Loops
&
Arrays

Today's Objectives

- Main concepts of arrays
- Performing tasks associated with arrays:
  - Creating an array
  - Initializing an array
  - Retrieving values stored in an array
  - Setting/Changing values in an array
  - Finding the length of an array
  - Using a for-loop to "Walk-through" the elements in an array
- Limitations when using arrays
  - Can't change the length of an existing array
  - Arrays can only hold the data types it was declared with)
- Performing manipulations on arrays:
  - Getting the first element of an array
  - Getting the last element of an array
  - Changing each element of an array
- Variable scope: what it is and why it is important
- Using increment/decrement assignments in a program
- Using the debugger in the IDE to walk through your code

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.

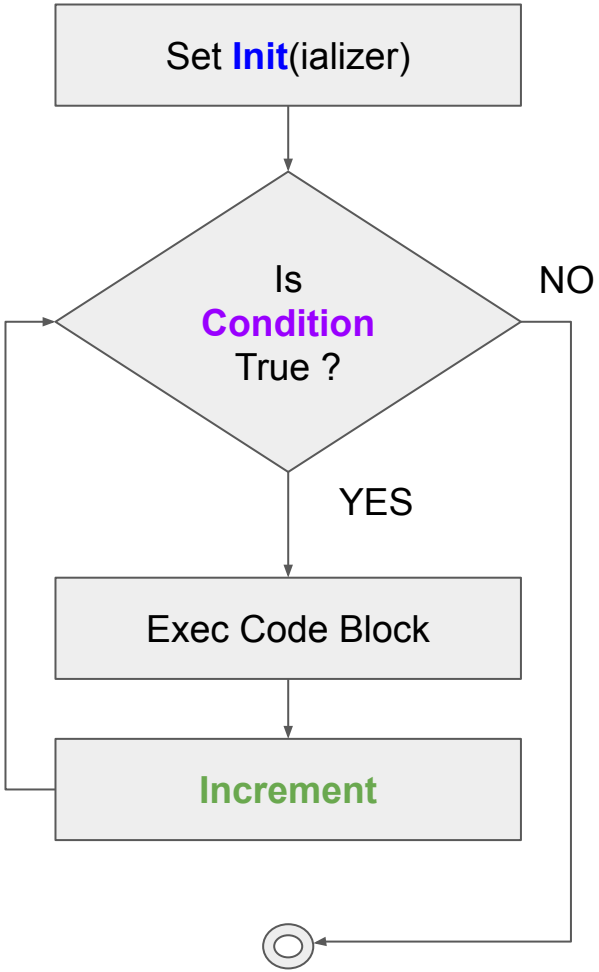
There are several types of loops in Java:

- For Loop (by far the most common)
- While Loop
- Do-While Loop

For Loop: Visualized

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

```
for( init; condition; increment )
{
      //Code Block
      Conditional code;
}
```



For Loop: Example

Here's what the for loop pattern looks like in code:

```
public class MyClass {

	public static void main(String[] args) {

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

		for (int i=0; i < 3; i++) {
			System.out.println(i);
		}
	}
}
```

This code will print **"Developers!"** five times, followed by the numbers 0 to 2.

For Loop: Example Visualized

Let's walk through this part of the code:

```
for (int i=0; i < 3; i++) {
        System.out.println(i);
}
```

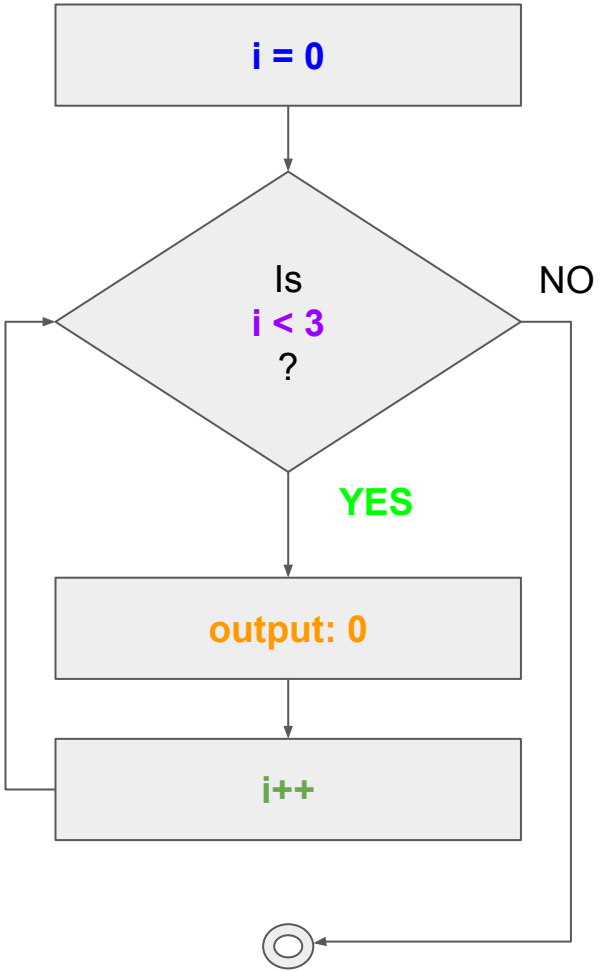
Each run through the loop is called an **iteration**.

