Kanal: **@Uzb_Dasturchilar** | Muallif: Qudrat Abdurahimov

1] dars

Create your first Java program

Let's start by compiling and running the following short sample program.

```
/*
 This is a simple Java program. Call this file "Example.java".
 */

public class Example {
  // Your program begins with a call to main().
  public static void main(String args[]) {
    System.out.println("Java.");
  }
}
```

In Java, a source file is called a compilation unit. It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the .java filename extension.

In Java, all code must reside inside a class. By convention, the name of the public class should match the its file name. And Java is case-sensitive.

Compiling the Program

To compile the program, execute the compiler, `javac`, specifying the name of the source file on the command line:

```
C:\>javac Example.java
```

The `javac` compiler creates a file called `Example.class`. `Example.class` contains the bytecode version of the program.

To run the program, use the Java interpreter, called `java`. Pass the class name `Example` as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
Java.
```

When Java source code is compiled, each individual class is put into its own file named `classname.class`.

A Closer Look at the Example.java

Here is the code we just created.

```
/*
 This is a simple Java program. Call this file "Example.java".
 */

public class Example {
  // Your program begins with a call to main().
  public static void main(String args[]) {
    System.out.println("Java.");
  }
}
```

The first part is a comment.

```
/*
   This is a simple Java program. Call this file "Example.java".
*/
```

Comment is a remark for a program. The contents of a comment are ignored by the compiler. The next line of code in the program is shown here:

```
public class Example {
```

The keyword `class` declares that a new class is being defined.

`Example` is the name of the class.

The entire class definition is between the opening curly brace ({) and the closing curly brace (}). The next line in the program is the single-line comment, shown here:

```
// Your program begins with a call to main().
```

A single-line comment begins with a `//` and ends at the end of the line. The next line of code is shown here:

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

Java applications begin execution by calling `main(String args[])`. Java is case-sensitive. Thus, `Main` is different from `main`.

!!!
2] dars

Keywords and Identifiers

A keyword is a word whose meaning is defined by the programming language.

Java Keywords and Reserved Words:

```
abstract    class       extends     implements    null        strictfp        true
assert      const       false       import        package     super           try
boolean     continue    final       instanceof    private     switch          void
break       default     finally     int           protected   synchronized
volatile
byte        do          float       interface     public      this            while
case        double      for         long          return      throw
catch       else        goto        native        short       throws
char        enum        if          new           static      transient
```

An identifier is a word used by a programmer to name a variable, method, class, or label.

Keywords and reserved words may not be used as identifiers.

An identifier must begin with a letter, a dollar sign ($), or an underscore (_); subsequent characters may be letters, dollar signs, underscores, or digits.

Some examples are:

```
foobar          // legal
Myclass         // legal
$a              // legal
3_a             // illegal: starts with a digit
!theValue       // illegal: bad 1st  char
```

Identifiers are case sensitive.

For example, `myValue` and `MyValue` are distinct identifiers

!!!

3] dars

The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

Primitive Type	Reserved Word	Size	Min Value	Max Value
Boolean	boolean	N/A	N/A	N/A
Character	char	16-bit	Unicode 0	Unicode 2^{16} - 1
Byte integer	byte	8-bit	-128	+127
Short integer	short	16-bit	-2^{1}	$+2^{15}$ - 1
Integer	int	32-bit	-2^{31}	$+2^{31}$ - 1
Long integer	long	64-bit	-2^{63}	$+2^{63}$ - 1
Floating-point	float	32-bit	1.4e-045	3.4e+038
Double precision floating-point	double	64-bit	4.9e-324	1.8e+308

byte, short, int, and long are for whole-valued signed numbers. float and double are fractional precision numbers.

char represents symbols in a character set, like letters and numbers. boolean represents true/false values.

Integers

Java defines four integer types: byte, short, int, and long.

Integer types are signed, positive and negative values.

The width and ranges of these integer types vary widely:

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

Floating Point Types

There are two kinds of floating-point types: float and double. float type represents single-precision numbers. double type stores double-precision numbers.

Floating-Point Types width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

!!

4] dars

byte

The smallest integer type is byte. byte is a signed 8-bit type that has a range from -128 to 127. byte type variables are useful when working with a stream of data from a network or file.

Byte variables are declared by use of the byte keyword. The following declares two byte variables called b and c:

```
byte b, c;
```

short

short is a signed 16-bit type. short type variable has a range from -32,768 to 32,767.

Here are some examples of short variable declarations:

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

int

int is a signed 32-bit type that has a range from -2,147,483,648 to 2,147,483,647.

When byte and short values are used in an expression they are promoted to int when the expression is evaluated.

octal (base eight)

Octal values are denoted in Java by a leading zero. valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range.

```
public class Main {

  public static void main(String[] args) {
    int i = 010;

    System.out.println(i);
  }
}
```

The output:

```
8
```

hexadecimal (base 16).

hexadecimal matches with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (0x or 0X).

The range of a hexadecimal digit is 0 to 15, so A through F (or a through f) are substituted for 10 through 15.

An integer literal can always be assigned to a long variable. An integer can also be assigned to a char as long as it is within range.

```
public class Main{
```

```
  public static void main(String[] argv){
    int f = 0XFFFFF;

    System.out.println(f);//1048575

  }
}
```

long

long is a signed 64-bit type and is used when an int type is not large enough. The range of long type is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

For example, here is a program that use long type to store the result.

```
public class Main {
    public static void main(String args[]) {
        long result= (long)Integer.MAX_VALUE * (long)10;
        System.out.println(result);//21474836470

    }
}
```

The result could not have been held in an int variable.

To specify a long literal, you need to tell the compiler that the literal value is of type long by appending an upper- or lowercase L to the literal.

For example, 0x7ffffffffffffffL or 123123123123L.

```
public class Main {
  public static void main(String args[]) {
    long l = 0x7ffffffffffffffL;

    System.out.println("l is " + l);
  }
}
```

The output generated by this program is shown here:

```
l is 576460752303423487
```

float

float type represents single-precision numbers. float is 32-bit width and its range is from 1.4e-045 to 3.4e+038 approximately.

float type variables are useful when you need a fractional component. Here are some example float variable declarations:

```
float high, low;
```

float literals

Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant.

```
public class Main {
    public static void main(String args[]) {
        float d =  3.14159F;
        System.out.print(d);//3.14159
    }
}
```

Java's floating-point calculations are capable of returning +infinity, -infinity, +0.0, -0.0, and NaN

```
public class Main {
  public static void main(String[] args) {
    Float f1 = new Float(Float.NaN);
    System.out.println(f1.floatValue());

    Float f2 = new Float(Float.NaN);
    System.out.println(f2.floatValue());
    System.out.println(f1.equals(f2));
    System.out.println(Float.NaN == Float.NaN);
    System.out.println();

  }
}
```

double

double represents double-precision numbers. double is 64-bit width and its range is from 4.9e-324 to 1.8e+308 approximately.

Here is a program that uses double variables to compute the area of a circle:

```
public class Main {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8888; // radius of circle
    pi = 3.1415926; // pi, approximately
    a = pi * r * r;
```

```
    System.out.println("Area of circle is " + a);
  }
}
```

The output:

```
Area of circle is 372.4859596381597
```

double type Literals

double type numbers have decimal values with a fractional component.

They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.

For example, 2.0, 3.14159, and 0.6667.

```
public class Main {
    public static void main(String args[]) {
        double d =  3.14159;
        System.out.print(d);//3.14159
    }
}
```

Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.

The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative.

For example, 6.02E23, 314159E-05, and 4e+100.

```
public class Main {
  public static void main(String[] argv) {
    double d1 = 6.022E23;
    double d2 = 314159E-05;
    double d3 = 2e+100;

    System.out.println("d1 is " + d1);
    System.out.println("d2 is " + d2);
    System.out.println("d3 is " + d3);
  }

}
```

The output generated by this program is shown here:

```
d1 is 6.022E23
d2 is 3.14159
d3 is 2.0E100
```

You can explicitly specify a double literal by appending a D or d.

