CST281 Object Oriented Programming

MODULE 2

Core Java Fundamentals

Syllabus

➢Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class.

➢Operators - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.

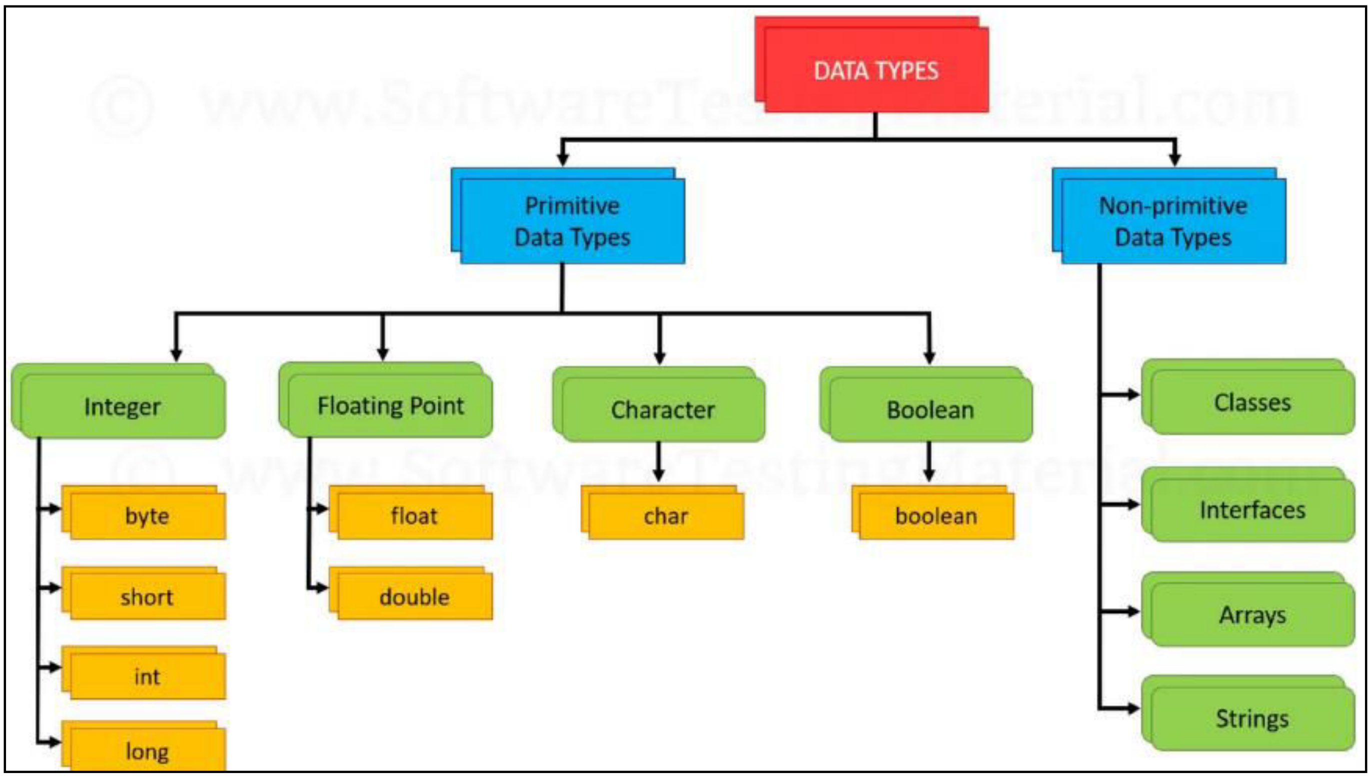
➢Control Statements - Selection Statements, Iteration Statements and Jump Statements.

➢Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, ***this* Keyword, Method Overloading, Using** Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command-Line Arguments, Variable Length Arguments.

DATA TYPES, OPERATORS & CONTROL STATEMENTS

DATA TYPES

- Data type defines the values that a variable can take, for example if a variable has int data type, it can only take integer values.
- Data types specify the different sizes and values that can be stored in the variable.
- There are two types of data types in Java:
  - Primitive data types
  - Non-primitive data types



Primitive Data Types (Fundamental Data Types)

- A primitive type is predefined by the language and is named by a reserved keyword.
- Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

Integers: byte, short, int, and long, which are for whole-valued signed numbers.
Floating-point numbers : float and double, which represent numbers with fractional precision.
Characters: char, which represents symbols in a character set, like letters and numbers.
Boolean: boolean, which is a special type for representing true/false values.

Integers

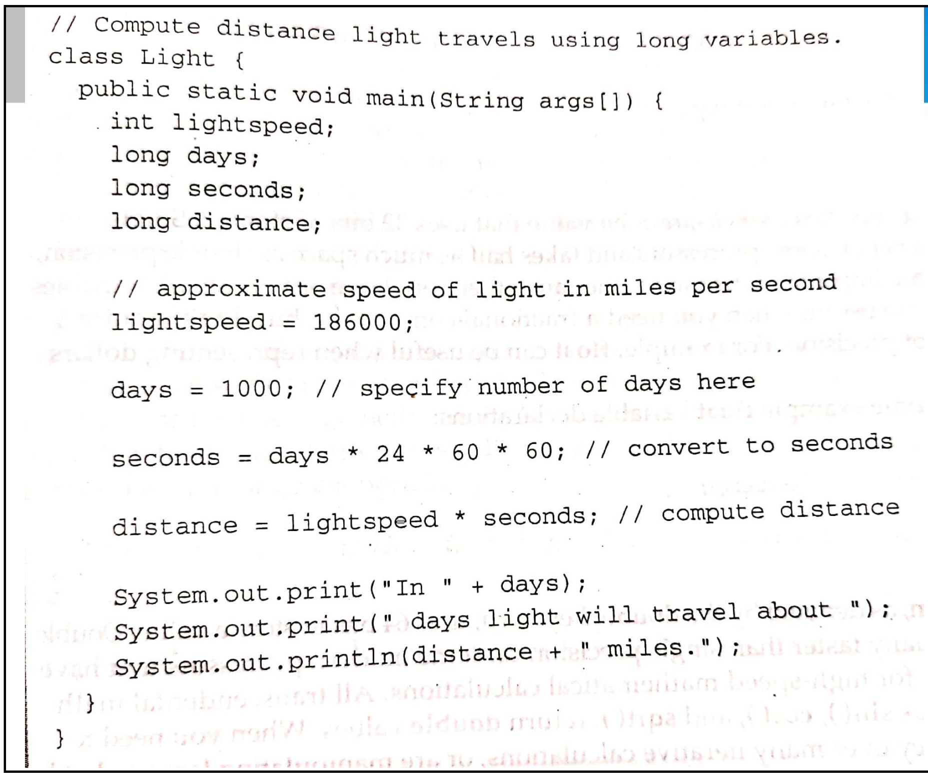
- Java defines four integer types: byte, short, int, and long

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

- byte:
  - The smallest integer type is byte.
  - This is a signed 8-bit. Variables of type byte are especially useful when you're working with a stream of data from a network or file.
  - when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
  - Byte variables are declared by use of the byte keyword.
  - For example, the following declares two byte variables called b and c:

```
byte b, c;
```

- Short is a signed 16-bit type.
  - you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
  - Example : short s;
- The most commonly used integer type is int.
  - In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
  - Example: int a;
- long
  - is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.
  - Example: long a;



```
// Compute distance light travels using long variables.
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

Floating-point numbers

- Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.
- The type float specifies a single-precision value that uses 32 bits of storage.
- Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.
- Example: float highTemp, lowTemp;

Double

- Double precision, as denoted by the doublekeyword, uses 64 bits to store a value.
- When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.
- Example : double pi, r, a;

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

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

Characters

- In Java, the data type used to store characters is char.
- Java uses Unicode to represent characters
- At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type.
- Example: char letterA= 'A‘;

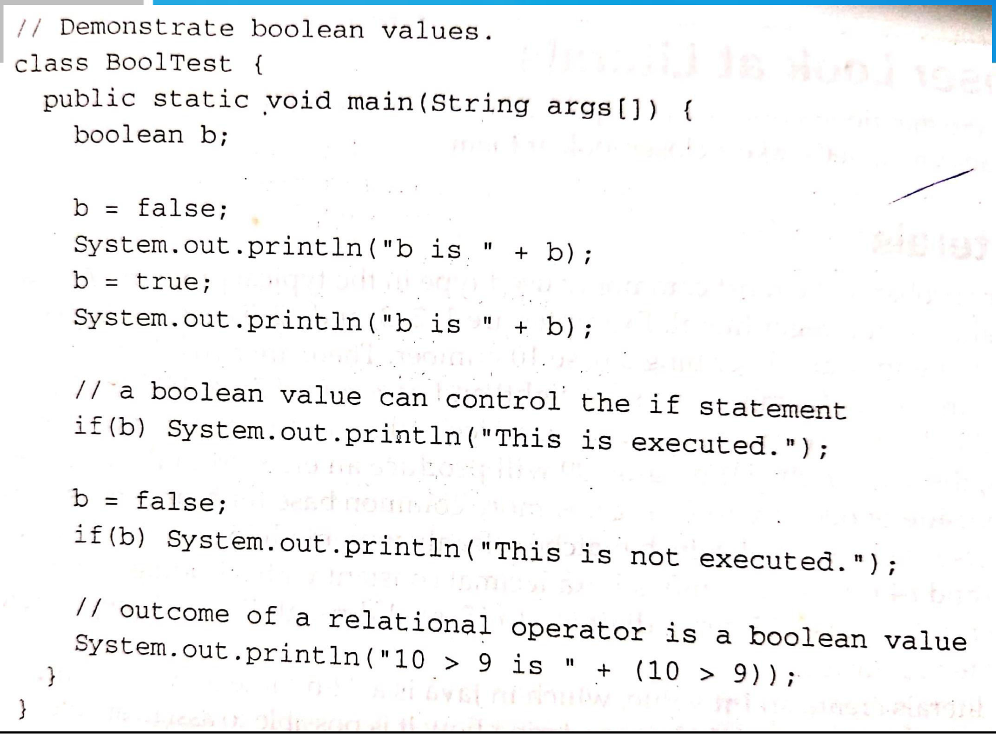
```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // code for X
    ch2 = 'Y';

