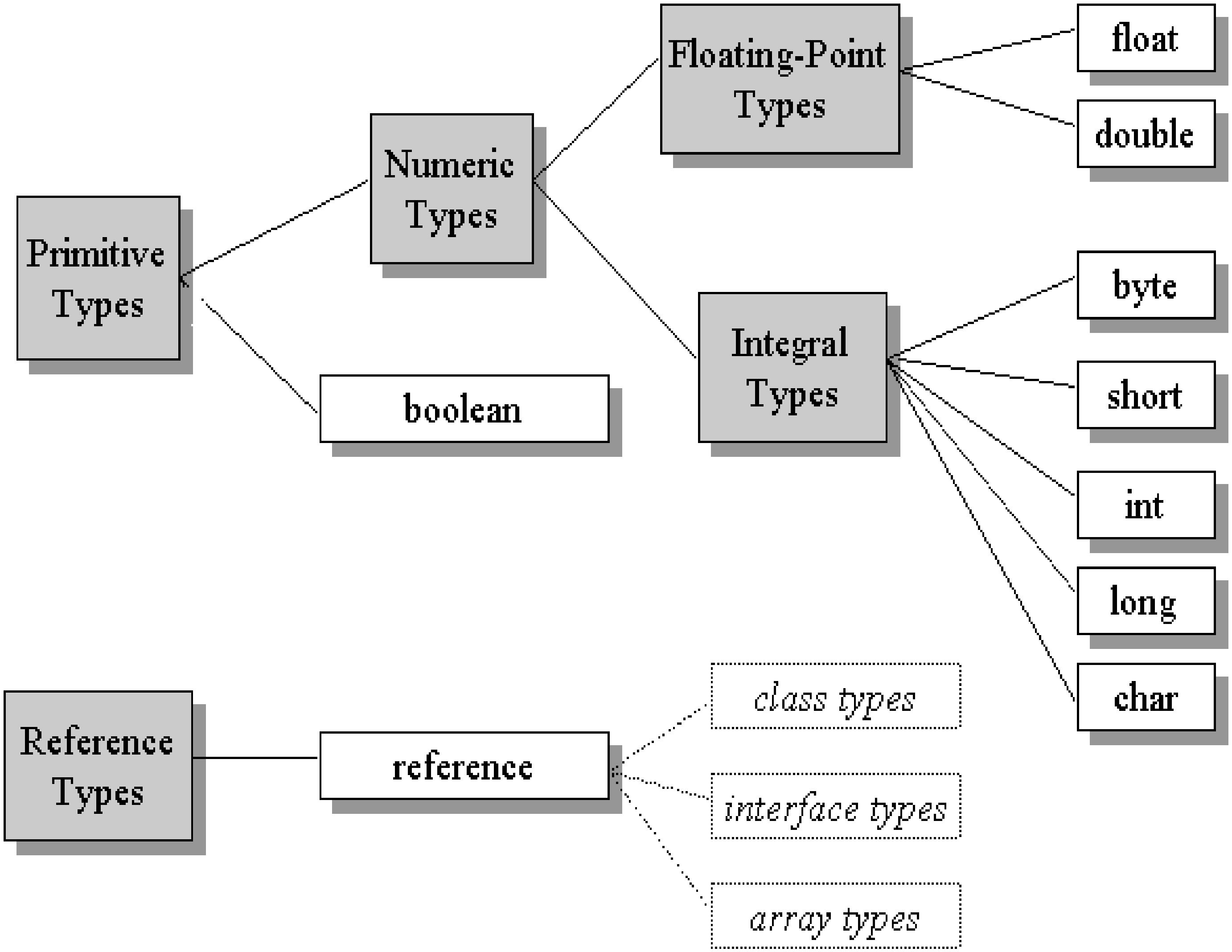
Fundamental Programming Structures in java

Data Types


Primitive Types
Numeric Types
Floating-Point Types
float
double
Integral Types
byte
short
int
long
char
boolean
Reference Types
reference
class types
interface types
array types

