ACM

LOOPS, ARRAYS, METHODS

Association for Computing Machinery

Discussion

Loops

Loops statements allow us to execute a statement multiple times.

Like conditional statements, they are controlled by Boolean expressions.

Java has three kinds of loops statements:

- the while loop
- the do loop
- the for loop

While Loop

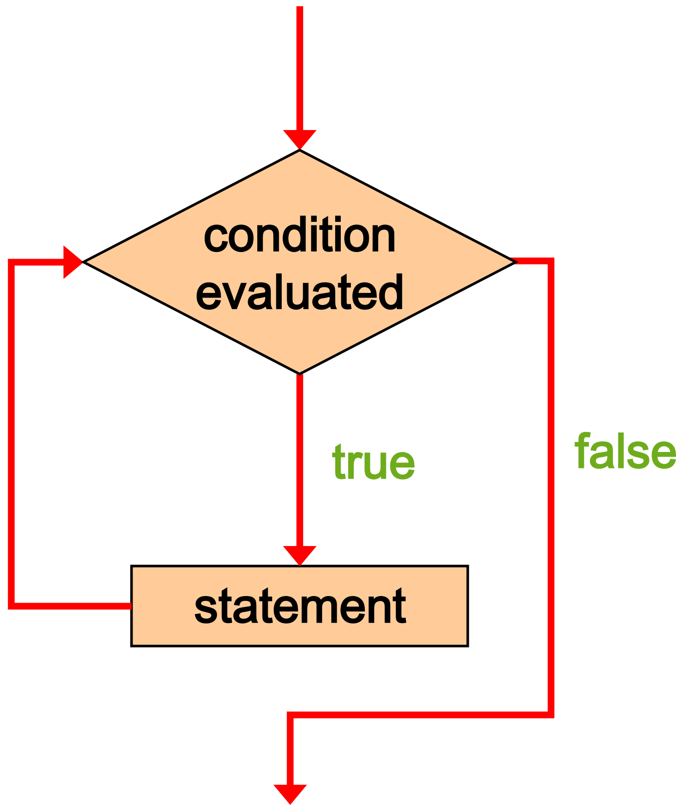
```
while ( condition ){
    statement;
}
```

If the `condition` is true, the `statement` is executed.

Then the condition is evaluated again, and if it is still true, the statement is executed again.

The statement is executed repeatedly until the condition becomes false.

While Loop



An example of a while statement:

```
int count = 1;
while (count <= 5){
    System.out.println (count);
    count++;
}
```

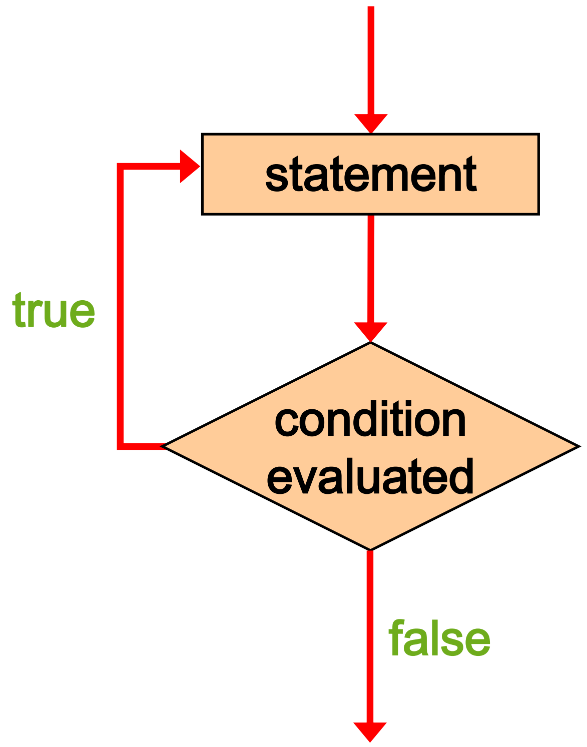
Do While loop

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

The `statement` is executed once initially, and then the `condition` is evaluated.

The statement is executed repeatedly until the condition becomes false.