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

This program displays the following output:

```
ch1 and ch2: X Y
```

Booleans

- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, true or false.
- This is the type returned by all relational operators, as in the case of a < b.
- Example: boolean bool;



```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

LITERALS IN JAVA

Literals

- A constant value in Java is created by using a literal representation.
  1. Integer Literals
  2. Floating-Point Literals
  3. Boolean Literals
  4. Character Literals
  5. String Literals

Integer Literals

- Any whole number value is an integer literal.
- Examples are 1, 2, 3, and 42
- There are three bases which can be used in integer literals
  1. Decimal(base 10)
  2. octal (base 8)
  3. hexadecimal (base 16).

- Normal decimal numbers
  - cannot have a leading zero.
  - can use digits from 0 to 9
- Octal values
  - are denoted by a leading zero.
  - can use digits from 0 to 7
  - E.g 012, 0356
- Hexadecimal constant
  - are denoted with a leading zero-x, (0x or 0X).
  - use digits from 0 to 9 and letters *A* through *F (or a through f)* E.g. 0x234, 0X3B5c

- An integer literal can always be assigned to a **long variable.**
  - Append an upper- or lowercase *L to the literal*
    - 9223372036854775807L
- integer can also be assigned to a **char** as long as it is within range.
- literal value is assigned to a **byte** or **short variable as** long as it is within range.

Floating-Point Literals

- Floating-point numbers represent decimal values with a fractional component.
- Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.
  - E.g. 3.14159, 2.0
- 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
  - E.g. 6.022E23, 314159E–05, 2e+100.

- Floating-point literals in Java are double precision by default.
- To specify a float literal, we must append an F or f to the constant.
- We can also explicitly specify a double literal by appending a D or d.
- The default double type consumes 64 bits of storage, while the less-accurate float type requires only 32 bits

Boolean Literals

- Boolean literals are simple.
- There are only two logical values that a boolean value can have,
  - true , false.
- The values of true and false do not convert into any numerical representation.
- The true literal in Java does not equal 1
- The false literal in Java does not equal 0.

Character Literals

- Characters in Java are indices into the Unicode character set.
- They are 16-bit values that can be converted into integers
  - and manipulated with the integer operators, such as the addition and subtraction operators.
- A literal character is represented inside a pair of single quotes.
  - All of the visible ASCII characters can be directly entered inside the quotes, such as ‘a’, ‘z’, and ‘@’.

Escape sequences

- For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need
- '\n' for the newline character.
- '\'' for the single-quote character

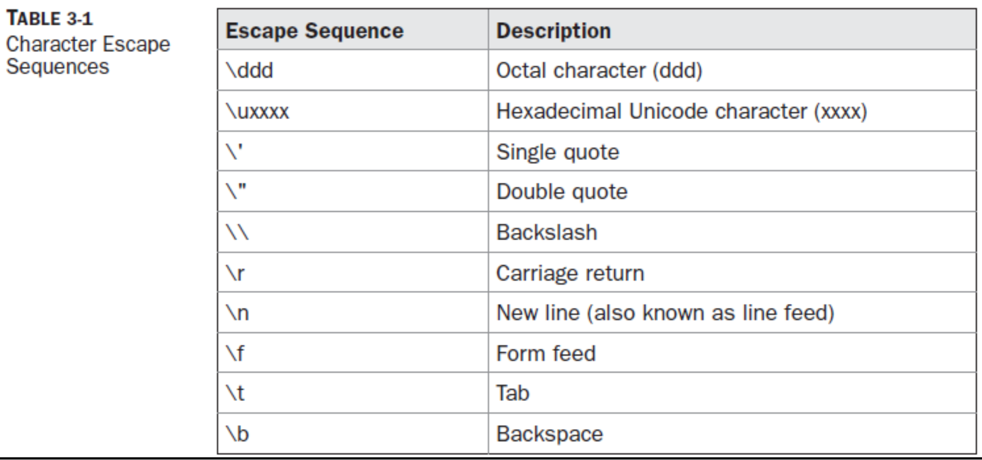


TABLE 3-1
Character Escape Sequences

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

String Literals

- String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes
- Examples of string literals are
  - "Hello World"
  - "two\nlines"
  - "\"This is in quotes\""

Variables In Java

29

Variables In JAVA

- Variable in Java is a data container that stores the data values during Java program execution.
- Variable is a memory location name of the data.
- variable="vary + able" that means its value can be changed.
- In order to use a variable in a program we need to perform 2 steps
  1. Variable Declaration
  2. Variable Initialization

30

- All variables must be declared before they can be used.
- The basic form of a variable declaration is :

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

- The type is one of Java's atomic types, or the name of a class or interface.
- The identifier is the name of the variable.
- Square bracket denote that =Value is optional in declaration.

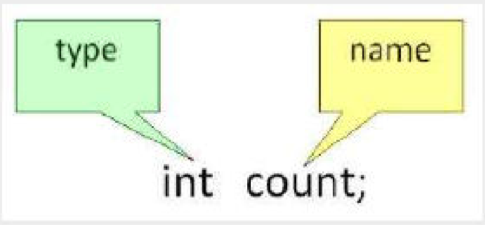
1. Variable Declaration

Syntax: data_type variable_name ;

Eg: int a,b,c;

float pi;

double d;



2. Variable Initialization

Syntax : data_type variable_name = value;

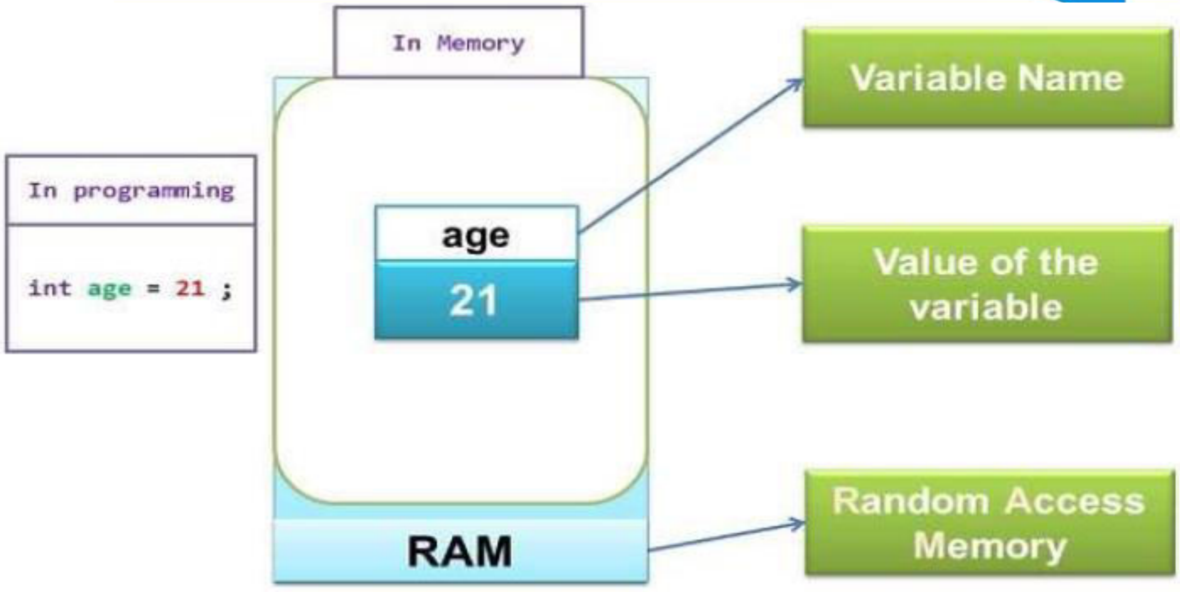
Eg: int a=2,b=4,c=6;

float pi = 3.14f;

double val = 20.22d;

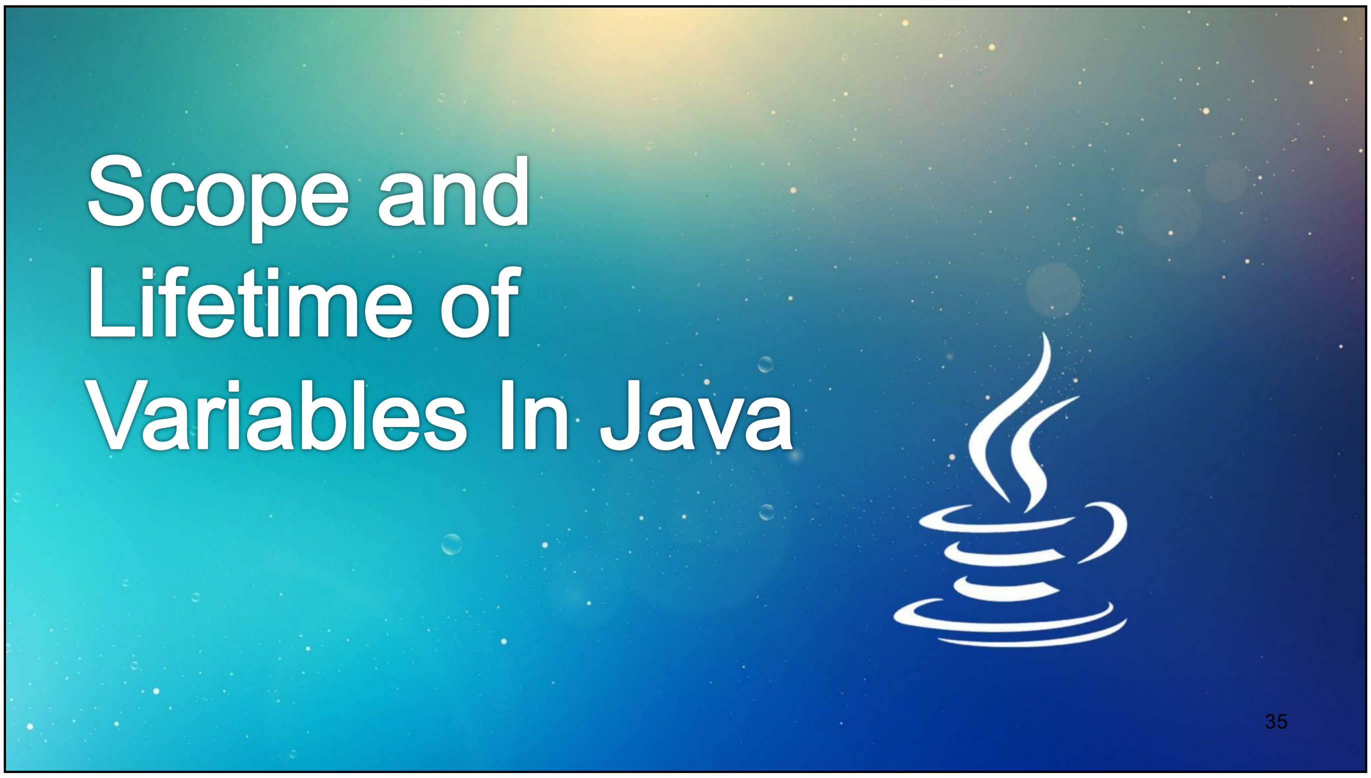
char a = 'v';

int num = 45.66;

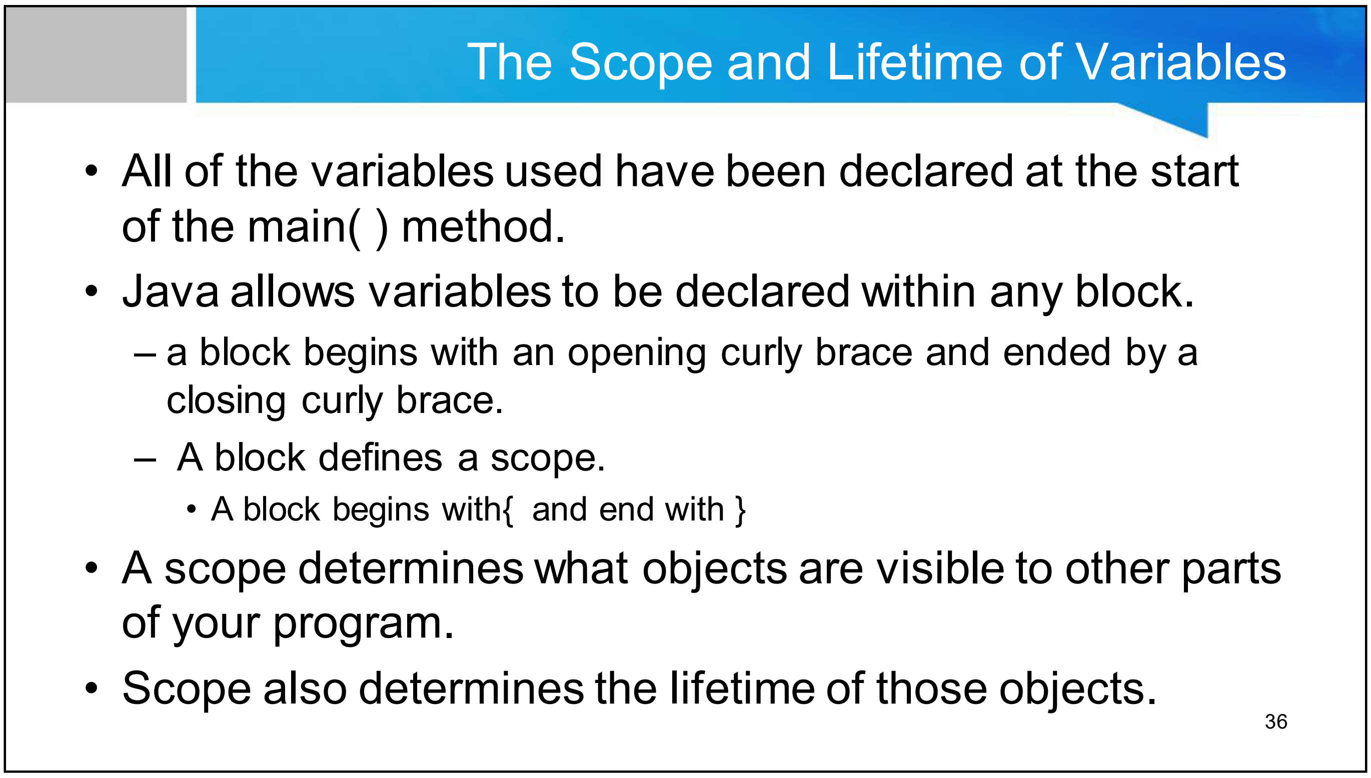


- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // c is dynamically initialized
      double c = Math.sqrt(a * a + b * b);

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



Scope and Lifetime of Variables In Java



The Scope and Lifetime of Variables

- All of the variables used have been declared at the start of the main( ) method.
- Java allows variables to be declared within any block.
  - a block begins with an opening curly brace and ended by a closing curly brace.
  - A block defines a scope.
    - A block begins with{ and end with }
- A scope determines what objects are visible to other parts of your program.
- Scope also determines the lifetime of those objects.

- Two major scopes are
  - Scope defined by a class
  - Scope defined by a method.
- Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
  - Local variable

The Scope and Lifetime of variables(contd.)

- Scopes can be nested.
  - Each time you create a block of code, we are creating a new, nested scope.
  - The outer scope encloses the inner scope.
  - This means that *objects declared in the* ***outer scope*** *will be* ***visible to code within the inner scope***.

