Chapter 2
Fundamental Programming Structures in Java

1. Writing Comments

The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code. Java has three ways of making comments.

- Single line comments begins with // and end at the end of the line. For example,
  // This is a simple Java program
- Multi line comment is enclosed in /* …… */. For example,
  /* This is a simple Java
  program */
- Documentation comment is enclosed in /** …….. */. This comment is used to produce documentation using **javadoc** tool. For example,
  /** This is a simple Java program */

Example:

```
// This is my first Java program.
public class Welcome
{
	public static void main(String[]args)
	{
		System.out.println("Hello World!");
	}
}
```

2. Identifiers and Keywords

Identifiers are the words that describe variable's name, method names, class names etc. An identifier can be any string of letters lower and upper characters, dollar sign ($), underscore (_), and digits with the exception that it cannot start with digits. Remember Java is case sensitive and identifier **radius** is different from **Radius**. For example,
Hello, hi, hi123, $wer, _dsf, sd_ew, etc. are valid identifiers.
1hi, hello Mr, Mr.X, hello-dude, etc. are not valid identifiers.
Keywords are the words that have distinct and special meaning in Java language and treated in special way by java compiler. So, we cannot use keywords as identifiers. There are many keywords in Java such as int, if, abstract, break, default, extends etc.

3. Literals

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values. Java language specifies five major types of literals namely **integer**, **floating point**, **character**, **string**, and **boolean**. For example,
75 is an integer literal
56.89 is a floating point literal
'y' is a character literal

“hello” is a string literal
true is a boolean literal.

4. Primitive Data Types

Java is strongly typed language so every variable in java must be declared of specific type explicitly so that the compiler understands what kind of variable we are considering. A data type determines the values it may contain, plus the operations that may be performed on it. A data type also defines the amount of memory that will be used when defining the data type and the valid range of values it can represent. There are eight primitive data types in java. The eight types constitutes four groups namely **integers**, **floating points**, **characters** and **boolean**.

4.1. Integer Types

Integers represent the numbers without the fractional part i.e. whole numbers. In Java four data types represent integers: **byte**, **short**, **int**, and **long**. The following table shows the memory required for each type and the valid range of values it can represent.

Integer Type	**Memory Required**	**Range of Values**
byte	1 byte (8 bits)	–128 to +127 (–2^7 to 2^7–1)
short	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
int	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
long	8 bytes (64 bits)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (–2^{63} to 2^{63}–1)

You choose the particular type of integers according to your need. For example if you need age of man in years to be represented, byte is the suitable one since we do not expect age of the man to be more than 127.
We declare Integer Types as:
***int** x;* ***byte** num;* ***short** snum;* ***long** lnum;*
We can initialize the above types while declaring or afterwards. In any case literals are used for initialization. For example:

int *second* = 5, *minute*; /*here two variables are declared where one is initialized
minute = 30; and the other is not. The other variable minute is initialized after declaration statement*/

Default value for these types, if not initialized is 0 for all types with usual literal conventions. There are 4 types of literal representations for integer types they are decimal literals like 10 and -34, octal literals with leading zero like 012 (decimal 10), hexadecimal literals with leading 0x or 0X (zero x or X) like 0x1A (decimal 26) and binary literals with leading 0b or 0B (zero b or B) like 0b111 (decimal 7). Here if we use literal for long integer we append the letter L or l at the normal literal for e.g. ***long** howlong* = 2147483648L.

4.2. Floating-Point Types

The types of numbers with the fractional parts are referred to as floating-point types. A common way in mathematics, as well as in computer science, to represent floating-point numbers is to provide an exponential representation of a number. This is defined as listing the significant digits with one digit written before the decimal point, and then defining the power of 10 to multiply by the number to generate its real value. So, 123 million could be written as 1.23E+8. We have two types of floating-point data types **float** and **double**. The following table shows the memory required for each type and the valid range of values it can represent.

Floating - Point Type	Memory Required	Range of Values
float	4 bytes (32 bits)	+/– 3.40282347E+38
double	8 bytes (64 bits)	+/– 1.7976931346231570E+308

The choice of the above type depends upon the precision we need. As an instance if you are representing currency, float will be ok, but if you are calculating the amount of light to apply to a surface rendering in graphics application, a double becomes more appropriate.
We declare Floating-Point Types as:
float *floatval***;** **double** *doubval*;
We can initialize the above types while declaring or afterwards as defined previously for integer types.
Default value for these types, if not initialized is 0.0 for both types with usual literal conventions. Here if we use literal for float we append the letter f or F at the number. For example, **float** *floatliteral* = 12.3456f, *floatlit2* = 1.234e4F. Similarly we can append d or D for double type (for double type d or D may be omitted). There are three special values for floating type numbers **Double.POSITIVE_INFINITY**, **Double.NEGATIVE_ INFINITY** and **Double.NaN** (also corresponds to **Float** class) to represent positive infinity, negative infinity and not a number respectively.

4.3. The char Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters. For example,
char ch = 'A';
Besides the \u escape sequences, there are several escape sequences for special characters as given below.