Do While loop

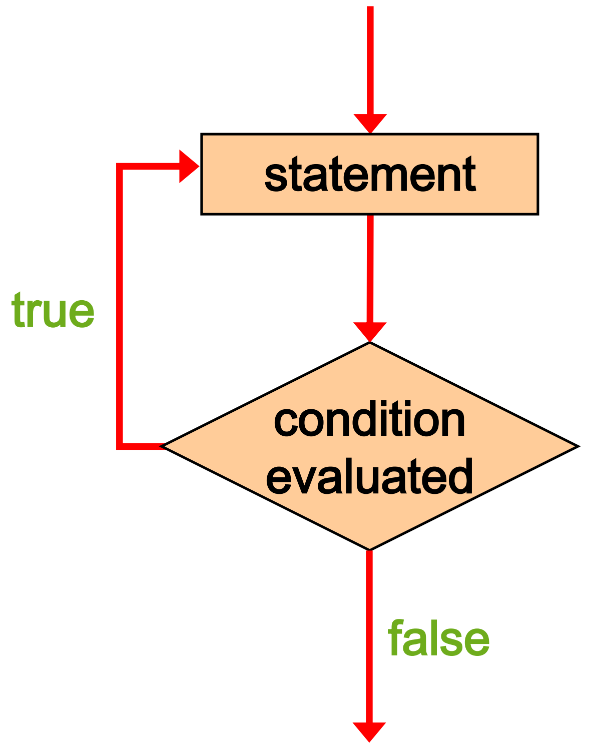


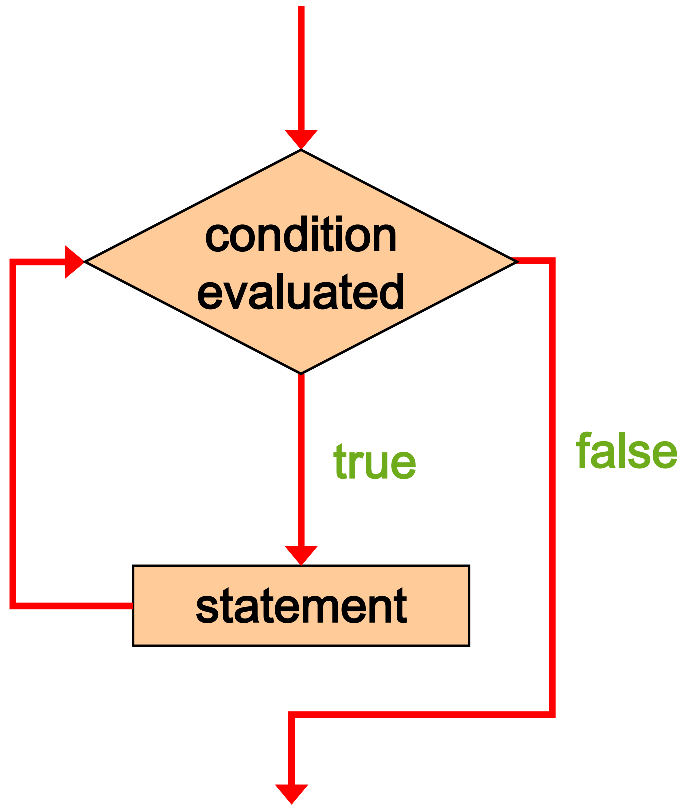
Do While loop

```
int count = 0;
do{
   count++;
   System.out.println (count);
} while (count < 5);
```

The body of a `do while` loop executes at least once.

