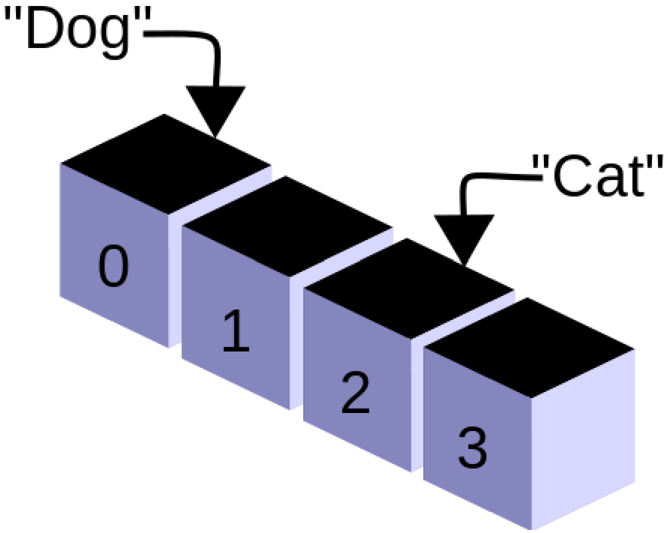
Module 1-4

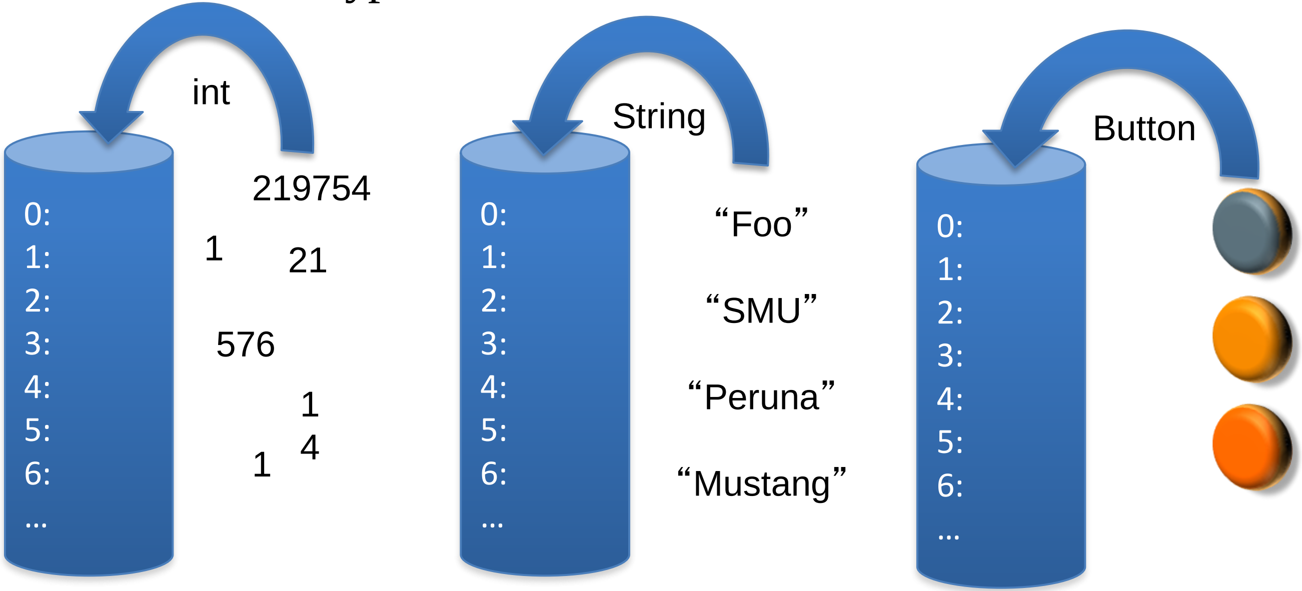
Loops and Arrays

Arrays



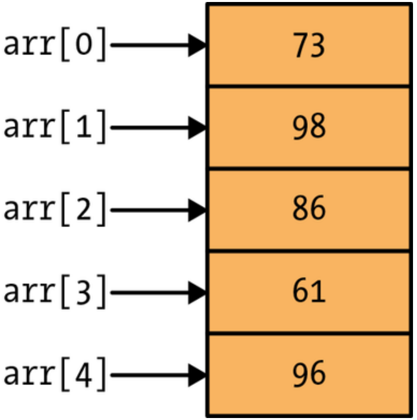
Arrays

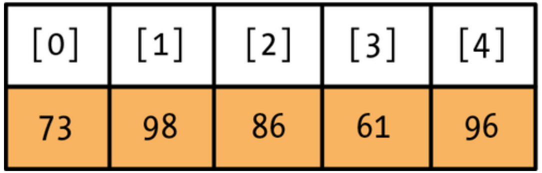
- An array is a collection of elements of the same type.
- Elements can be either primitive types or non-primitive / reference types.



Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- Every location in an array has an **index** number.
- Index identifiers begin with zero (0) and are always positive.
- An element of an array can be retrieved with it’s index number