Character	Meaning	Unicode Equivalent
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000a
\r	Carriage Return	\u000d
\”	Double Quote	\u0022
\’	Single Quote	\u0027
\\	Back Slash	\u005c

4.4. The boolean Type

The last primitive data type in the Java programming language is the **boolean** data type. A boolean data type has one of two values: **true** or **false**. These are not strings, but reserved words in the Java programming language. The declaration and initialization of Boolean variable is as:
boolean boolval = **false**;

Note: your compiler may generate error message if you do not initialize Boolean variable.

5. Variables and Constants

As in every programming language, variables are used to store values. Constants are variables whose values don't change.

In Java, every variable has a *type*. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;
int vacationDays = 2; //initializing variables
long earthPopulation, nepalPopulation;
boolean done;
```

The convention adopted by the Java world in naming variables is to start with a lowercase letter, and if the name has multiple words, capitalize the first letter of every word. For example: myVaraible, hisChair, redHat. If we are naming class, the convention is to start with capital letter like MyClass.

To declare a variable to be a **constant**, or not changing, Java uses the keyword **final** to denote that a variable is in fact a constant. For example, we know that the value of PI is constant so we can declare PI with the keyword **final** and ensure that its value cannot change as **final double** PI = 3.14285;

If you try to compile a statement that modifies a constant value, such as PI with

```
PI = PI * 2;
```

The compiler generates an error saying that you cannot assign a value to a final variable. By convention, programmers usually capitalize all the letters in a constant's name so that it can be found with only a quick look of the source code. Pi, therefore, was declared as **final double** PI = 3.14285;

6. Enumerated Types

Sometimes, a variable should only hold a restricted set of values. For example, you may sell pizza in four sizes: small, medium, large, and extra-large. You can define your own *enumerated type* whenever such a situation arises. An enumerated type has a finite number of named values. For example,

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

A variable of type Size can hold only one of the values listed in the type declaration, or the special value null that indicates that the variable is not set to any value at all. For xample,

```
Size s = Size.MEDIUM;
```

For example,

```
public class EnumTest
{
        enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
        public static void main(String[]args)
        {

                Size s = Size.MEDIUM;
                System.out.println(s);
        }
```

}

Note: Enumerated types must not be local.

7. Type Conversion and Casting

It is often necessary to convert from one numeric type to another. Here we consider three different conversions.

a. Automatic Type Promotion in Expressions

When using numeric operands in expressions, the type does not necessarily have to be the same. Whenever two incompatible operands are operated then there will be automatic type promotion as follows:

- If any of the operands is **double**, the entire expression is promoted to **double**.
- If one operand is **float**, the entire expression is promoted to **float**.
- If one operand is **long**, the whole expression is promoted to **long**.
- All **byte** and **short** values are promoted to **int**.

This type of conversion is safe and does not result in the loss of information.

b. Conversion through Assignment

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.

The rule for conversion between primitive numeric types is that a type can be assigned to a compatible type or a wider type, but not a narrower type. So, the following conversions are permissible by the compiler:

- byte – short, int, long, float, double
- short – int, long, float, double
- int – long, float, double
- long – float, double
- float – double
- char – int

Example: **int** *i* = 10; **long** *l* = *i*; // legal **double** *d* = *i*; // legal **short** *s* = *i*; // Illegal

c. **Casting Incompatible Types**

To create a *narrowing conversion* between two incompatible types, we must use cast. Casting tells the compiler to convert the data to the specified type even though it might lose data. Casting is performed by prefixing the variable or value by the desired data type enclosed in parentheses:

datatype *variable* = (***datatype***)*value*;

For example: **int** *i* = 10; short *s* = (**short**)*i*;

long *l* = 100; **byte** *b* = (**byte**)*l*;

This type of casting must be carefully done since it may result in loss of data.

8. Operators

An operator is a symbol that is used to perform some action on one, two or three operands. The operators that perform on one operand are called **unary operators**, that perform on

two operands are called **binary operators**, and that perform on three operands are called **ternary operators**.
The Java operators and their corresponding associativity are as given in table below where high precedence operators appear before low precedence operators and operators within the same group have same precedence. Operators on the same level are processed from left to right, except for those that are right to left as indicated in the table below.

Operator	Description	Associativity
[ ]	Array element reference	Left to Right
.	Member selection	
( )	Method call	
++	Post increment	Left to Right
--	Post decrement	
~	Bitwise NOT	Right to Left
!	Logical NOT	
-	Unary minus	
+	Unary plus	
++	Pre increment	
--	Pre decrement	
(type)	Casting	
New	Object and array creation	
*	Multiplication	Left to Right
/	Division	
%	Modulus	
-	Addition	Left to Right
+	Subtraction	
<<	Bitwise shift left	Left to Right
>>	Bitwise shift right	
>>>	Bitwise shift right zero fill	
<	Less than	Left to Right
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
instanceof	Type comparison	
==	Equal to	Left to Right
!=	Not equal to	
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
|	Bitwise Inclusive OR	Left to Right
&&	Logical AND	Left to Right
||	Logical OR	Left to Right
?:	Conditional Operator	Right to Left
=	Simple assignment	Right to Left
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	Shorthand assignment	

8.1. The [], ( ) and . Operators

The [ ] operator is used for accessing the array elements at particular position. The ( ) operator is used for calling methods. The . operator is used for accessing members of a class. We discuss these operators later when we study array and object oriented concepts.

8.2. The ++ and -- Operators

These operators are unary operators. The ++ operator is called increment operator and the -- operator is called decrement operator. The increment operator increases its operand by 1 and the decrement operator decreases its operand by 1. For example, the statement

x = x + 1;

can be rewritten as

x++; or ++x;

Similarly, the statement

x = x – 1;

is equivalent to

x--; Or --x;

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `x++;` and `++x;` will both end in x being incremented by one. The only difference is that the prefix version (`++x`) evaluates to the incremented value, whereas the postfix version (`x++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference. For example,