First time through the loop:

i is 0

output is: 0

i is incremented to 1



For Loop: Example Visualized

Let's walk through this part of the code:

```
for (int i=0; i < 3; i++) {
        System.out.println(i);
}
```

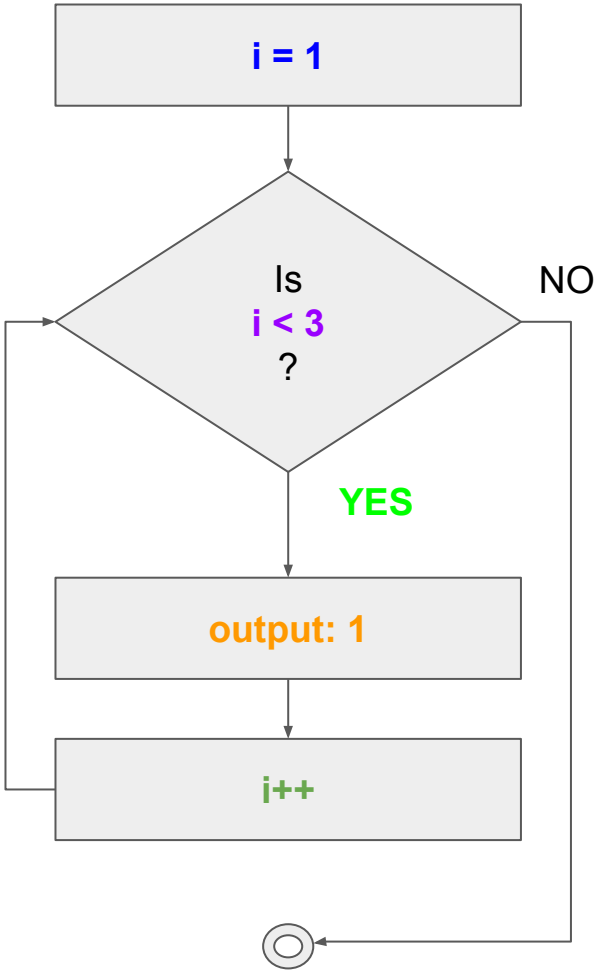
Each run through the loop is called an **iteration**.

Second time through the loop:

i is 1

output is: 1

i is incremented to 2



For Loop: Example Visualized

Let's walk through this part of the code:

```
for (int i=0; i < 3; i++) {
        System.out.println(i);
}
```

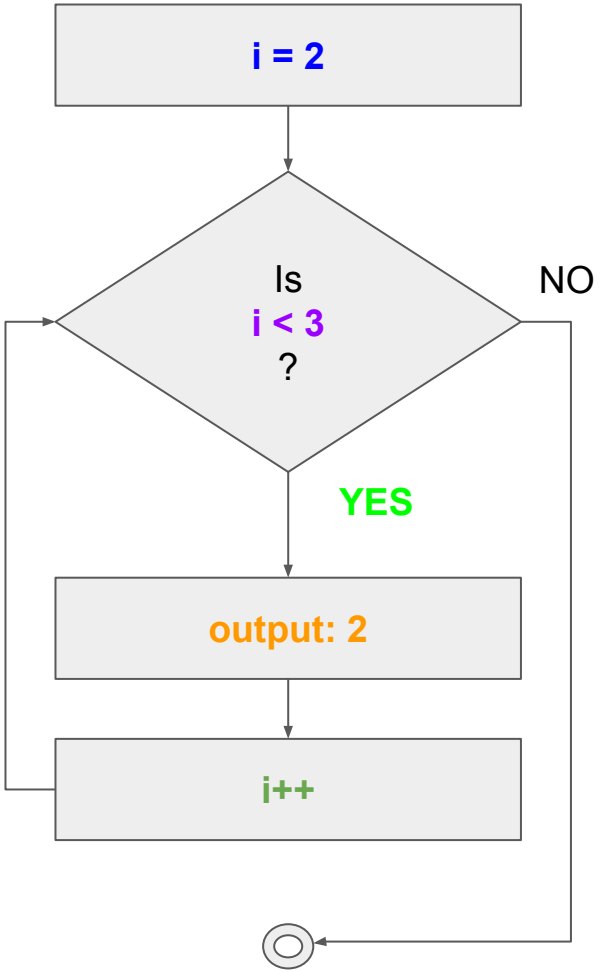
Each run through the loop is called an **iteration**.