Java Primitive Data Types

Type	Values	Default	Size	Range
byte	signed integers	0	8 bits	-128 to 127
short	signed integers	0	16 bits	-32768 to 32767
int	signed integers	0	32 bits	-2147483648 to 2147483647
long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	+/-1.4E-45 to +/-3.4028235E+38, +/-infinity, +/-0, NAN
double	IEEE 754 floating point	0.0	64 bits	+/-4.9E-324 to +/-1.7976931348623157E+308, +/-infinity, +/-0, NaN
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
boolean	true, false	false	1 bit used in 32 bit integer	NA

To print default value

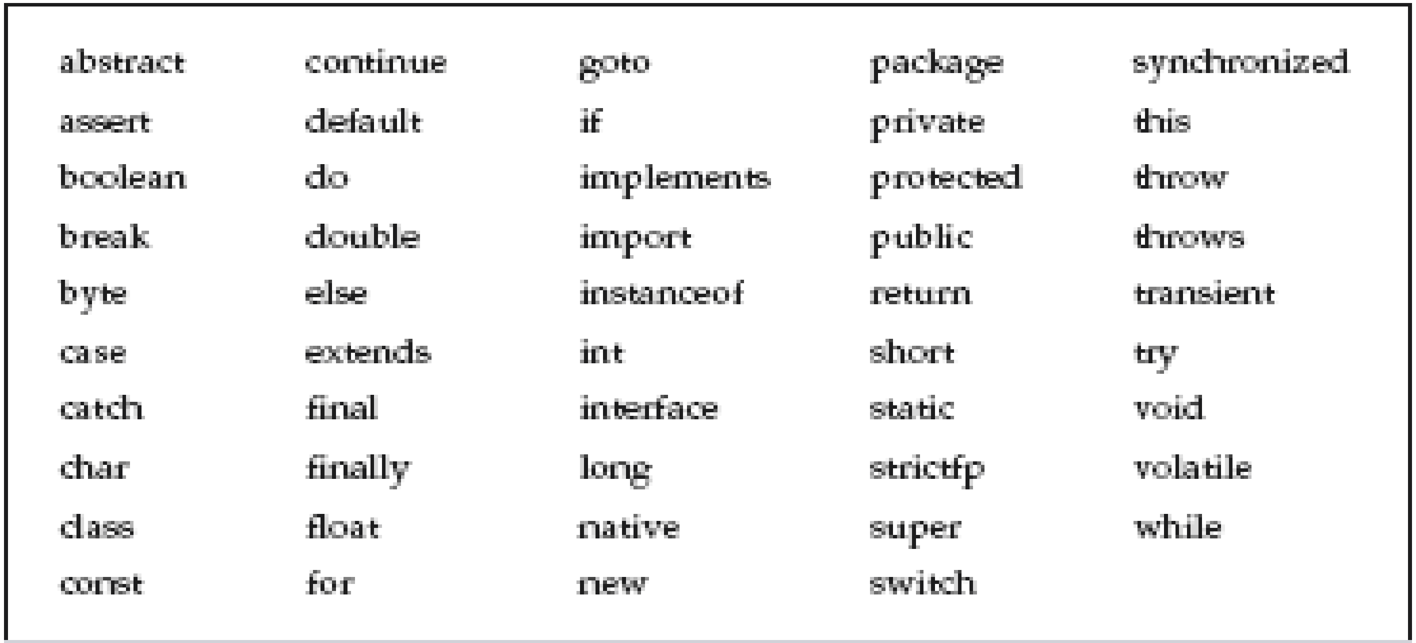
```
public class First
{
    static boolean x;
    public static void main(String args[])
    {
            System.out.println(“value of x:”+x);
    }
}
```

Java Tokens

- Smallest individual units in a program are known as tokens.
- 5 types:
  - Reserve keyword
  - Identifier
  - Literals
  - Operators
  - Separators

Keywords

- Written in lower case
- Keyword cannot be used as names for a variable, classes, methods and so on.
- 49 reserved words:



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

Identifiers

- Identifiers
  - Used to name local variables
  - Names of attributes
  - Names of classes, methods, objects, labels, packages and interfaces

Java Identifiers

- Naming Rules
  - Must start with a letter
  - After first letter, can consist of letters, digits (0,1,…,9)
  - The underscore “_” and the dollar sign “$” are considered letters
- Variables
  - All variables must be declared in Java
  - Can be declared almost anywhere (scope rules apply)
  - Variables have default initialization values
    - Integers: 0
    - Reals: 0.0
    - Boolean: False
  - Variables can be initialized in the declaration
  - *type identifier [= value][, identifier [= value] ...] ;*