```
public class Main {
    public static void main(String args[]) {
        double d =  3.14159D;
        System.out.print(d);//3.14159
    }
}
```

Java's floating-point calculations are capable of returning +infinity, -infinity, +0.0, -0.0, and NaN

dividing a positive number by 0.0 returns +infinity. For example, System.out.println(1.0/0.0); outputs Infinity.

```
public class Main{
  public static void main(String[] args) {
    System.out.println(1.0/0.0);
  }

}
```

Infinity

Dividing a negative number by 0.0 outputs -infinity. For example, System.out.println(-1.0/0.0); outputs -Infinity.

```
public class Main{
  public static void main(String[] args) {
    System.out.println(-1.0/0.0);
  }

}
```

Output:

```
-Infinity
```

Dividing 0.0 by 0.0 returns NaN. square root of a negative number is NaN

For example, System.out.println(0.0/0.0) and System.out.println(Math.sqrt(-1.0)) output NaN.

Dividing a positive number by +infinity outputs +0.0. For example, System.out.println(1.0/(1.0/0.0)); outputs +0.0.

Dividing a negative number by +infinity outputs -0.0. For example, System.out.println(-1.0/(1.0/0.0)); outputs -0.0.

```
public class Main {
  public static void main(String[] args) {
```

```
    Double d1 = new Double(+0.0);
    System.out.println(d1.doubleValue());

    Double d2 = new Double(-0.0);
    System.out.println(d2.doubleValue());
    System.out.println(d1.equals(d2));
    System.out.println(+0.0 == -0.0);


  }
}
```

The following code parses command-line arguments into double precision floating-point values

```
public class Main {
  public static void main(String[] args) {
    if (args.length != 3) {
      System.err.println("usage: java Main value1 op value2");
      System.err.println("op is one of +, -, *, or /");
      return;
    }
    try {
      double value1 = Double.parseDouble(args[0]);
      double value2 = Double.parseDouble(args[2]);
      if (args[1].equals("+")) {
        System.out.println(value1 + value2);
      } else if (args[1].equals("-")) {
        System.out.println(value1 - value2);
      } else if (args[1].equals("*")) {
        System.out.println(value1 * value2);
      } else if (args[1].equals("/")) {
        System.out.println(value1 / value2);
      } else {
        System.err.println("invalid operator: " + args[1]);
      }
    } catch (Exception nfe) {
      System.err.println("Bad number format: " + nfe.getMessage());
    }
  }
}
```

char

In Java, char stores characters.

Java uses Unicode to represent characters. Unicode can represent all of the characters found in all human languages.

Java char is a 16-bit type. The range of a char is 0 to 65,536.

There are no negative chars.

More information about Unicode can be found at http://www.unicode.org.

Here is a program that demonstrates char variables:

```
public class Main {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88; // code for X

    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);//ch1 and ch2: X Y
  }
}
```

ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X.

char can be used as an integer type and you can perform arithmetic operations.

```
public class Main {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);//ch1 contains X

    ch1 = (char)(ch1 + 1); // increment ch1
    System.out.println("ch1 is now " + ch1);//ch1 is now Y
  }
}
```

char Literals

Characters in Java are indices into the Unicode character set. character is represented inside a pair of single quotes.

For example, 'a', 'z', and '@'.

```
public class Main {
  public static void main(String[] argv) {
    char ch = 'a';

    System.out.println("ch is " + ch);//ch is a

  }
}

public class Main {
  public static void main(String[] argv) {
    char ch = '@';

    System.out.println("ch is " + ch);//ch is @
    ch = '#';
```

```
    System.out.println("ch is " + ch);//ch is #
    ch = '$';

    System.out.println("ch is " + ch);//ch is $
    ch = '%';

    System.out.println("ch is " + ch);//ch is %

  }
}
```

The escape sequences are used to enter impossible-to-enter-directly characters.

'\'' is for the single-quote character.

'\n' for the newline character.

For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'.

For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, '\u0061' is the ISO-Latin-1 'a' because the top byte is zero.

'\ua432' is a Japanese Katakana character.

The following table shows the character escape sequences.

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

```
public class Main {
  public static void main(String[] argv) {
    char ch = '\'';

    System.out.println("ch is " + ch);//ch is '

  }
}
```

boolean

Java has a boolean type for logical values. It can have only one of two possible values, true or false.

This is the type returned by all relational operators.

Here is a program that demonstrates the boolean type:

```
public class Main {
  public static void main(String args[]) {
    boolean boolVariable;
    boolVariable = false;
    System.out.println("b is " + boolVariable);
    boolVariable = true;
    System.out.println("b is " + boolVariable);

    if (boolVariable) {
      System.out.println("This is executed.");
    }

    boolVariable = false;
    if (boolVariable) {
      System.out.println("This is not executed.");
    }
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

Output:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Boolean Literals

Boolean literals are only two logical values: true and false. The values of true and false do not convert into any numerical representation.

The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean.

```
public class Main {
  public static void main(String[] argv) {
    boolean b = true;

    int i = b;

  }
}
```

If you try to compile the program, the following error message will be generated by compiler.

```
Main.java:5: incompatible types
found   : boolean
required: int
    int i = b;
            ^
1 error
```

!!

5] dars

Variable stores value in a Java program

A variable is defined by an identifier, a type, and an optional initializer. The variables also have a scope(visibility / lifetime).

Declaring a Variable

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

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

- `type` could be `int` or `float`.
- `identifier` is the variable's name.
- Initialization includes an equal sign and a value.

To declare more than one variable of the specified type, use a comma-separated list.

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing d and f.
```

The following variables are defined and initialized in one expression.

```
public class Main {
  public static void main(String[] argv) {
    byte z = 2; // initializes z.
    double pi = 3.14; // declares an approximation of pi.
    char x = 'x'; // the variable x has the value 'x'.
  }
}
```

Variable cannot be used prior to its declaration.

```
public class Main {
  public static void main(String[] argv) {
```

```
    count = 100; // Cannot use count before it is declared!
    int count;
  }
}
```

Compiling the code above generates the following error message:

```
D:\>javac Main.java
Main.java:4: cannot find symbol
symbol  : variable count
location: class Main
    count = 100; // Cannot use count before it is declared!
    ^
1 error
```

Operators

Java operators can be divided into the following four groups:

- arithmetic,
- bitwise,
- relational,
- logical.

Operators are classified as prefix, postfix, and infix.

- A prefix operator is a unary operator that precedes its operand, for example -6.
- A postfix operator is a unary operator that trails its operand, for example x++.
- An infix operator is a binary or ternary operator that is placed between the binary operator's two or the ternary operator's three operands, for example x + 5

!!

6] dars

Arithmetic Operators

Arithmetic operators are used in mathematical expressions.

The following table lists the arithmetic operators:

Operator	Result
+	Addition
-	Subtraction (unary minus)
*	Multiplication
/	Division

Operator	Result
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use arithmetic operators on boolean types, but you can use them on char types.

The Basic Arithmetic Operators

The basic arithmetic operations are addition, subtraction, multiplication, and division. They behave as you would expect. The minus operator also has a unary form which negates its single operand.

```
public class Main {

  public static void main(String args[]) {

    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

  }
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
```

Floating Point Arithmetic

The modulus operator, %, returns the remainder of a division operation.

```java
public class Main {
  public static void main(String args[]) {
    int x = 42;

    System.out.println("x mod 10 = " + x % 10);
  }
}
```

When you run this program you will get the following output:

```
x mod 10 = 2
```

The modulus operator can be applied to floating-point types as well as integer types.

```java
public class Main {
  public static void main(String args[]) {
    double y = 42.25;

    System.out.println("y mod 10 = " + y % 10);
  }
}
```

When you run this program you will get the following output:

```
y mod 10 = 2.25
```

The following code uses the modulus operator to calculate the prime numbers:

```java
  public class Main {
  public static void main(String[] args) {
    int limit = 100;
    System.out.println("Prime numbers between 1 and " + limit);
    for (int i = 1; i < 100; i++) {
      boolean isPrime = true;
      for (int j = 2; j < i; j++) {
        if (i % j == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime)
        System.out.print(i + " ");
    }
  }
}
```

The output:

```
Prime numbers between 1 and 100
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Arithmetic Compound Assignment Operators

Statements like the following

```
a = a + 4;
```

can be rewritten as

```
a += 4;
```

Both statements perform the same action: they increase the value of a by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

In this case, the %= obtains the remainder of a/2 and puts that result back into a.