```
public class IncDec
{
	public static void main(String args[])
	{
		int a = 1, b = 2, c;
		c = a++ + --b;
		c++;
		System.out.println("a = " + a);
		System.out.println("b = " + b);
		System.out.println("c = " + c);
	}
}
```

Output:

```
a = 2
b = 1
c = 3
```

8.3. The cast Operator

To create a *narrowing conversion* between two incompatible types, we must use cast operator. Casting tells the compiler to convert the data to the specified type even though it might lose data. Casting is performed by prefixing the variable or value by the desired data type enclosed in parentheses. For example,

double x = 9.997;

int nx = (int) x;
This type of casting must be carefully done since it may result in loss of data.

8.4. The new Operator

The **new** operator is used to instantiate the class; that is, used to create an object of the class. This operator is also used to create an array. We discuss this operator later when we study object oriented concepts and arrays.

8.5. Arithmetic Operators

There are five arithmetic operators in Java. All the arithmetic operators are given below:

- The / operator is used for division. This operator denotes integer division if both arguments are integers, and floating-point division otherwise.
- The * operator is used for multiplication.
- The + operator is used for addition. This operator is also used for unary plus and string concatenation.
- The – operator is used for subtraction. This operator is also used for unary minus (e.g. -10).
- The % operator is used to obtain remainder after integer division (also called **modulus** operator).

Examples:

7.5/5 = 1.5   7/5 = 1   2+3 = 5   3-2 = 1   5%3 = 2

8.6. Assignment Operators

There are different assignment operators in Java.

- The = operator is used to assign value of an expression to another expression. For e.g. if we want to assign value 5 to the variable five, then we do ***five = 5;***, similarly, ***str = "hello";*** assigns string value "hello" to the variable str.
- **Arithmetic assignment operators (+=, -=, *=, /=, %=):** The += operator is used to assign variable a value obtained by adding the previous value of the variable and value of the other expression. For example a += b; is equivalent to the statement a = a + b;. Other assignment operators are similarly interpreted.

8.7. Relational and Equality Operators

Relational and equality operators are very important in any programming. They are used for comparing the operands. The outcome of using these operators is a **boolean** value. Java has various such operators as given below. The first two are equality operators and the other operators are relational operators.

- The == operator is used to verify whether two operands are equal.
- The != operator is used to verify whether two operands are not equal.
- The > operator is used to verify whether the first operand is greater than the second operand or not.
- The >= operator is used to verify whether the first operand is greater than or equals to the second operand or not.
- The < operator is used to verify whether the first operand is less than the second operand or not.

- The <= operator is used to verify whether the first operand is less than or equals to the second operand or not.
- The **instanceof** operator is used to test whether the object is an instance of the specified type (class or subclass or interface). Discussed later.

Example:

```
int a = 15, b = 6;
System.out.println(a == b); //false
System.out.println(a != b); //true
System.out.println(a > b);  //true
System.out.println(a >= b); //true
System.out.println(a < b);  //false
System.out.println(a <= b); //false
```

8.8. Logical Operators

The logical operators operate on Boolean operand(s) and results the value as true or false. Logical operators in Java are as follows.

- The **logical NOT (!)** operator is used to negate the boolean value i.e. if the value of the Boolean variable is **true** then its negation will be **false**.
- The **logical AND (&&)** operator outputs the result true if both the operands on which the operator is operating are true, false otherwise.
- The **logical OR (||)** operator outputs the result false if both the operands on which the operator is operating are false, true otherwise.

Example:

```
int a = 15, b = 6, c = 5;
System.out.println(!(a > b)); //false
System.out.println(a > b && a < b); //false
System.out.println(a > b || a < b); //true
```

8.9. Bitwise Operators

The purpose of bitwise operators is to operate on the given integral operand (**long**, **int**, **short**, **char**, and **byte**) in low level i.e. bit level to represent the data. The following are the bitwise operators in java.

- The ~ operator is used to negate individual bits of the operand.
- The & operator does the ANDing of individual bits and produces output. The ANDing of two bit results in 1 if both the bits are 1, 0 otherwise.
- The | operator does the ORing of individual bits and produces output. The ORing of two bits results in 0 if both the bits are 0, 1 otherwise.
- The ^ operator does the XORing of individual bits and produces output. The XORing of two bit results in 1 if exactly one of the bits is 1 and the other is 0, 0 otherwise.
- The << operator shifts all of the bits in a value to the left a specified number of times. The top (leftmost) bits due to left shift are lost and the rightmost bits shifted are filled in with 0's.
- The >> operator shifts all of the bits in a value to the right a specified number of times. The top (leftmost) bits due to right shift are filled in with previous content of

the top bit. This is called ***sign extension*** that serves to present the sign of the number shifted.

- The >>> operator shifts all of the bits in a value to right a specified number of times. The leftmost bits due to shifting are 0.
- **Bitwise assignment operators (&=, ^=, |=, <<=, >>=, >>>=):** &= operator is used to assign variable a value obtained by ANDing the individual bits of the previous value of the variable and value of the other expression. For example a &= b; is equivalent to the statement a = a & b; Other assignment operators are similarly interpreted.

Examples: In the examples below, we assume 8 bit (byte) values.

- ~2 = ~(00000010) = 11111101 = -3.
- ~-2 = ~(11111110) = 00000001 = 1.
- 5&2 = 00000101&00000010 = 00000000 = 0.
- -5&-2 = 11111011&11111110 = 11111010 = -6.
- 5|2 = 0000 0101 | 0000 0010 = 00000111 = 7.
- -5|-2 = 11111011|11111110 = 11111111 = -1.
- 5^2 = 00000101 ^ 0000 0010 = 00000111 = 7.
- -5^-2 = 11111011^11111110 = 00000101 = 5.
- 35>>2 = 00100011 >> 2 = 00001000 = 8.
- -5>>1 = 11111011>>1 = 11111101 = -3.
- 35>>>2 = 0010 0011 >> 2 = 0000 1000 = 8.
- -5>>>1 = 11111111111111111111111111111011 >>> 1 = 0111111111111111 1111111111111101 = 2147483645 (this operator is only meaningful for 32 and 64 bit values).
- 35<<2 = 00100011 << 2 = 10001100 = 140.
- -5<<1 = 1111 1011<<1 = 1111 0110 = -10.

8.10. The Conditional Operator (?:)

This operator is **ternary operator** i.e. it requires three operands. The syntax for this operator is ***condition ? statement1:statement2***. Here we interpret like: if condition is true then statement 1 is executed else statement 2 is executed. For e.g. ***int min = (x <= y) ? x : y;*** means if x <= y then min = x, otherwise min = y.

9. Operator Associativity

If we come up with the situation where operators are from the same precedence group then the order of evaluation is provided by associativity. For example, if we have 3 * 5 % 3, what is the result (3 * 5) % 3 or 3 * (5 % 3)? First expression is 15 % 3, which is 0. The second expression is 3 * 2, which is 6. Here, since * and the % operators have the same precedence, precedence does not give the answer. So we take the advantage of associativity and calculate the expression 3 * 5 % 3 as 15 % 3 since * and % are left to right associative. To come out of this kind of confusing situation, we can use parenthesis to define precedence.

10. Expressions

Java **expressions** consist of combination of variables, literals, operators, and method call that result in a single value. For example, a = b + 5 is an expression.
The data type of the value returned by an expression depends on the elements used in the expression. We can also specify exactly how an expression will be evaluated using parenthesis. For example, (a + b) * c.

11. Reading Keyboard Input using Scanner

Java provides various ways to read input from the keyboard. One method is to use Scanner class from java.util package. This class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character. For example,

```
import java.util.Scanner;
public class InputTest
{
        public static void main(String args[])
        {
                int a, b, c;
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter two numbers:");
                a = sc.nextInt();
                b = sc.nextInt();
                c = a + b;
                System.out.println("Sum = " + c);
        }
}
```

12. Control Statements

Like any other programming language, Java has three types of control statements: ***conditional statements***, ***looping statements***, and ***jump statements***.

12.1. Conditional Statements

Conditional statements allow us to decide a statement of code or a block of code to execute depending upon the given condition. These statements are also called **selection statements** or **decision making statements**. Java supports two conditional statements: **if** and **switch**.

12.1.1. The if Statement

This type of statements are used when we have to select a statement or a block of statements if certain condition is satisfied and take some other action if the condition is not satisfied. There are different forms of if statements as given below.

The *if only* Statement

Consider the situation when we have to perform an action only if certain condition is matched and do nothing if the condition is not matched. This situation can be best described by if only statements. The syntax for if only statement is

```
if( condition )
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

If the condition is true then statements in block are executed else nothing is done. For example,

```
if(amount >= 1000)
        discount = amount * 0.05;
```

The *if-else* Statement

When there is a need of executing a statement (or block) if a condition matches and another statement (or block) is to be executed otherwise, then we use if – else statement. It has the general syntax as given below:

```
if (condition)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
else
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

In this situation, if the condition is satisfied the first block is executed, otherwise the second block is executed. For example,