Java Identifiers

- Example Declarations

```
int speed;                          // integer, defaults to 0
int speed = 100;                    // integer, init to 100
long distance = 3000000000L;        // “L” needed for a long
float delta = 25.67f;               // “f” needed for a float
double delta = 25.67;               // Defaults to double
double bigDelta = 67.8E200d;        // “d” is optional here
boolean status;                     // defaults to “false”
boolean status = true;

char c = 88;                        //code for X
c++;                                // output: Y
```

- Potential Problems (for the C/C++ crew)

```
long double delta = 3.67E204;  // No “long double” in Java
unsigned int = 4025890231;   // No unsigned ints in Java
```

General Naming conventions

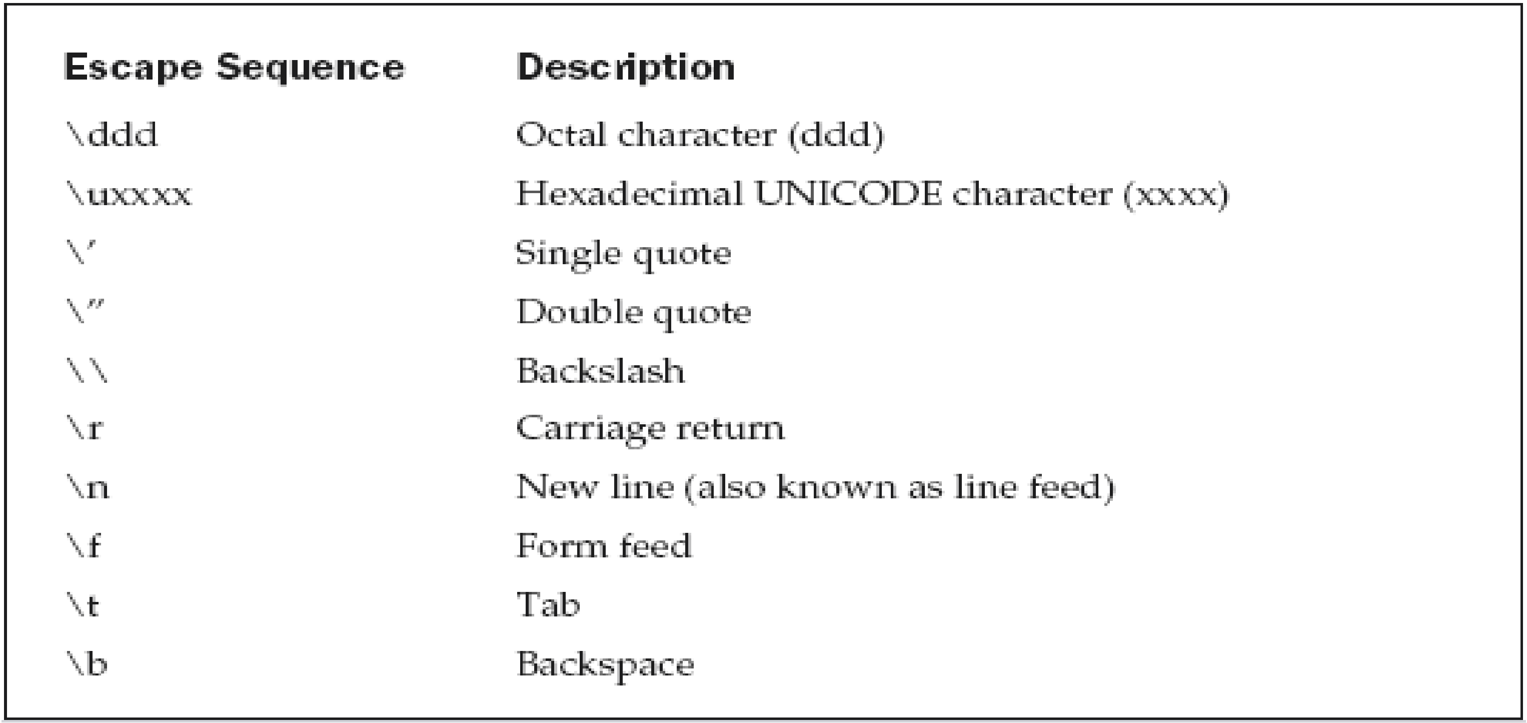
- Name of public methods and instance variables start with lower case letters. E.g. average, sum
- More than one words. E.g. FirstProg
- All private and local variable user _. E.g. avg_marks
- All classes and interface start with upper case. E.g. Student
- Constant value variables all upper case. E.g. PI