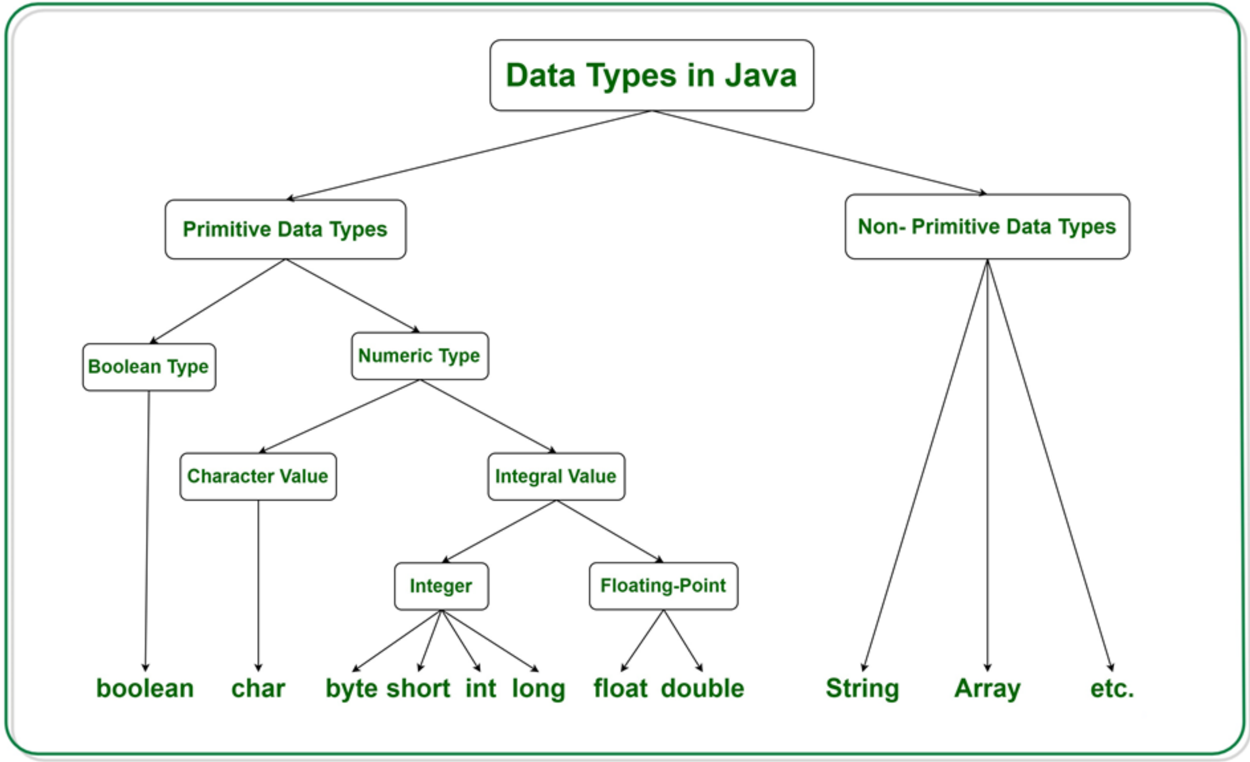


[0]	[1]	[2]	[3]	[4]
73	98	86	61	96

Arrays

- The Array itself is a reference type.
- When you create an array,
  - Assign it to a variable.
  - Declare the type of data it contains.
    - Use *primitive* or *non-primitive* / *reference* type name followed by []
  - Give it a variable name to refer to it by.
  - Create it using the **`new`** keyword.
  - Set the size of the array.

Data Types in Java



Arrays

For example:

```
String[] myStringArray = new String[5];
```

*Variable **myStringArray** is declared as type Array of Strings.*

Why do we use arrays?

Suppose we have the following statements:

```
//Use individual variables to store car models:
String car0 = "Volvo“;
String car1 = “BMW”;
String car2 = “Ford”;
String car3 = “Lexus”;

//Use an array to store car models:
String[] cars = new String[4];
String cars[0] = "Volvo“;
String cars[1] = “BMW”;
String cars[2] = “Ford”;
String cars[3] = “Lexus”;

//Use an array to store car models - Array Initializer
String[] cars = {"Volvo", "BMW", "Ford", “Lexus"};
String cars[] = {"Volvo", "BMW", "Ford", “Lexus"};
```

Why do we use arrays?

Advantages over individual variables

- Array stores data elements of the same data type.
- Maintains multiple variable names using a single name. Arrays help to maintain large data under a single variable name. This avoid the confusion of using multiple variables.
- Arrays can be used for sorting and searching data elements.

(String[] cars = {"Volvo", "BMW", "Ford", “Lexus"};)

Why do we use arrays?

```
//String array example
public class StringArrayEx1
{
  public static void main(String[] args)
  {
    String[] cars = new String[4];
    cars[0] = "Volvo";
    cars[1] = "BMW";
    cars[2] = "Ford";
    cars[3] = "Lexus";
    //Use ‘for’ loop to print cars
    for ( String car  : cars)
    {
      System.out.println(car);
     }
   }
}
```

Why do we use arrays?

Output:

```
Volvo
BMW
Ford
Lexus
```

Arrays: Enhanced *for* Statement

- Iterates through the elements of an array without using a counter, thus avoiding the possibility of “stepping outside” the array.

- Syntax:

```
for ( parameter : arrayName )
      statement
```

  - where *parameter* has a type and an identifier, and *arrayName* is the array through which to iterate.

  - Parameter type must be consistent with the type of the elements in the array.

Example:

```
String[] cars = {"Volvo", "BMW", "Ford", “Lexus"};
for(String car  : cars)
{
```

Why do we use arrays?

```
//Use individual variables to store car models.
public class StringArrayEx2
{
  public static void main(String[] args)
  {
    String car0 = "Volvo";
    String car1 = "BMW";
    String car2 = "Ford";
    String car3 = "Lexus";

    //Print the car one by one
    System.out.println(car0);
    System.out.println(car1);
    System.out.println(car2);
    System.out.println(car3);
  }
}
```

Why do we use arrays?

```
//Arrays.sort example
import java.util.Arrays;
public class StringArrayEx3
{
   public static void main(String[] args)
   {
      String[] cars = {"Volvo", "BMW", "Ford", "Lexus"};
      //Sort the cars in alphabetical order with the Arrays.sort method
       Arrays.sort(cars);
       //Print the sorted array
       for(String car  : cars)
       {
           System.out.println(car);
        }
    }
}
```

Why do we use arrays?

Output:

```
BMW
Ford
Lexus
Volvo
```

Why do we use arrays?

```
//Arrays.binarySearch example
import java.util.Arrays;
public class StringArrayEx4
{
   public static void main(String[] args)
   {
      String[] cars = {"Volvo", "BMW", "Ford", "Lexus"};
      //Sort the cars in alphabetical order with the Arrays.sort method
       Arrays.sort(cars);
      int indx = Arrays.binarySearch(cars, “Ford”);
       //Print indx
       System.out.println(“The car is found in index: “ + indx);
      }
}
```

Output:

```
The car is found in index: 1
```

Why do we use arrays?

```
public class StringArrayEx2
{
   public static void main(String[] args)
   {
      String car0 = "Volvo";
      String car1 = "BMW";
      String car2 = "Ford";
      String car3 = "Lexus";

      //Difficult to sort the above car models..
      System.out.println(car0);
      System.out.println(car1);
      System.out.println(car2);
      System.out.println(car3);
   }
}
```

Why do we use arrays?

```
//Read input from console and store them in an array.
import java.util.Scanner;
public class StringReadAndWrite
{
   public static void main(String[] args)
   {
      Scanner sc=new Scanner(System.in);
      //Ask the user for the size of the array.
      System.out.println(“Enter the length of String array");
      int n=sc.nextInt();
      String[] sarray=new String[n];
      //Ask the user to input the string one by one.
      for(int i=0;i<n;i++)
      {
          System.out.println(“Enter the "+(i+1)+" city :");
          sarray[i]=sc.next();
       }
       System.out.println(“Cities entered by user:");
       for(String s:sarray){
          System.out.println(s);
       }
   }
}
```

Why do we use arrays?