```
{//outer
    {//inner
        {//innermost
        }
    }
}
```

The Scope and Lifetime of variables(contd.)

```
// This fragment is wrong!
count = 100;  // cannot use variable before it is declared!
int count;
```

- Variables are created when their scope is entered, and destroyed when their scope is left.
  - This means that a variable will not hold its value once it has gone out of scope.

- Variable can be reinitialized each time it enters the block in which it is declared

```
class LifeTime {
    public static void main(String args[]) {
        int x;
        for(x = 0; x < 2; x++)
        {
            int y = -1; // y is initialized each time block is entered
            System.out.println("y is: " + y);    // this always prints -1
            y = 100;
            System.out.println("y is now: " + y);
        }
}}
```

- Although blocks can be nested, you cannot **declare a variable to have the** same name as one in an **outer scope**.

```
// This program will not compile
class ScopeErr {
    public static void main(String args[])
    {       int bar = 1;
            {   // creates a new scope
            int bar = 2; // Compile-time error
            //  bar already defined in outer scope!
            }
    }
}
```

Type Conversion and Casting In Java

Type Conversion and Casting

- If the two types are **compatible**, then Java will perform the **conversion automatically(implicitly)**.
  - it is always possible to assign an **int value to a long variable.**
- The conversion between incompatible types are to be done **explicitly.**

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are **compatible**.
  - The **destination** type is **larger** than the source type.

$$\text{Destination} \xleftarrow{\hspace{3cm}} = \quad \text{source}$$

(***same*** *type or* ***larger***)

- When these two conditions are met, a ***widening conversion*** *takes place.*

Java’s Automatic Conversions(contd.)

- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
  - No automatic conversions from the numeric types to char or boolean.
- Java also performs an automatic type conversion when a literal integer constant is stored into variables of type byte, short, long, or char.

byte → short →int →long → float → double

WIDENING CONVERSION

SMALL-----------------------------------→ LARGE

Casting Incompatible Types

- If we want to assign an int value to a byte variable.
  - This conversion will not be performed automatically, because a byte is smaller than an int.

  byte variable=integer

  (small) (large) ✖

- This is called narrowing conversion.
- To create a conversion between two incompatible types, we must use a cast.

- A cast *is* simply an explicit type conversion. It has this general form:

  (target-type) value

  - target-type specifies the desired type to which value is to be converted.

  int a;

  byte b;

  b = (byte) a;

- If the integer's value is larger than the range of a byte, it will be reduced to modulo *(the remainder of an integer division)* by the byte's range(256).

- A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation.
  - If the value 1.23 is assigned to an integer, the resulting value will simply be 1.

```
int a=1.23; // here variable  a stores only 1
// .23 will have been truncated
```

- If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

E.g.

```
byte b;
int i = 257;
b=(byte) i;
```

Here byte(-128 to 127) is smaller than 257, so the value stored in b is 257 mod 256=1

- When the large value is cast into a byte variable, the *result* is the remainder of the division of value by 256 byte → short →int →long → float → double

WIDENING CONVERSION
(AUTOMATIC / IMPLICIT)

SMALL----------------------------------→ LARGE

double→ float →long →int → short → byte

NARROWING CONVERSION

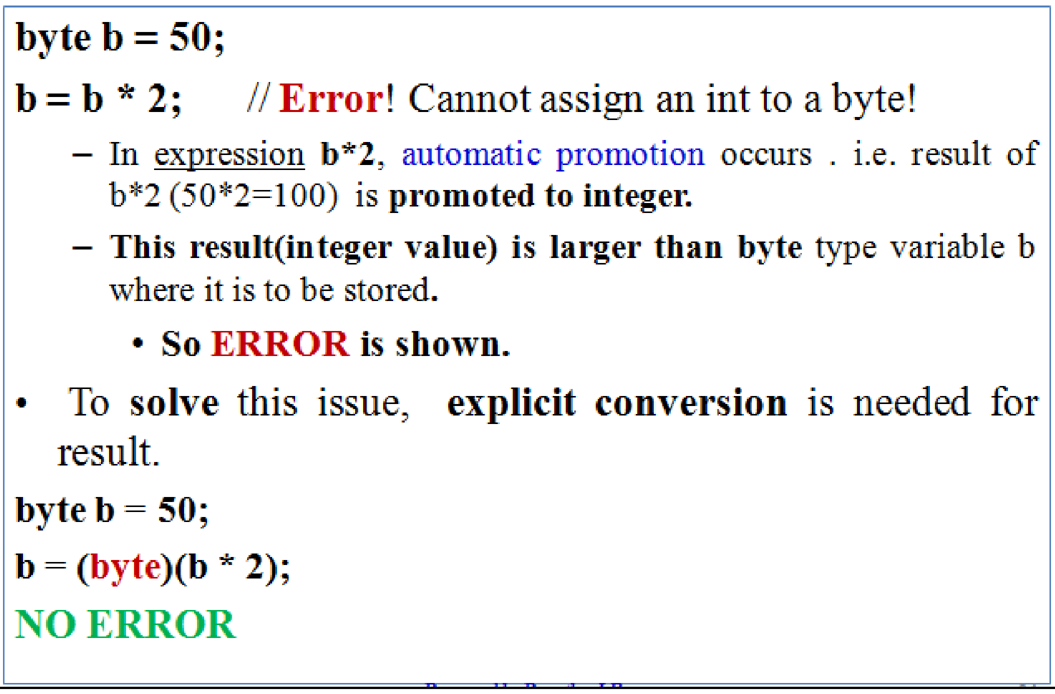
LARGE ----------------------------------→SMALL

EXPLICIT

Automatic Type Promotion in Expressions

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c; // conversions may occur in expressions.
```

- Here intermediate term a * b (40*50=2000) exceeds the range of its byte operands(-128 to 127) a and b.
- To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- So no error.
- Variable d will contain 20



byte b = 50;

b = b * 2; // **Error**! Cannot assign an int to a byte!