Literals

- Constant values are stored in variables.
- 5 types:
  - Integer literal
    - Decimal : int i=17;
    - Octal : int i =021; // (17 = 021)
    - Hexadecimal : int i= 0x11; // (17 = 0x11)
  - Floating point literal
    - float f = 10.7f;
    - double d =10.7;   //by default float is assigned to double
  - String literal
    - string str = “Hello”;
    - “two\nlines”
    - “\”This is in quotes\””
  - Boolean literal
    - `boolean status = true;`

Con’t

- Character literal
  - Java Character set:
  - Characters are defined by unicode character set.
  - Unicode is a 16 bits character coding system.
  - ASCII code + additional character are included.
  - char a= ‘a’;



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

Operators (LARACIBS)

- Logical operator (&&,||,!)
- Arithmetic operator (+, -, *, /, %)
- Relational operator (>,<,>=, <=, !=,==)
- Assignment operator (=)
- Conditional operator (?,:)
- Increment/Decrement operator (++,--)
- Bitwise operator (&,|,^,~,<<,>>)
- Special operator (instance of, .)

Logical Operators (&&,||,!)

Assume boolean variables A holds true and variable B holds false then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Logical operator returns true/false

E.g. if(age>55 && salary<1000)

Arithmetic Operators(+, -, *, /, %)

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0

3 types:
Integer Arithmetic (E.g. 15/10 = 1)
Real Arithmetic (E.g. 15.0/10.0 = 1.0)
Mixmode Arithmetic (E.g. 15/10.0 = 1.5)

Relational Operators (>,<,>=, <=, !=,==)

Assume variable A holds 10 and variable B holds 20 then: returns true/false

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

```
int done;
if(!done) ... // Valid in C/C++
if(done) ... // but not in Java.
```

```
if(done == 0)) ... // This is Java-style.
if(done != 0) ...
```

Assignment Operators (=)

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A

Shorthand Assignment Operators

Operator	Description	Example
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Conditional operator (?,:)

- Conditional operator is also known as the ternary operator.
- variable x = exp1 ? exp2 :exp3

```
public class Test
{
        public static void main(String args[])
       {
                   int a , b; a = 10; b = (a == 1) ? 20: 30;
                  System.out.println( "Value of b is : " + b );
                  b = (a == 10) ? 20: 30;
                  System.out.println( "Value of b is : " + b );
       }
 }
```

Output:

```
    Value of b is : 30
    Value of b is : 20
```

Increment/Decrement Operators (++,--)

Pre(++a, --a) and post(a++,a--)

Operator	Description	Example
++	Increment - Increase the value of operand by 1	B++ gives 21
--	Decrement - Decrease the value of operand by 1	B-- gives 19

```
int a = 10;
int d = 25;
System.out.println("a++ = " + (a++) );
System.out.println("b-- = " + (a--) );
// Check the difference in d++ and ++d
System.out.println("d++ = " + (d++) );
System.out.println("++d = " + (++d) );
```

Output:

```
a++ = 10
 b-- = 11
 d++ = 25
++d = 27
```

Bitwise Operator (&,|,^,~,<<,>>,>>>)

- Manipulation of data values at bit level.
- Applied to int, short, long, byte, char.
- Not applied on float or double.
- Bitwise operator works on bits and perform bit by bit operation. Assume if

  a = 60;

  b = 13;
- Now in binary format they will be as follows:

```
    a = 0011 1100
    b = 0000 1101
    -----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a   = 1100 0011
```

```
if (denom != 0 && num / denom > 10)
if(c==1 & e++ < 100) d = 100;
```

Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Precedence of Java Operators