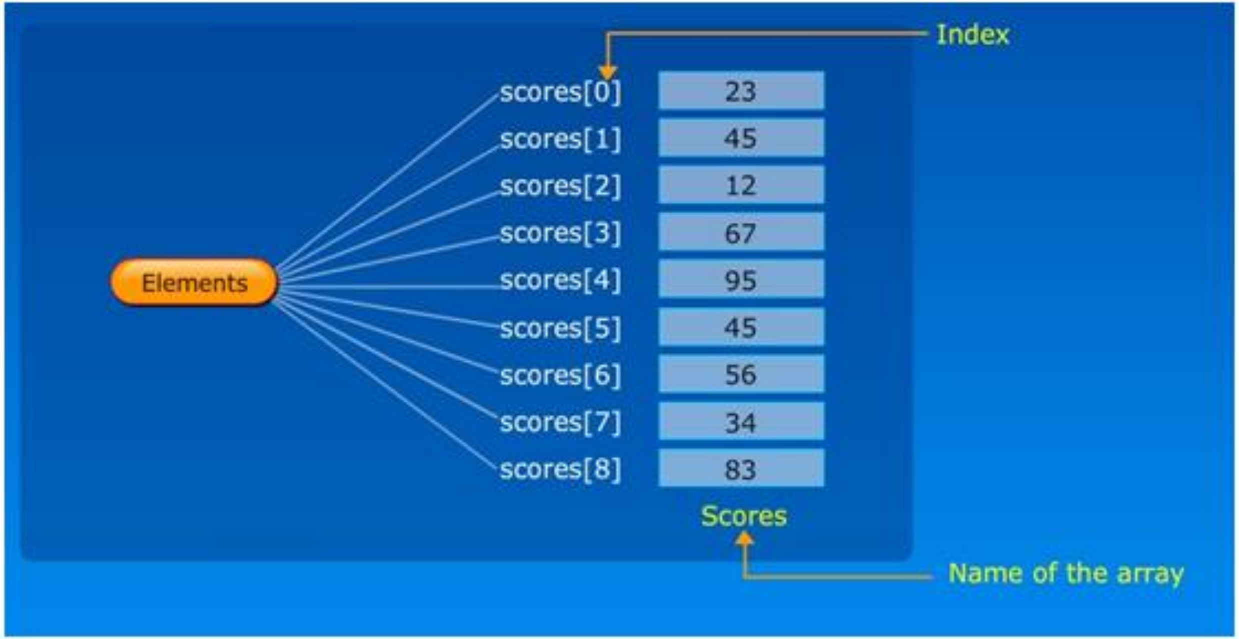
Sample run:

```
Enter the length of String array:
4
Enter the 1 city :
London
Enter the 2 city :
Dallas
Enter the 3 city :
Tokyo
Enter the 4 city :
Paris
Cities entered by user:
London
Dallas
Tokyo
Paris
```

Why do we use arrays?

```
import java.util.Scanner;
public class StringReadAndWrite
{
    public static void main(String[] args)
    {
       Scanner sc=new Scanner(System.in);
       System.out.println(“Enter the length of String array");
       int n=sc.nextInt();
       String[] sarray=new String[n];
       for(int i=0;i<n;i++)
       {
            System.out.println(“Enter the "+(i+1)+" String :");
            sarray[i]=sc.next();
        }
        System.out.println("Strings entered by user:");
        for(String s:sarray){
            System.out.println(s);
        }
         //Add code here to sort the array and print it.
    }
}
```

Arrays



scores[0]	23
scores[1]	45
scores[2]	12
scores[3]	67
scores[4]	95
scores[5]	45
scores[6]	56
scores[7]	34
scores[8]	83

Arrays

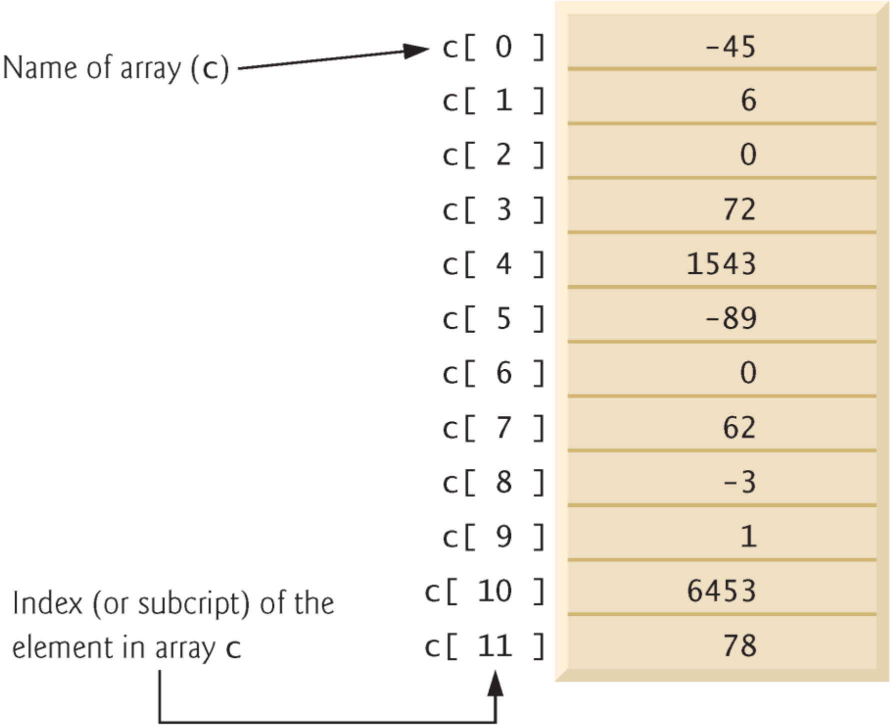
```
int[] myIntArray = new int[10];
```

myIntArray[0]	0
myIntArray[1]	0
myIntArray[2]	0
myIntArray[3]	0
myIntArray[4]	0
myIntArray[5]	0
myIntArray[6]	0
myIntArray[7]	0
myIntArray[8]	0
myIntArray[9]	0

```
myIntArray[2] = 12;
myIntArray[8] = 695;
```

myIntArray[0]	0
myIntArray[1]	0
myIntArray[2]	12
myIntArray[3]	0
myIntArray[4]	0
myIntArray[5]	0
myIntArray[6]	0
myIntArray[7]	0
myIntArray[8]	695
myIntArray[9]	0