- In expression **b*2**, automatic promotion occurs . i.e. result of b*2 (50*2=100) is **promoted to integer.**
- **This result(integer value) is larger than byte** type variable b where it is to be stored.
  - **So ERROR is shown.**

- To **solve** this issue, **explicit conversion** is needed for result.

byte b = 50;

b = (byte)(b * 2);

NO ERROR

The Type Promotion Rules

- First, all byte, short, and char values are promoted to int.
- If one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float.
- If any of the operands is double, the result is double.

```
class Promote {

public static void main(String args[]) {

byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;

double d = .1234;

double result = (f * b) + (i / c) - (d * s);

}

}
```

f * b , b is promoted to a float (result float)

i / c, c is promoted to int, and the result is of type **int**.

d * **s**, the value of s is promoted to double – result double

float plus an **int** is a **float.**

float minus the **double** is promoted to **double**

RESULT double

Operators In Java

Operators

- An operator is a symbol that tells the computer to perform
- certain mathematical or logical manipulation.
- Java operators can be divided into following categories:
  1. Arithmetic Operators
  2. Relational Operators
  3. Bitwise Operators
  4. Logical Operators
  5. Assignment Operators
  6. conditional operator (Ternary)

Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String args[]) {
    // arithmetic using integers
    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);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

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

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

- The modulus operator, %, returns the remainder of a division operation.
- It can be applied to floating-point types as well as integer types

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;

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

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

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

- Java provides special operators that can be used to combine an arithmetic operation with an assignment.

$\boxed{a = a + 4;} \Rightarrow \boxed{a \mathrel{+}= 4;}$

- There are compound assignment operators for all of the arithmetic, binary operators.

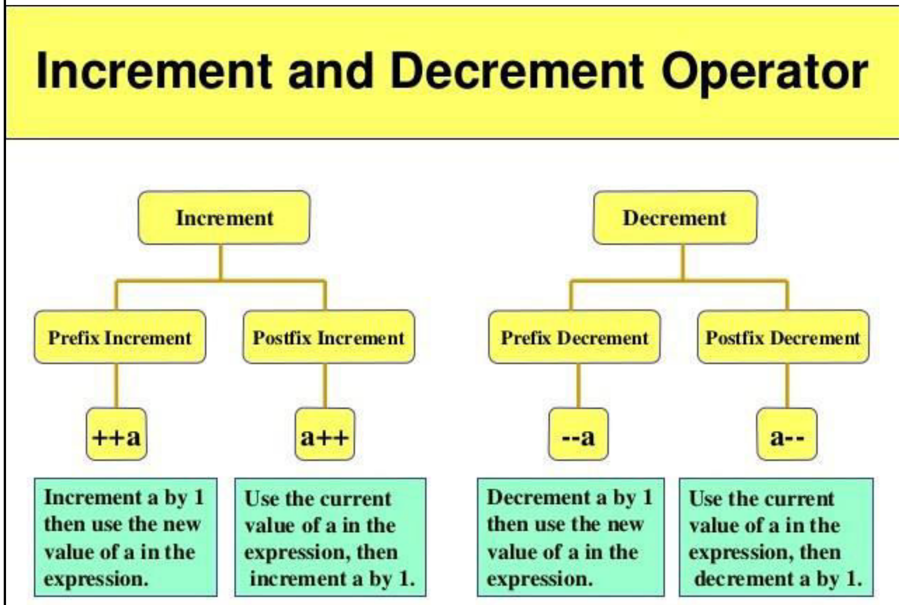
var = var op expression; $\Rightarrow$ var op= expression;

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement


Increment and Decrement Operator
Increment
Decrement
Prefix Increment
Postfix Increment
Prefix Decrement
Postfix Decrement
++a
a++
--a
a--
Increment a by 1 then use the new value of a in the expression.
Use the current value of a in the expression, then increment a by 1.
Decrement a by 1 then use the new value of a in the expression.
Use the current value of a in the expression, then decrement a by 1.

- The ++ and the – – are Java’s increment and decrement operators
- The increment operator increases its operand by one.
- The decrement operator decreases its operand by one

Pre-Increment Post increment(Prefix/Postfix)

- In prefix form the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form the previous value is obtained for use in the expression, and then the operand is modified.
  - Pre increment E.g

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

x 43
y 43

  - Post increment E.g

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

x 43
y 42

```
// Demonstrate ++.
class IncDec {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
    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 = 4
d = 1
```

Bitwise operators

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

Bitwise logical oprators

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Examples

```
  00101010    42
& 00001111    15
----------
  00001010    10
```

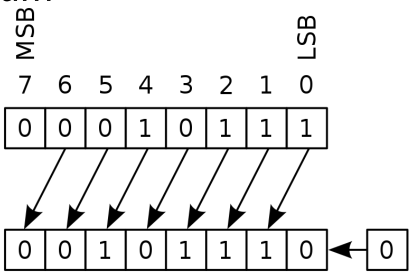
```
  00101010    42
| 00001111    15
----------
  00101111    47
```

```
  00101010    42
^ 00001111    15
----------
  00100101    37
```

```
~00101010
becomes
 11010101
```

Left shift

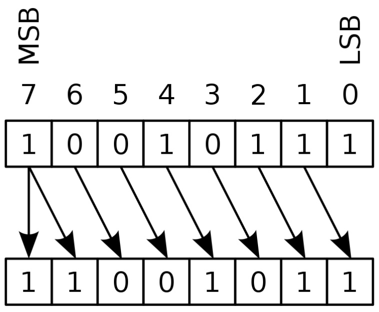
- The left shift operator, **<<, shifts all of the bits in a value to the left a specified number of times.**
- It has this general form: *value << num*
- *num specifies the number of positions to left-shift the value in value.*
- **<<** moves all of the bits in the specified value to the left by the number of bit positions specified by *num*



Right shift

- Each time you shift a value to the right, it divides that value by two—and discards any remainder.
- When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit.
- This is called sign extension and serves to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4

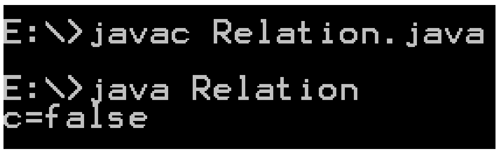
```
11111000    –8
>>1
11111100    –4
```



- E.g.
```
int a = 32;
a = a >> 2; // a now contains 8
```
- E.g.
```
int a = 35;
a = a >> 2; // a still contains 8
```
```
00100011    35
>> 2
00001000     8
```

Boolean Logical Operators

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



```
E:\>javac Relation.java

E:\>java Relation
c=false
```

int a = 4;

int b = 1;

boolean c = a < b;//c contains false. 4 is not less than 1

Here the result of **a<b (**which is **false)** is stored in c.

E.g.

int done;

if(!done) ... // Valid in C/C++

if(done) ... // but not valid in Java.

- The logical Boolean operators, **&, |, and ^, operate on boolean values in the same way** that they operate on the bits of an integer.

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

Short-Circuit Logical Operators

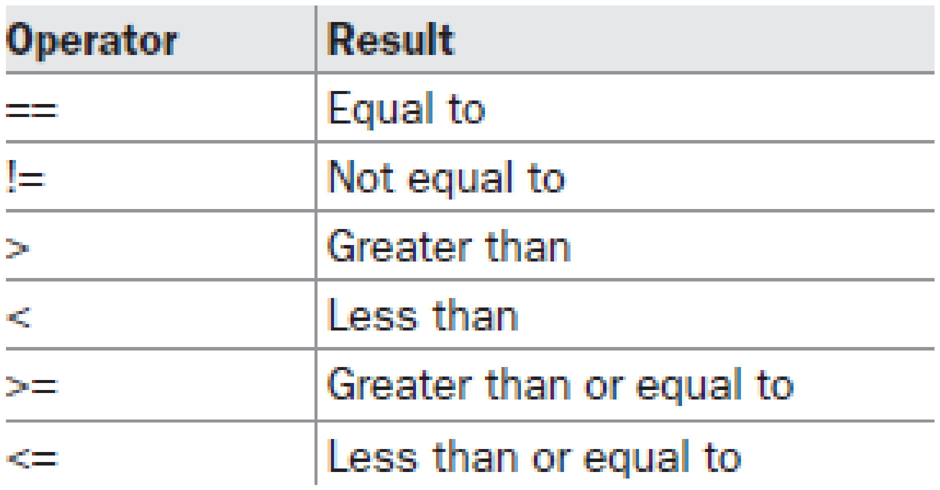
- Secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators.
- The OR operator results is true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is.
- If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

- E.g

if (**denom != 0** && num / denom > 10)

- Here if denom is 0 the second expression is not validated
  - So there is no risk of causing a run-time exception when denom is zero.
- If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero.

Relational operators



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

Assignment Operator

- var = expression;
- Here, the type of var must be compatible with the type of expression.
- It allows you to create a chain of assignments

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

Ternary (conditional or three-way) operator

- The ? Operator has this general form:

expression1 ? expression2 : expression3

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

```
int a=3,b=5;
int c=(a>b?a:b);
```

- Here a>b is **false** so the value of b is stored in c.

Operator Precedence

- Operator precedence determines the order in which the operators in an expression are evaluated.

int myInt = 12 - 4 * 2;

- What will be the value of myInt? Will it be (12 - 4)*2, that is, 16? Or it will be 12 - (4 * 2), that is, 4?
- When two operators share a common operand, 4 in this case, the operator with the highest precedence is operated first.
- In Java, the precedence of * is higher than that of -. Hence, the multiplication is performed before subtraction, and the value of myInt will be 4.