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Separator

- ( )
  - Encloses arguments in method definitions and calling
  - adjusts precedence in arithmetic expressions
  - surrounds cast types and delimits test expressions in flow control statements
- { }
  - defines blocks of code and automatically initializes arrays
- [ ]
  - declares array types and dereferences array values
- ;
  - terminates statements
- ,
  - separates successive identifiers in variable declarations
  - chains statements in the test, expression of a for loop
- .
  - Selects a field or method from an object; separates package names from sub-package and class names
- :
  - Used after loop labels

Type conversion and casting

Automatic type conversion takes place if 2 conditions are met:

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

- *Widening conversion*
  - **int** type is always large enough to hold all valid **byte** values, so no explicit casting required.
- Integer and floating-point types, are compatible with each other.
- The numeric types are not compatible with **char** or **boolean**.
- **char** and **boolean** are not compatible with each other.

Casting Incompatible Types

- What if you want to assign an **int** value to a **byte** variable?
  - a **byte** is smaller than an **int**.
  - conversion is called a *narrowing conversion,* since you are explicitly making the value narrower so that it will fit into the target type.
- Conversion between two incompatible types
  - (*target-type*) *value*

```
int a;
byte b;
b = (byte) a;
```

Con’t

```
class Conversion {
public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;
    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);
    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);
    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
}
}
```

Output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Automatic Type Promotion in Expressions

- Java automatically promotes each **byte** or **short** operand to **int** when evaluating an expression.

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

- Causes compile-time error

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

- Solution:

```
byte b = 50;
b = (byte)(b * 2);
which gives output as  100.
```

The Type Promotion Rules

- All **byte** and **short** 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**.

- E.g.

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

Arrays : One-Dimensional Arrays

- An array is data structure that stores a collection of the same type.
- Declaring : int month_days[]; // int[] month_days;
- Creating memor location : month_days= new int[12];
- int month_days[] = new int[12];

```
class AutoArray {
  public static void main(String args[]) {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30,
                                    31,30, 31 };
        System.out.println("April has " + month_days[3] + " days.");
  }
}
```

