // an array of int

 int index = 4;

 a[index] = index = 2;//

The evaluation proceeds as follows:

a[index] = index = 2;

 a[4] = index = 2;

 a[4] = (index = 2);// index gets the value 2. = is right associative.

 a[4] = 2; // The value of a[4] is changed from 50 to 2.

#### Numeric Type Conversions on Assignment

* If the destination and source have the same type in an assignment, then, obviously, the source and the destination are type compatible. and the source value need not be converted.
* if a widening primitive conversion is permissible, then the widening conversion is applied implicitly

 // Implicit Widening Primitive Conversions

 int smallOne = 1234;

 long bigOne = 2000; // Implicit widening: int to long.

 double largeOne = bigOne;// Implicit widening: long to double.

 double hugeOne = (double) bigOne; // Cast redundant but allowed

* implicit narrowing primitive conversions on assignment can occur in cases where all of the following conditions are fulfilled:
* the source is a constant expression of either byte, short, char, or int type
* the destination type is either byte, short, or char type
* the value of the source is determined to be in the range of the destination type at compile time.

Eg. // Above conditions fulfilled for implicit narrowing primitive conversions.

 short s1 = 10; // int value in range.

char c1 = 32; // int value in range.

byte b1 = 40; // int value in range.

// Above conditions not fulfilled for implicit narrowing primitive conversions.

 int i2 = -20;

 final int i3 = i2;

 final int i4 = 200;

 short s3 = (short) i2; // Not constant expression.

 char c3 = (char) i3; // final value of i3 not determinable.

 char c4 = (char) i2; // Not constant expression.

 byte b4 = (byte) 128; // int value not in range.

* All other narrowing primitive conversions will produce a compile-time error on assignment, and will explicitly require a cast

 float huge = (float) 1.7976931348623157d; // double to float.

 long giant = (long) 4415961481999.03D; // (1) double to long.

 int big = (int) giant; // (2) long to int.

 short small = (short) big; // (3) int to short.

#### Arithmetic Operators: \*, /, %, +, -

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Unary | + | Addition | - | Subtraction |   |   |
| Binary | \* | Multiplication | / | Division | % | Remainder |
|   | + | Addition | - | Subtraction |  |  |

In the table above:

* The precedence of the operators is in decreasing order.
* The operators in the same row have the same precedence.

The unary operators have **right associativity** and the binary operators have **left associativity**.

Eg.of unary operators: int value = - -10; // (-(-10)) is 10

Eg.of Binary operators: int value= 2 \* 4 - 7 % 5 /2 ;// is 7

#### Arithmetic Compound Assignment Operators: \*=, /=, %=, +=, -=

A compound assignment operator has the following syntax:

**<variable> <op>= <expression>**

and the following semantics:

**<variable> = (<type>) (<variable> <op> (<expression>))**

| **Expression:** | **Given T as the Numeric Type of x, the Expression Is Evaluated as:** |
| --- | --- |
| x \*= a | x = (T) ((x) \* (a)) |
| x /= a | x = (T) ((x) / (a)) |
| x %= a | x = (T) ((x) % (a)) |
| x += a | x = (T) ((x) + (a)) |
| x -= a | x = (T) ((x) - (a)) |

**Eg.** int i = 2;

i \*= i + 4; // (1) Evaluated as i = (int) (i \* (i + 4)).

#### Variable Increment and Decrement Operators: ++, --

Variable increment (++) and decrement (--) operators come in two flavors: prefix and postfix.

**Prefix increment or decrement operator has the following semantics:**

++i or --i adds 1 to i or subtract 1 from i first, then uses the new value of i as the value of the expression. It is equivalent to the following statements.

**Postfix increment or decrement operator has the following semantics:**

j++ or j-- uses the current value of j as the value of the expression first, then adds 1 to j or subtract 1 from j.

#### Relational Operators: <, <=, >, >=

Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators. The evaluation results in a boolean value. Relational operators are **nonassociative**.

#### Primitive Data Value Equality: ==, !=

The equality operator == and the inequality operator != can be used to compare primitive data values, including boolean values. Equality operators have left associativity.

Eg. int a, b, c;

 a = b = c = 5;

 boolean valid1 = a == b == c; // (1) Illegal.

 boolean valid2 = a == b && b == c; // (2) Legal.

 boolean valid3 = a == b == true; // (3) Legal.