Any statement of the form

```
var = var op expression;
```

can be rewritten as

```
var op= expression;
```

Here is a sample program that shows several op= operator assignments:

```
public class Main {

  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 1;
    b *= 2;
    c += a * b;
    c %= 3;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);

  }
}
```

The output of this program is shown here:

```
a = 2
b = 4
c = 2
```

Increment and Decrement

- `++` and `--` are Java's increment and decrement operators.
- The increment operator, `++`, increases its operand by one.
- The decrement operator, `--`, decreases its operand by one.

For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

This statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

The increment and decrement operators are unique in that they can appear both in postfix form and prefix form.

In the postfix form they follow the operand, for example, i++.
In the prefix form, they precede the operand, for example, --i.

The difference between these two forms appears when the increment and/or decrement operators are part of a larger expression.

In the prefix form, the operand is incremented or decremented before the value is used in the expression. In postfix form, the value is used in the expression, and then the operand is modified.

Examples of Pre-and Post- Increment and Decrement Operations

Initial Value of x	Expression	Final Value of y	Final Value of x
5	y = x++	5	6
5	y = ++x	6	6
5	y = x--	5	4

Initial Value of x	Expression	Final Value of y	Final Value of x
5	y = --x	4	4

For example:

```
x = 42;
y = ++x;
```

y is set to 43, because the increment occurs before x is assigned to y. Thus, the line

```
y = ++x;
```

is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of x is obtained before the increment operator is executed, so the value of y is 42.

In both cases x is set to 43. The line

```
y = x++;
```

is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

The following program demonstrates the increment operator.

```
public class Main {

  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = ++b;
    int d = a++;

    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);

  }
}
```

The output of this program follows:

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

Bitwise Operators

Java bitwise operators can be applied to the integer types: `long`, `int`, `short`, `char`, `byte`. Bitwise Operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Left Shift

It has this general form:

```
  value << num
```

The following code shifts byte type variable.

```
public class Main {
  public static void main(String args[]) {
    byte a = 64, b;
    int i;
    i = a << 2;
    b = (byte) (a << 2);
    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
```

```
  }
}
```

The output generated by this program is shown here:

```
Original value of a: 64
i and b: 256 0
```

Each left shift has the effect of doubling the original value. The following program illustrates this point:

```
public class Main {
  public static void main(String args[]) {
    int num = 0xFFFFFFF;

    for (int i = 0; i < 4; i++) {
      num = num << 1;
      System.out.println(num);

    }
  }
}
```

The program generates the following output:

```
536870910
1073741820
2147483640
-16
```

The Right Shift

<![CDATA[The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:]]>

```
  value >> num
```

num specifies the number of positions to right-shift.

The following code fragment shifts the value 32 to the right by two positions:

```
public class Main {
  public static void main(String[] argv) {

    int a = 32;
    a = a >> 2;
    System.out.println("a is " + a);
```

```
  }
}
```

The output:

```
a is 8
```

The Unsigned Right Shift

<![CDATA[Java's unsigned, shift-right operator, >>>, always shifts zeros into the high-order bit.]]>

```java
public class Main {
  public static void main(String[] argv) {
    int a = -1;
    a = a >>> 24;

    System.out.println("a is " + a);
  }
}
```

The output:

```
a is 255
```

Bitwise Operator Assignments

Bitwise operator assignments combines the assignment with the bitwise operation. The following two statements are equivalent:

```java
a = a >> 4;
a >>= 4;
```

The following two statements are equivalent:

```java
a = a | b;
a |= b;
```

The following program demonstrates the bitwise operator assignments:

```java
public class Main {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;
    a |= 2;
```

```
    b >>= 2;
    c <<= 2;
    a ^= c;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);

  }
}
```

The output of this program is shown here:

```
a = 15
b = 0
c = 12
```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```
public class Main {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println(" a = " + a);
    System.out.println(" b = " + b);
    System.out.println(" a|b = " + c);
    System.out.println(" a&b = " + d);
    System.out.println(" a^b = " + e);
    System.out.println("~a&b|a&~b = " + f);
    System.out.println(" ~a = " + g);

  }
}
```

Here is the output from this program:

```
 a = 1
 b = 2
 a|b = 3
 a&b = 0
 a^b = 3
~a&b|a&~b = 3
 ~a = 14
```

Relational Operators

The relational operators determine the relationship between two operands. The relational operators are:

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

For example, the following code fragment is perfectly valid:

```
public class Main {
  public static void main(String[] argv) {
    int a = 4;
    int b = 1;
    boolean c = a < b;

    System.out.println("c is " + c);
  }
}
```

The result of a < b (which is false) is stored in c.

```
c is false
```

The outcome of a relational operator is a boolean value.

In the following code, the `System.out.println` outputs the result of a relational operator.

```
public class Main {
  public static void main(String args[]) {
    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
10 > 9 is true
```

Boolean Logical Operators

The Boolean logical operators operate only on boolean operands.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
? :	Ternary if-then-else

The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

The following program demonstrates the boolean logical operators.

```
public class Main {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println(" a = " + a);
    System.out.println(" b = " + b);
    System.out.println(" a|b = " + c);
    System.out.println(" a&b = " + d);
    System.out.println(" a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
```

```
    System.out.println(" !a = " + g);

  }
}
```

The output:

```
 a = true
 b = false
 a|b = true
 a&b = false
 a^b = true
!a&b|a&!b = true
 !a = false
```

Short-Circuit Logical Operators

The OR operator results in true when one operand is true, no matter what the second operand is.
The AND operator results in false when one operand is false, no matter what the second operand is.
If you use the || and &&, Java will not evaluate the right-hand operand when the outcome can be determined by the left operand alone.

The following code shows how you can use short-circuit logical operator to ensure that a division operation will be valid before evaluating it:

```
public class Main {
  public static void main(String[] argv) {
    int denom = 0;
    int num = 3;
    if (denom != 0 && num / denom > 10) {
      System.out.println("Here");
    } else {
      System.out.println("There");
    }
  }
}
```

The output:

```
There
```

The following code uses a single & ensures that the increment operation will be applied to `e` whether `c` is equal to 1 or not.

```
public class Main {
  public static void main(String[] argv) {
    int c = 0;
    int e = 99;
    int d = 0;
    if (c == 1 & e++ < 100)
      d = 100;
```

```
    System.out.println("e is " + e);
    System.out.println("d is " + d);
  }
}
```

The output:

```
e is 100
d is 0
```

The ? Operator

The ? operator is a ternary (three-way) operator. The ? has this general form:

```
expression1 ? expression2 : expression3
```

- `expression1` can be any expression that evaluates to a `boolean` value.
- If expression1 is `true`, then `expression2` is evaluated.
- Otherwise, `expression3` is evaluated.

The expression evaluated is the result of the `?` operation.

Both `expression2` and `expression3` are required to return the same type, which can't be void. Here is an example of ? operator:

```
public class Main {
  public static void main(String[] argv) {
    int denom = 10;
    int num = 4;
    double ratio;

    ratio = denom == 0 ? 0 : num / denom;
    System.out.println("ratio = " + ratio);
  }
}
```

The output:

```
ratio = 0.0
```

Here is another program that demonstrates the `?` operator. It uses it to obtain the absolute value of a variable.

```
public class Main {
  public static void main(String args[]) {
    int i, k;
```

```
    i = 10;
    k = i < 0 ? -i : i;
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i;
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

The following table shows the order of precedence for Java operators, from highest to lowest.