Third time through the loop:

i is 2

output is: 2

i is incremented to 3



For Loop: Example Visualized

Let's walk through this part of the code:

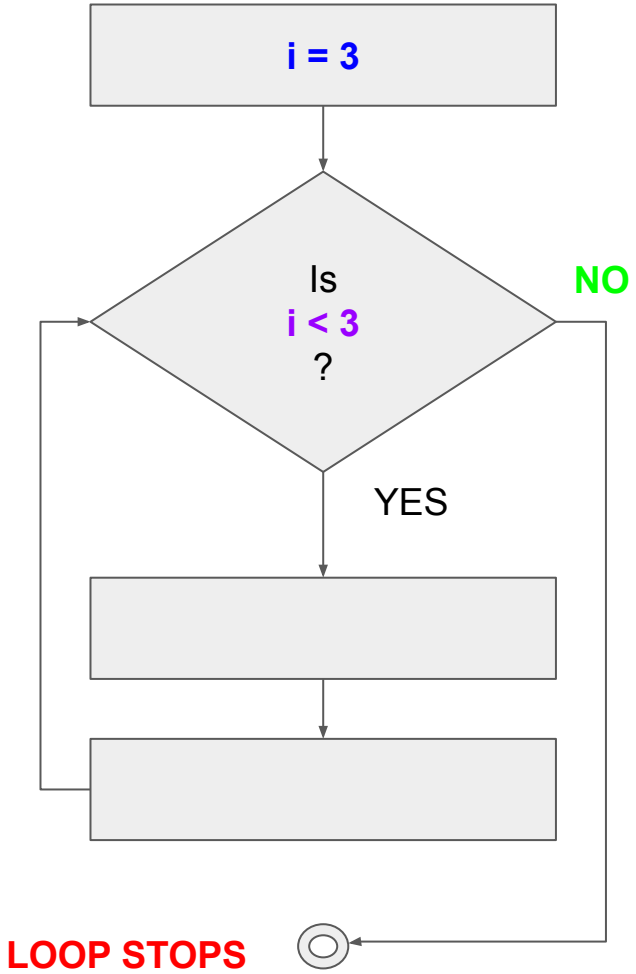
```
for (int i=0; i < 3; i++) {
        System.out.println(i);
}
```

Each run through the loop is called an **iteration**.

Fourth time through the loop:

i is 3

loop stops



While Loop vs. Do-While Loop

While and Do-While Loops continue looping until a condition is no longer true.

While Loop

```
int i = 0;

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

If condition is not met on initial run, code in loop will never execute.

Do-While Loop

```
int i = 0;

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

Loops is guaranteed to run at least once before the loop condition is checked.,

- 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.

Remember Blocks?

Code that needs to belong together as a single unit can be written in a **block**.

- A **block** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

- Blocks have a list of statements within them and are enclosed with braces `{ . . }`

Variable Scope

A variable's **scope** defines where in the program a variable exists (i.e. can be referenced). When code execution reaches a point where a variable is no longer referenceable, the variable is said to be **out of scope**.

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. Therefore, if a variable is declared outside of a block, it is accessible within the inner block.

Variable Scope

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
}

System.out.println(outside);
System.out.println(i);
```

The scope of **outside** is the whole code.

The scope of **i** is only within the loop block.

Variable Scope

Will print value of **outside** (5).

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
}

System.out.println(outside);
System.out.println(i);
```

The scope of **outside** is the whole code.

The scope of **i** is only within the loop block.

Variable Scope

Will print value of **outside** (5).

Will print value of **outside** (5) and the value of **i**.

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
}

System.out.println(outside);
System.out.println(i);
```

The scope of **outside** is the whole code.

The scope of **i** is only within the loop block.

Variable Scope

Will print value of **outside** (5).

Will print value of **outside** (5) and the value of **i**.

Will print value of **outside** (5) but trying to print the value of **i** will cause an **error** because **i** is **out of scope**!

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
}

System.out.println(outside);
System.out.println(i);
```

The scope of **outside** is the whole code.

The scope of **i** is only within the loop block.