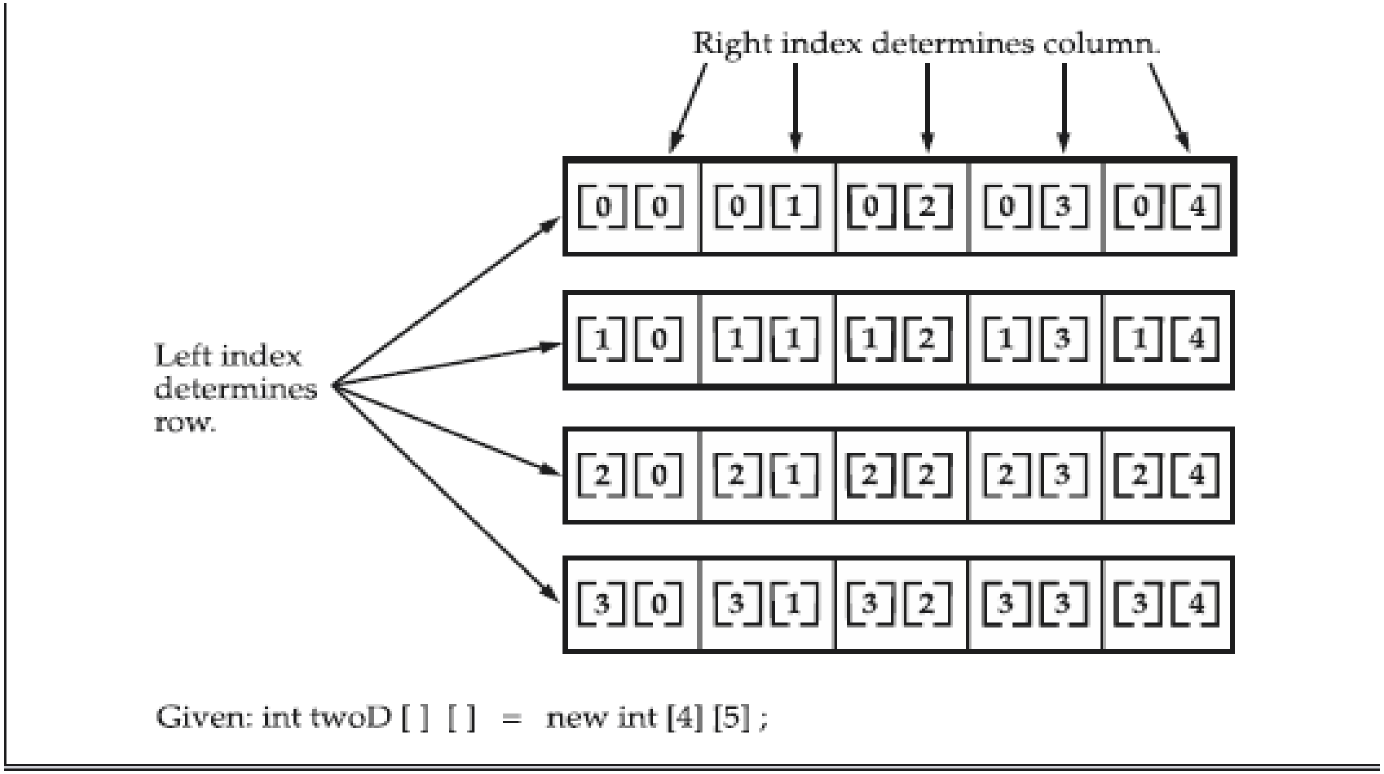
Con’t

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

```
int[] winning_numbers;
winning_numbers = numbers; // refer to same array
numbers[0] = 13;            // changes both
```

Multidimensional Arrays : Arrays of Arrays

- Two-Dimensional Arrays
- int twoD[][] = new int[4][5];



Right index determines column.

Left index determines row.

[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]

Given: int twoD [ ] [ ] = new int [4] [5] ;

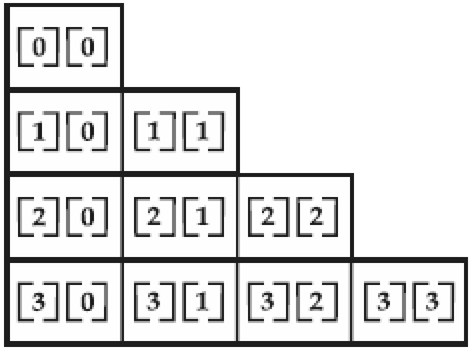
Con’t

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;
    for(i=0; i<4; i++)
        for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
    }
    for(i=0; i<4; i++) {
        for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
                System.out.println();
    }
   }
}
```

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Con’t

```
// 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();
      }
   }
}  // size can be computed at run time, but can't be changed - allocated on heap
```



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

Con’t

```
// Initialize a two-dimensional array.
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();
     }
   }
}
```

```
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
```

Three-Dimensional Arrays

```
// Demonstrate a three-dimensional array.
class threeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            for(k=0; k<5; k++)
                threeD[i][j][k] = i * j * k;
    for(i=0; i<3; i++) {
        for(j=0; j<4; j++) {
            for(k=0; k<5; k++)
                System.out.print(threeD[i][j][k] + " ");
        System.out.println();
        }
    System.out.println();
    }
  }
}
```

Con’t

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
```

Control Structures

1. If……..else statements : Decision Making Statement

if	Nested ifs	The if-else-if Ladder

if (condition)
statement1;
else statement2;
	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;
		if(*condition)*
		statement;
		else if*(condition)*
		statement;
		else if(*condition)*
		statement;
		...
		else
		statement;

For loop

2. For loop : entry controlled loop

Syntax:

```
for(initialization; condition; increment)
{
statements;
}
```

- for(int n=10; n>0; n--)
- for(a=1**, b=4; a<b; a++, b--)**
- for(int i=1; !done; i++)
- for(; !done;)   // Parts of the for loop can be empty.
- for(; ;)   //infinite loop

Nested for Loops

- Nested Loops

```
for(initialization; condition; increment)
{
   for(initialization; condition; increment)
   {
       statements;
   }
}
```

While loop

3. While loop : entry controlled loop

Syntax:

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

E.g.

```
// The target of a loop can be empty.
class NoBody {
   public static void main(String args[]) {
           int i, j;
           i = 100;
           j = 200;
           // find midpoint between i and j
           while(++i < --j) ; // no body in this loop
           System.out.println("Midpoint is " + i);
   }
}
```