1. () [] . (Highest)
2. ++ - - ~ !
3. * / %
4. + -
5. <![CDATA[>> >>> <<]]>
6. <![CDATA[>>= < <=]]>
7. == !=
8.
9. ^
10. |
11.
12. ||
13. ?:
14. = op= (Lowest)

Operator Precedence

The following table shows the order of precedence for Java operators, from highest to lowest.

1. () [] . (Highest)
2. ++ - - ~ !
3. * / %

4. + -
5. <![CDATA[>> >>> <<]]>

6. <![CDATA[>>= < <=]]>

7. == !=
8.
9. ^
10. |
11.
12. ||
13. ?:
14. = op= (Lowest)

!!

Simplest if statement

Java has two selection statements: `if` and `switch`. Simplest if statement form is shown here:

```
if(condition)
   statement;
```

`condition` is a boolean expression.

If `condition` is `true`, then the statement is executed.
If `condition` is `false`, then the statement is bypassed.

Here is an example:

```
public class Main {

  public static void main(String args[]) {
    int num = 99;
    if (num < 100) {
      System.out.println("num is less than 100");

    }
  }
}
```

The output generated by this program is shown here:

```
num is less than 100
```

Using `if` statement to compare two variables

```
public class Main {

  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if (x < y){
      System.out.println("x is less than y");
    }

    x = x * 2;
    if (x == y){
      System.out.println("x now equal to y");
    }

    x = x * 2;
    if (x > y){
      System.out.println("x now greater than y");
    }

    if (x == y){
      System.out.println("===");
    }
  }
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

Using a boolean value to control the if statement

The value of a `boolean` variable is sufficient, by itself, to control the if statement.

```
public class Main {
  public static void main(String args[]) {
    boolean b;
    b = false;
    if (b) {
      System.out.println("This is executed.");
    } else {
      System.out.println("This is NOT executed.");
    }

  }
}
```

There is no need to write an `if` statement like this:

```
if(b == true) ...
```

The output generated by this program is shown here:

```
This is NOT executed.
```

Simplest if statement

Java has two selection statements: `if` and `switch`. Simplest if statement form is shown here:

```
if(condition)
   statement;
```

`condition` is a boolean expression.

If `condition` is `true`, then the statement is executed.
If `condition` is `false`, then the statement is bypassed.

Here is an example:

```
public class Main {

  public static void main(String args[]) {
    int num = 99;
    if (num < 100) {
      System.out.println("num is less than 100");

    }
  }
}
```

The output generated by this program is shown here:

```
num is less than 100
```

Using `if` statement to compare two variables

```
public class Main {

  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if (x < y){
      System.out.println("x is less than y");
    }

    x = x * 2;
    if (x == y){
      System.out.println("x now equal to y");
    }
```

```
    x = x * 2;
    if (x > y){
      System.out.println("x now greater than y");
    }

    if (x == y){
      System.out.println("===");
    }
  }
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

Using a boolean value to control the if statement

The value of a `boolean` variable is sufficient, by itself, to control the if statement.

```
public class Main {
  public static void main(String args[]) {
    boolean b;
    b = false;
    if (b) {
      System.out.println("This is executed.");
    } else {
      System.out.println("This is NOT executed.");
    }

  }
}
```

There is no need to write an `if` statement like this:

```
if(b == true) ...
```

The output generated by this program is shown here:

```
This is NOT executed.
```

switch statement

The `switch` statement is a multiway branch statement.

It provides a better alternative than a large series of `if-else-if` statements.
Here is the general form of a switch statement:

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

Duplicate case values are not allowed.

A `break` statement jumps out of switch statement to the first line that follows the entire `switch` statement.
Here is a simple example that uses a `switch` statement:

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 6; i++)
      switch (i) {
        case 0:
          System.out.println("i is zero.");
          break;

        case 1:
          System.out.println("i is one.");
          break;

        case 2:
          System.out.println("i is two.");
          break;

        case 3:
          System.out.println("i is three.");
          break;

        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

The break statement is optional

If you omit the `break`, execution will continue on into the next case. For example, consider the following program:

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 12; i++)
      switch (i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

Nested switch Statements

For example, the following fragment is a valid nested `switch` statement.

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 6; i++)
      switch(i) {
        case 0:
          switch(i+1) { // nested switch
            case 0:
              System.out.println("target is zero");
              break;
```

```
            case 1:
              System.out.println("target is one");
              break;
          }
          break;
        case 2: // ...
      }
   }
}
```

The output:

```
target is one
```

while loop

The `while` loop repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
   // body of loop
}
```

- The `condition` can be any `Boolean` expression.
- The body of the loop will be executed as long as the conditional expression is `true`.
- The curly braces are unnecessary if only a single statement is being repeated.

Here is a while loop that counts down from 10, printing exactly ten lines of "tick":

```
public class Main {
  public static void main(String args[]) {
    int n = 10;
    while (n > 0) {
      System.out.println("n:" + n);
      n--;
    }
  }
}
```

When you run this program, you will get the following result:

```
n:10
n:9
n:8
n:7
n:6
n:5
```

```
n:4
n:3
n:2
n:1
```

The body of the `while` loop will not execute if the condition is `false`. For example, in the following fragment, the call to `println()` is never executed:

```
public class Main {
  public static void main(String[] argv) {
    int a = 10, b = 20;
    while (a > b) {
      System.out.println("This will not be displayed");
    }
    System.out.println("You are here");
  }
}
```

The output:

```
You are here
```

The body of the `while` can be empty. For example, consider the following program:

```
public class Main {
  public static void main(String args[]) {
    int i, j;
    i = 10;
    j = 20;

    // find midpoint between i and j
    while (++i < --j)
      ;
    System.out.println("Midpoint is " + i);
  }
}
```

It generates the following output:

```
Midpoint is 15
```

do-while statement

To execute the body of a while loop at least once, you can use the do-while loop. Its general form is

```
do {
   // body of loop
```

```
} while (condition);
```

Here is an example to show how to use a `do-while` loop. It generates the same output as before.

```
public class Main {
  public static void main(String args[]) {
    int n = 10;
    do {
      System.out.println("n:" + n);
      n--;
    } while (n > 0);
  }
}
```

The loop in the preceding program can be written as follows:

```
n:10
n:9
n:8
n:7
n:6
n:5
n:4
n:3
n:2
n:1

public class Main {
  public static void main(String args[]) {
    int n = 10;
    do {
      System.out.println("n:" + n);
    } while (--n > 0);
  }
}
```

The output:

```
n:10
n:9
n:8
n:7
n:6
n:5
n:4
n:3
n:2
n:1
```

The following program implements a very simple help system with `do-while` loop and `switch` statement.

```
public class Main {
  public static void main(String args[]) throws java.io.IOException {
    char choice;
```

```
    do {
      System.out.println("Help on:");
      System.out.println(" 1. A");
      System.out.println(" 2. B");
      System.out.println(" 3. C");
      System.out.println(" 4. D");
      System.out.println(" 5. E");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while (choice < '1' || choice > '5');
    System.out.println("\n");
    switch (choice) {
      case '1':
        System.out.println("A");
        break;
      case '2':
        System.out.println("B");
        break;
      case '3':
        System.out.println("C");
        break;
      case '4':
        System.out.println("D");
        break;
      case '5':
        System.out.println("E");
        break;
    }
  }
}
```

Here is a sample run produced by this program:

```
Help on:
 1. A
 2. B
 3. C
 4. D
 5. E
Choose one:
```

for Loop

The simplest form of the `for` loop is shown here:

```
   for(initialization; condition; iteration) statement;
```

`initialization` sets a loop control variable to an initial value.

`condition` is a Boolean expression that tests the loop control variable.
If `condition` is true, the for loop continues to iterate.
If `condition` is false, the loop terminates.

The `iteration` determines how the loop control variable is changed each time the loop iterates.

Here is a short program that illustrates the for loop:

```
public class Main {
  public static void main(String args[]) {
    int i;

    for (i = 0; i < 10; i = i + 1)
      System.out.println("This is i: " + i);
  }
}
```

This program generates the following output:

```
This is i: 0
This is i: 1
This is i: 2
This is i: 3
This is i: 4
This is i: 5
This is i: 6
This is i: 7
This is i: 8
This is i: 9
```

`i` is the loop control variable.