Variable Scope: Nested Blocks

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
        if (i == 4} {
                int inside = i + 5;
                System.out.println(inside);
        }
        System.out.println(inside);
}

System.out.println(outside);
System.out.println(i);
```

Variable Scope: Nested Blocks

The `if` block is nested within the `for` loop block.

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
        if (i == 4} {
                int inside = i + 5;
                System.out.println(inside);
        }
        System.out.println(inside);
}

System.out.println(outside);
System.out.println(i);
```

Variable Scope: Nested Blocks

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
        if (i == 4} {
                int inside = i + 5;
                System.out.println(inside);
        }
        System.out.println(inside);
}

System.out.println(outside);
System.out.println(i);
```

The `if` block is nested within the `for` loop block.

The `if` block has access to the variables in the `for` loop block.

The `for` loop block does **NOT** have access to the variables in the `if` block.

Variable Scope: Nested Blocks

```
int outside = 5;

System.out.println(outside);

for (int i=0; i < 3; i++) {
        System.out.println(outside);
        System.out.println(i);
        if (i == 4} {
                int inside = i + 5;
                System.out.println(inside);
        }
        System.out.println(inside);
}

System.out.println(outside);
System.out.println(i);
```

The `if` block is nested within the `for` loop block.

The `if` block has access to the variables in the `for` loop block.