#### Object Reference Equality: ==, !=

The equality operator == and the inequality operator != can be applied to object references to test whether they denote the same object.

Pizza pizza\_A = new Pizza("Sweet&Sour"); // new object

Pizza pizza\_B = new Pizza("Sweet&Sour"); // new object

Pizza pizza\_C = new Pizza("Hot&Spicy"); // new object

String banner = "Come and get it!"; // new object

boolean test = banner == pizza\_A; // (1) Compile-time error.

boolean test1 = pizza\_A == pizza\_B; // false

boolean test2 = pizza\_A == pizza\_C; // false

pizza\_A = pizza\_B; // Denote the same object, are aliases.

boolean test3 = pizza\_A == pizza\_B; // true

The comparison banner == pizza\_A in (1) is illegal, because String and Pizza types are not type compatible. The values of test1 and test2 are false, because the three references denote different objects, regardless of the fact that pizza\_A and pizza\_B are both sweet and sour pizzas. The value of test3 is true, because now both pizza\_A and pizza\_B denote the same object.

#### Boolean Logical Operators:!, ^, &, |

Boolean logical operators include the unary operator ! (logical complement) and the binary operators & (logical AND), | (logical inclusive OR), and ^ (logical exclusive OR, a.k.a. logical XOR).

boolean b1, b2, b3 = false, b4 = false;

b1 = 4 == 2 & 1 < 4; // false, evaluated as (b1 = ((4 == 2) & (1 < 4)))

b2 = b1 | !(2.5 >= 8); // true

b3 = b3 ^ b2; // true

b4 = b4 | b1 & b2; // false

Order of evaluation is illustrated for the last example:

 (b4 = (b4 | (b1 & b2)))

 (b4 = (false | (b1 & b2)))

 (b4 = (false | (false & b2)))

 (b4 = (false | (false & true)))

 (b4 = (false | false))

 (b4 = false)

It can also be applied to integral operands for bitwise operations.

 int v1=3 ,v2=2;

 int result1 = ~v1; // -3

 int result2 = v1 & v2; //2

 int result3 = v1 | v2; //3

 int result4 = v1 ^ v2; // 1

#### Conditional Operators: &&, ||

Conditional operators && and || are similar to their counterpart logical operators & and |, except that their evaluation is **short-circuited**. The left-hand operand is evaluated before the right one, if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated.

 && have high precedence than | |.

boolean b1 = 4 == 2 && 1 < 4; // false, short-circuit evaluated as

 // (b1 = ((4 == 2) && (1 < 4)))

boolean b2 = !b1 || 2.5 > 8; // true, short-circuit evaluated as

 // (b2 = ((!b1) || (2.5 > 8)))

boolean b3 = !(b1 && b2); // true

boolean b4 = b1 || !b3 && b2; // false, short-circuit evaluated as

 // (b4 = (b1 || ((!b3) && b2)))

Order of evaluation for computing the value of boolean variable b4 proceeds as follows:

 (b4 = (b1 || ((!b3) && b2)))

 (b4 = (false || ((!b3) && b2)))

 (b4 = (false || ((!true) && b2)))

 (b4 = (false || ((false) && b2)))

 (b4 = (false || false))

 (b4 = false)

#### Shift Operators: <<, >>, >>>

The binary shift operators form a new value by shifting bits either left or right a specified number of times in a given integral value.

|  |  |  |
| --- | --- | --- |
| Shift left | a << n | Shift all bits in a left n times, filling with 0 from the right |
| Shift right with sign bit | a >> n | Shift all bits in a right n times, filling with the sign bit from the left. |
| Shift right with zero fill | a >>> n | Shift all bits in a right n times, filling with 0 from the left |

##### The Shift-left Operator <<

int i = 12;

int result = i << 4; // 192

The bits in the int value for i are shifted left four places as follows:

 i << 4

 = 0000 0000 0000 0000 0000 0000 0000 1100 << 4

 = 0000 0000 0000 0000 0000 0000 1100 0000

 = 0x000000c0

 = 192

Each left-shift corresponds to multiplication of the value by 2. In the above example, 12\*24 is 192

##### The Shift-right-with-sign-fill Operator >>

 int i = 12;

 int result = i >> 2; // 3

The value for i is shifted right with sign-fill two places.

 i >> 2

 = 0000 0000 0000 0000 0000 0000 0000 1100 >> 2

 = 0000 0000 0000 0000 0000 0000 0000 0011

 = 0x00000003

 = 3

##### The Shift-right-with-zero-fill Operator >>>

 12 >>> 2

 = 0000 0000 0000 0000 0000 0000 0000 1100 >>> 2

 = 0000 0000 0000 0000 0000 0000 0000 0011

***NB: The easiest Method to calculate Shift Operator is***

1. ***if a>>n we can calculate a = a/2n***
2. ***if a<<n we can calculate a = a\*2n***

### 2.6 Precedence and Associativity Rules for Operators

**Precedence rules** are used to determine which operator should be applied first if there are two operators with different precedence. The operator with the highest precedence is applied first.