`i` is initialized to zero in the initialization.
At the start of each iteration, the conditional test x < 10 is performed.
If the outcome of this test is true, the println() statement is executed, and then the iteration portion of the loop is executed.

This process continues until the conditional test is false.

The following code loops reversively:

```
public class Main {
  public static void main(String args[]) {
    for (int n = 10; n > 0; n--)
      System.out.println("n:" + n);
  }
}
```

The output:

```
n:10
n:9
n:8
n:7
n:6
n:5
```

```
n:4
n:3
n:2
n:1
```

Control Variables Inside the for Loop

It is possible to declare the variable inside the initialization portion of the for.

```
public class Main {
  public static void main(String args[]) {
    for (int n = 10; n > 0; n--){
      System.out.println("tick " + n);
    }
  }
}
```

The output:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

The scope n ends when the for statement ends. Here is a program that tests for prime numbers using for loop statement.

```
public class Main {
  public static void main(String args[]) {
    int num;
    boolean isPrime = true;
    num = 50;
    for (int i = 2; i <= num / 2; i++) {
      if ((num % i) == 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime)
      System.out.println("Prime");
    else
      System.out.println("Not Prime");

  }
}
```

The output:

```
Not Prime
```

Using the Comma

Java allows two or more variables to control a for loop.

And you can include multiple statements in both the initialization and iteration portions of the for.
Each statement is separated from the next by a comma.
Here is an example:

```
public class Main {
  public static void main(String args[]) {

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

    }
  }
}
```

The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

For Loop Variations

The three sections of the `for` can be used for any purpose. Parts of the `for` loop can be empty.

```
public class Main {
  public static void main(String args[]) {
    int i = 0;
    boolean done = false;
    for (; !done;) {
      System.out.println("i is " + i);
      if (i == 10)
        done = true;
      i++;

    }
  }
}
```

The output:

```
i is 0
i is 1
i is 2
i is 3
```

```
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9
i is 10
```

Declare multiple variables in for loop Example

You can define more than one variables and use them inside the for loop.

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0, j = 1, k = 2; i < 5; i++)
      System.out.println("I : " + i + ",j : " + j + ", k : " + k);
  }
}
```

The output:

```
I : 0,j : 1, k : 2
I : 1,j : 1, k : 2
I : 2,j : 1, k : 2
I : 3,j : 1, k : 2
I : 4,j : 1, k : 2
```

Nested for Loops

For example, here is a program that nests `for` loops:

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 10; i++) {
      for (int j = i; j < 10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

for each loop

The new version eliminates the loop counter.

The new syntax is:

```
for (type variable_name:array){

}
```

The type must be compatible with the array type.

```
public class Main {
  public static void main(String args[]) {
    String[] arr = new String[]{"java2s.com","a","b","c"};
    for(String s:arr){
      System.out.println(s);
    }
  }
}
```

The output:

```
java2s.com
a
b
c
```

Use for-each style for on a two-dimensional array.

```
public class Main {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];
    // give nums some values
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 5; j++)
        nums[i][j] = (i + 1) * (j + 1);
    // use for-each for to display and sum the values
    for (int x[] : nums) {
      for (int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90

// Search an array using for-each style for.
public class Main {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;
    // use for-each style for to search nums for val
    for (int x : nums) {
      if (x == val) {
        found = true;
        break;
      }
    }
    if (found)
      System.out.println("Value found!");
  }
}
```

break to Exit a Loop

When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Here is a simple example:

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 100; i++) {
      if (i == 10)
        break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
```

```
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

The `break` statement can be used with while loop as well. For example, here is the preceding program coded by use of a while loop.

```
public class Main {
  public static void main(String args[]) {
    int i = 0;
    while (i < 100) {
      if (i == 10)
        break; // terminate loop if i is 10
      System.out.println("i: " + i);
      i++;
    }
    System.out.println("Loop complete.");
  }
}
```

The output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

The `break` statement can be used with infinite loops.

```
public class Main {
  public static void main(String args[]) {
    int i = 0;
    while (true) {
      if (i == 10){
        break; // terminate loop if i is 10
      }
      System.out.println("i: " + i);
      i++;
```

```
    }
    System.out.println("Loop complete.");
  }
}
```

The output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

When used inside a set of nested loops, the `break` statement will only break out of the inner-most loop. For example:

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 5; i++) {
      System.out.print("Pass " + i + ": ");
      for (int j = 0; j < 100; j++) {
        if (j == 10)
          break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Pass 3: 0 1 2 3 4 5 6 7 8 9
Pass 4: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

The `break` that terminates a `switch` statement affects only that `switch` statement and not any enclosing loops.

```
public class Main {
  public static void main(String args[]) {
    for (int i = 0; i < 6; i++)
      switch (i) {
        case 1:
          System.out.println("i is one.");
          for (int j = 0; j < 5; j++) {
```

```
            System.out.println("j is " + j);
          }
          break;
        case 2:
          System.out.println("i is two.");
          break;

        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

The output:

```
i is greater than 3.
i is one.
j is 0
j is 1
j is 2
j is 3
j is 4
i is two.
i is greater than 3.
i is greater than 3.
i is greater than 3.
```

continue

`continue` statement forces an early iteration of a loop.

In `while` and `do-while` loops, a `continue` statement causes control to be transferred to the conditional expression that controls the loop.
In a `for` loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

The following code shows how to use a continue statement.

```
public class Main {
  public static void main(String[] argv) {
    for (int i = 0; i < 10; i++) {
      System.out.print(i + " ");
      if (i % 2 == 0)
        continue;
      System.out.println("");
    }
  }
}
```

The code above generates the following result.

```
0 1
2 3
4 5
6 7
8 9
```

Using continue with a label

`continue` may specify a label to describe which enclosing loop to continue.

```
public class Main {
  public static void main(String args[]) {
    outer: for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
        if (j > i) {
          System.out.println();
          continue outer;
        }
        System.out.print(" " + (i * j));
      }
    }
    System.out.println();
  }
}
```

Here is the output of this program:

```
 0
 0 1
 0 2 4
 0 3 6 9
 0 4 8 12 16
 0 5 10 15 20 25
 0 6 12 18 24 30 36
 0 7 14 21 28 35 42 49
 0 8 16 24 32 40 48 56 64
 0 9 18 27 36 45 54 63 72 81
```

return statement returns from a method.

The `return` statement immediately terminates the method in which it is executed.

```
public class Main {
  public static void main(String args[]) {
    boolean t = true;
    System.out.println("Before return");
    if (t)
      return; // return to caller
    System.out.println("after");
  }
```

```
}
```

The output from this program is shown here:

```
Before return
```

Comments

There are three types of comments defined by Java.

1. Single-line,
2. Multiline and
3. Documentation comment.

Single-line comment

Java single line comment starts from `//` and ends till the end of that line.

```
public class Main {
  // This is a single line comment.
  public static void main(String[] argv) {
  }

}
```

Multiline comment

Java multiline comment is between `/*` and `*/`.

```
public class Main {
  /* This
     is
     a
     Multiline
     comment.
  */
  public static void main(String[] argv) {
  }

}
```

Documentation comment.

Documentation comment is used to produce an HTML file that documents your program. A Javadoc comment occupies one or more lines of source code. The documentation comment begins with a `/**` and ends with a `*/`. Everything from /** through */ is ignored by the compiler.

The following example demonstrates a Javadoc comment:

```java
/**
* Application entry point
*
* @param args array of command-line arguments passed to this method
*/
public static void main(String[] args)
{
// TODO code application logic here
}
```

This example begins with a Javadoc comment that describes the main() method. /** and */ contains a description of the method, which could include HTML tags such as <p> and <code>/</code>, and the @param Javadoc tag (an @-prefixed instruction).

The following list identifies several commonly used tags:

- @author identifies the source code's author.
- @deprecated identifies a source code entity that should no longer be used.
- @param identifies one of a method's parameters.
- @see provides a see-also reference.
- @since identifies the software release where the entity first originated.
- @return identifies the kind of value that the method returns.