Do while Vs. While





For loop

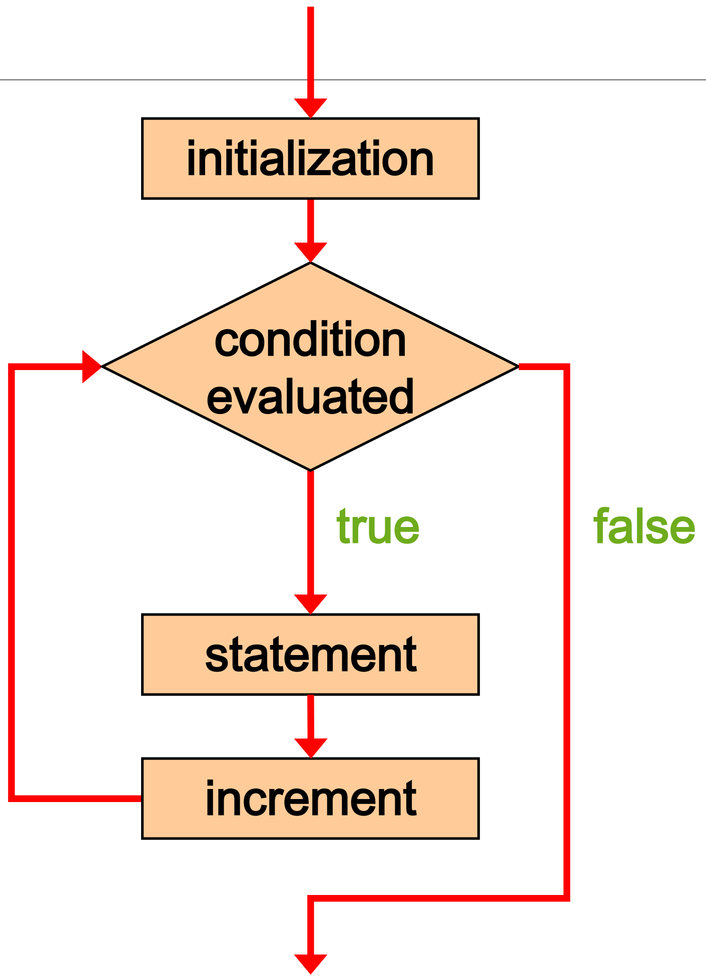
The *initialization* is executed once before the loop begins

The *statement* is executed until the *condition* becomes false

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

The *increment* portion is executed at the end of each iteration

For loop


initialization
condition evaluated
true
false
statement
increment

For loop

```
for (int count=1; count <= 5;
count++){
   System.out.println (count);
}
```

The initialization section can be used to declare a variable

Like a `while` loop, the condition of a `for` loop is tested prior to executing the loop body

Therefore, the body of a `for` loop will execute zero or more times

Discussion

Nested loops

```
count1 = 1;
while (count1 <= 10){
   count2 = 1;
   while (count2 <= 20)   {
      System.out.println ("Here");
      count2++;
   }
   count1++;
}
```

Break

Remember the `break` keyword that we used to stop a `switch` statement from executing more than one statement?

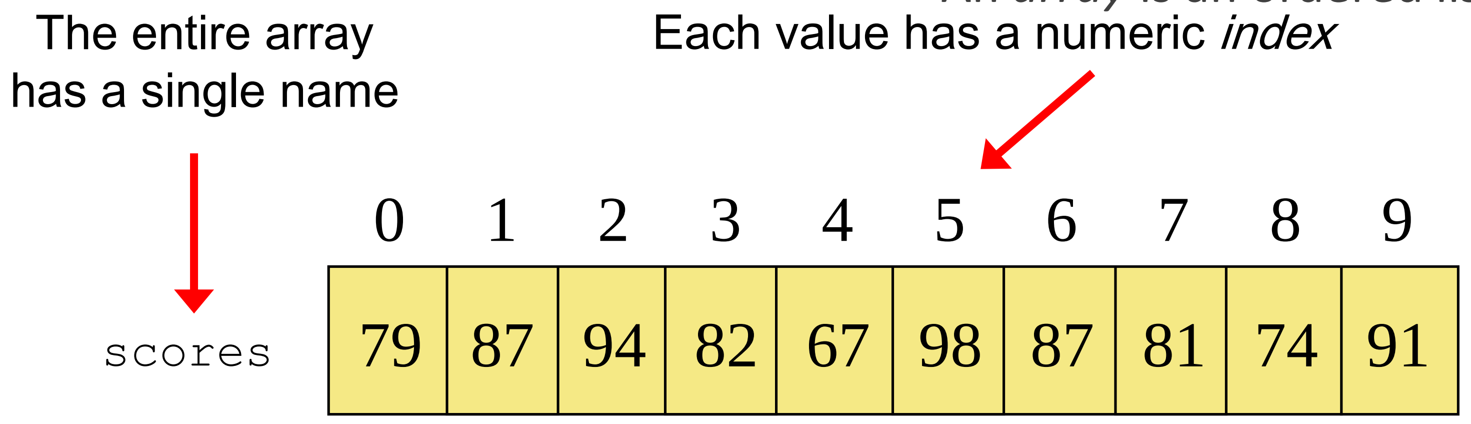
`break` can also be used to exit an infinite loop.

But it is almost always best to use a well-written while loop.