**Associativity rules** are used to determine which operator should be applied first if there are two operators with the same precedence, and these follow each other in the expression.

Left associativity implies grouping from left to right and Right associativity implies grouping from right to left.

* The operators are shown with decreasing precedence from the top of this table.
* Operators within the same row this table have the same precedence.
* All binary operators, except for the relational and assignment operators, **associate from left to right**. The relational operators are nonassociative.
* Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.

|  |  |
| --- | --- |
| Postfix operators | [] . (parameters) expression++ expression-- |
| Unary prefix operators | ++expression --expression +expression -expression ~ ! |
| Unary prefix creation and cast | new (type) |
| Multiplicative | \* / % |
| Additive | + - |
| Shift | << >> >>> |
| Relational | < <= > >= instanceof |
| Equality | == != |
| Bitwise/logical AND | & |
| Bitwise/logical XOR | ^ |
| Bitwise/logical OR | | |
| Conditional AND | && |
| Conditional OR | || |
| Conditional | ?: |
| Assignment | = += -= \*= /= %= <<= >>= >>>= &= ^= |= |

### 2.7 The Class String

There is no primitive type for strings in Java. However, there is a class called String that can be used to store and process strings of characters.

**The class String** is a predefined class that is automatically made available to you when you are programming in Java. Objects of type String are strings of characters that are written within double quotes.

For example, the following declares blessing to be the name for a String variable:

 String blessing;

The following assignment statement sets the value of blessing so that blessing serves as another name for the String object "Live long and prosper.":

blessing = "Live long and prosper.";

The declaration and assignment can be combined into a single statement, as follows:

String blessing = "Live long and prosper.";

When you use the + operator on two strings, the result is the string obtained by connecting the two strings to get a longer string. This is called **concatenation**. So, when it is used with strings, the + is sometimes called **the concatenation operator**.

For example, consider the following:

String noun = "Strings";

String sentence;

sentence = noun + "are cool.";

System.out.println(sentence)//Strings are cool

 System.out.println("100" + 42); // 10042

A class is the name for a type whose values are objects. Objects are entities that store data and can take actions. For example, objects of the class String store data consisting of strings of characters, such as "Hello". The actions that an object can take are called methods. For example, the method length() returns the number of characters in a String object.

int n = "Hello".length();// the value of n is 5

or

String greeting = "Hello";

int n = greeting.length();

#### String Methods:

The class String has a number of useful methods that can be used for string-processing applications. A method can return different values depending on what happens in your program. However, each method can return values of only one type. For example, the method length of the class String always returns an int value. the type given before the method name is the type of the values returned by that method. Since length always returns an int value, the entry for length begins

int length()