The `for` loop block does **NOT** have access to the variables in the `if` block.

This will cause an error because `inside` is out of scope!!!

Arrays: Life without them as we know it now

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**!

Introducing Arrays...

An **array** is a series of elements having the **same data types**.

Examples:

- A roster of names
- The 10-Day weather report (temperatures)
- In sports, the points earned per inning / quarter / half

In Java, this would mean that we are creating:

- An array of **String** elements
- An array of **double** elements
- An array of **int** elements

Arrays Visualized

Let's see what the roster of names example would look like:

Larry	Curly	Moe

- You can think of an array as a **series of slots** that hold **data of the same data type** (Strings in this example).

Arrays Visualized

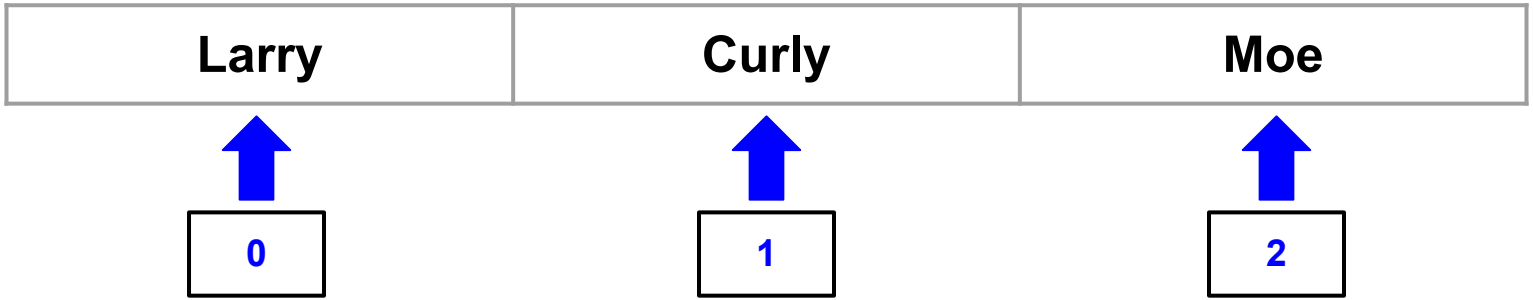
Let's see what the roster of names example would look like:

Larry	Curly	Moe

- You can think of an array as a **series of slots** that hold **data of the same data type** (Strings in this example).
- You can refer to each element in the array by an **index** (kind of like an address in a set of mailboxes).

Arrays Visualized

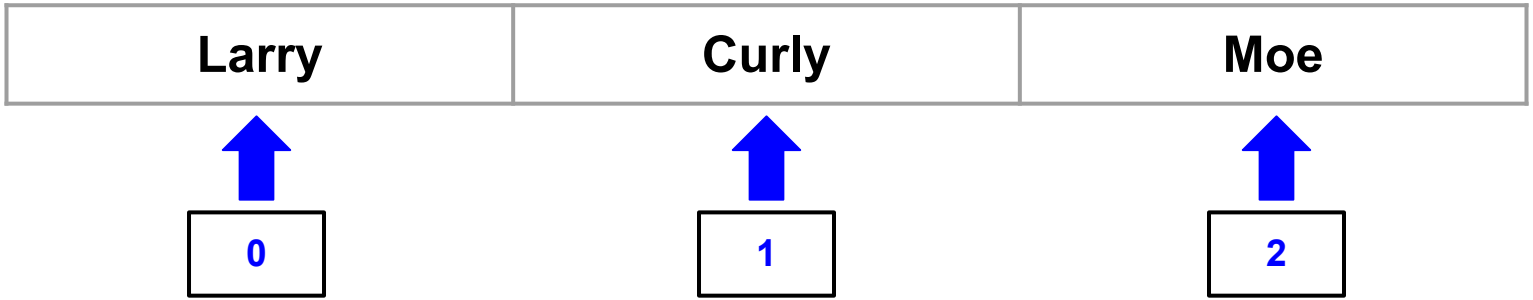
Let's see what the roster of names example would look like:



- You can think of an array as a **series of slots** that hold **data of the same data type** (Strings in this example).
- You can refer to each element in the array by an **index** (kind of like an address in a set of mailboxes).