The following code has more documentation comments

```java
/**
 * A simple class for introducing a Java application.
 *
 * @author yourName
 */
public class HelloWorld {
  /**
   * Application entry point
   *
   * @param args
   *          array of command-line arguments passed to this method
   */
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

We can extract these documentation comments into a set of HTML files by using the JDK's javadoc tool, as follows:

```
javadoc HelloWorld.java
```

!!

Introduction to Arrays

A Java array is an ordered collection of primitives, object references, or other arrays. Java arrays are homogeneous: except as allowed by polymorphism, all elements of an array must be of the same type.

Each variable is referenced by array name and its index. Arrays may have one or more dimensions.

One-Dimensional Arrays

A one-dimensional array is a list of similar-typed variables. The general form of a one-dimensional array declaration is:

```
type var-name[ ];
```

- `type` declares the array type.
- `type` also determines the data type of each array element.

The following declares an array named days with the type "array of int":

```
int days[];
```

- `days` is an array variable.
- The value of days is set to `null`.

Allocate memory for array

You allocate memory using `new` and assign it to array variables. `new` is a special operator that allocates memory. The general form is:

arrayVar = new type[size];

- `type` specifies the type of data being allocated.
- `size` specifies the number of elements.
- `arrayVar` is the array variable.

The following two statements first create an int type array variable and then allocate memory for it to store 12 int type values.

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

days refers to an array of 12 integers.
All elements in the array is initialized to zero.

Array creation is a two-step process.

1. declare a variable of the desired array type.

2. allocate the memory using `new`.

In Java all arrays are dynamically allocated.
You can access a specific element in the array with [index].
All array indexes start at zero.
For example, the following code assigns the value 28 to the second element of days.

```
public class Main {
  public static void main(String[] argv) {
    int days[];
    days = new int[12];

    days[1] = 28;

    System.out.println(days[1]);
  }
}
```

It is possible to combine the declaration of the array variable with the allocation of the array itself.

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

Array Element Initialization Values

Element Type	Initial Value
byte	0
int	0
float	0.0f
char	'\u0000'
object reference	null
short	0
long	0L
double	0.0d
boolean	false

Alternative Array Declaration Syntax

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

```
type[] var-name;
```

For example, the following two declarations are equivalent:

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

The following declarations are also equivalent:

```
char d1[][] = new char[3][4];
char[][] d2 = new char[3][4];
```

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. For example, the following declares a two-dimensional array variable called `twoD`.

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

This allocates a 4-by-5 array and assigns it to twoD. This array will look like the one shown in the following:

```
   [leftIndex][rightIndex]


   [0][0] [0][1] [0][2] [0][3] [0][4]
   [1][0] [1][1] [1][2] [1][3] [1][4]
   [2][0] [2][1] [2][2] [2][3] [2][4]
   [3][0] [3][1] [3][2] [3][3] [3][4]
```

The wrong way to think about multi-dimension arrays

```
+----+----+----+
|   1|   2|   3|
+----+----+----+
|   4|   5|   6|
+----+----+----+
|   7|   8|   9|
+----+----+----+
```

right way to think about multi-dimension arrays

```
+--+        +----+----+----+
|  |--------|   1|   2|   3|
+--+        +----+----+----+     +----+----+----+
|  |-----------------------------|   4|   5|   6|
+--+   +----+----+----+          +----+----+----+
|  |---|   7|   8|   9|
+--+   +----+----+----+
```

An irregular multi-dimension array

```
+--+        +----+----+
|  |--------|   1|   2|
+--+        +----+----+          +----+----+----+
|  |-----------------------------|   4|   5|   6|
+--+   +----+----+----+----+     +----+----+----+
|  |---|   7|   8|   9|  10|
+--+   +----+----+----+----+
```

The following code use nested for loop to assign values to a two-dimensional array.

```java
public class Main {
  public static void main(String args[]) {
    int twoD[][] = new int[4][5];
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 5; j++) {
        twoD[i][j] = i*j;
      }
    }
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 5; j++) {
        System.out.print(twoD[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

This program generates the following output:

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

Three-dimensional array

The following program creates a 3 by 4 by 5, three-dimensional array.

```java
public class Main {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];

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

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

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
```

```
0 0 0 0 0
0 0 0 0 0

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

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

Jagged array

When you allocate memory for a multidimensional array, you can allocate the remaining dimensions separately. For example, the following code allocates the second dimension manually.

```
public class Main {
  public static void main(String[] argv) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[5];
    twoD[1] = new int[5];
    twoD[2] = new int[5];
    twoD[3] = new int[5];
  }
}
```

When allocating dimensions manually, you do not need to allocate the same number of elements for each dimension.

The following program creates a two-dimensional array in which the sizes of the second dimension are unequal.

```
public class Main {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    for (int i = 0; i < 4; i++){
      for (int j = 0; j < i + 1; j++) {
        twoD[i][j] = i + j;
      }
    }
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < i + 1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0
1 2
2 3 4
3 4 5 6
```

The array created by this program looks like this:

```
    [0][0]
    [1][0] [1][1]
    [2][0] [2][1] [2][2]
    [3][0] [3][1] [3][2] [3][3]
```

Initialize multidimensional arrays

Enclose each dimension's initializer within its own set of curly braces.

```
public class Main{
  public static void main(String args[]) {
    double m[][] = {
            { 0, 1, 2, 3 },
            { 0, 1, 2, 3 },
            { 0, 1, 2, 3 },
            { 0, 1, 2, 3 }
        };
    for(int i=0; i<4; i++) {
      for(int j=0; j<4; j++){
        System.out.print(m[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

When you run this program, you will get the following output:

```
0.0 1.0 2.0 3.0
0.0 1.0 2.0 3.0
0.0 1.0 2.0 3.0
0.0 1.0 2.0 3.0
```

Arrays length

Array size, arrayName.length, holds its length. The following code outputs the length of each array by using its `length` property.

```
public class Main {
  public static void main(String args[]) {
```

```
    int a1[] = new int[10];
    int a2[] = {1, 2, 3, 4, 5};
    int a3[] = {4, 3, 2, 1};
    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 5
length of a3 is 4
```

Calculate with Array

Calculate Average value of Array elements

```
public class Main {
  public static void main(String[] args) {

    int[] intArray = new int[] { 1, 2, 3, 4, 5 };
    // calculate sum
    int sum = 0;
    for (int i = 0; i < intArray.length; i++){
      sum = sum + intArray[i];
    }
    // calculate average
    double average = sum / intArray.length;

    System.out.println("average: " + average);
  }
}
```

The output:

```
average: 3.0
```

Create Fibonacci Series with array

```
import java.util.Arrays;

public class Main {
  public static void main(String[] args) {
    int length = 20;
    long[] series = new long[length];
```

```
    series[0] = 0;
    series[1] = 1;
    for (int i = 2; i < length; i++) {
      series[i] = series[i - 1] + series[i - 2];
    }
    System.out.print(Arrays.toString(series));
  }
}
```

The output:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181]
```

The following code use two-dimensional double type array to do Matrix calculation

```
class Matrix {
  private double[][] doubleArray;

