Primitive Data Types	Variables	Literals	Operators	Control Structures

Primitive Data Types

- Types that are directly supported by the underlying hardware are known as Primitive Data Types. Most programming languages have a notion of *primitive data types*
- The set of primitive types in Java is defined in terms of the Java Virtual Machine, rather than in terms of a particular physical CPU; therefore, the Java primitives are identical on all platforms.
- The C++ primitive types are mapped directly to hardware-supported data types on the underlying physical CPU, and thus are not necessarily identical across all platforms. Moreover, the primitive types of C++ are largely derived from those of C.
- The character type in C++ was designed to allow ASCII character set. Whereas the Java character type was designed to allow for supporting a wide range of world languages.
- C++ allows all integer types to be explicitly declared as signed or unsigned.
- C++ regards the type char as an integer type and whether it is signed or unsigned by default is platform-dependent.
- In some contexts the C++ type char can be used as a one-byte integer, equivalent to the Java type byte.

The following table summarizes the primitive types in C++ and their relationship to the Java types. Note that the type structure of the two languages is similar.

C++ Type name	Typical Meaning	Closest Java type(s)
(char in some contexts)	8 bit integer	byte
short int (short)	16 bit integer	short
long int (long)	32 bit integer	int
(none)	64 bit integer	long
int	16 or 32 bit integer (platform specific)	short or int (depending on platform)
char	8 bit character	(byte in some contexts)
wchar_t	16 bit character	char
bool	false or true	boolean
float	32 bit real	float
double	64 bit real	double
long double	96 or 128 bit real	(none)

All 3 are signed by default

Where a name is given in parentheses after a C++ type name, it is a commonly used abbreviation for the type name.

8 primitive data types in Java

Finally, the type bool is a relatively recent addition to C++. Historically, the C++ comparison operators (like ==, <, etc.) returned an integer value (0 for false, 1 for true); the Boolean operators (&&, ||, and !) operated on integer values, and statements like if () and while() expected an integer expression, with 0 being interpreted as false and any nonzero value as true. It is still legal to use integers where a boolean value is expected.

(The bool values false and true are functionally equivalent to the integers 0 and 1.)

Variables

A variable provides us with named storage that our programs can manipulate. Each variable in Java or C++ has a specific type. All variables must be declared before they can be used. Following is the basic form of a variable declaration

data type variable [= value][, variable [= value] ...] ;

Variable can be of following types :

- **Local** variables − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed. The scope of local variables is limited to the block enclosed in braces ({}) where they are declared.

  There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

- **Instance** variables − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class** variables − Class variables are variables declared within a class, outside any method, with the static keyword.
- **Global** Variable (only available in C++) - A global variable is declared in the main body of the source code, outside all functions. Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

Valid identifier (variable) naming rules:

- Name contains only letters, digits , underscore characters (_) or a dollar sign ($) [only permitted for Java]. Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid.
- All variable names must begin with a letter of the alphabet, _ or $ (only for Java). The convention is to always use a letter of the alphabet. The dollar sign and the underscore are discouraged.
- In no case they can begin with a digit.
- Upper case and lower case letters are distinct.
- They cannot match any keyword operator.
- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word like gearRatio and currentGear.
- Examples of legal identifiers in : age, $salary (only allowed in Java) , _value, __1_value.

Declaring Variables

C++	**int a;** **float mynumber;** // Variable of two different types **int a, b, c;** // more than one variable of same type **unsigned short int NumberOfSisters;** **signed int MyAccountBalance;** **short Year;** **short int Year;** // two are equivalent **int a=5;** // initial value = 5 **int b(2);** // initial value = 2 **int result;** // initial value undetermined
Java	**int a, b, c;** // Declares three ints, a, b, and c. **int a = 10, b = 10;** // Example of initialization **byte B = 22;** // initializes a byte type variable B. **double pi = 3.14159;** // declares and assigns a value of PI. **char a = 'a';** // the char variable a iis initialized with value 'a'

Known as constructor initialization, is done by enclosing the initial value between parentheses (()):

Constant variables

Both Java and C++ allow a variable declaration to specify that the variable is really a constant - i.e. its value cannot be changed after it is declared. C++ uses the word "const" for this, while Java uses "final". The following, therefore, are equivalent:

- C++ : const int bjorksFavoriteNumber = 42;
- Java : final int bjorksFavoriteNumber = 42;

Const and final are NOT equivalent when applied to methods:

The following are not at all equivalent:

- C++ : const Foo * bar()
- Java : final Foo bar()

Means bar() returns a pointer to a Foo, which cannot be used to modify the Foo

bar() cannot be overridden in any subclass of the current class.

String Variables

Both C++ and Java provide string class (string in C++, java.lang.String in Java). This is not a primitive type, but it behaves in a similar way as primitive types do in its most basic usage.