Do….While loop

4. Do….While loop : exit controlled loop

Syntax:

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

- The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Switch statement : Decision Making Statement

5. Switch statement : Alternative to if-else-if ladder statement

Syntax:

```
switch(variable)
{
   case(value1):
          statements;
          break;
   case(value2):
          statements;
          break;
   default:
          statements;
          break;
}
```

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

Con’t

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
        for(int i=0; i<4; i++)
        switch(i) {
                 case 0:
                 case 1:
                    System.out.println("i is less than 1");
                    break;
                 case 2:
                 case 3:
                    System.out.println("i is less than 3");
                    break;
                 default:
                     System.out.println("i is 10 or more");
                 }
        }
}
```

Nested switch Statements

```
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: //
```

Jump Statements

break

Syntax: break *label;*

- Uses of break statment

1)It terminates a statement sequence in a switch statement.

2) It can be used to exit a loop.

3) It can be used as a “civilized” form of goto.

- E.g. for 2: break out of the innermost loop.

```
for(int i=0; i<3; 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();
}
```

```
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
```

Con’t

```
// Using break to exit from nested loops
// It can be used as a “civilized” form of goto.
class BreakLoop4 {
   public static void main(String args[]) {
          outer: for(int i=0; i<3; i++) {
                    System.out.print("Pass " + i + ": ");
                    for(int j=0; j<100; j++) {
                              if(j == 10) break outer;     // exit both loops
                              System.out.print(j + " ");
                    }
          System.out.println("This will not print");
          }
   System.out.println("Loops complete.");
   }
}
```

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

Con’t

```
// This program contains an error.
// cannot break to any label which is not defined for an enclosing block
class BreakErr {
  public static void main(String args[]) {
    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }
    for(int j=0; j<100; j++) {
    if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

continue

- Ignore the current iteration and start next iteration
- Will not come out of loop

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

Output:

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

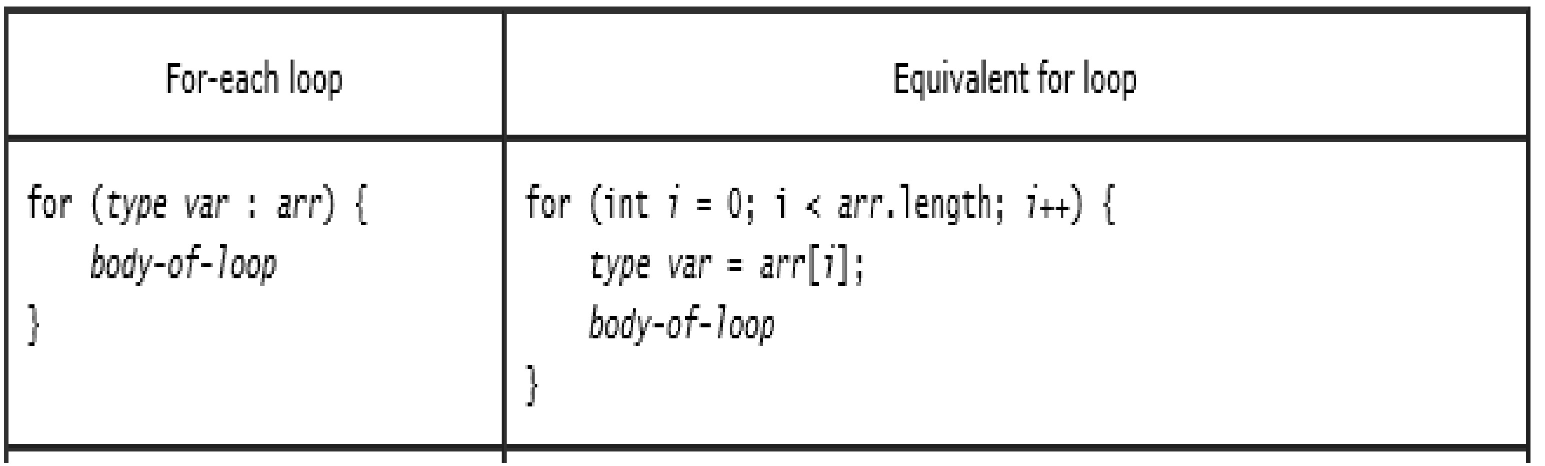
return

- The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method.
- return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ).