Arrays



Name of array (c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

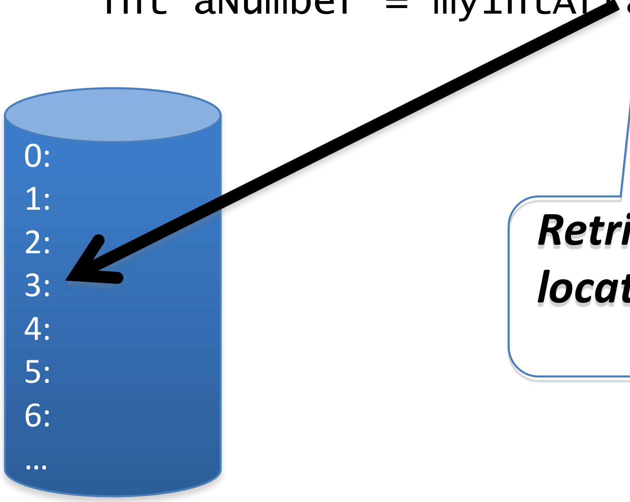
Index (or subcript) of the element in array c

A 12-element array.

Arrays

- An element of an array can be retrieved using its index location:

```
int aNumber = myIntArray[3];
```



Retrieves the integer at index location #3….. the <u>fourth</u> value

Arrays

- The size of an array can be determined using the length variable:

```
int sizeOfMyArray = myStringArray.length;
```

CAREFUL…

The highest index location of an array is its length minus one.

Doesn’t Exist!!

```
int x = myStringArray.length;

int y = myStringArray[x];
```

```java
// Fig. 6.2: InitArray.java
// Initializing the elements of an array to default values of zero.

public class InitArray
{
   public static void main( String[] args )
   {
      int[] array; // declare array named array

      array = new int[ 10 ]; // create the array object

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

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

Fig. 6.2 | Initializing the elements of an array to default values of zero.

Array Initializer

- You can create an array and initialize its elements with an array initializer—a comma-separated list of expressions (called an initializer list) enclosed in braces.

- Array length is determined by the number of elements in the initializer list.

- The following statement creates a five-element array with index values 0–4

```
int[] n = { 10, 20, 30, 40, 50 };
```

- Element `n[0]` is initialized to `10`, `n[1]` is initialized to `20`, and so on.

- When the compiler encounters an array declaration that includes an initializer list, it counts the number of initializers in the list to determine the size of the array, then sets up the appropriate `new` operation "behind the scenes."

```java
// Fig. 6.3: InitArray.java
// Initializing the elements of an array with an array initializer.

public class InitArray
{
   public static void main( String[] args )
   {
      // initializer list specifies the value for each element
      int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

Fig. 6.3 | Initializing the elements of an array with an array initializer.

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Objectives

- Understand and explain the concepts of arrays
- Perform tasks associated with arrays:
  - Create an array
  - Initialize an array
  - Retrieve values stored in an array
  - Set/Change values in an array
  - Find the length of an array
  - Use a for-loop to "walk-through" the elements in an array
- Explain the limitations when using arrays
  - Can't change the length of an existing array
  - Arrays can only hold the data types it was declared with

Objectives

- Describe how to perform the following manipulations on arrays:
    - Get the first element of an array
    - Get the last element of an array
    - Change each element of an array
- Explain the concepts related to variable scope and why it is important
- Use the increment/decrement shortcut assignments properly in a program
- Use the debugger in their IDE to walk through the code

Arrays

Arrays are a collection of elements having the same data types.

- Examples:
    - A roster of names
    - The weather report
    - In sports, the points earned per inning / quarter / half
- In Java, this would mean that we are creating:
    - An array of Strings
    - Also an array of doubles
    - An array of ints

Arrays: What if we didn’t use them

Let’s define the points scored per quarter in a basketball game.

```
public class Basketball {

   public static void main(String[] args) {
      int homeTeamQ1Score = 20;
      int homeTeamQ2Score = 14;
      int homeTeamQ3Score = 18;
      int homeTeamQ4Score = 23;

      int awayTeamQ1Score = 20;
      int awayTeamQ2Score = 26;
      int awayTeamQ3Score = 10;
      int awayTeamQ4Score = 27;
   }
}
```

Suppose we needed to create variables that tracked the scores per quarter. There are 4 quarters in a basketball game, and there are 2 teams...so we would need 8 variables!

Arrays: What if we **did** use them

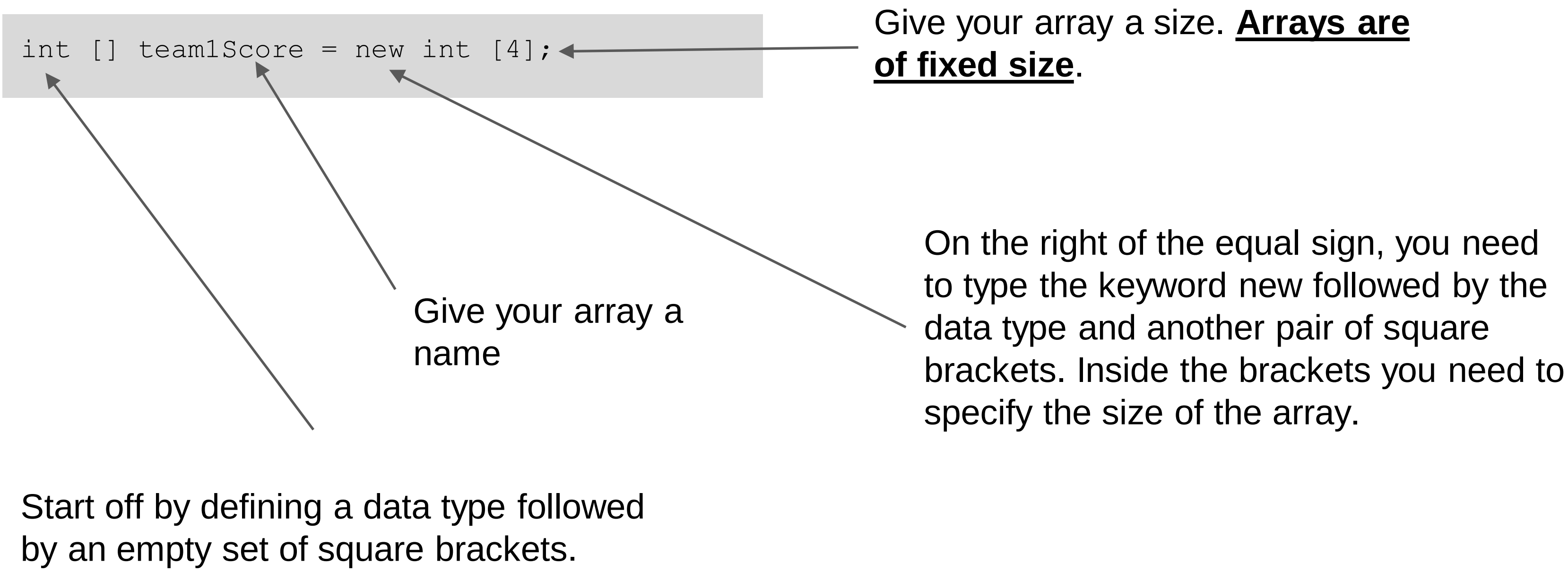
The previous example can be implemented with an array.

```
public class Basketball {

   public static void main(String[] args) {

      int [] team1Score = new int [4];
      int [] team2Score = new int [4];

      team1Score[0]= 20;
      team1Score[1]= 14;
      team1Score[2]= 18;
      team1Score[3]= 23;

      team2Score[0]= 20;
      team2Score[1]= 26;
      team2Score[2]= 10;
      team2Score[3]= 27;
   }
}
```