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution

<table>
<tr><th>C++</th><th>Java</th></tr>
<tr>
<td>To declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the std namespace
<pre>
// my first string
#include <iostream>
#include <string>
using namespace std;
int main ()
{
    string mystring = "A string";
    cout << mystring;
    return 0;
}
</pre>
Both initialization are valid with strings:
string mystring = "A string";
string mystring (A string");
</td>
<td>String greeting = "Hello world!";

Compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor.
The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

<pre>
public class StringDemo {

  public static void main(String args[]) {
    char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
    String helloString = new String(helloArray);
    System.out.println(helloString);
  }
}
</pre>
</td>
</tr>
</table>

C-Style Character Strings in C++

C++ also uses "Cstyle" strings, inherited from C. The C language represents a string by an array of characters, terminated with the null character, '\0'.

Because of the equivalence of arrays and pointers, this leads to strings being represented by the type char *. (Such strings are often declared as const char * when there is no need to manipulate the contents of the string.) The C library includes a variety of functions for working with such null-terminated array of char strings. These are declared in the header file <cstring>. Of particular significance is that when the compiler encounters as quoted string, it represents it by a C-style char * string, not a C++ string class object.

Literals

Literal means any number, text, or other information that represents a value. So a literal is some data that's presented directly in the code, rather than indirectly through a variable or function call. The data constituting a literal cannot be modified by a program, but it may be copied into a variable for further use. Following table represents different types of literals used in Java and C++.

	Java	C++
Integer Literal	byte, int, long, and short can be expressed in decimal, hexadecimal or octal number systems as well. **75 // decimal ,** **0113 // octal , 0 prefix indicatex octal** **0x4b // hexadecimal, 0x prefix indicates hexadecimal** All of these represent the same number: 75	Similar rules is also applicable to C++. However, we can force them to either be unsigned by appending the u character to it, or long by appending l: **75 // int ,** **75u // unsigned int,** **75l // long,** **75ul // unsigned long** In both cases, the suffix can be specified using either upper or lowercase letters.
Floating Point Literal	a) Two types for floating-point numbers exist: float and double. b) The default type for floating point literals is double c) As the name implies, a double has 2x the precision of float. In general a double has 15 decimal digits of precision, while float has 7. **d)** They express numbers with decimals and/or exponents. **Example** **3.14159 // 3.14159 ,** **6.02e23 // 6.02 x 10^23 ,** **1.6e-19 // 1.6 x 10^-19,** **3.0 // 3.0** **.01f // Floating-point literal** e) the suffix F (or f) is appended to designate the data type of a floating-point literal as float.	Same rules applie for C++. In addition you can explicitly express a long double numerical literal, by using l suffixes respectively: **3.14159L // long double** Any of the letters that can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters.
Character Literals	**a)** Single **printable** character literal is expressed in a pair of single quote such as **'a', '#', '3'.** **b)** Special characters (nonprintable character) can be expressed using a single backslash character at the beginning like newline (\n) or tab (\t). For example**:** **'\n' , '\t'** c) **Unicode** escape sequence (\u) is used to represent printable and non-printable characters. Example :	In C++ we can also use the same rule except **no Unicode character** is representable using char literal.

	'\u0041' Capital letter A '\u0030' Digit 0 '\u0022' Double quote '\u0020' Space	
String Literal	In Java a string is not a basic data type, rather it is an object. These strings are not stored in arrays as in C language. String Literal can be represented using double quote. Example : String myString = "How are you?"; "" // the empty string "\"" // a string containing " // representation when formed from two string literals "This is a " + "two-line string" Wide characters are used mainly to represent non-English or exotic character sets.	Same rules are also applicable for C++. In addition following representations are also applicable : a) For more than a single line literal value , put a backslash sign (\) at the end of each unfinished line. **"string expressed in \ two lines"** b) Concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character: **"this forms" "a single" "string" "of characters"** c) Finally, if we want the string literal to be explicitly made of wide characters (**wchar_t**), instead of narrow characters (char), we can precede the constant with the L prefix: **L"This is a wide character string"**

Operators in C++ and Java

Operators are special symbols that perform specific operations and then return a result. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence	
Operators	**Precedence**
postfix	*expr*++ *expr*−−
unary	++*expr* −−*expr* +*expr* −*expr* ~ !
multiplicative	* / %
additive	+ −

shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

a) The Simple Assignment Operator =

This operator can also be used on objects to assign *object references*.

Example : int cadence = 0; int speed = 0; int gear = 1;

Some specialty of assignment operator in C++ :

- a = 2 + (b = 5); is equivalent to: b = 5; a = 2 + b; so final value of a is 7
- a = b = c = 5; It assigns 5 to the all the three variables: a, b and c.

b) The Arithmetic Operators

"%" means divides one operand by another and returns the remainder as its result.