  Matrix(int nrows, int ncols) {
    doubleArray = new double[nrows][ncols];
  }

  int getCols() {
    return doubleArray[0].length;
  }

  int getRows() {
    return doubleArray.length;
  }

  double getValue(int row, int col) {
    return doubleArray[row][col];
  }

  void setValue(int row, int col, double value) {
    doubleArray[row][col] = value;
  }
}

public class Main {
  public static void main(String[] args) {
    Matrix a = new Matrix(1, 3);
    a.setValue(0, 0, 1); // | 1 2 3 |
    a.setValue(0, 1, 2);
    a.setValue(0, 2, 3);
    dump(a);
    Matrix b = new Matrix(3, 2);
    b.setValue(0, 0, 4); // | 4 7 |
    b.setValue(1, 0, 5); // | 5 8 |
    b.setValue(2, 0, 6); // | 6 9 |
    b.setValue(0, 1, 7);
    b.setValue(1, 1, 8);
    b.setValue(2, 1, 9);
    dump(b);
    dump(multiply(a, b));
  }

  static void dump(Matrix m) {
    for (int i = 0; i < m.getRows(); i++) {
```

```
      for (int j = 0; j < m.getCols(); j++){
        System.out.print(m.getValue(i, j) + " ");
      }
      System.out.println();
    }
    System.out.println();
  }

  static Matrix multiply(Matrix a, Matrix b) {
    if (a.getCols() != b.getRows()) {
      throw new IllegalArgumentException("rows/columns mismatch");
    }
    Matrix result = new Matrix(a.getRows(), b.getCols());
    for (int i = 0; i < a.getRows(); i++) {
      for (int j = 0; j < b.getCols(); j++) {
        for (int k = 0; k < a.getCols(); k++) {
          result.setValue(i, j, result.getValue(i, j) + a.getValue(i, k) *
b.getValue(k, j));
        }
      }
    }

    return result;
  }
}
```

!!!

Type Conversion and Casting

If the two types are compatible, then Java will perform the conversion automatically. For example, assign an int value to a long variable.

For incompatible types we must use a cast. Casting is an explicit conversion between incompatible types.

Java's Automatic Conversions

An automatic type conversion will be used if the following two conditions are met:

1. The two types are compatible.
2. The destination type is larger than the source type.

`int` type is always large enough to hold all valid `byte` values, so an automatic type conversion takes place.

```
public class Main {
  public static void main(String[] argv) {
    byte b = 10;
    int i = 0;

    i = b;
```

```
    System.out.println("b is " + b);
    System.out.println("i is " + i);
  }
}
```

The output:

```
b is 10
i is 10
```

For widening conversions, integer and floating-point types are compatible with each other.

```
public class Main {
  public static void main(String[] argv) {
    int i = 1234;
    float f;

    f = i;

    System.out.println("i is " + i);
    System.out.println("f is " + f);
  }
}
```

The output:

```
i is 1234
f is 1234.0
```

The numeric types are not compatible with char or boolean

```
public class Main{
   public static void main(String[] argv){
      char ch = 'a';
      int num = 99;
      ch = num ;
    }
}
```

Compiling the code above will generate the following error message.

char and boolean are not compatible with each other.

```
public class Main {
  public static void main(String[] argv) {
    char ch = 'a';
    int num = 99;
    ch = num;
  }
}
```

Compiling the code above will generate the following error message.

```
D:\>javac Main.java
Main.java:5: possible loss of precision
found   : int
required: char
    ch = num;
         ^
1 error
```

Java performs an automatic type conversion when storing a literal `integer` constant into variables of type `byte`, `short`, or `long`.

```
public class Main {
  public static void main(String[] argv) {
    byte b = 1;
  }
}
```

But you cannot store a value out of the byte scope

```
public class Main{
   public static void main(String[] argv){
      byte b = 11111;
    }
}
```

When compiling, it generates the following error message:

```
D:\>javac Main.java
Main.java:3: possible loss of precision
found   : int
required: byte
      byte b = 11111;
               ^
1 error
```

Casting Incompatible Types

A narrowing conversion is to explicitly make the value narrower. A cast is an explicit type conversion. It has this general form:

```
(target-type) value
```

`target-type` specifies the desired type. The following code casts `int` type value to `byte` type value.

```
public class Main {
  public static void main(String[] argv) {
```

```
    int a = 1234;
    byte b;

    b = (byte) a;

    System.out.println("a is " + a);
    System.out.println("b is " + b);
  }
}
```

The output:

```
a is 1234
b is -46
```

Truncation happens when assigning a floating-point value to an integer value. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.

```
public class Main {

  public static void main(String args[]) {
    byte b;
    int i = 1;
    double d = 1.123;

    System.out.println("Conversion of double to int.");
    i = (int) d;
    System.out.println("d: " + d + " i: " + i);

    System.out.println("Conversion of double to byte.");
    b = (byte) d;
    System.out.println("d: " + d + " b: " + b);

  }
}
```

This program generates the following output:

```
Conversion of double to int.
d: 1.123 i: 1
Conversion of double to byte.
d: 1.123 b: 1
```

Automatic Type Promotion in Expressions

For example, examine the following expression:

```
public class Main {
  public static void main(String[] argv) {
```

```
    byte a = 40;
    byte b = 50;
    byte c = 100;
    int d = a * b / c;
  }
}
```

The result of a * b exceeds the range of byte. To handle this kind of problem, Java automatically promotes each byte or short operand to int. a * b is performed using integers.

Automatic promotions can cause compile-time errors.

```
public class Main {
  public static void main(String[] argv) {
    byte b = 5;
    b = b * 2; // Error! Cannot assign an int to a byte!
  }
}
```

Compiling the code above generates the following errors:

```
D:\>javac Main.java
Main.java:4: possible loss of precision
found   : int
required: byte
    b = b * 2; // Error! Cannot assign an int to a byte!
          ^
1 error
```

If you understand the consequences of overflow, use an explicit cast.

```
public class Main {
  public static void main(String[] argv) {
    byte b = 50;
    b = (byte) (b * 2);

    System.out.println("b is " + b);
  }
}
```

The output from the code above is:

```
b is 100
```

The Type Promotion Rules

Widening conversions do not lose information about the magnitude of a value.

For example, an `int` value is assigned to a `double` variable.

This conversion is legal because doubles are wider than ints.

Java's widening conversions are

- From a byte to a short, an int, a long, a float, or a double
- From a short to an int, a long, a float, or a double
- From a char to an int, a long, a float, or a double
- From an int to a long, a float, or a double
- From a long to a float or a double
- From a float to a double

Widening conversions:

```
char->int
byte->short->int->long->float->double
```

Here are the Type Promotion Rules:

1. All `byte` and `short` values are promoted to `int`.
2. If one operand is a `long`, the whole expression is promoted to `long`.
3. If one operand is a `float`, the entire expression is promoted to `float`.
4. If any of the operands is `double`, the result is `double`.

In the following code, `f * b`, `b` is promoted to a `float` and the result of the subexpression is `float`.

```
public class Main {

  public static void main(String args[]) {
    byte b = 4;
    float f = 5.5f;
    float result = (f * b);
    System.out.println("f * b = " + result);
  }
}
```

The output:

```
f * b = 22.0
```

In the following program, `c` is promoted to `int`, and the result is of type `int`.

```
public class Main {
  public static void main(String args[]) {
    char c = 'a';
    int i = 50000;
    int result = i / c;
    System.out.println("i / c is " + result);
```

```
  }
}
```

The output:

```
i / c is 515
```

In the following code the value of `s` is promoted to `double`, and the type of the subexpression is `double`.

```
public class Main {
  public static void main(String args[]) {
    short s = 1024;
    double d = .1234;
    double result = d * s;
    System.out.println("d * s is " + result);
  }
}
```

The output:

```
d * s is 126.3616
```

!!

Autoboxing and auto-unboxing

Type Wrappers

Java uses primitive types, such as int or double, to hold the basic primitive data types. Primitive types, rather than objects, are used for the sake of performance.

Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.

Autoboxing and Auto-unboxing

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

For example, the following code constructs an Integer object that has the value 100:

```
public class Main {

  public static void main(String[] argv) {
    Integer iOb = 100; // autobox an int
  }
}
```

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox iOb, you can use this line:

```
public class Main {

  public static void main(String[] argv) {
    Integer iOb = 100; // autobox an int
    int i = iOb; // auto-unbox
  }

}
```

Here is a program using autoboxing/unboxing:

```
public class Main {
  public static void main(String args[]) {
    Integer iOb = 100; // autobox an int
    int i = iOb; // auto-unbox
    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```

Output:

```
100 100
```

Autoboxing and Methods

autoboxing occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type.

autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

For example, consider this example:

```
public class Main {
  static int m(Integer v) {
    return v; // auto-unbox to int
  }
```

```
  public static void main(String args[]) {
    Integer iOb = m(100);
    System.out.println(iOb);
  }
}
```

!!

enum type

An enumeration is created using the enum keyword.