We created 2 arrays of integers, team1Score and team2Score. We have set the length of each one of these arrays to 4 elements.

Arrays: Declaration Syntax

An array has the following syntax:



```
int [] team1Score = new int [4];
```

Give your array a size. **Arrays are of fixed size**.

On the right of the equal sign, you need to type the keyword new followed by the data type and another pair of square brackets. Inside the brackets you need to specify the size of the array.

Give your array a name

Start off by defining a data type followed by an empty set of square brackets.

Arrays: Assigning Items

We can assign items to individual elements in an array:

```
int [] team1Score = new int [4];

team1Score[0]= 20;
team1Score[1]= 14;
team1Score[2]= 18;
team1Score[3]= 23;
```

index	0	1	2	3
value	20	14	18	23

Note that this array has 4 slots.

These slots are called “elements.” We can access elements by specifying a number starting from 0. This number that designates the element is called an index.

Variable Scope

- Code can be grouped together in **blocks**
  - **Block** is a group of zero or more statements between braces
  - Can be used anywhere a single statement is allowed.
- **Scope** defines where the variable can be referenced
  - **Referenced** means exists, or can be accessed
- **Rules of Scope:**

1. Variables declared inside of a function or block **{..}** are local variables and only available within that block. This includes loops.
2. Blocks can be nested within other blocks and therefore if a variable is declared outside of a block, it is accessible within the inner block.

Variable Scope Example

```
    public static void main(String[] args){
            {
                // The variable x has scope within
                // brackets
                int x = 10;
                System.out.println(x);
            }

            // Uncommenting below line would produce
            // error since variable x is out of scope.

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

Variable Scope Example

```
public static void main(String args[]){
        int outside = 5;
        { // inner block
            int inside = 7;
            outside += 15; // outside = outside + 15;
            System.out.println(inside); // will print 7
        }

        // will print 20
        System.out.println(outside);
    }
```

Variable Scope Example

```
public static void main(String args[]) {
       int x = 25;
       if (x >= 20) {
           x /= 5;
           System.out.println(x);
       }

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

Loops

Loops are designed to perform a task until a condition is met.

- Examples:
  - Print a list of all numbers between 0 and 10.
  - Print all the items in a grocery list.

- There are several types of loops in Java:
  - For Loop (by far the most common)
  - While Loop
  - Do-While Loop

Loops: Visualized

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

For loop order of execution:

1. Initialization
2. Condition
3. Block
4. Altering

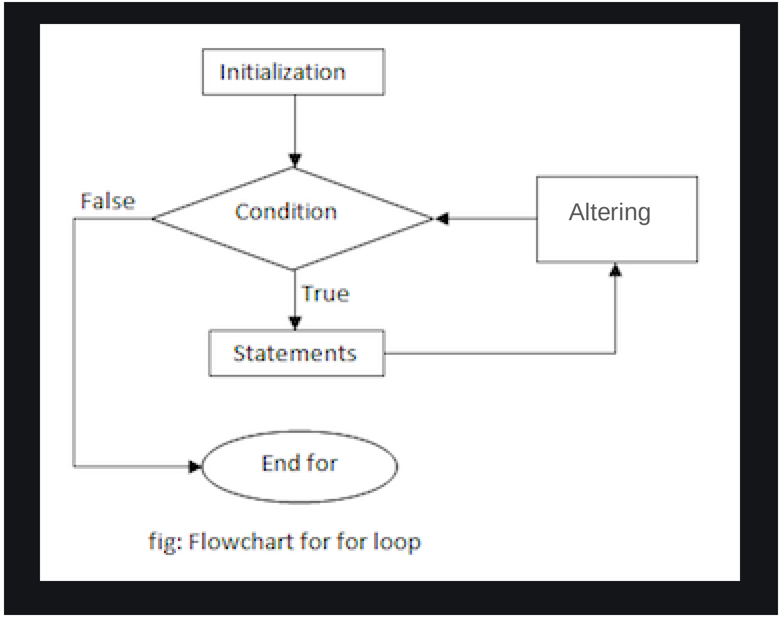


fig: Flowchart for for loop

Loops: For Loops

For Loops are the most common types of loops. They follow this pattern:

```
for (//initialize index ; //check condition ; //increment or decrement index) {

        // action to repeat

}
```

Loops: For Loop Examples

Here is an example:

```
public class MyClass {

           public static void main(String[] args) {

              for (int i=0; i < 5; i++) {
                 System.out.println(i);
              }
           }
}
```

This code will print the numbers 0 to 4.

Loops: For Loop Visualized

Let’s consider this example again:

```
for (int i=0; i < 5; i++) {
     System.out.println(i);
}
```

Each run of the loop is called an iteration. You can generally tell how many iterations the loop will run for just by looking at the code. In this example, we expect 5 iterations.

Iteration #	**Value of i at beginning**	**Action**	Value of i at end	Is i less than 5?
1	0	Prints 0	1	Yes
2	1	Prints 1	2	Yes
3	2	Prints 2	3	Yes
4	3	Prints 3	4	Yes
5	4	Prints 4	5	No

Loops: While & Do-While Loops

Here is an example:

```
int i = 0;

while (i < 5) {
      System.out.println(i);
      i++;
}
```

```
int i = 0;

do {
      System.out.println(i);
      i++;
} while (i < 5);
```

- For both the while and do-while you must increase or decrease the index manually.
- The do-while is guaranteed to execute **at least once**.