**Some Methods in the Class String**

|  |
| --- |
|  |
| int length() |
| Returns the length of the calling object (which is a string) as a value of type int.EXAMPLE After program executesString greeting = "Hello!";greeting.length() returns6 |
| boolean equals(Other\_String) |
| Returns true if the calling object string and the Other\_Stringare equal. Otherwise, returns false.EXAMPLE After program executes String greeting = "Hello"; greeting.equals("Hello") returns true greeting.equals("Good-Bye")returns false greeting.equals("hello")returns falseNote that case matters. "Hello" and "hello" are not equal because one starts with an uppercase letter and the other starts with a lowercase letter. |
| boolean equalsIgnoreCase(Other\_String) |
| Returns true if the calling object string and the Other\_String are equal, considering upper- and Lowercase versions of a letter to be the same. Otherwise, returns false.EXAMPLEAfter program executes  String name = "mary!"; greeting.equalsIgnoreCase("Mary!") returns true |
| String toLowerCase( ) |
| Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase. EXAMPLEAfter program executes String greeting = "Hi Mary!"; greeting.toLowerCase() returns "hi mary!" |
| String toUpperCase( ) |
| Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase. EXAMPLE After program executes String greeting = "Hi Mary!"; greeting.toUpperCase() returns "HI MARY!" |
| char charAt(Position) |
| Returns the character in the calling object string at the Position. Positions are counted 0, 1, 2, etc. EXAMPLEAfter program executes  String greeting = "Hello!"; greeting.charAt(0) returns 'H', and  greeting.charAt(1) returns 'e' |
| String substring(start) |
| Returns the substring of the calling object string starting from Start through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position Start is included in the value returned. EXAMPLE After program executes String sample = "AbcdefG"; sample.substring(2) returns"cdefG". |
| String substring(start,End) |
| Returns the substring of the calling object string starting from position Start through, but not including, position End of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position Start is included in the value returned, but the character at position End is not included. EXAMPLE After program executes String sample = "AbcdefG"; sample.substring(2, 5) returns "cde". |
| int indexOf(A\_String) |
| Returns the index (position) of the first occurrence of the string A\_String in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1if A\_String is not found. EXAMPLEAfter program executes String greeting = "Hi Mary!"; greeting.indexOf("Mary") returns3, and  greeting.indexOf("Sally") returns−1 |
| int indexOf(A\_String,Start) |
| Returns the index (position) of the first occurrence of the string A\_String in the calling object string that occurs at or after position Start. Positions are counted 0, 1, 2, etc. Returns −1 if A\_String is not found. EXAMPLE After program executes String name = "Mary, Mary quite contrary"; name.indexOf("Mary", 1) returns 6.The same value is returned if 1is replaced by any number up to and including 6.name.indexOf("Mary", 0) returns0.name.indexOf("Mary", 8) returns–1. |
| int lastIndexOf(A\_String) |
| Returns the index (position) of the first occurrence of the string A\_String in the calling object string that occurs at or after position Start. Positions are counted 0, 1, 2, etc. Returns −1 if A\_String is not found. EXAMPLE After program executes String name = "Mary, Mary quite contrary"; name.indexOf("Mary", 1) returns 6.The same value is returned if 1 is replaced by any number up to and including 6. name.indexOf("Mary", 0) returns 0. name.indexOf("Mary", 8) returns –1. |
| int compareTo(A\_String) |
| Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered. , but all the uppercase letters precede all the lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive numberEXAMPLE After program executes String entry = "adventure"; entry.compareTo("zoo") returns a negative number,  entry.compareTo("adventure") returns 0, and  entry.compareTo("above") returns a positive number |
| int compareToIgnoreCase(A\_String) |
| EXAMPLE After program executesString entry = "adventure";entry.compareToIgnoreCase("Zoo") returns a negative number, entry.compareToIgnoreCase("Adventure") returns 0, and "Zoo".compareToIgnoreCase(entry)returns a positive number. |

#### Escape Sequences:

\" Double quote.

\' Single quote.

\\ Backslash.

\n New line. Go to the beginning of the next line.

\r Carriage return. Go to the beginning of the current line.

\t Tab. White space up to the next tab stop.

For example, the statement

System.out.println("To be or\nNot to be.");

will write the following to the screen:

To be or

Not to be.

### 2.8 Array

An array is a data structure that defines an indexed collection of a fixed number of homogeneous data elements. This means that all elements in the array have the same data type.

A position in the array is indicated by a non-negative integer value called the index. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot increase to accommodate more elements.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. Each array object has a final field called length. The first element is always at index 0 and the last element at index n-1, where n is the value of the length field in the array.

#### one-dimensional arrays:

Simple arrays are one-dimensional arrays, that is, a simple sequence of values.

An array variable declaration has either the following syntax:

<element type>[] <array name>;

or

<element type> <array name>[];

Where <element type> can be a primitive data type or a reference type. The array variable <array name> has the type <element type>[]. Note that the array size is not specified. This means that the array variable <array name> can be assigned an array of any length, as long as its elements have <element type>.

It is important to understand that the declaration does not actually create an array. It only declares a reference that can denote an array object.

Example:

char a[ ];

double[ ] array1, array2; the same with **double array1[ ],array2[ ];**

When the [] notation follows the type, all variables in the declaration are arrays.

An array can be constructed for a specific number of elements of the element type, using the new operator. The resulting array reference can be assigned to an array variable of the corresponding type.