```
if(amount >= 1000)
        discount = amount * 0.05;
else
        discount = amount * 0.03;
```

The if else statement resembles with the conditional operator (?:). Conditional operator is a ternary operator (demands 3 operands), and is used in certain situations, replacing if-else statement. Its general form is:

exp1 ***?*** *exp2* ***:*** *exp3;*

If **exp1** is true, **exp2** is executed and whole expression is assigned with value of **exp2**. If the exp1 is false, then **exp2** is executed, and whole expression is assigned with **exp2** value. For example,

```
c = a > b ? a+b : a-b;
```

is equivalent to

```
if(a>b)
        c=a+b;
else
        c=a-b;
```

The if-else-if ladder Statement

If we have the situation where there are different actions that are executed depending upon different condition with same type of instance then if-else-if is useful. It has general syntax as given below:

```
if (condition 1)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
else if (condition 2)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
…………………………
else if ( condition n)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
else
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

Here if condition1 is true first block of statements is executed, otherwise if condition2 is true second block of statements is executed, for true condition n, nth block of statements is executed and final block of statements is executed if no condition matches. For example,

```
if(amount >= 5000)
        discount = amount * 0.1;
```

```
else if (amount >= 4000)
        discount = amount * 0.07;
else if (amount >= 3000)
        discount = amount * 0.05;
else
          discount = amount * 0.03;
```

12.1.2. The *switch* Statement

It is a multiple branch selection statement, which successively tests the value of an expression against a list of constants. When a match is found, the statement(s) associated with that constant are executed. Its syntax is:

```
switch (expression)
{
case constant1:
        statement(s)
        break;
case constant2:
        statement(s)
        break;
   ...
case constantn:
        statement(s)
        break;
default:
        statement(s)
}
```

A case constant can be

- A constant expression of type char, byte, short, or int
- An enumerated constant
- Starting with Java 7, a string literal

The value of expression is tested against the constants present in the case labels. When a match is found, the statement sequence, if present, associated with that case is executed until the break statement or the end of the switch statement is reached. The statement following default is executed if no matches are found. The default is optional, and if it is not present, no action takes place if all matches fail. For example,

```
public class SwitchTest
{
        public static void main(String[]args)
        {
                int day = Integer.parseInt(args[0]);
                switch(day)
                {
                        case 1:
                                System.out.print("Sunday");
                                break;
                        case 2:
```

```
                    System.out.print("Monday");
                    break;
                case 3:
                    System.out.print("Tuesday");
                    break;
                case 4:
                    System.out.print("Wednesday");
                    break;
                case 5:
                    System.out.print("Thrusday");
                    break;
                case 6:
                    System.out.print("Friday");
                    break;
                case 7:
                    System.out.print("Saturday");
                    break;
                default:
                    System.out.print("Wrong day!");
            }
        }
}
```

Things to remember with switch:

- A switch statement can only be used to test for equality of an expression. We cannot check for other comparisons like <, <=, >, >=
- No two case constants can be same
- Omission of a break statement causes execution to go to next case label
- The default label is executed when no case constants matches the expression value

12.2. Looping Statements

When there is a need of executing a task a specific number of times until a termination condition is met, we use looping statements. These statements are also called **iteration statements** or **repetitive statements**. Java provides three ways of writing looping statements. They are `while` *statement*, `do-while` *statement*, and `for` *statement*.

12.2.1. The *while* Statement

The `while` statement executes the statements as long as the given condition remains true. The general syntax of while statement is:

```
while(condition)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

Here the statements within the brace are executed as long as the condition is `true`. When the `condition` becomes `false`, the `while` loop stops executing these statements and exits out of the loop. For example,

```
int i=1;
while( i <= 10 )
{
        System.out.println( "i=" + i );
        i++;
}
```

Remember: Do not forget to increment i, otherwise loop will never terminate i.e. code goes into infinite loop. So when using loop, remember to guarantee its termination.

12.2.2. The *do-while* Statement

The do while loop always executes its body at least once. So, to guarantee, at least one time, execution of statements do while loop is used. The basic syntax of do while loop is:

```
do
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
} while(condition);
```

Here the statements inside the braces are executed at least once, because its condition is at the bottom of the loop. For example,

```
int i=1;
do
{
        System.out.println( "i=" + i );
        i++;
} while( i <= 10 );
```

12.2.3. The *for* Statement

When we have the fixed number of the iteration known then we use for loop (normally, while loop is also possible). The basic syntax of `for` statement is:

```
for(initialization; condition; increment/decrement )
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

First before loop starts, the initialization statement is executed that initializes the variable or variables in the loop. Second the condition is evaluated, if it is true, the body of the `for` loop will be executed. Finally, the increment/decrement statement will be executed, and the condition will be evaluated, this continues until the condition is false. For example,

```
for(int  i=1;i <= 10; i++)
```

```
        System.out.println( "i=" + i );
```

Remember: Variable declaration in initialization part of the loop is permissible and generally is a good practice. If you have already declared and initialized the variable you could leave the initialization section blank as shown below

```
int i = 1;
for(;i <= 10; i++)
        System.out.println( "i=" + i );
```