Arrays: Iterating

Going back to our basketball example, let’s say we want to print the score for each quarter. This might be the first thing that comes to mind:

```
int [] team1Score = new int [4];

System.out.println(team1Score[0]);
System.out.println(team1Score[1]);
System.out.println(team1Score[2]);
System.out.println(team1Score[3]);
```

This seems like a very inefficient way to accomplish this task. There must be a better way! (Spoiler Alert: there is)

Arrays: Iterating

We can use a loop to sequentially iterate through each element of the array.

```
public class Basketball {

    public static void main(String[] args) {

       int[] team1Score = new int[4];

       team1Score[0] = 20;
       team1Score[1] = 14;
       team1Score[2] = 18;
       team1Score[3] = 23;

       for (int i = 0; i < team1Score.length; i++){

           System.out.println(team1Score[i]);
       }
   }
}
```

- The value of team1Score.length will be 4.
- The loop will iterate three times, once for i=0, once for i=1, and once for i=2. After that, i is no longer less than team1Score.length.
- Note that i is also used to specify the position of the array so that it knows which element to print.

Arrays: Iterating (in slow motion)

Given the previous example, an array containing {20, 14, 18, 23}

Iteration #	**i**	**team1Score[i]**	**Is i < team1Score.length ?**
1	0	20	i has increased to 1, 1 < 4 so yes
2	1	14	i has increased to 2, 2 < 4 so yes
3	2	18	i has increased to 3, 3 < 4 so yes
4	3	23	i has increased to 4, 4 < 4 so no. No more iterations, loop ends.

Arrays: Alternative Declaration

If you know the values you want to place in an array ahead of time, consider using this condensed format to declare:

```
int[] team1Score = {4, 3, 2};
String[] lastNames = { "April", "Pike", “Kirk”};
```

The Increment/Decrement Operator

The increment (++) and decrement operator (--) increases or decreases a number by 1 respectively. You have seen this in the context of a for loop. Here is a more general example (the output is 94):

```
int x = 93;
x++;
System.out.println(x);
```

- If the operator is in behind a variable it is a **postfix operator** (i.e. x++).
  - A postfix operator is evaluated first, then incremented.
- If the operator is in front a variable it is a **prefix operator** (i.e. ++x).
  - A prefix operator is incremented first, then evaluated.

The Increment/Decrement Operator: Example

A choice of having a prefix or a postfix in certain calculations can have unexpected consequences:

```
int a = 3;
System.out.println(++a + 4); // prints 8

int b = 3;
System.out.println(b++ + 4); // prints 7
```

- In the first example, we have a prefix operator, a is increased first and becomes 4, it is used in the operation (4+4).
- In the second example we have a postfix operator, a is used in the operation first (3+4), then increased.

Second Deck
Loops and Arrays

Module 1: 04

Today’s Objectives

- Arrays
- Variable Scope
- Loops
- Troubleshooting and Debugging

Problem: We can hold a single value in a variable, for example: `String name = “John”`, but what if we want to hold multiple values for a similar purpose?

We could create a variable for each item:

```
            String instructor1 = “John”;
            String instructor2 = 
“Rachelle”;
            String instructor3 = “Steve”;
            String instructor4 = “Matt”;
            String instructor5 = “Dan”;
```

What problems would this cause?

We could reuse the same variable:

```
            String instructor = “John”;
            instructor  = “Rachelle”;
            instructor  = “Steve”;
            instructor  = “Matt”;
            instructor  = “Dan”;
```

What problems would this cause?

Arrays

An array is a series of values of the **same data type** that are held together in a wrapper that can be treated as a single thing.

Arrays are **created using the *new* keyword** and **Arrays are a fixed size** (The number of items the array will contain must be set when the array is created and cannot be changed after it is set.).

```
String[] instructors = new String[5];
```

In Java arrays can technically be defined as

```
int[] nums;
```

or

```
int nums[];
```

The second (`int nums[]`) syntax is a leftover from C, and while syntactically valid, is considered bad form due to decreased readability.

Array Default Values

Data Types have default values. When an array is created with the new keyword, each of the items is created with that data types default value.

Data Type	Default
byte, short, long, int, char	0
double, float	0.0
boolean	false
String	null

```
int[] nums = new int[3];
```

Index	Value
0	0
1	0
2	0

```
boolean[] b = new boolean[5];
```

Index	Value
0	false
1	false
2	false
3	false
4	false

Array Elements and Indexes

Arrays are a set of values, like a variable, of the specified data type. These values are called **Elements**.

The elements in the array have an **index**. An index is a numeric value that ***starts at 0*** and provides a way to identify a specific element in the array. The index is used to set or get an element.

Setting values for elements in an array:

```
instructors[0] = “John”;
instructors[1] = “Rachelle”;
instructors[2] = “Steve”;
instructors[3] = “Matt”;
instructors[4] = “Dan”;
```

Index	Value
0	John
1	Rachelle
2	Steve
3	Matt
4	Dan

Getting values from an Array

Values from arrays can be retrieved using the index.

```
String name = instructors[2]; ← name will contain “Steve”
```

The array variable with the index, can be used like any other variable.

```
    String greeting = “Hello, “ + instructors[1];
```

What will the variable greeting contain?

instructors

Index	Value
0	John
1	Rachelle
2	Steve
3	Matt
4	Dan

Updating Array Elements

The value of an element in an Array can be changed using its index.

```
instructors[0] = “The John Fulton”;
```

Index	Value
0	The John Fulton
1	Rachelle
2	Steve
3	Matt
4	Dan

instructors

Index	Value
0	John
1	Rachelle
2	Steve
3	Matt
4	Dan

How will the following code change the Array?

```
instructors[4] = “Doug”;
```

How will the following code change the Array?

```
instructors[1] = instructors[1] + “sh”;
```

How will the following code change the Array?

```
instructors[5] = “Dan Again”;
```

instructors

Index	Value
0	John
1	Rachelle
2	Steve
3	Matt
4	Dan

What will the variable doubleNum contain?

```
int doubleNum = nums[1] * 2;
```

What will the variable third contain?

```
int third = nums[0] / 3;
```

How will the following code change the Array?

```
nums[2] = nums[4];
```

How will the following code change the Array?

```
nums[1] = nums[1] * nums[0];
```

int[] nums

Index	Value
0	10
1	20
2	25
3	15
4	1

Arrays declared elsewhere...