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

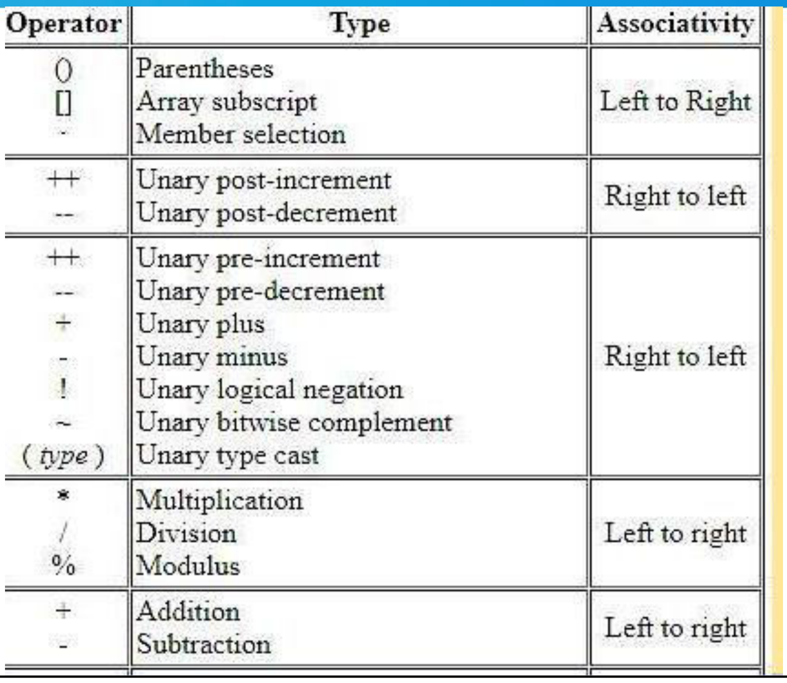
Associativity of operators

- When an expression has two or more operators with the same precedence, the expression is evaluated according to its **associativity**.
  - It is the order of applying operators

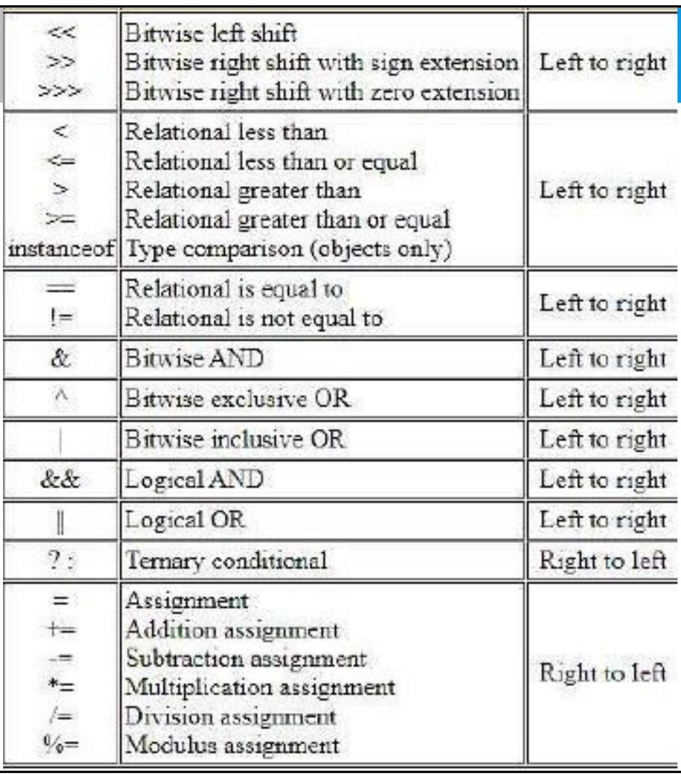
a = b = c;

- Here, the value of c is assigned to variable b. Then the value of b is assigned of variable a. Why? It's because the associativity of = operator is from right to left.

Operator Associativity



Operator	Type	Associativity
() [] .	Parentheses Array subscript Member selection	Left to Right
++ --	Unary post-increment Unary post-decrement	Right to left
++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
* / %	Multiplication Division Modulus	Left to right
+ -	Addition Subtraction	Left to right



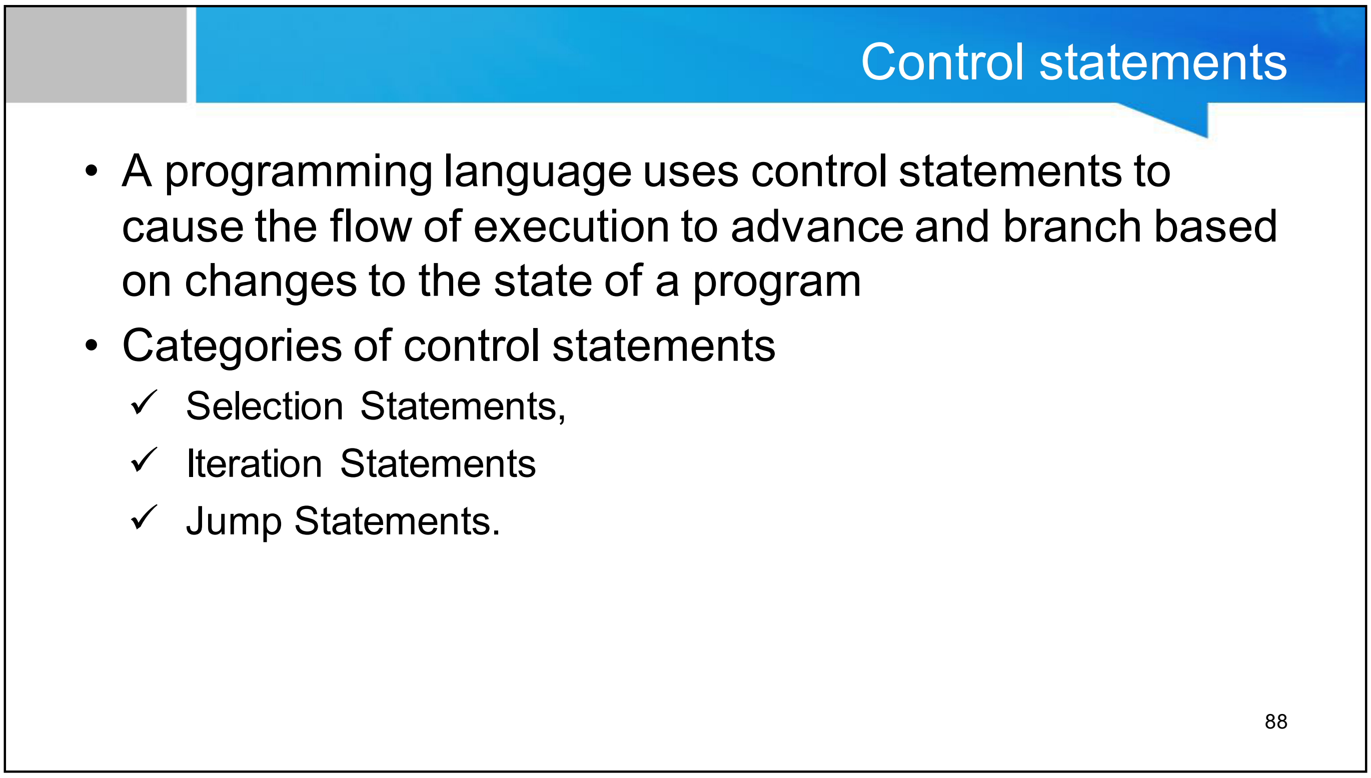
<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
== !=	Relational is equal to Relational is not equal to	Left to right
&	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
? :	Ternary conditional	Right to left
= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Associativity

- Right to Left associative
  - – Unary operators
  - – Assignment operators
  - – Conditional(ternary) operators)
- All other operators are Left to Right associative

Control statements (if) In Java

87



Control statements

- A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program
- Categories of control statements
  - ✓ Selection Statements,
  - ✓ Iteration Statements
  - ✓ Jump Statements.

88

- Selection statements allow the program
  - to choose different paths of execution based on condition (outcome of an expression or the state of a variable).
- Iteration statements enable program execution
  - to repeat one or more statements (that is, iteration statements form loops).
- Jump statements allow your program
  - to execute in a nonlinear fashion.

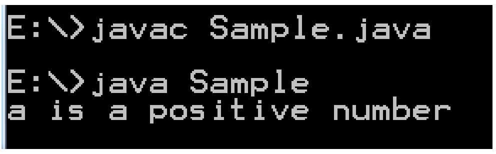
- Also called decision making statements.
- Selection statements control the flow of program's execution based upon conditions known only during run time. It helps to choose different paths of execution based on condition.
- Java supports two selection statements:
  - ✓if
  - ✓switch

if statement

- if statement is Java's conditional branch statement.
  - ✓It can be used to route program execution through different paths.
- Syntax of simple if statement

```
if (condition)
{
 // block of code to be executed if the condition is true
..................
}
```

```
class Sample{
public static void main(String args[])
{
    int a=5;
    if(a>0)
    {
      System.out.println(" a is a positive number");
    }
}
}
```



```
E:\>javac Sample.java

E:\>java Sample
a is a positive number
```

If-else statement

- General form of the if statement:

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

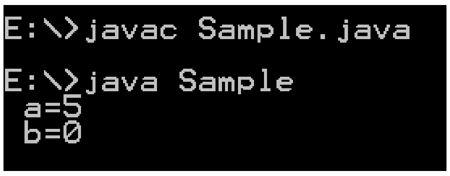
- Statement may be a single statement or a compound statement enclosed in curly braces (that is, a block).
- The condition is any expression that returns a boolean value.
- The else clause is optional.

Working of if-else

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

- If the condition is true, then statement1 is executed.
- Otherwise, statement2 (if it exists) is executed.
- Both statements will not be executed at the same time.

```
class Sample{
public static void main(String args[]) {
int a=5, b=3;
if(a < b) a = 0;
else b = 0;
System.out.println(" a=" + a);
System.out.println(" b=" + b);
}
}
```



```
E:\>javac Sample.java

E:\>java Sample
 a=5
 b=0
```

- If statement can be controlled using a boolean variable.
- E.g.

```
...
boolean dataAvailable;
// ...
if (dataAvailable)          //if dataAvailable is true
   ProcessData();           //call this function
else
   waitForMoreData();  //call this function
..
```

Nested ifs

- A nested if is an if statement that is the inside (target of) another if or else.
- The else statement always refers to the
  - nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if(i == 10)
   {
   if(j < 20) a = b;
   if(k > 100) c = d;              // this if is
   else a = c;                     // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

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

- The if statements are executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If NONE of the conditions is true, then the final else statement will be executed.
- The last else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

```
   April is in the Spring.
```

Control statements (switch) In Java

Switch statement

- The switch statement is Java’s multiway branch statement.
- It is an better alternative than a large series of if-else-if statements.