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

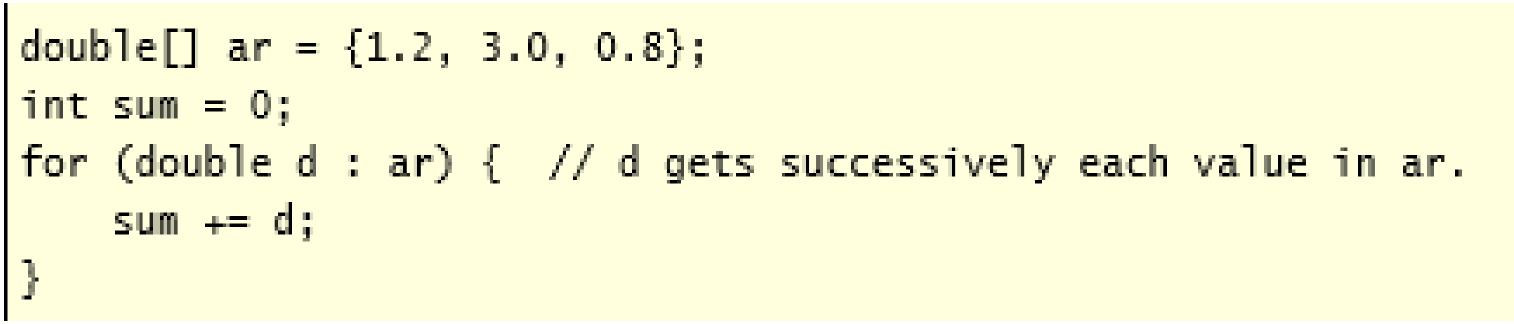
Output: Before the return.

For-each loop

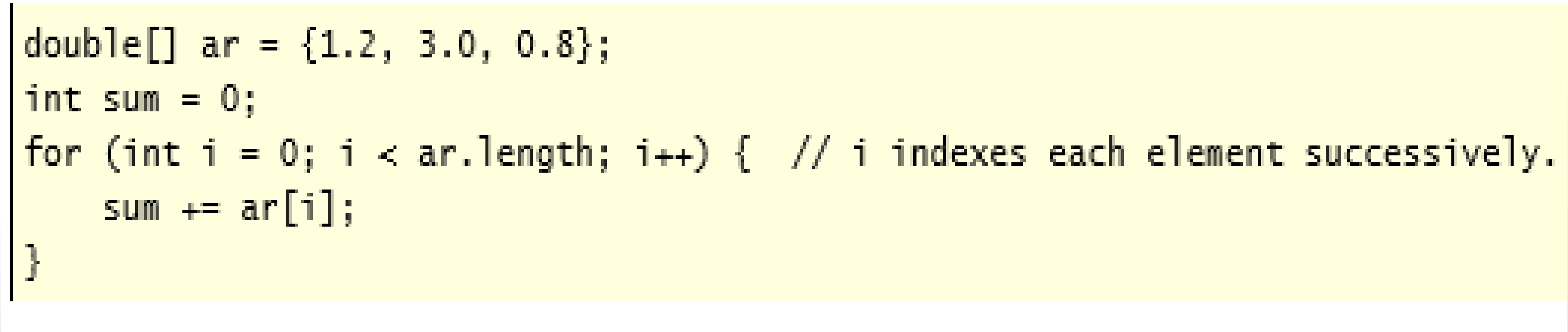


For-each loop	Equivalent for loop

for (*type var* : *arr*) {
    body-of-loop
}
	for (int *i* = 0; i < *arr*.length; *i*++) {
	    type var = *arr*[*i*];
	    body-of-loop
	}



```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (double d : ar) {  // d gets successively each value in ar.
    sum += d;
}
```



```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (int i = 0; i < ar.length; i++) {  // i indexes each element successively.
    sum += ar[i];
}
```

Con’t

- 2 Dimensional

```
class p4{
  public static void main(String args[]){

    int x[][]= {
                    {1,2},
                    {3,4},
                    {5,6},
             };
    for(int z[] : x)
    {
            for(int y : z)
                    System.out.print(y);
            System.out.println(" ");
    }
  }
}
```

Output:

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