Given the following Array: `int[] nums` that we know has been created and filled with values, how could we:

1. Access the first element in the Array?

1. Find out how many elements are in the array?

1. Access the last element in the Array?

Static Initialization

If the elements that will be in an array are known when the array is created, then those items can be used to create the array. The size will be populated from the number of elements in the list.

```
int[] nums = new int[] { 10, 20, 30 };
```

Is the same as:

```
int[] nums = new int[3];
int[0] = 10;
int[1] = 20;
int[2] = 30;
```

```
String[] instructors = new String[] {
                                    “John”,
“Rachelle”,
                                    “Steve”,
“Matt”, “Dan”};
```

Is the same as:

```
String[] instructors = new String[5];
instructors [0] = “John”;
instructors [1] = “Rachelle”;
instructors [2] = “Steve”;
instructors [3] = “Matt”;
instructors [4] = “Dan”;
```

Arrays Summary

1. Arrays are fixed-length - the size is set when created and cannot be changed
2. Arrays have a 0 based index
3. Items in an array are called Elements
4. Everything Element an array must be the same data type
5. Elements may be accessed to retrieved, changed, or set using their index
6. Arrays are created using the new keyword
7. The first element in an array is always at index 0
8. The number of elements in an array can be gotten using `array.length`

`int[] numbers = new int[10];` ← creates an array that holds 10 integers

numbers[1] ← access the second element in the array

numbers[2] = 10; ← sets the value of second element in the array to 10

numbers.length ← gets the number of elements in the array

numbers[numbers.length - 1] ← access the last element in the array

Variable Scope

A variables **scope** defines where in a program that the variable is valid (*can be referenced*). When code execution reaches a point where the variable *can no longer be referenced*, then the variable is **out of scope**.

In Java, scope is determined by blocks.

1. Variables declared inside a block ({...}) are only available within that block.
2. Blocks can be *nested* within other blocks. Variables declared in a block are also available in any nested blocks. However, they are not available in a blocks *parent* block.

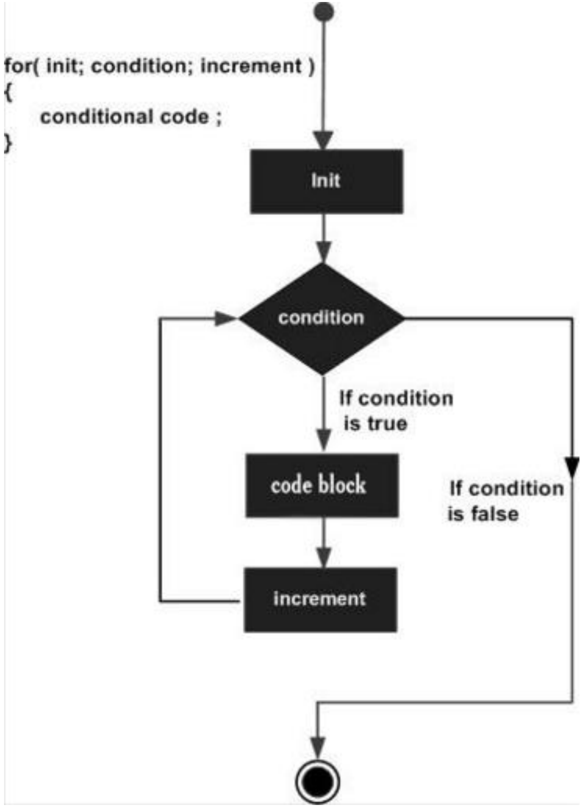
Java does not have the concept of Global scoped variables

```
{
            int x = 10;
      {
                  int y = 20;
            {
                        int z
            = 5;
                        y = y
            + z;
          ←}what variables are available here?     If y is available, what is its value?
      }
        ← what variables are available here?
}
```

Loops

Loops allow code to repeated for a set number of times.   Loops use a boolean condition, and execute the code while that condition is true and stop execution once the condition becomes false.

The boolean condition must start true, and something in the loop must change the loop so that it eventually becomes false, and ends the loop.



```
for( init; condition; increment )
{
    conditional code ;
}
```

for loop

1. Declare an incrementer variable.
2. Create a boolean condition that is true
3. Define an iterator expression that changes the incementer. This controls how many times the loop will execute the code block.

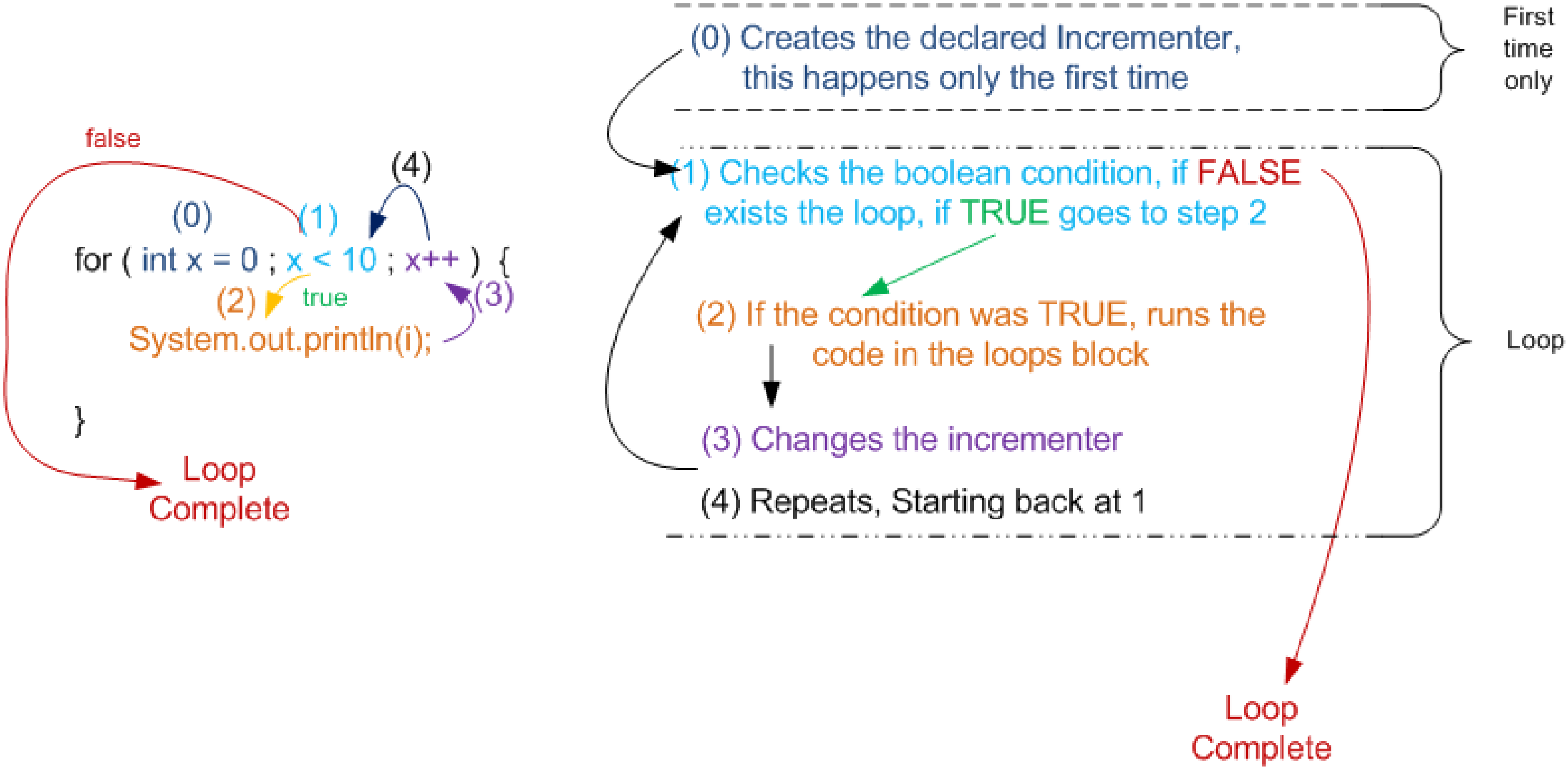
When the incrementer variable is declared in the for() it is in scope for the code block established by the loop, and out of scope once the loop ends.

```
for (  incrementer ; boolean condition ; iterator expression ) {
                // Code that repeats while condition is true

}
```

```
for (int i = 0; i < 10; i++) {

        System.out.println(i);

}
```