We can also include more than one statement in the initialization and increment/decrement portions of the for loop. For example,

```
int a, b;
for (a = 1, b = 4; a < b; a++, b--)
{
        System.out.println("a = " + a);
        System.out.println("b = " + b);
}
```

Note: The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
for ( ; ; )
{
        Statement 1
        Statement 2
        ……………
        Statement n
 }
```

Comparison of Three Loops:

Depending on the position of the test expression (condition) in the loop, a loop may be classified either as the **entry-controlled loop** or **exit-controlled loop**.

In the entry-controlled loop, the test expressions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. Examples are *while* and *for* loops.

In case of an exit-controlled loop, the test expression is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time and conditionally after that. Example is *do – while* loop.

Nested Control Statements

We can nest any control flow statement (conditional and iteration statement) within another to any depth. When one loop is nested inside another loop, the inner loop is first terminated and again restarted when the first loop starts for the next incremented value. The outer loop is terminated last. For example,

```
for(int i = 1; i<=2;i++)
{
          System.out.println("i = " + i);
          for(int j=1; j<=3;j++)
                    System.out.println("j = " + j);
}
```

We can also nest a conditinal statemen inside a looping statement as given in the example below.

```
for(int i = 1; i<=100; i++)
{
		if(i % 5 == 0 && i % 7 == 0)
			System.out.println(i);
}
```

12.3. Jump Statements

Java, for the transfer of control from one part to another, supports three jump statements: **break**, **continue**, and **return**.

12.3.1. The *break* Statement

The use of break statement causes the immediate termination of the loop (all type of loops) and switch from the point of break statement and the passage program control to the statements following the loop. In case of nested loops if break statement is in inner loop only the inner loop is terminated. For example,

```
for( int i=1; i<= 10; i++ )
{
	if( i == 5 )
		break;
	System.out.print( " i=" + i );
}
```

12.3.2. The *continue* Statement

Continue statement causes the execution of the current iteration of the loop to cease, and then continue at the next iteration of the loop. For example,

```
for( int i=1; i<= 10; i++ )
{
	if( i == 5 || i == 7 )
		continue;
	System.out.println("i = " + i);
}
```

12.3.3. The *return* Statement

It is used to transfer the program control back to the caller of the method. The general form of this statement is ***return expression;*** which returns the value to the caller of the method. For example,

```
class ReturnTest
{
	public static int add(int a, int b)
	{
		return a + b;
	}
	public static void main(String[]args)
	{
```

```
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int c = add(a, b);
               System.out.print("Sum = " + c);
        }
}
```

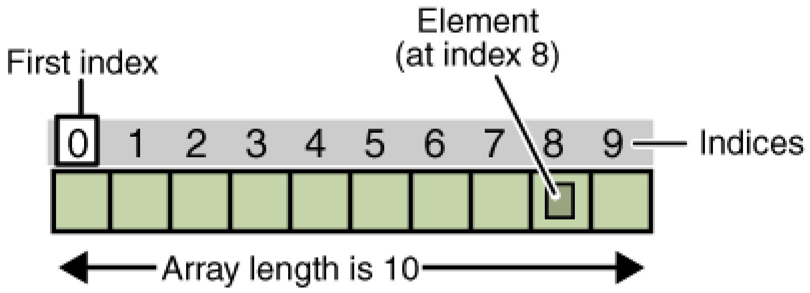
13. Array

An array is a group of contiguous same type variables that are referred to by a common name. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an *element*, and each element is accessed by its numerical *index*.

Array offers a convenient means of grouping related information since array is the collection of same type of data elements. Arrays of any type (primitive and non-primitive) can be created and may have one or more dimensions.

13.1. One-Dimensional Array

In one-dimensional arrays, a list of items can be given one variable name using only one subscript. This type of array is also called *single-subscripted* variable. To access an element in this array we use single index (subscript) value and in Java this index value starts with 0. Hence, the 9th element is accessed at index 8. The figure below shows an array of 10 elements.



<u>Declaration</u>:

The general form of a one-dimensional array declaration is

type[] arrayName; **or** *type arrayName [];*

For example,

int intArray []; **or** *int [] intArray;*

We can declare several arrays at the same time as follows:

int[]nums1, nums2, nums3; **or** *int nums1[], nums2[], nums3[];*

<u>Creation</u>:

After declaring an array, we need to create it in the physical memory. Java allows us to create arrays using **new** operator only, as shown below:

arrayName = new type[size];

For example,

intArray = new int[10]; *//array of int with 10 elements*

Note: It is also possible to combine the two steps – declaration and creation – into one as shown below:

int[]intArray = new int[10];

<u>Initialization</u>:
When an array is created we can initialize it after the creation or within the declaration itself. Within the declaration, we can initialize it as shown follows:
int [] intArray = {10, 20, 31, 45, 56};
This is initialization at the declaration time and at this situation the size of the array is 5. Remember all the elements are enclosed within the braces separated by commas.
Initialization after the creation is done like this:

```
int [] intArray = new int[5];
intArray[0] = 10;
intArray[1] = 20;
intArray[2] = 31;
intArray[3] = 45;
intArray[4] = 56;
```

<u>Accessing Array Elements</u>:
Each array element is accessed by its numerical index. For example,

```
int[]intArray = {10, 20, 31, 45, 56};
System.out.println(intArray[0]);
System.out.println(intArray[2]);
System.out.println(intArray[3]);
```

Output:

```
10
31
45
```

<u>Array Length</u>:
In Java, all arrays store the allocated size in a variable named **length**. We can access the length of the array **intArray** using **intArray.length** and the value of **intArray.length** in the above example is 5.

<u>Example</u>:

```
class ArrayTest
{
	public static void main(String[]args)
	{
		int[]nums = {8, 4, 15, 20, 7, 29};
		int sum = 0;
		for(int i = 0; i < nums.length; i++)
			sum += nums[i];
		System.out.print("Sum = " + sum);
	}
}
```

Output:

```
Sum = 83
```

<u>The for-each Loop</u>:

Java has a powerful looping construct that allows you to loop through each element in an array (or any other collection of elements) without having to fuss with index values. Its syntax is:

for (*variable* : *collection*) *statement*

For example,

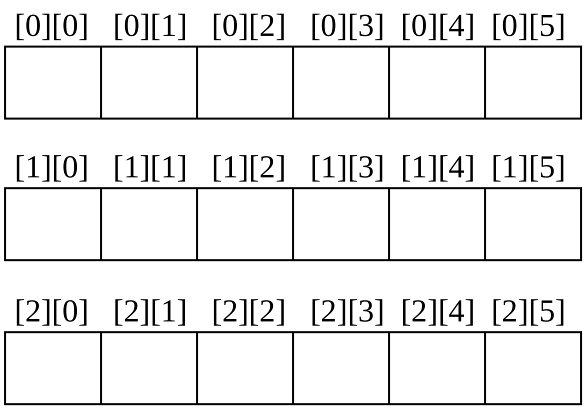
```
public class ArrayTest
{
   public static void main(String[]args)
   {
      int[]nums = {8, 4, 15, 20, 7, 29};
      int sum = 0;
      for(int i:nums)
         sum += i;
      System.out.print("Sum = " + sum);
   }
}
```

13.2. Multi-Dimensional Array

Multidimensional arrays are actually arrays of arrays. The declaration, creation and initialization schemes are similar to the one-dimensional arrays but with subtle differences. For multidimensional arrays, we specify each additional index using another set of square brackets. For example, we can create, initialize and access two-dimensional array as follows:

int array2D [][] = new int [3][6];   //array creation

This code creates an array of dimension 3 by 6 as shown below.



We can initialize two-dimensional arrays as follows:

char charArray2D[2][3] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’};

Or equivalently,

char charArray2D[][] = {{‘a’, ‘b’, ‘c’}, {‘d’, ‘e’, ‘f’}};

To access the array we use the same subscript form but there is slight difference in that the two-dimensional array starts from inner subscripts to outer one. For example, charArray2D[0][0].

Example Piece of Code:

char charArray2D[][] = {{‘a’, ‘b’, ‘c’}, {‘d’, ‘e’, ‘f’}};

```
for(int i = 0; i < 2; i++)   //for inner subscripts
{
        for(int j = 0; j < 3; j++)   //for outer subscripts
                System.out.print(charArray2D[i][j]);
        System.out.println();
}
```

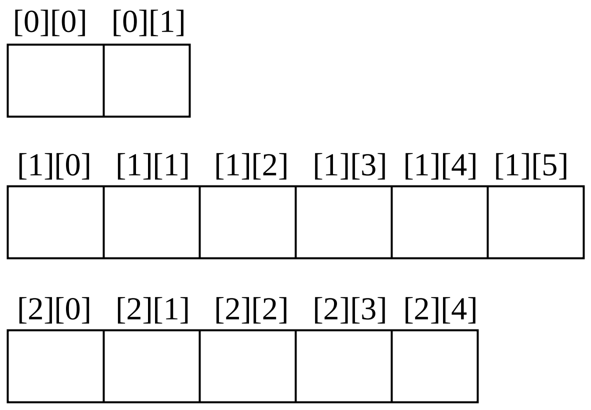
Output:

```
abc
def
```

When multidimensional array is created it is not necessary to define memory for all dimensions. The definition for the first (leftmost) dimension is sufficient. For example,