Syntax of switch

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

switch(expression){…..}

- The expression inside switch must be of type byte, short, int, or char;
  - each of the values specified in the case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement)

Working of switch

- The value of the expression inside switch is compared with each of the literal values in the case statements.
  - If a match is found, the code sequence following that case statement is executed.
  - If none of the constants in the case matches the value of the expression, then the default statement is executed.
  - default statement is optional.
  - If no case matches and no default is present, then no further action is taken.

- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code after the entire switch statement.
- This has the effect of "jumping out" of the switch.

```
// A simple example of the switch.
class SampleSwitch {
  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.
```

```
// In a switch, break statements are optional.
class MissingBreak {
  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

- We can use a switch as part of the statement sequence of an outer switch. This is called a nested switch

```
switch(count) {
   case 1:
        switch(target) { // nested switch
                      case 0:
                             System.out.println("target is zero");
                             break;
                      case 1:  // no conflicts with outer switch
                             System.out.println("target is one");
                             break;
                      }
        break;
   case 2: // ...   }
```

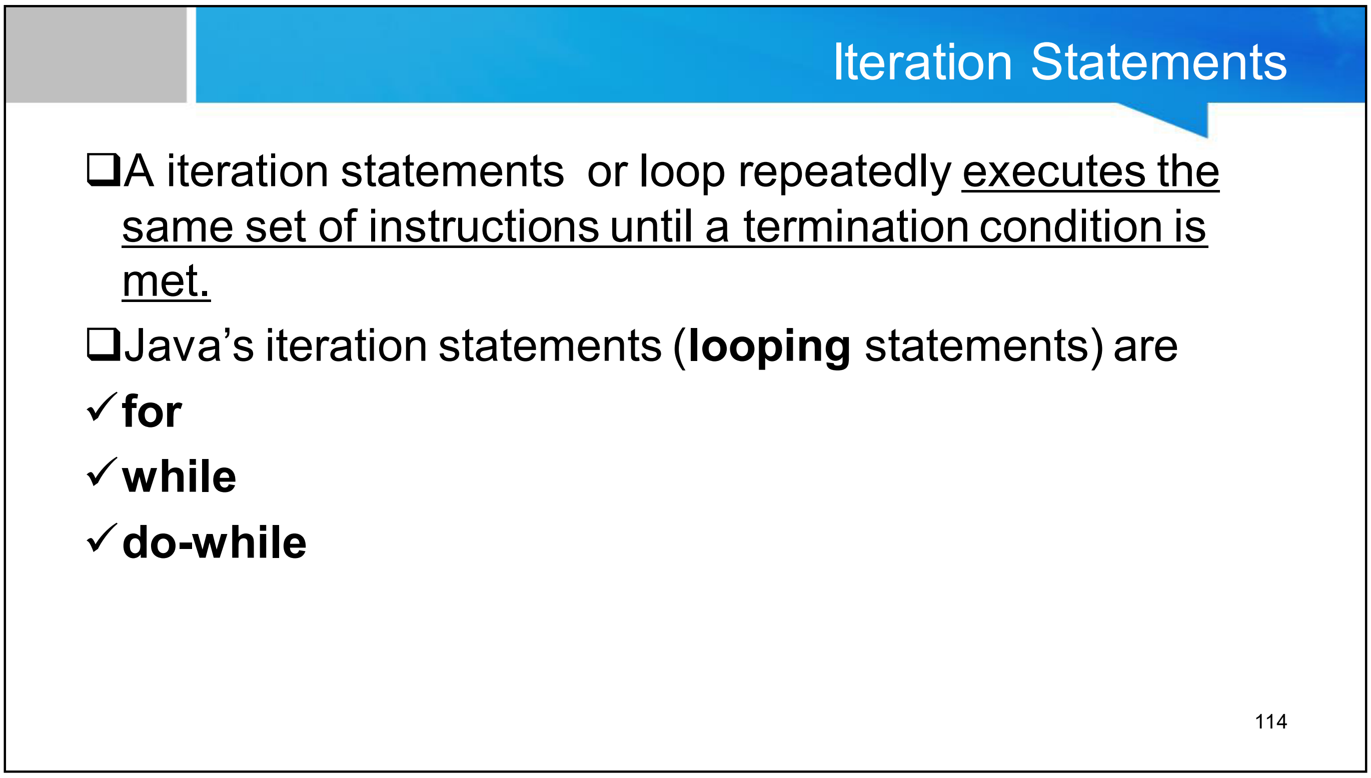
Features of the switch statement

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression.
  - switch looks only for a match between the value of the expression inside switch and one of its case constants.
- No two case constants in the same switch can have identical values.
  - But a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs.

Switch(features)

- When Java compiler compiles a switch statement, it will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression.
- So a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses.
- The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression.
- The compiler has no such knowledge of a long list of if expressions

Iteration statements In Java (while)



Iteration Statements

- ❑A iteration statements or loop repeatedly executes the same set of instructions until a termination condition is met.
- ❑Java's iteration statements (**looping** statements) are
- ✓**for**
- ✓**while**
- ✓**do-while**

while

- The while loop is Java's most fundamental loop statement. It is ENTRY CONTROLLED loop.
  - The statements inside the body of while is executed only if the condition inside while is true.
- It repeats a statement or block while its controlling expression is true.
- General form:

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

Working of while

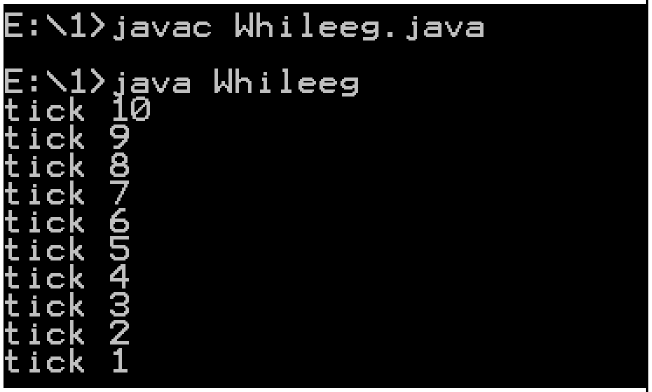
```
while(condition)
{
// body of loop
}
```

- The condition can be any Boolean expression.
  - The body of the loop will be executed as long as the conditional expression is true.
  - When condition becomes false, control passes to the next line of code immediately after the loop.

- The curly braces are not needed if only a single statement is being repeated.

```
while(condition)
   Statement;
```

```
// Demonstrate the while loop.
class Whileeg {
    public static void main(String args[]) {
        char  ch='a'
        while(ch=='a') {
            System.out.println("tick " + n);
            n--;
        }
    }
}
```



```
E:\1>javac Whileeg.java

E:\1>java Whileeg
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

- The body of the while (or any other of Java's loops) can be empty.
  - This is because a **null statement** (one that consists only of a **semicolon**) is syntactically valid in Java.

```
while(condition) ;
```

Here if condition is true no statement is executed as part of while

```
class Whileeg {
public static void main(String args[])
{
   float n = 10.5;
   while(n > 0)
    {
   System.out.println("tick " + n);
   }
}
}
```

OUTPUT

```
tick 10
tick 10
tick 10
…..

.



.
```

INFINITE LOOP

Iteration statements In Java (do-while)

do-while

- The **do-while loop always executes its body at least** once, because its conditional expression is at the bottom of the loop.

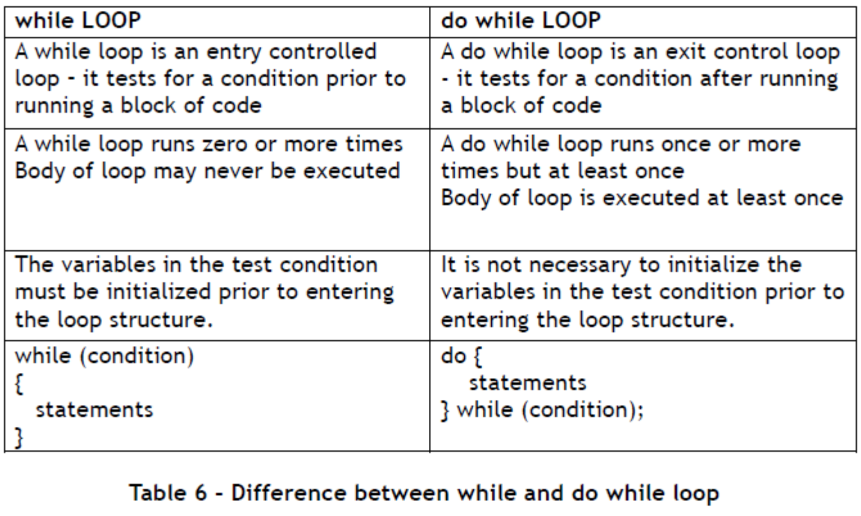
```
do
{
   //statements
}
while(condition);
```

Working of do-while

do-while is EXIT CONTROLLED loop.

```
do
{
  //statements
}
while(condition);
```

1. Initially the **statements** inside the do-while loop is executed
2. then only the **condition** inside while is checked.
3. Then the loop is executed only if that condition is true.
   – That is condition is checked only during exit from do-while loop.



while LOOP	do while LOOP
A while loop is an entry controlled loop - it tests for a condition prior to running a block of code	A do while loop is an exit control loop - it tests for a condition after running a block of code
A while loop runs zero or more times Body of loop may never be executed	A do while loop runs once or more times but at least once Body of loop is executed at least once
The variables in the test condition must be initialized prior to entering the loop structure.	It is not necessary to initialize the variables in the test condition prior to entering the loop structure.

while (condition)
{
statements
}
	do {
	statements
	} while (condition);