Output:

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



```
for ( int x = 0 ; x < 10 ; x++ )  {
        System.out.println(i);

}
```

(0) Creates the declared Incrementer, this happens only the first time

(1) Checks the boolean condition, if FALSE exists the loop, if TRUE goes to step 2

(2) If the condition was TRUE, runs the code in the loops block

(3) Changes the incrementer

(4) Repeats, Starting back at 1

Increment / Decrement Shorthand

i++ → use the current value of i, and then i = i + 1;

++i → i = i + 1, then use the new value of i

i-- → use the current value of i, then i = i - 1;

--i → i = i - 1, then use the new value of i

For readability, the general standard is to only use these increment / decrement shorthand in the for() of a loop, or on lines by themselves.

for (; ; i++) ← **OK**
i++;
← **OK**

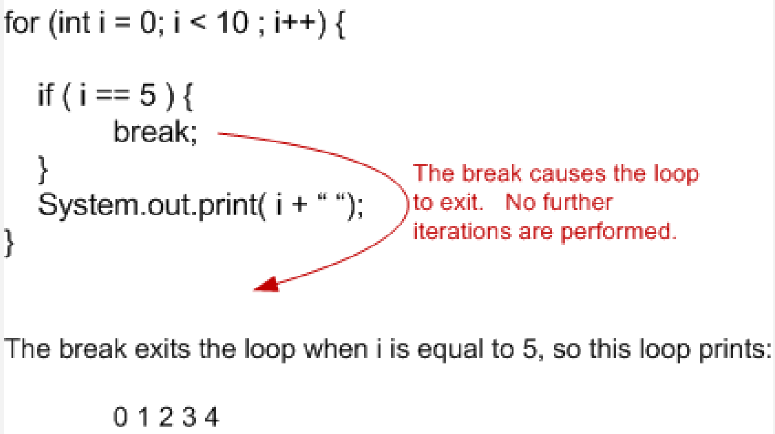
x = 120 + --y - 5; ← **Discouraged**

Assignment Shorthand

x += y → x = x + y;
x -= y → x = x - y;
x *= y → x = x * y;
x /=y → x = x / y;

Break and Continue

Putting a **break;** statement in the code block of a loop, immediately ends the loop.



```
for (int i = 0; i < 10 ; i++) {

   if ( i == 5 ) {
          break;
   }
   System.out.print( i + “ “);
}
```

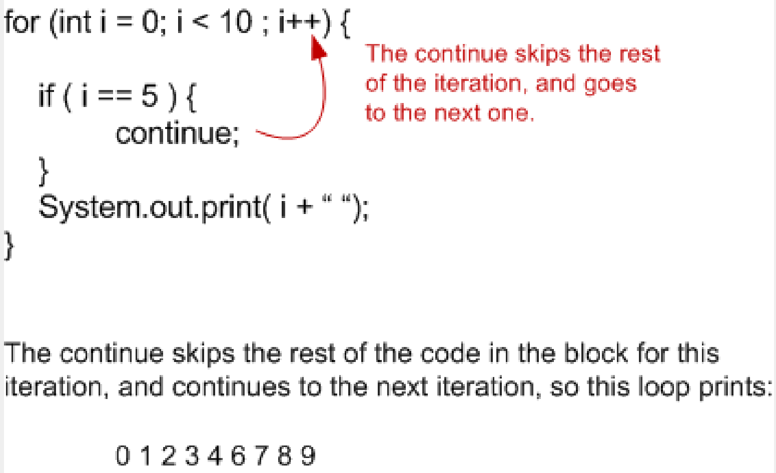
The break causes the loop to exit.   No further iterations are performed.

The break exits the loop when i is equal to 5, so this loop prints:

```
         0 1 2 3 4
```

Visual Explanation

A **continue;** statement in the code block, ends the current iteration early, and continues the loop on the next iteration.



```
for (int i = 0; i < 10 ; i++) {

   if ( i == 5 ) {
          continue;
   }
   System.out.print( i + “ “);
}
```

The continue skips the rest of the iteration, and goes to the next one.

The continue skips the rest of the code in the block for this iteration, and continues to the next iteration, so this loop prints:

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

Visual Explanation

Loops with Arrays

instructors

Index	Value
0	John
1	Rachelle
2	Steve
3	Matt
4	Dan

The incrementor in a loop can start at 0, and be increased by 1, which matches the index of each element in an array. If we then create a boolean condition, that stops the loop when it reaches the last index of the array, it would effectively loop through each index in the array, allowing the code to access each element in the block.

```
for (int i = 0; i < instructors.length ; i++) {

        System.out.println ( instructors[i] );

}
```

```
Output:     John                        ( i == 0 → instructors[ 0 ] )
                Rachelle                ( i == 1 → instructors[ 1 ] )
                Steve                           ( i == 2 → instructors[
2 ] )
                Matt                            ( i == 3 → instructors[
3 ] )
```

Visual Explanation

5-minute Review

- Arrays
- Variable Scope
- for Loops
- for Loops with Arrays