- The **index** will be an **int** value which increases with each new slot..

Arrays Visualized

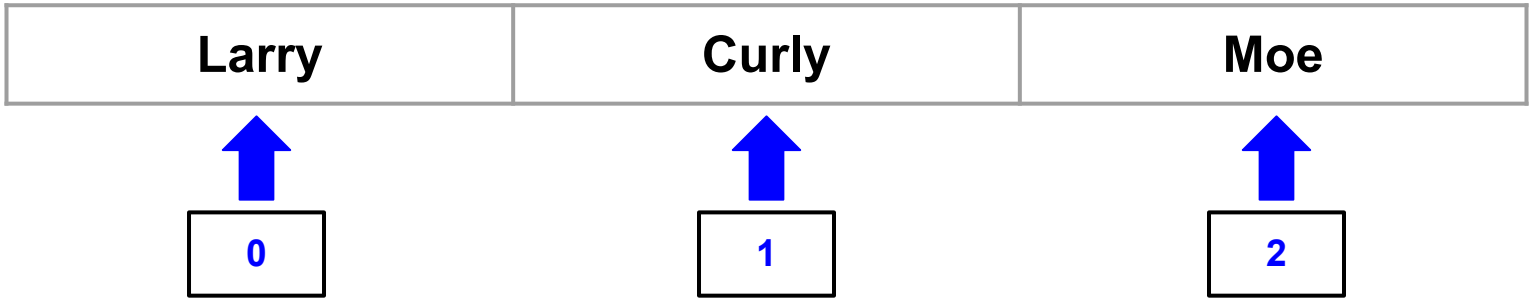
Let's see what the roster of names example would look like:



But wait! How come the index starts at 0? Shouldn't it start at 1?

Arrays Visualized

Let's see what the roster of names example would look like:



But wait! How come the index starts at 0? Shouldn't it start at 1?

You can think of the index as an offset. So if you start at the beginning of the array (the first slot), your offset would be 0 since you are already there. When you go to the next slot, the offset would be 1, and so on. **Array indexes always start at zero.**.

Arrays: Life with arrays

The scores example we looked at before can be implemented using 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`**
  - **`team2Score`**
- We have set the length of each of these arrays to 4 elements.

Arrays: Declaration Syntax

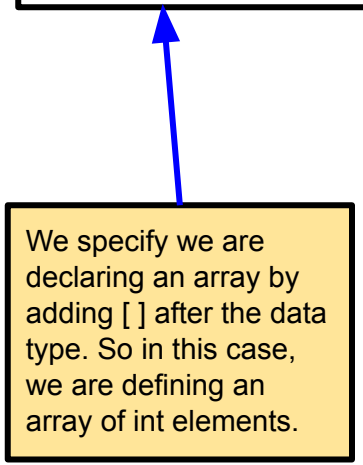
An array has the following syntax:

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

Arrays: Declaration Syntax

An array has the following syntax:

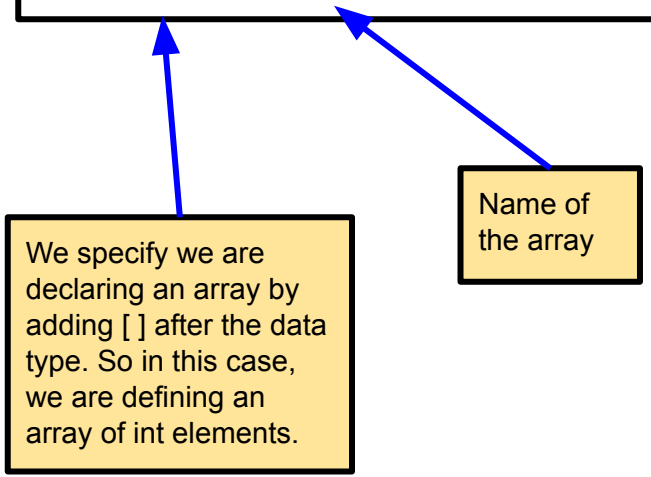
```
int [] team1Score = new int [4];
```



Arrays: Declaration Syntax

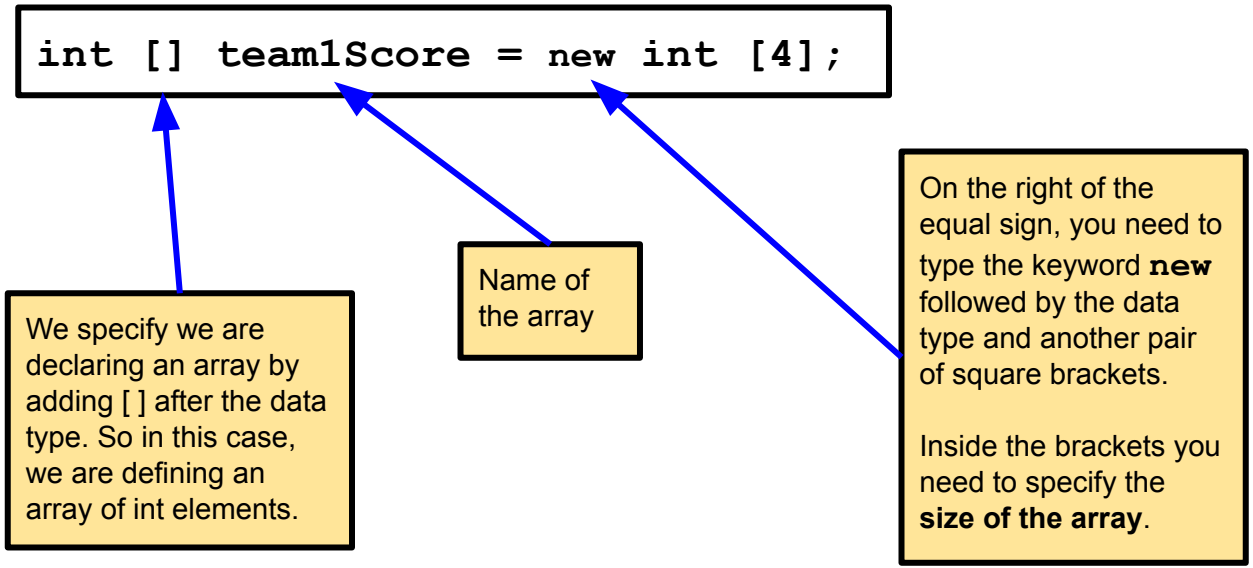
An array has the following syntax:

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



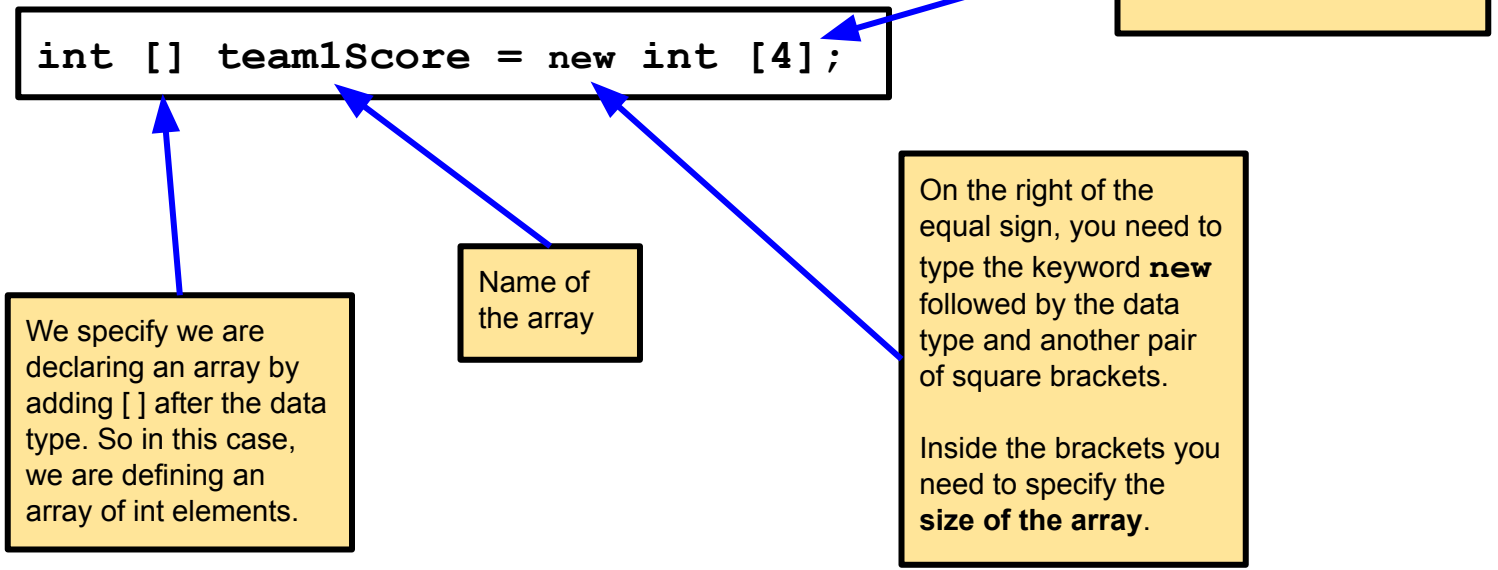
Arrays: Declaration Syntax

An array has the following syntax:



Arrays: Declaration Syntax

An array has the following syntax:



Arrays: Alternative Declaration Syntax

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