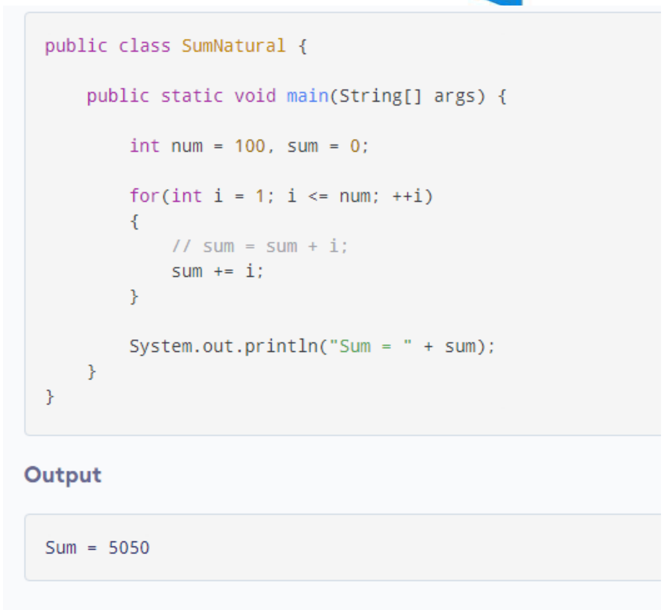
Table 6 - Difference between while and do while loop

Iteration statements In Java (for)

for

- It is an iteration statement
- looping

```
for(initialization; condition; iteration) {
    // body
}
```



```
public class SumNatural {

    public static void main(String[] args) {

        int num = 100, sum = 0;

        for(int i = 1; i <= num; ++i)
        {
            // sum = sum + i;
            sum += i;
        }

        System.out.println("Sum = " + sum);
    }
}
```

Output

```
Sum = 5050
```

Working of for loop

- When the loop first starts, the **initialization** portion of the loop is executed. It acts as a loop control variable (counter).
  - the initialization expression is only executed once.
- Next, condition is evaluated.(Boolean expression)
  - It usually tests the loop control variable against a target value.
  - If this expression is **true**, then the body of the loop is executed.
  - If it is **false**, the loop terminates.
- Next, the iteration portion of the loop is executed.
  - increments or decrements the loop control variable.
- Next, condition is evaluated.
- And the process continues until condition becomes **false**

```
for( ; ; )
{
// ...
}
```

INFINITE LOOP

The For-Each Version of the for Loop

```
for(type var : collection)
    statement-block;
```

- Here, type specifies the type and var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
    System.out.println(x);
```

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

- With each pass through the loop, x is automatically given a value equal to the next element in nums.
  - Thus, on the first iteration, x contains 1;
  - on the second iteration, x contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

Nested loops

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print("i="+i+" j="+j + "\t\t");
System.out.println();
}
}
}
```

OUTPUT

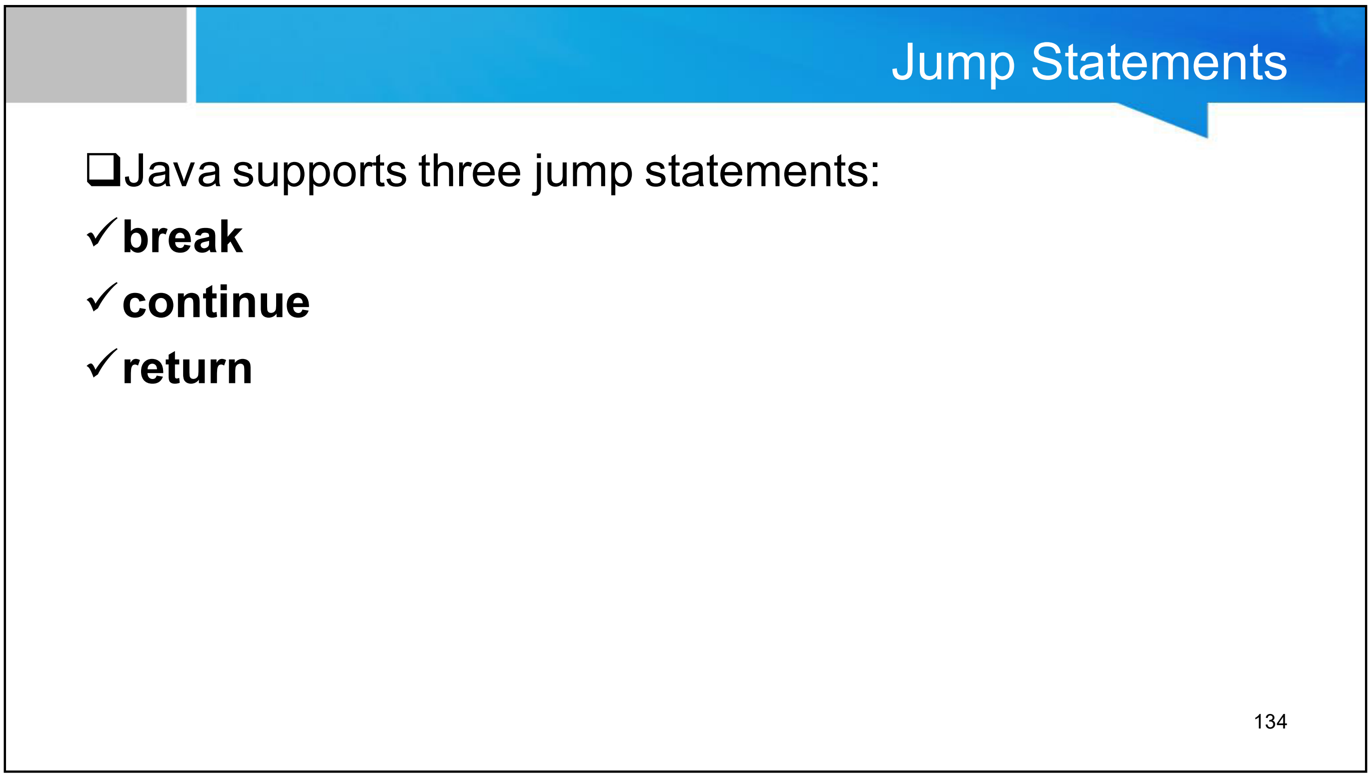
```
i=0 j=0     i=0 j=1     i=0 j=2
i=1 j=0     i=1 j=1     i=1 j=2
i=2 j=0     i=2 j=1     i=2 j=2
i=3 j=0     i=3 j=1     i=3 j=2
```

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print(i +"\t\t");
System.out.println();
}
}
}
```

OUTPUT

```
0    0    0
1    1    1
2    2    2
3    3    3
```

Jump statements In Java



Jump Statements

❑Java supports three jump statements:

✓**break**

✓**continue**

✓**return**

break statement

- Three uses.

✓First it **terminates** a statement sequence in a switch statement.

✓Second, it can be used to **exit** a loop.

✓Third, it can be used as a “civilized” form of goto.

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
    for(int i=0; i<6; i++)
    { if(i == 3)
          break; // terminate loop if i is 3
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
    }
}
```

OUTPUT

```
i: 0
i: 1
i: 2
Loop complete.
```

Using break as a Form of Goto

- By using this form of **break, you can, for example, break out of** one or more blocks of code.
- The general form of the labeled break statement is :

  break *label;*

```
// Using break as a civilized form of goto.
class Breakeg {
    public static void main(String args[]) {
        boolean t = true;
        first: {
            second: {
                third: {
                System.out.println("Before the break.");
                if(t) break second; //break of second block
                System.out.println("This won't execute");
                }
            System.out.println("This won't execute");
            }
          System.out.println("After second block.");
          }
}}
```

OUTPUT

```
Before the break.
After second block..
```

continue statement

- In while and do-while loops, a continue statement causes control to be transferred directly 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.
- For all three loops, any intermediate code after continue is bypassed(skipped).

```
// Using break to exit a loop.
class continueeg {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
        {
            if(i == 3)
              continue; // skip remaining stmts if i is 3
             // continue loop.control goes to iteration
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

OUTPUT

```
i: 0
i: 1
i: 2
i: 4
i: 5
Loop complete.
```

return statement

- The return statement is used to explicitly return from a method.
  - The return causes program control to transfer back to the caller of the method.
- When return statement is executed the method terminates.
- The return causes execution to return to the Java run-time system
- Methods that have a return type other than void return a value to the calling method(function)

return *value;*

  - Here, *value is the value is returned to the calling function*

```
// Demonstrate return.
class Return {
    public static void main(String args[]) {
        boolean t = true;
        System.out.println("Before the return.");
        if(t) return;
        System.out.println("This won't execute.");
    }
}
```

OUTPUT
```
Before the return
```

Arrays In Java

Arrays

- An array is a group of like-typed(same type) variables that are referred to by a common name.
- Arrays of any type can be created
- Arrays may have one or more dimensions.
- A specific element in an array is accessed by its index.
  - Index means position It starts from 0.
    - Index of first element is 0, second element is 1 etc.

One-Dimensional Arrays

- create an array variable of the desired type.
- Declaration syntax 1

  type variablename[ ];

  E.g. int a[];
- Declaration syntax 2

  type[ ] variablename;
- The following two declarations are equivalent:

  int a[];

  int[]a;

  Here this declaration means that a is an array variable, but no array actually exists. No space is allocated for it in memory

- We have to link array with an actual, physical array of integers.
- So we must allocate space using new and assign it to array variable .
  - new is a special operator that allocates memory.

  variable=new type[size];

  E.g.

  int a[];

  a= new int[12];

  int a[]=new int[12];

  - After this statement executes, variable a will refer to an array of 12 integers

- Obtaining an array is a two-step process.
  1. First, we must **declare** a variable of the desired array type.
  2. Second, we must **allocate the memory** that will hold the array, using **new**, and assign it to the array variable
- In Java all arrays are ***dynamically allocated.***
- It is possible to combine the declaration of the array variable with the allocation.

E.g.

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

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

Store value in array

```
class Array {
public static void main(String args[])
  {
       int a[];
       a = new int[4];
       a[0] = 1;
       a[1] = 3;
       a[2] = 2;
       a[3]=5;
   }
   }
```

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

Array initialization

- Arrays can be initialized(give values) when they are declared.
- An array initializer is a list of comma-separated expressions surrounded by curly braces.
- No need for new operator

```
class AutoArray {
    public static void main(String args[])
{
    int a[] = { 1,3,2,5};
}
}
```

- If you try to access elements **outside** the range of the array (negative numbers or numbers greater than the length of the array), it will cause a **run-time error**.
- E.g

```
int a[]=new int[10];
a[-3]=5;//ERROR
a[11]=7; //ERROR ARRAY INDEX OUT OF BOUNDS
```