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator (in C++ it is also know an modulo)

The + operator can also be used for concatenating (joining) two strings together

c) Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=) (C++ & Java)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression is equivalent to value += increase; value = value + increase;

a -= 5; a = a - 5; a /= b; a = a / b; price *= units + 1; price = price * (units + 1);

and the same for all other operators.

d) The Unary Operators (C++ & Java)

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
−	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
−−	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

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

e) Relational and equality operators (==, !=, >, <, >=, <=)

Suppose a=2, b=3 and c=6,

```
(a == 5)        // evaluates to false since a is not equal to 5.
(a*b >= c)      // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c)     // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a)    // evaluates to true.
```

f) Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT.
The && and || operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

&& Conditional-AND || Conditional-OR

```
Example :  if((value1 == 1) && (value2 == 2))     System.out.println("value1 is 1 AND value2 is 2");
           if((value1 == 1) || (value2 == 1))     System.out.println("value1 is 1 OR value2 is 1");

    !(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
```

g) Conditional operator (?) (Java & C++)

This operator is also known as the *ternary operator* (in Java) because it uses three operands. In the following example, this operator should be read as: "If someCondition is true, assign the value of

value1 to result. Otherwise, assign the value of value2 to result." The following program, ConditionalDemo2, tests the ?: operator:

If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3          // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3        // returns 4, since 7 is equal to 5+2.
5>3 ? a : b           // returns the value of a, since 5 is greater than 3.
a>b ? a : b           // returns whichever is greater, a or b.
```

h) Comma operator (,) (only for C++)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code: **a = (b=3, b+2);**

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

i) Bitwise Operators (&, |, ^, ~, <<, >>) (Java & C++)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

j) Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int  i;    float f = 3.14;    i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses: **i = int (f);**

Both ways of type casting are valid in C++.

k) sizeof() (only for C++)

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:
a = sizeof (char); This will assign the value 1 to a because char is a one-byte long type.
The value returned by sizeof is a constant, so it is always determined before program execution.

l) instanceof (only for Java)

The instanceof operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface. The following program, InstanceofDemo, defines a parent class (named Parent), a simple interface (named MyInterface), and a child class (named Child) that inherits from the parent and implements the interface.

Here Child class inherited from Parent class and implemented MyInterface

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

      Parent obj1 = new Parent();      Parent obj2 = new Child();
      System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
      System.out.println("obj1 instanceof Child: "  + (obj1 instanceof Child));
      System.out.println("obj1 instanceof MyInterface: "
        + (obj1 instanceof MyInterface));
      System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
      System.out.println("obj2 instanceof Child: "  + (obj2 instanceof Child));
      System.out.println("obj2 instanceof MyInterface: "
        + (obj2 instanceof MyInterface));
   }
}
class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}
```

Output:

```
obj1 instanceof Parent: true          obj1 instanceof Child: false
obj1 instanceof MyInterface: false    obj2 instanceof Parent: true
obj2 instanceof Child: true           obj2 instanceof MyInterface: true
```

When using the instanceof operator, keep in mind that **null is not an instance of anything**.

Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, every language provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

- A block (compound statement) is a group of statements enclosed in braces: { }: { statement1; statement2; statement3; }
- Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block).

1. The if-then and if-then-else Statements

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true. The structure is similar to in Java and C++. Following is an example of C++.

if (condition) statement
if (condition) statement1 else statement2
	if (x == 100) cout << "x is 100";
	if (x == 100)
	{ cout << "x is "; cout << x; }

2. The selective structure: switch.

The C++ syntax shown below similar to Java syntax.

- switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.
- If expression was not equal to constant1 it will be checked against constant2 and so on.
- Finally, if the value of expression did not match any of the previously specified constants, the program will execute the statements included after the default: label, if it exists (since it is optional).

```
switch (expression)
{
    case constant1:
       group of statements 1;
       break;
    case constant2:
       group of statements 2;
       break;
    .
```

The switch statement uses labels instead of blocks (like if-else). This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):).

```
    .
    default:
      default group of statements
}
```

The following code example in Java, SwitchDemo2, shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

```
class SwitchDemo2 {
   public static void main(String[] args) {
      int month = 2;       int year = 2000;       int numDays = 0;

      switch (month) {
         case 1: case 3: case 5:
         case 7: case 8: case 10:
         case 12:
            numDays = 31;       break;
         case 4: case 6:
         case 9: case 11:
            numDays = 30;      break;
         case 2:
            if (((year % 4 == 0) &&   !(year % 100 == 0))  || (year % 400 == 0))
               numDays = 29;
            else
               numDays = 28;
            break;
         default:
            System.out.println("Invalid month.");         break;
      }
      System.out.println("Number of Days = "   + numDays);
   }
}
```