```
// create an int array and initialize it.
int[] team1Score = {5, 4, 3, 2};

// create a String array and initialize it.
String[] lastNames = { "April", "Pike", "Kirk"};
```

Arrays: Alternative Declaration Syntax

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

```
// create an int array and initialize it.
int[] team1Score = {5, 4, 3, 2};

// create a String array and initialize it.
String[] lastNames = { "April", "Pike", "Kirk"};
```

Rather than using the **`new`** keyword and specifying the size of the array, we specify a set of data enclosed in braces (`{ .. }`).

The array will be created with the correct size for the data size and will be initialized with the specified data.

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;
```

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;
```

Note that this array has 4 slots.

These slots are called **elements**. We can access an element by specifying the name of the array with the index we want to access in brackets.

Remember, the index numbers start at 0, so `team1Score[0]= 20;` puts the value 20 in the first slot, `team1Score[1]= 14;` puts the value 14 in the second slo, and so on.

Be aware that if you try to access an element beyond the size of the array you will get an **error**.

Arrays: Assigning Items

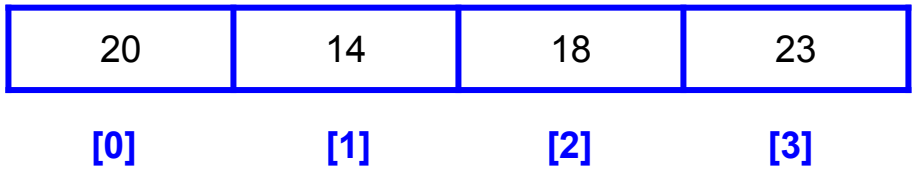
We can assign items to individual elements in an array:

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

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

After this code is run, we will have an array that looks like this:

team1Score



Note that this array has 4 slots.

These slots are called **elements**. We can access an element by specifying the name of the array with the index we want to access in brackets.

Remember, the index numbers start at 0, so `team1Score[0]= 20;` puts the value 20 in the first slot, `team1Score[1]= 14;` puts the value 14 in the second slo, and so on.

Be aware that if you try to access an element beyond the size of the array you will get an **error**.

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];

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

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

… but this approach has merely transferred the problem of having multiple variables for each score to a new problem of having a `println` for each *Element* in the array.

How could we simplify our code?

Arrays: Iterating

Remember how our for loop worked? We can use a for loop to iterate through the index values and print each one based on the current index!

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

Arrays: Iterating

Remember how our for loop worked? We can use a for loop to iterate through the index values and print each one based on the current index!

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

- Arrays have a **`length`** property that we can use to find out how many elements the array has.

Arrays: Iterating

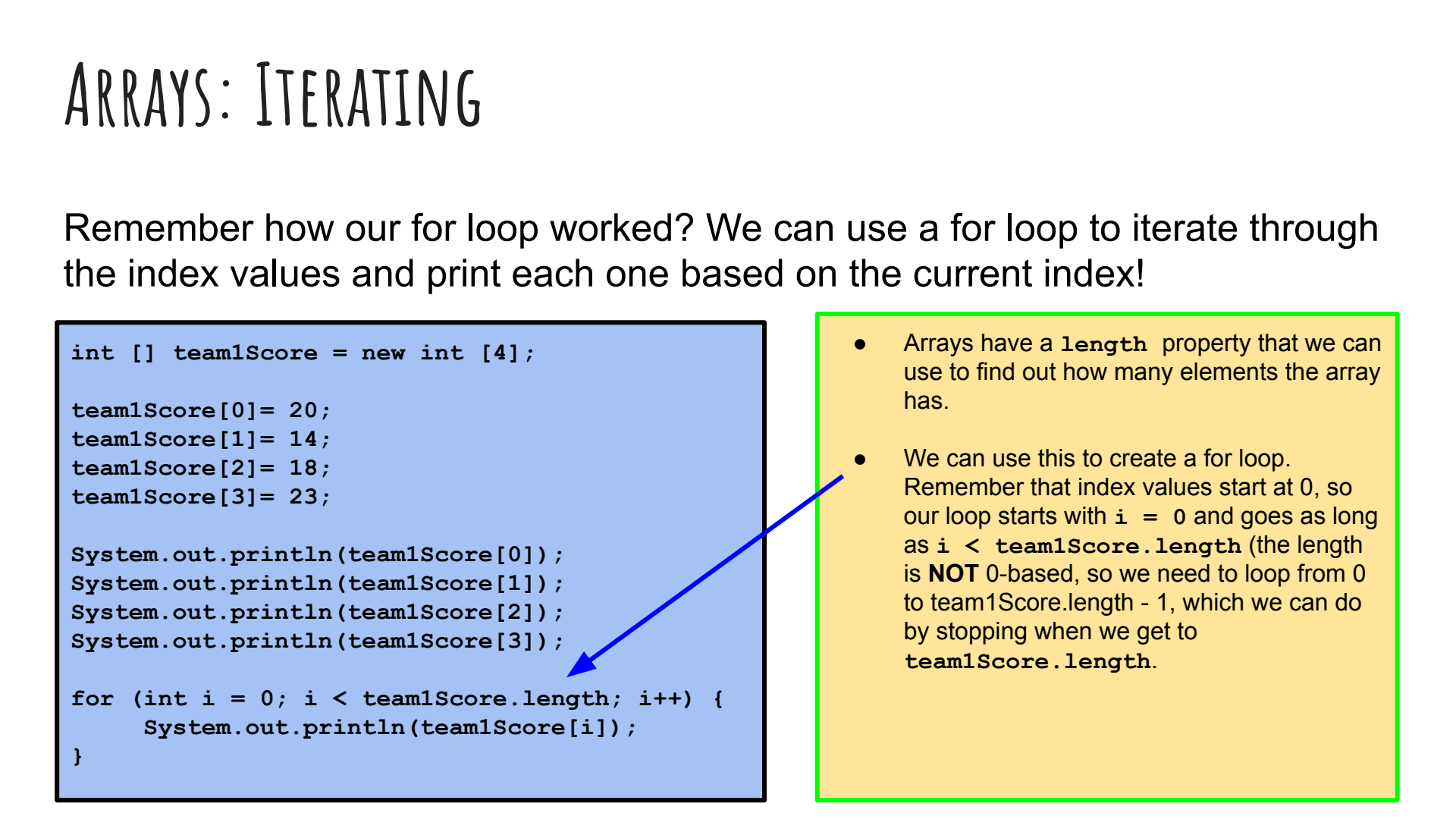
Remember how our for loop worked? We can use a for loop to iterate through the index values and print each one based on the current index!

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```



- Arrays have a **`length`** property that we can use to find out how many elements the array has.