For example, here is a simple enumeration:

```
// An enumeration.
enum Direction {
   East, South, West, North
}
```

The identifiers East, South are called enumeration constants. Enumeration constants is implicitly declared as a public, static final member of Direction.

The following program demonstrates the Direction enumeration:

```
// An enumeration of direction varieties.
enum Direction {
  East, South, West, North
}

public class Main {
  public static void main(String args[]) {
    Direction dir;
    dir = Direction.South;
    // Output an enum value.
    System.out.println("Value of dir: " + dir);
    System.out.println();
    dir = Direction.South;
    // Compare two enum values.
    if (dir == Direction.South)
      System.out.println("ap contains GoldenDel.\n");
    // Use an enum to control a switch statement.
    switch (dir) {
    case South:
      System.out.println("south");
      break;
    case East:
      System.out.println("East");
      break;
    case West:
      System.out.println("West");
      break;
    case North:
      System.out.println("North.");
      break;
    }
```

```
  }
}
```

The following code use enum with switch statement.

```
enum Coin {
  PENNY, NICKEL, DIME, QUARTER
}

public class Main {

  public static void main(String[] args) {
    Coin coin = Coin.NICKEL;
    switch (coin) {
    case PENNY:
      System.out.println("1 cent");
      break;
    case NICKEL:
      System.out.println("5 cents");
      break;
    case DIME:
      System.out.println("10 cents");
      break;
    case QUARTER:
      System.out.println("25 cents");
      break;
    default:
      assert false;
    }
  }
}
```

values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: values( ) and valueOf( ).

Their general forms are:

```
public static enum-type[ ] values( )
public static enum-type valueOf(String str)
```

The values( ) method returns an array that contains a list of the enumeration constants. The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.

In both cases, enum-type is the type of the enumeration.

The following program demonstrates the values( ) and valueOf( ) methods:

```
enum Direction {
```

```
  East, South, West, North
}

public class Main {
  public static void main(String args[]) {
    Direction dir;
    // use values()
    Direction all[] = Direction.values();
    for (Direction a : all)
      System.out.println(a);
    System.out.println();
    // use valueOf()
    dir = Direction.valueOf("South");
    System.out.println(dir);
  }
}
```

enum as Class

You can give constructors, add instance variables and methods, and implement interfaces for enum types.

When you define a constructor for an enum, the constructor is called when each enumeration constant is created.

Each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
// Use an enum constructor, instance variable, and method.
enum Direction {
  South(10), East(9), North(12), West(8);
  private int myValue;
  // Constructor

  Direction(int p) {
    myValue = p;
  }

  int getMyValue() {
    return myValue;
  }
}

public class Main {
  public static void main(String args[]) {

    System.out.println(Direction.East.getMyValue());

    for (Direction a : Direction.values())
      System.out.println(a + " costs " + a.getMyValue());
  }
}
```

Two restrictions that apply to enumerations.

- an enumeration can't inherit another class.
- an enum cannot be a superclass.

You can add fields, constructors, and methods to an enum. You can have the enum implement interfaces.

```
enum Coin {
  PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
  private final int denomValue;

  Coin(int denomValue) {
    this.denomValue = denomValue;
  }

  int denomValue() {
    return denomValue;
  }

  int toDenomination(int numPennies) {
    return numPennies / denomValue;
  }
}

public class Main {
  public static void main(String[] args) {
    int numPennies = 1;
    int numQuarters = Coin.QUARTER.toDenomination(numPennies);
    numPennies = Coin.QUARTER.denomValue();
    int numDimes = Coin.DIME.toDenomination(numPennies);
    numPennies = Coin.DIME.denomValue();
    int numNickels = Coin.NICKEL.toDenomination(numPennies);
    numPennies = Coin.NICKEL.denomValue();
    for (int i = 0; i < Coin.values().length; i++) {
      System.out.println(Coin.values()[i].denomValue());
    }

  }
}
```

enum type Inherit Enum

All enum inherit java.lang.Enum. java.lang.Enum's methods are available for use by all enumerations.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal( ) method:

```
final int ordinal( )
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero.

You can compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method.

```
final int compareTo(enum-type e)
```

enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant.

If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using equals( ).

Those two objects will only be equal if they both refer to the same constant, within the same enumeration.

The following program demonstrates the ordinal( ), compareTo( ), and equals( ) methods:

```
// Demonstrate ordinal(), compareTo(), and equals().
enum Direction {
  East, South, West, North
}

public class Main {
  public static void main(String args[]) {
    Direction ap, ap2, ap3;
    // Obtain all ordinal values using ordinal().
    for (Direction a : Direction.values()){
      System.out.println(a + " " + a.ordinal());
    }
    ap = Direction.West;
    ap2 = Direction.South;
    ap3 = Direction.West;
    System.out.println();
    // Demonstrate compareTo() and equals()
    if (ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);
    if (ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);
    if (ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);
    System.out.println();
    if (ap.equals(ap2))
      System.out.println("Error!");
    if (ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);
    if (ap == ap3)
      System.out.println(ap + " == " + ap3);
  }
}
```

enum type Inherit Enum

All enum inherit java.lang.Enum. java.lang.Enum's methods are available for use by all enumerations.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal( ) method:

```
final int ordinal( )
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero.

You can compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method.

```
final int compareTo(enum-type e)
```

enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant.

If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using equals( ).

Those two objects will only be equal if they both refer to the same constant, within the same enumeration.

The following program demonstrates the ordinal( ), compareTo( ), and equals( ) methods:

```
// Demonstrate ordinal(), compareTo(), and equals().
enum Direction {
  East, South, West, North
}

public class Main {
  public static void main(String args[]) {
    Direction ap, ap2, ap3;
    // Obtain all ordinal values using ordinal().
    for (Direction a : Direction.values()){
      System.out.println(a + " " + a.ordinal());
    }
    ap = Direction.West;
    ap2 = Direction.South;
    ap3 = Direction.West;
    System.out.println();
    // Demonstrate compareTo() and equals()
    if (ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);
    if (ap.compareTo(ap2) > 0)
```

```
      System.out.println(ap2 + " comes before " + ap);
    if (ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);
    System.out.println();
    if (ap.equals(ap2))
      System.out.println("Error!");
    if (ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);
    if (ap == ap3)
      System.out.println(ap + " == " + ap3);
  }
}
```

Overriding toString() to return a Token constant's value

```
enum Token {
  IDENTIFIER("ID"), INTEGER("INT"), LPAREN("("), RPAREN(")"), COMMA(",");
  private final String tokValue;

  Token(String tokValue) {
    this.tokValue = tokValue;
  }

  @Override
  public String toString() {
    return tokValue;
  }

}
public class Main{
  public static void main(String[] args) {
    for (int i = 0; i < Token.values().length; i++){
      System.out.println(Token.values()[i].name() + " = " +
Token.values()[i]);
    }

  }

}


IDENTIFIER = ID
INTEGER = INT
LPAREN = (
RPAREN = )
COMMA = ,
```

Assign a different behavior to each constant.

```
enum Converter {
  DollarToEuro("Dollar to Euro") {
    @Override
    double convert(double value) {
      return value * 0.9;
    }
  },
  DollarToPound("Dollar to Pound") {
    @Override
    double convert(double value) {
      return value * .8;
    }
  };
  Converter(String desc) {
    this.desc = desc;
  }

  private String desc;

  @Override
  public String toString() {
    return desc;
  }

  abstract double convert(double value);

}
public class Main{
  public static void main(String[] args) {
    System.out.println(Converter.DollarToEuro + " = " +
Converter.DollarToEuro.convert(100.0));
    System.out.println(Converter.DollarToPound + " = " +
Converter.DollarToPound.convert(98.6));
  }

}
```

- **enum underline constant**

Kanalimiz a’zolari uchun maxsus! Yuqori malakali dasturchilar uchun qo’llanma.

Telegram kanal: **@Uzb_Dasturchilar**

Youtube kanal: **Uzb Dasturchilar TV**