```
String str2D [][] = new String[3][];
str2D[0] = new String[2];
str2D[1] = new String[6];
str2D[2] = new String[5];
```

This code creates the following array called **ragged/jagged arrays**. Ragged arrays are the arrays in which different rows have different lengths.



Exercises

1. Write a program to find area and perimeter of rectangle. Use Scanner class to read input for length and breadth.
2. Write a program to calculate discount on the basis of following assumption:
   a) If purchased amount is greater than or equal to 1000, discount is 5%
3. Write a program to calculate discount on the basis of following assumption:
   a) If purchased amount is greater than or equal to 1000, discount is 5%
   b) If purchased amount is less than 1000, discount is 3%
4. Write a program to calculate discount on the basis of following assumption:
   a) If purchased amount is greater than or equal to 5000, discount is 10%
   b) If purchased amount is greater than or equal to 4000 and less than 5000, discount is 7%
   c) If purchased amount is greater than or equal to 3000 and less than 4000, discount is 5%
   d) If purchased amount is greater than or equal to 2000 and less than 3000, discount is

3%

e) If purchased amount is less than 2000, discount is 2%

5. Write a program to check whether a number is even or odd.
6. Write a program to compare two numbers.
7. Write a program to find the largest number among four numbers.
8. Write a program to calculate the simple interest on the basis of following assumption:
   b) If balance is greater than or equal to 10000, interest is 7 %
   c) If balance is greater than or equal to 50000 and less than 100000 interest is 5 %
   d) If balance is less than 50000, interest is 3%
9. Admission to a professional course is subject to the following conditions:

a) Marks in mathematics	>=60
b) Marks in physics	>=50
c) Marks in chemistry	>=40
d) Total in all three subjects	>=200

   Or

   Total in mathematics and physics >=150

   Given the marks in three subjects, write a program to process the applications to list eligible candidates.
10. The rates of tax on gross salary are as shown below:

Income	**Tax**
Less than 10,000	Nill
Rs. 10,000 to 19,999	10%
Rs. 20,000 to 39,999	15%
Rs. 40,000 to above	20%

   Write a program to compute the net salary after deducting the tax for the given information and print the same.
11. Jet Company gives 5% commission to its salesman if their monthly sales are less than Rs. 10,000 and a 10% commission if it is equal to or greater than Rs. 10,000. Write a program to calculate commission at the end of the month.
12. Write a program using switch statement to display EXCELLENT, VERY GOOD, GOOD, SATISFACTORY, or FAIL if the user enters A, B, C, D, or E respectively.
13. Write a program to display number of days in a month using switch statement.
14. Write a program using switch statement to develop a simple calculator for +, -, *, /, and % operators.
15. Write a program to display your name 10 times.
16. Write a program to display first n natural numbers. Also display their sum, product, and average.
17. Write a program that displays the temperatures from 0 degrees Celsius to 100 degrees Celsius and its Fahrenheit equivalent. [Hint: F = C * 9/5 + 32]
18. Write a program that displays all prime numbers from 1 to 100.
19. A palindrome is a sequence of characters that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write an application that reads in a five-digit integer and determines whether it is a palindrome. If the number is not five digits long, display an error message and allow the user to enter a new value.
20. Write the program to determine the sum of the following harmonic series for a given value of n (input from command line).

1 + 1/2 + 1/3 + 1/4 + … + 1/n.

21. The straight-line method of computing the early depreciation of the value of an item is given by

    Depreciation = (purchase price – salvage value)/years of service.

    Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.
22. Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.
23. Write a program to reverse the digits of a whole number. [Hint: Use modulus operator to extract the last digit and integer division by 10 to get the n – 1 digit number from the n digit number]
24. Write a program that computes the sum of the digits of the given integer number.
25. Write a program to find factorial of a number. [Hint: factorial n = n! = n * (n – 1) * (n – 2) *…* 2 * 1]
26. The number in the sequence 1 1 2 3 5 8 13 21 … are called Fibonacci numbers. Write a program to find nth Fibonacci number. [Hint: nth Fibonacci number is sum of (n-1)st and (n-2)nd Fibonacci numbers]
27. Write a program to print the following outputs using for loops

```
1                 $ $ $ $ $            1            1
2 2                 $ $ $ $           2 2           0 1
3 3 3                 $ $ $          3 3 3          1 0 1
4 4 4 4                 $ $         4 4 4 4         0 1 0 1
5 5 5 5 5                 $        5 5 5 5 5        1 0 1 0 1
```

28. Write a program to print the Floyd's triangle as shown below

```
1
2 3
4 5 6
7 8 9 10
11……..15
.
.
79………………91
```

29. Write a program to display the following menu
    a. To find area of circle
    b. To check the given number is odd or even
    c. To find the sum of N numbers
    d. Exit

Perform above task until the user wants to exit.

30. Write a program find sum, product, and average of 10 floating point numbers stored in an array.
31. Given a list of marks of 20 students ranging from 0 to 100, write a program to compute and print the number of students who have obtained marks
    a. in the range 81 to 100
    b. in the range 61 to 80
    c. in the range 41 to 60, and
    d. in the range 0 to 40
32. Write a program to find the minimum and maximum among 10 numbers.

33. Write a program to add elements stored in odd indices in an array of integer.
34. Write a program to add two matrices.
35. Write a program that accepts the elements of 3×3 matrix and calculate the sum of all elements of the matrix.
36. Write a program to read 4×4 matrix and find sum of each row.
37. Write a program to find transpose of a matrix.
38. Write a program to multiply two matrices and display the resultant matrix.