- We can use this to create a for loop. Remember that index values start at 0, so our loop starts with **`i = 0`** and goes as long as **`i < team1Score.length`** (the length is **NOT** 0-based, so we need to loop from 0 to team1Score.length - 1, which we can do by stopping when we get to **`team1Score.length`**.

Arrays: Iterating

Remember how our for loop worked? We can use a for loop to iterate through the index values and print each one based on the current index!

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

- Arrays have a **`length`** property that we can use to find out how many elements the array has.
- We can use this to create a for loop. Remember that index values start at 0, so our loop starts with **`i = 0`** and goes as long as **`i < team1Score.length`** (the length is **NOT** 0-based, so we need to loop from 0 to team1Score.length - 1, which we can do by stopping when we get to **`team1Score.length`**.
- We then use the current value of **`i`** as the index value to access in the array and print the value at that element.

Arrays: Iterating

Let's walk through and see what happens...

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

When we start the loop:

`i = 0;`

We check:

Is `i < team1Score.length`
(`team1Score.length` will be 4)?

Since `i` is 0, it is less than 4 and we get into the code block.

We print `team1Score[i]` and since `i = 0`, that is `team1Score[0]` and so we print:

`20`

Before we go to the next iteration, we increment `i`

Arrays: Iterating

Let's walk through and see what happens...

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

Next iteration through the loop:

`i = 1;`

We check:

Is `i < team1Score.length`
(`team1Score.length` will be 4)?

Since `i` is 1, it is less than 4 and we get into the code block.

We print `team1Score[i]` and since `i = 1`, that is `team1Score[1]` and so we print:

`14`

Before we go to the next iteration, we increment `i`

Arrays: Iterating

Let's walk through and see what happens...

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

Next iteration through the loop:

`i = 2;`

We check:

Is `i < team1Score.length`
(`team1Score.length` will be 4)?

Since `i` is 2, it is less than 4 and we get into the code block.

We print `team1Score[i]` and since `i = 2`, that is `team1Score[2]` and so we print:

18

Before we go to the next iteration, we increment `i`

Arrays: Iterating

Let's walk through and see what happens...

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

Next iteration through the loop:

`i = 3;`

We check:

Is `i < team1Score.length`
(`team1Score.length` will be 4)?

Since `i` is 3, it is less than 4 and we get into the code block.

We print `team1Score[i]` and since `i = 3`, that is `team1Score[3]` and so we print:

`23`

Before we go to the next iteration, we increment `i`

Arrays: Iterating

Let's walk through and see what happens...

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

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

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

for (int i = 0; i < team1Score.length; i++) {
     System.out.println(team1Score[i]);
}
```

Next iteration through the loop:

`i = 4;`

We check:

Is `i < team1Score.length`
(`team1Score.length` will be 4)?

Since `i` is 4, it is **NOT** less than 4 and the loop stops.

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 behind a variable it is a postfix operator (i.e. x++).
  - A variable with 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 variable with 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.