This is the output from the code:
Number of Days = 29

Using Strings in switch Statements

In **Java SE 7 and later**, you can use a String object in the switch statement's expression. The following code example, StringSwitchDemo, displays the number of the month based on the value of the String named month:

```
public class StringSwitchDemo {

   public static int getMonthNumber(String month) {

      int monthNumber = 0;         if (month == null) {  return monthNumber;  }

      switch (month.toLowerCase()) {
```

```
            case "january":     monthNumber = 1;     break;
            case "february":    monthNumber = 2;     break;
            case "march":       monthNumber = 3;     break;
            case "april":       monthNumber = 4;     break;
            case "may":         monthNumber = 5;     break;
            case "june":        monthNumber = 6;     break;
            case "july":        monthNumber = 7;     break;
            case "august":      monthNumber = 8;     break;
            case "september":   monthNumber = 9;     break;
            case "october":     monthNumber = 10;    break;
            case "november":    monthNumber = 11;    break;
            case "december":    monthNumber = 12;    break;
            default:            monthNumber = 0;     break;
        }
        return monthNumber;
    }
    public static void main(String[] args) {

        String month = "August";
        int returnedMonthNumber =  StringSwitchDemo.getMonthNumber(month);
        if (returnedMonthNumber == 0)    {        System.out.println("Invalid month");      }
        else {
                System.out.println(returnedMonthNumber);
        }
    }
}
```

The String in the switch expression is compared with the expressions associated with each case label as if the String.equals method were being used.

The output from this code is 8.

3. The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

for (*initialization*; *termination*; *increment*) { *statement(s)* }

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.
- For **Java** : The three expressions of the for loop are optional; an **infinite loop** can be created as follows:

```
// infinite loop
for ( ; ; ) {    // your code goes here  }
```

- For **C++** : Initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written (for (;n<10;)).

Java	C++

The following program, uses the general form of the for statement to print the numbers 1 through 10 to standard output:
 	
class ForDemo {
public static void main(String[] args){
for(int i=1; i<11; i++){
System.out.println("Count is: " + i);
}
}
}
 	
The for statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the ***enhanced for*** statement, and can be used to make your loops more compact and easy to read. Example:
 	
The following program, uses the enhanced for loop through the array:
 	
class EnhancedForDemo {
public static void main(String[] args){
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
for (int item : numbers) {
System.out.println("Count is: " + item);
}
}
}
In this example, the variable item holds the current value from the numbers array.
	#include <iostream>
	using namespace std;
	int main ()
	{
	for (int n=10; n>0; n--) {
	cout << n << ", ";
	}
	cout << "FIRE!\n";
	return 0;
	}
	10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
	• Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:
	for (n=0, i=100 ; n!=i ; n++, i--)
	{
	// whatever here...
	}

4. The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
   statement(s)
}
```

Similar syntax in C++ and java

The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following Java program:

```
class WhileDemo {
   public static void main(String[] args){
      int count = 1;
      while (count < 11) {
         System.out.println("Count is: " + count);
         count++;
      }
   }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){    // your code goes here  }
```

The Java and C++ programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

Similar syntax in C++ and java

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once.

```
class DoWhileDemo {
   public static void main(String[] args){
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count < 11);
   }
}
```

5. Branching Statements

The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following Java program:

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

      int[] arrayOfInts = { 32, 87, 3, 589,12, 1076, 2000, 8, 622, 127 };
      int searchfor = 12;
```

We can run a "for" loop for array in this way also

```
    //int[] numbers = {1,2,3,4,5,6,7,8,9,10};
     //for (int item : numbers) { }
      int i;
      boolean foundIt = false;

      for (i = 0; i < arrayOfInts.length; i++) {
         if (arrayOfInts[i] == searchfor) {
            foundIt = true;
            break;
         }
      }
      if (foundIt) {
         System.out.println("Found " + searchfor + " at index " + i);
      } else {
         System.out.println(searchfor + " not in the array");
      }
   }
}
```

This program searches for the number 12 in an array. The break terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop.
This program's output is:
Found 12 at index 4

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. Labeled break can also be used for coming out of outer loop.

The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, ContinueDemo , steps through a String, counting the occurences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

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

      String searchMe = "peter piper picked a " + "peck of pickled peppers";
      int max = searchMe.length();          int numPs = 0;

      for (int i = 0; i < max; i++) {
         // interested only in p's
         if (searchMe.charAt(i) != 'p')
            continue;
         // process p's
         numPs++;
      }
      System.out.println("Found " + numPs + " p's in the string.");
   }
}
```

Here is the output of this program:
Found 9 p's in the string.

The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. return ++count;

When a method is declared void, use the form of return that doesn't return a value. return;

In case of C++ it is optional for a void function

The exit function (only for C++)

It defined in the cstdlib library. The purpose of exit is to terminate the current program with a specific exit code. Its prototype is: void exit (int exitcode);

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.