Discussion

Arrays

An *array* is an ordered list of values



An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9

Arrays

A particular value in an array is referenced using the array name followed by the index in brackets

For example, the expression

```
scores[2]
```

refers to the value `94` (the 3rd value in the array)

That expression represents a place to store a single integer and can be used wherever an integer variable can be used

Arrays

For example, an array element can be assigned a value, printed, or used in a calculation :

```
     scores[2] = 89;

     scores[first] = scores[first] + 2;

     mean = (scores[0] + scores[1])/2;

     System.out.println ("Top = " +
scores[5]);
```

Arrays

The values held in an array are called *array elements*

An array stores multiple values of the same type – the *element type*

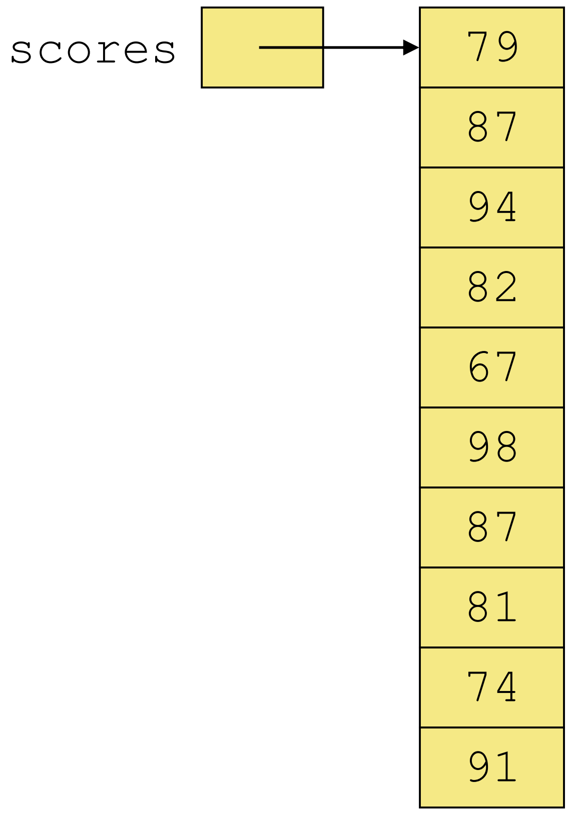
The element type can be a primitive type or an object reference

Therefore, we can create an array of integers, an array of characters, an array of `String` objects, an array of `Coin` objects, etc.

In Java, the array itself is an object that must be instantiated

Arrays

Another way to depict the `scores` array:



Declaring Arrays

The `scores` array could be declared as follows:

```
int[] scores = new int[10];
```

Note that the array type does not specify its size, but each object of that type has a specific size

The reference variable `scores` is set to a new array object that can hold 10 integers

An array is an object, therefore all the values are initialized to default ones (here 0)

Declaring Arrays

Some other examples of array declarations:

```
     float[] prices = new float[500];

boolean[] flags;
     flags = new boolean[20];

char[] codes = new char[1750];
```

```
      int LIMIT = 15, MULTIPLE = 10;

int[] list = new int[LIMIT];
```

```
// Initialize the array values
for (int i= 0; i< LIMIT; i++){
    list[i] = i* MULTIPLE;}

list[5] = 999;  // change one array value

// Print the array values
for (int value : list)
    System.out.print (value + "  ");
```

Alternate Array Syntax

The brackets of the array type can be associated with the element type or with the name of the array

Therefore the following two declarations are equivalent:

```
float[] prices;

float prices[];
```

The first format generally is more readable and should be used

Initializer Lists

An *initializer list* can be used to instantiate and fill an array in one step

The values are delimited by braces and separated by commas

Examples:

```
int[] units = {147, 323, 89, 933, 540,
               269, 97, 114, 298, 476};

char[] letterGrades = {'A', 'B', 'C', 'D', ’F'};
```

Initializer Lists

Note that when an initializer list is used:

- the `new` operator is not used
- no size value is specified

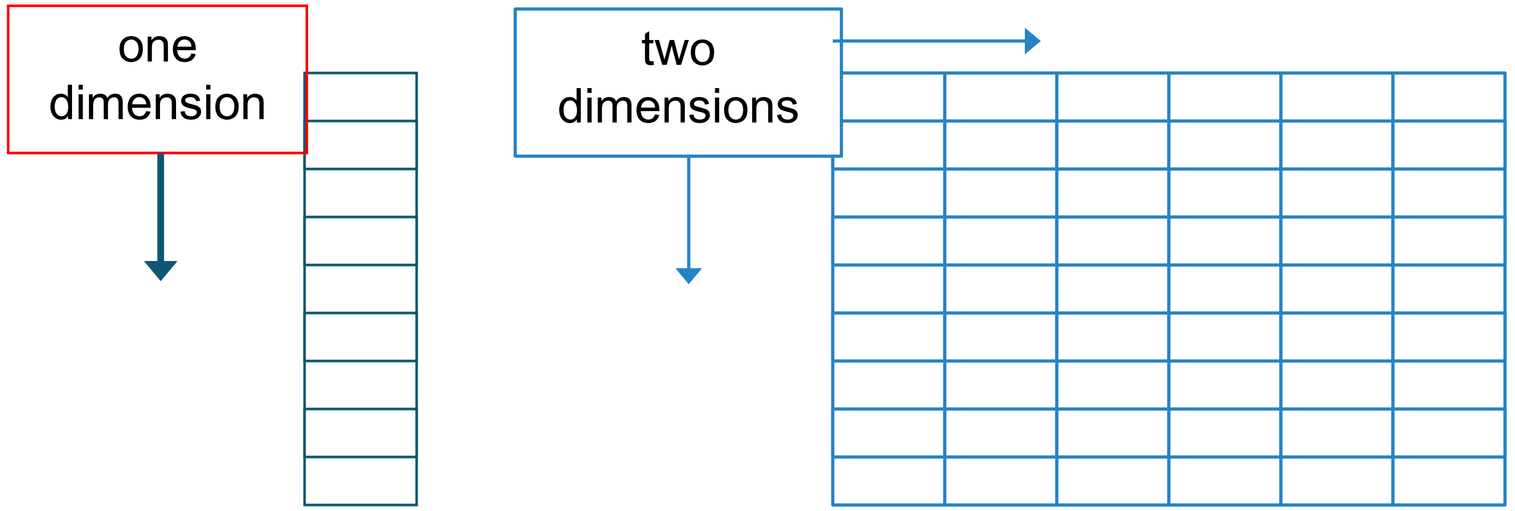
The size of the array is determined by the number of items in the initializer list

An initializer list can be used only in the array declaration

Two-Dimensional Arrays

A *one-dimensional array* stores a list of elements

A *two-dimensional array* can be thought of as a table of elements, with rows and columns



Two-Dimensional Arrays

Expression	Type	Description
`table`	`int[][]`	2D array of integers, or array of integer arrays
`table[5]`	`int[]`	array of integers
`table[5][12]`	`int`	integer

```
 public static void main (String[] args) {
      int[][] table = new int[5][10];

      // Load the table with values
      for (int row=0; row < table.length; row++)
for (int col=0; col < table[row].length; col++)
table[row][col] = row * 10 + col;

      // Print the table
      for (int row=0; row < table.length; row++) {
for (int col=0; col < table[row].length; col++)
System.out.print (table[row][col] + "\t");
System.out.println();
}}
```

Output

```
0     1    2    3    4    5    6    7    8    9
10   11   12   13   14   15   16   17   18   19
20   21   22   23   24   25   26   27   28   29
30   31   32   33   34   35   36   37   38   39
40   41   42   43   44   45   46   47   48   49
```

Discussion

Methods, arguments and return values

Java methods are like C/C++ functions. General case:

returnType methodName (*arg1*, *arg2*, … *argN*) {

  methodBody

}

A method definition consists of a method header and a method body. Here are all the parts of a method:

Modifiers: The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

Return Type: A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void**.

Methods, arguments and return values

•**Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.

•**Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

•**Method Body:** The method body contains a collection of statements that define what the method does.

Return

The return keyword exits a method optionally with a value

```
int storage(String s) {return s.length() * 2;}
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

Example

```
 public static int max(int num1, int num2) {
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Method

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

```
int largest = max(num1,num2);
```

Calling a Method

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

```
System.out.println("Welcome to Java”);
```

Passing Parameters by Values:

When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.
For example, the following method prints a message n times:

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

Here, you can use nPrintln("Hello", 3) to print "Hello" three times. The nPrintln("Hello", 3) statement passes the actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times.

Discussion