<array name> = new <element type> [<array size>];

The minimum value of <array size> is 0 (i.e., arrays with zero elements can be constructed in Java). If the array size is negative, a NegativeArraySizeException is thrown.

Given the following above array declarations the arrays can be constructed as follows:

 a=new char[10];

 array1=new String[4];

 array2=new String[3];

The array declaration and construction can be combined as follow :

<element type1>[] <array name> = new <element type2>[<array size>];

When the array is constructed, all its elements are initialized to the default value for <element type>.

Example int[ ] anIntArray = new int[10]; // Default element value: 0.

Java provides the means of declaring, constructing, and explicitly initializing an array in one declaration statement:

<element type>[] <array name> = { <array initialize list> };

Example

char[ ] charArray = {'a', 'h', 'a'};

int[ ] anIntArray = {1, 3, 49, 2, 6, 7, 15, 2, 1, 5};// the array size is 10

The whole array is referenced by the array name, but individual array elements are accessed by specifying an index with the [] operator. The array element access expression has the following syntax:

 <array name> [<index expression>]

Example int a=anIntArray[3]//the value of a is 2

In one dimensional to find the length of the array we can use length instance variable

<array name >.length //this return the size of the array

Example char a[]={‘a’,’b’,’c’,’d’};

 int arraySize=a.length;// the value of arraySize is 4

#### Multidimensional Arrays:

Since an array element can be an object reference and arrays are objects, array elements can themselves reference other arrays. In Java, an array of arrays can be defined as follows:

<element type>[ ][ ]...[ ] <array name>;

or

<element type> <array name>[ ][ ]...[ ];

The following declarations are all equivalent:

 int[][] mXnArray; // 2-dimensional array

 int[] mXnArray[]; // 2-dimensional array

 int mXnArray[][]; // 2-dimensional array

Combine the declaration with the construction of the multidimensional array

 int[][] mXnArray = new int[4][5]; // 4 row x 5 column matrix of ints

Multidimensional arrays can also be constructed and explicitly initialized using array initializer blocks discussed for simple arrays. Note that each row is an array which uses an array initializer block to specify its values:

double[ ][ ] identityMatrix = {

 {1.0, 0.0, 0.0, 0.0 }, // 1. row

 {0.0, 1.0, 0.0, 0.0 }, // 2. row

 {0.0, 0.0, 1.0, 0.0 }, // 3. row

 {0.0, 0.0, 0.0, 1.0 } // 4. row

 }; // 4 x 4 Floating-point matrix

identityMatrix.length; //returns length of rows

identityMatrix[0].length; //returns length of column at row 0

#### Use of = and == with Arrays

Array types are reference types; that is, an array variable contains the memory address

of the array it names. The assignment operator copies this memory address. For

example, consider the following code:

double [] a = new double [10];

double [] b = new double [10]; int i;

for (i = 0; i < a.length; i++)

a[i] = i;

b = a;

System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);

a[2] = 42;

System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);

This will produce the following output:

a[2] = 2.0 b[2] = 2.0

a[2] = 42.0 b[2] = 42.0

The equality operator = =does not test two arrays to see if they contain the same values. It tests two arrays to see if they are stored in the same location in the computer’s memory. For example, consider the following code:

int [ ] c = new int [10];

int [ ] d = new int [10];

int i;

for (i = 0; i < c.length; i++)

c[i] = i;

for (i = 0; i < d.length; i++)

d[i] = i;

if (c = = d)

System.out.println("c and d are equal by ==.");

else

System.out.println("c and d are not equal by ==.");

This produces the output

c and d are not equal by = =

### 2.9 Console Input and Output

#### Screen Output:

***System.out.println , system.out.print and system.out.printf***

***System.out is an object that is part of the Java language.***

**println Output**

You can output one line to the screen using System.out.println. The items that are output can be quoted strings, variables, numbers, or almost any object you can define in Java. To output more than one item, place a plus sign between the items. SYNTAX

System.out.println(Item\_1+Item\_2+ ... +Last\_Item);

EXAMPLE

System.out.println("Welcome to Java.");

System.out.println("Elapsed time = " + time + " seconds");

**println versus print**

The only difference betweenSystem.out.println and System.out.print is that with println, the next output goes on a new line, whereas with print, the next output is placed on the same line.