```
// Average value in an array.
class Average {
public static void main(String args[])
{
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;
    for(i=0; i<5; i++)
      result = result + nums[i];
    System.out.println("Average is " + result / 5);
    }
}
```

int[] num1, nums2, nums3; // create three arrays

– creates three array variables num1,num2,num3 of type int.

- It is the same as writing

int num1[], nums2[], nums3[];

Multidimensional Arrays

- Multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- E.g 2 D array declaration

int b[][]= new int[4][5];

This allocates a 4 by 5 array and assigns it to variable **b.**

4 rows and 5 columns

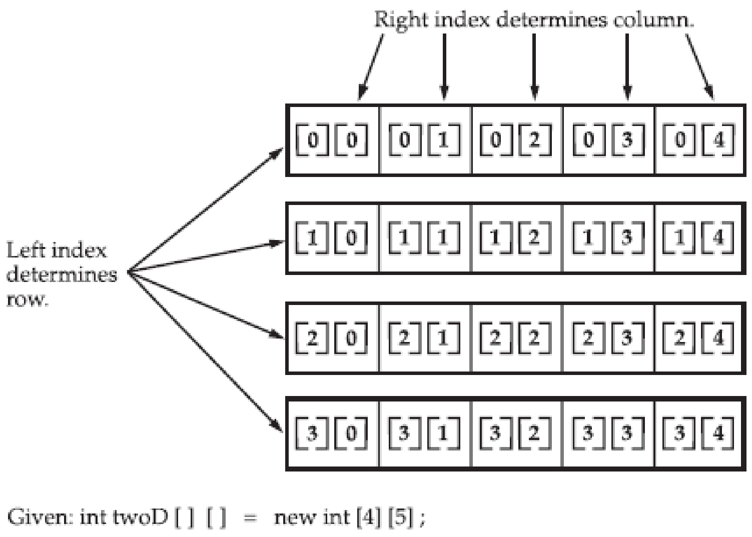


FIGURE 3-1 A conceptual view of a 4 by 5, two-dimensional array

- The following declarations are also equivalent:

```
char twod[][] = new char[3][4];
char[][] twod = new char[3][4];
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.

```
int a[][] = new int[2][];

a[0] = new int[3];

a[1] = new int[3];
```

$\leftarrow$

```
int a[][]= new int[2][3];
```

  - Here a is 2D array with two rows. First row a[0] has 3 columns. Second row a[1] has 3 columns.

```
class TwoDArray {
public static void main(String args[]) {
int a[][]= new int[2][3];
int i, j, k = 0;
for(i=0; i<2; i++)
   {
   for(j=0; j<3; j++)
    {
       a[i][j] = k;
       k++;
   }
   }
for(i=0; i<2; i++)
   { for(j=0; j<3; j++)
   {System.out.print(a[i][j] + " ");}
     System.out.println();
   } }}
```

```
OUTPUT
0 1 2
3 4 5
```

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

  int a[][] = new int[2][];

  a[0] = new int[1];

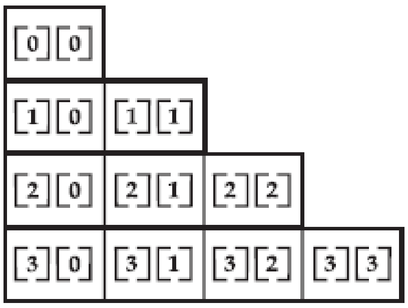
  a[1] = new int[2];

  – Here array **a** has 2 rows.

  – First row a[0] has 1 column.

  – Second row a[1] has 1 column.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  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];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {

          twoD[i][j] = k;
          k++;
        }

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



This program generates the following output:

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

Multidimensional array initialization

- Enclose each dimension's initializer(values) within its own set of curly braces.
- We can use **expressions** as well as **literal values** inside of array initializers.
- Eg.

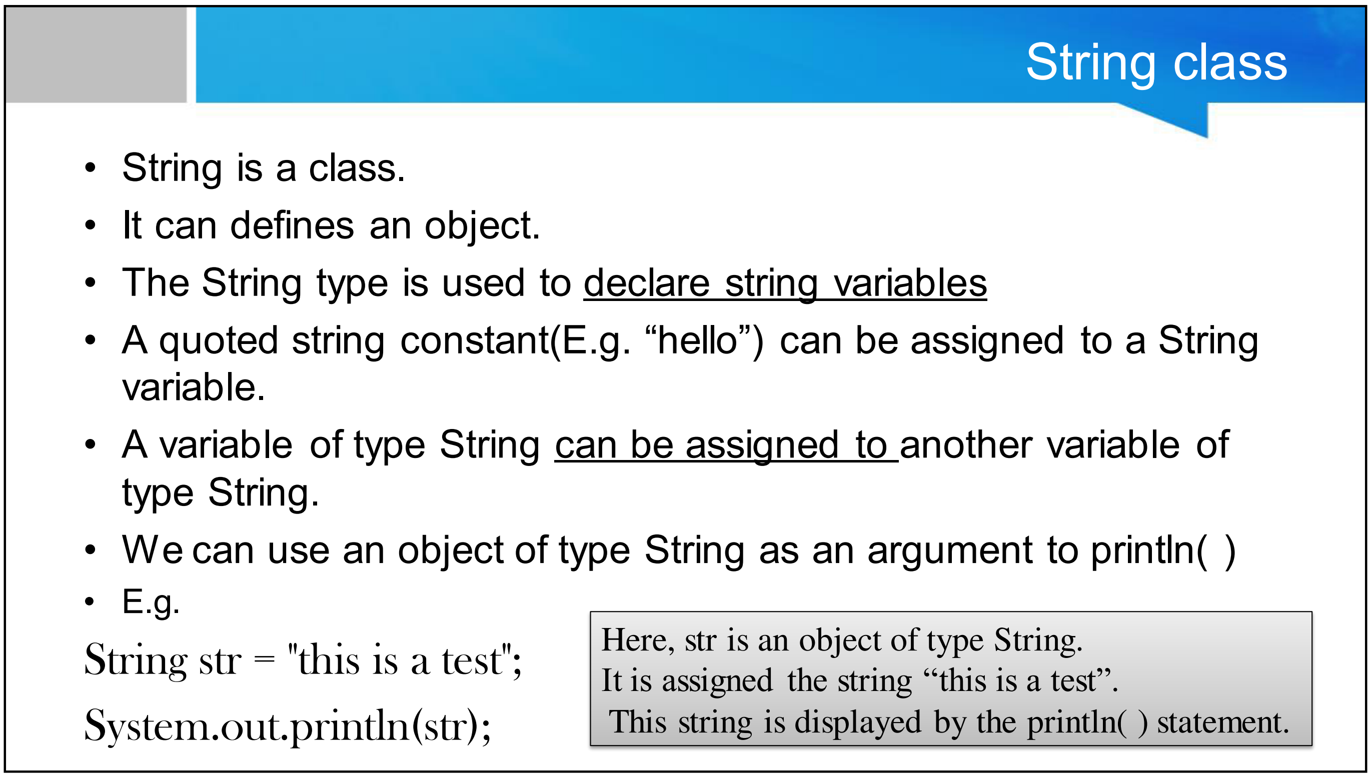
```
int a[][]={ {1,2,3} , {3,4,5}} ;
```

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

OUTPUT

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

String Class In Java



String class

- String is a class.
- It can defines an object.
- The String type is used to declare string variables
- A quoted string constant(E.g. “hello”) can be assigned to a String variable.
- A variable of type String can be assigned to another variable of type String.
- We can use an object of type String as an argument to println( )
- E.g.

```
String str = "this is a test";
System.out.println(str);
```

Here, str is an object of type String.
It is assigned the string “this is a test”.
This string is displayed by the println( ) statement.

```
class Sample {
public static void main(String args[])
    {
    String s="Hello"
    System.out.print(s);
    }
}
```

```
OUTPUT
Hello
```

- In Java, string is basically an object that represents sequence of char values .
- An array of characters works same as Java string.
- For example:

```
char[] ch={'H','e','l','l','o'};
String s=new String(ch);
   /*This statement converts character array ch to string and store
   in string object s.*/
```

This is same as

```
String s="Hello"; //creating string by java string literal
```

String methods

- length()
  - The length of a string can be found with the length() method.

```
class Sample {
   public static void main(String args[])
    {
    String s="Hello";
    System.out.print("Length=",s.length());
    }
}
```

```
OUTPUT
Length=5
```

- toUpperCase() and toLowerCase()
  - To convert from lower to upper and upper to lower respectively

```
class Sample {
public static void main(String args[])
    {
    String s="Hello World";
    System.out.println(s. toUpperCase());
    System.out.println(s. toLowerCase());
    }
}
```

```
OUTPUT
HELLO WORLD
hello world
```

- indexOf()
  - The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace)

```
class Sample {
public static void main(String args[])
    {
    String s="I am fine.I am ok";
    System.out.println(s.indexOf("am"));
    }
}
```

OUTPUT

```
2
```

- String concatenation
  - Method 1:The + operator can be used between strings to combine them. This is called concatenation
  - Method 2:We can use **concat()** method to concatenate two strings.

```
class Sample {
public static void main(String args[])
{
  String s1="Sachin ";
   String s2="Tendulkar";
   String s3=s1+s2;
   System.out.println(s3);//Sachin  Tendulkar
}}
```

OUTPUT

```
Sachin Tendulkar
```

- If we add a number and a string, the result will be a string concatenation.

```
class Sample {
public static void main(String args[])
    {
    String s1="10 " , s2="12";
    int a=13;
    System.out.println(s1+s2);
    System.out.println(s1+a);
    }
}
```

OUTPUT
```
1012
1013
```

Vector Class In Java

Vector class

- Vector implements a dynamic array
- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

- All vectors start with an initial capacity(size).
- After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects.
- The amount of extra space allocated during each reallocation is determined by the *increment* that you specify when you create the vector.
- If we don't specify an *increment*, the vector's size is doubled by each allocation cycle.

- Vector is declared like this:

```
class Vector<E>
```

- Here, E specifies the type of element that will be stored.
- Vector constructors are

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

- Vector( ) creates a default vector, which has an initial size of 10.
- Vector(int size) creates a vector whose initial capacity is specified by size.
- Vector(int size, int incr) creates a vector whose minitial capacity is specified by size and whose increment is specified by incr.
  - The increment specifies the number of elements to allocate each time that a vector is resized upward.
- Vector(Collection<? extends E> c) creates a vector that contains the elements of collection c.