EXAMPLE

System.out.print("Tom ");

System.out.print("Dick ");

System.out.println("and ");

System.out.print("Harry ");

This produces the following output:

Tom Dick and

Harry

Or we can say that

System.out.println( SomeThing ); is equivalent to System.out.print( SomeThing + "\n");

**Formatting Output with printf**

Method printf's first argument is a format string that may consist of fixed text and format specifiers. Fixed text is output by printf just as it would be output by print or println. Each format specifier is a placeholder for a value and specifies the type of data to output. Format specifiers also may include optional formatting information. Format specifiers begin with a percent sign (%) and are followed by a character that represents the data type.

|  |  |  |
| --- | --- | --- |
| CONVERSION CHARACTER | TYPE OF OUTPUT | EXAMPLES |
| d | Decimal(ordinary)integer  | System.out.printf(“my age is %d”,23); |
| f | Floating point | System.out.printf(“%f is less than %f”,3.0,4.0);  |
| s | string | System.out.printf(“fname:%s and lname:%s”,”abd”,”hon”); |
| c | Character  | System.out.printf(“goo%c morning”,’d’); |

Example : System.out.printf(“fname:%s and lname:%s”,”abd”,”hon”);

The output is : fname:abd and lname:hon

#### Console Input Using the Scanner Class:

The Scanner class can be used to obtain input from files as well as from the keyboard. To set things up for keyboard input, you need the following at the begin of the file with the keyboard input code:

 import java.util.Scanner;

You also need the following before the first keyboard input statement:

Scanner ObjectName= new Scanner(System.in);

Then the Object\_Name can then be used with the following methods to read and return various types of data typed on the keyboard. Values to be read should be separated by whitespace characters, such as blanks and/or new lines.

**ObjectName.nextInt()**

Returns the next value of type int that is typed on the keyboard.

**ObjectName.nextLong()**

Returns the next value of type long that is typed on the keyboard.

**ObjectName.nextByte()**

Returns the next value of type byte that is typed on the keyboard.

**ObjectName.nextShort()**

Returns the next value of type short that is typed on the keyboard.

**ObjectName.nextDouble()**

Returns the next value of type double that is typed on the keyboard.

**ObjectName.nextFloat()**

Returns the next value of type float that is typed on the keyboard.

**ObjectName.next()**

Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

**ObjectName.nextBoolean()**

Returns the next value of type Boolean that is typed on the keyboard. The values of true and false are entered as the strings "true" and "false". Any combination of upper- and/or lowercase letters is allowed in spelling "true" and "false".

**ObjectName.nextLine()**

Reads the rest of the current keyboard input line and returns the characters read as a value of type String . Note that the line terminator ‘\n’ is read and discarded; it is not included in the string returned.

**ObjectName.useDelimiter(New\_Delimiter);**

Changes the delimiter for keyboard input with ObjectName. The New\_Delimiter is a value of type String. After this statement is executed, New\_Delimiter is the only delimiter that separates words or numbers. See the subsection “Other Input Delimiters”.

Example

 import java.util.Scanner;

 public class ScannerDemo2 {

 public static void main(String[] args) {

 int n1, n2;

Scanner scannerObject = new Scanner(System.in);

System.out.println("Enter two whole numbers");

System.out.println("separated by one or more spaces:");

n1 = scannerObject.nextInt();

n2 = scannerObject.nextInt();

System.out.println("You entered " + n1 + " and " + n2);

System.out.println("Next enter two numbers.");

System.out.println("Decimal points are allowed.");

double d1, d2;

d1 = scannerObject.nextDouble();

d2 = scannerObject.nextDouble();

System.out.println("You entered " + d1 + " and " + d2);

System.out.println("Next enter two words:");

String word1 = scannerObject.next();

String word2 = scannerObject.next();

System.out.println("You entered \"" + word1 + "\" and \"" + word2 + "\"");

String junk = scannerObject.nextLine(); //To get rid of '\n'

System.out.println("Next enter a line of text:");

String line = scannerObject.nextLine();

System.out.println ("You entered: \"" + line + "\"");

 }

 }

The output:

Enter two whole numbers

Separated by one or more spaces:

42 43

You entered 42 and 43

Next enter two numbers.

A decimal point is OK.

9.99 57

You entered 9.99 and 57.0

Next enter two words:

Jelly beans

You entered "jelly" and "beans"

Next enter a line of text:

Java flavored jelly beans are my favorite.

You entered "Java flavored jelly beans are my favorite."