Unit 2 Tutorials: Arrays and Loops

INSIDE UNIT 2

Arrays

- Introduction to Arrays
- Manipulating Arrays
- Array Algorithms
- Java Generics
- Collection Types
- Multiple Dimensions
- Debugging Arrays and other Collection Types
- Tic-Tac-Toe Program

Loops

- Introduction to Loops
- Loops Using While & Do...While
- Loops Using For
- Loops Using Enhanced For (for each iteration)
- Nested Loops
- Debugging Loops
- Revisiting the Tic-Tac-Toe Program

Complex Methods

- Method Parameters and Arguments
- The Return Statement
- Overloading Methods
- Debugging Complex Methods
- Finishing the Tic-Tac-Toe Program

Introduction to Arrays

by Sophia

WHAT'S COVERED

In this lesson, you will learn about using a data structure called an array, to store and access multiple data values of the same type. You will also learn array access. Specifically, this lesson covers:

1. Arrays

When working with code, there are times that you will be working with one element at a time. Using a name or a single price are examples of the use of an array. There are other instances where there is a need to work with larger sets of data. An **array** is a special type of variable that represents multiple values that are all of the same data type. Most programming languages have a structure that allows users to store multiple values of the same type, in addition to storing single values. To better understand the code used when working with arrays, it's important to understand what an array is, how it works, and why one would want to use an array in code.

CONCEPT TO KNOW

An **array** is a named collection of contiguous storage locations. These are storage locations that are next to each other. They contain data items of the same type. Each of the items (values) is called an **element**. The elements are arranged in a linear fashion. Each element has an **index**, which is also called a subscript. It is an integer that specifies its location in the array. The first item in the array has the index 0, and the index of the last element is one less than the length of the array.

Java arrays can be single or multi-dimensional. The first part of this tutorial will focus on single-dimensional arrays. Multi-dimensional arrays will be covered later in this tutorial. When declaring an array, the declaration begins with the data type. When declaring an array, the data type is immediately followed by a pair of square brackets.

The type and square brackets are followed by the name (identifier), which is similar to how plain variables are declared:

EXAMPLE

```
int[] scores;
```

The statement above declares an array of int values but does not provide a size. The array has to have its size set before data can be read from or written to it. Specifying the initial values for the array is one way to set the size.

The initial values are specified in a comma-separated list in curly brackets:

EXAMPLE

```
int[] scores = {100, 99, 98, 97, 96};
```

This array can be pictured like this:

Scores					
Index	0	1	2	3	4
Value (int)	100	99	98	97	96

An array of five String values can be declared like this:

EXAMPLE

```
String cheeseChoice = {"Cheddar", "Swiss", "Gouda", "American", "Mozzarella"};
```

This array can be illustrated like this:

cheeseChoice					
Index	0	1	2	3	4
Value (String)	"Cheddar"	"Swiss"	"Gouda"	"American"	"Mozzarella"

Arrays can also be declared without specifying the actual values. To declare an empty array with a certain size, use the new keyword followed by the data type and size.

This is done in square brackets as demonstrated in this code:

EXAMPLE

```
int[] scores = new int[5];
```

An empty array that can hold five String values can be declared like this:

EXAMPLE

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

BIG IDEA

When working with arrays, it is important to keep in mind that once the type and size of an array have been declared, these cannot be changed, as the type and size are immutable. We will discuss a workaround that allows copying the contents of an array to a larger array later in the challenge.

When experimenting by defining arrays in Java, you will find that it is possible to declare an array by placing the square brackets after the name of the array rather than after the data type for the array.

The following declaration works, but it is not the preferred approach:

EXAMPLE

```
int scores[] = new int[5];
```

This style of array declaration, with the square brackets after the name rather than the data type, is found in programming languages such as C and C++. While this syntax may have once made early versions of Java seem more familiar to C and C++ programmers, it is no longer recommended. Standard practice in Java places the square brackets after the data type, as shown earlier in this section.

THINK ABOUT IT

Why do we use arrays? Think of a real-world example of an array as a parking lot. Look at the entire parking lot as a whole and think of it as a single object. Inside of the parking lot there are parking spots that are uniquely identified by a number. There may be parking spots that have a car, a truck, a motorcycle, or may even be empty. Within a list, the first spot is identified as a 0. This means that if there are 10 elements in a list, the index values go from 0 to 9. If there are five spots, the index values of the spots go from 0 to 4.

Index values in a list look like this:

↪ EXAMPLE

```
String[] parkingLot = {"motorcycle", "", "truck", "car", "car"};
```

This code can be illustrated as in the following table:

parkingLot List					
Index (spot #)	0	1	2	3	4
Value	"motorcycle"	""	"truck"	"car"	"car"

TERMS TO KNOW

Array
An array is a named collection of contiguous storage locations—storage locations that are next to each other —that contain data items of the same type. An array also has a fixed size.

Element
An element is one of the values in an array.

Index
The index (also known as the subscript or position) is an integer value that indicates an element in an array.

2. Array Access

As discussed previously, array elements are indexed. This means that the first element in the array has an index of 0. Then, the second element has an index of 1. This structure continues until all elements are indexed. Each element in the array is ordered using this index.

Each element in the array will follow this syntax:

↪ EXAMPLE

```
array[x]
```

In this example below, the array is replaced with the name of the array that is to be accessed, and the x is replaced with the desired index (position) number. The first element is always 0 and not 1. This means that the elements in the array have a defined order and that order will not change unless we modify the order. You can assign array values using the name of the array and the assignment operator, or the = sign.

A value can also be assigned to an individual element in the array using the assignment operator (=) and the name of the array and the index, as seen below:

↪ EXAMPLE

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

After these statements, the numbers array contains the integer values of 10, 20, and 30.

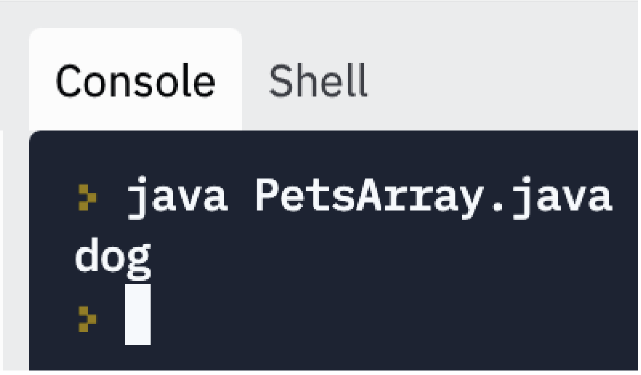
CONCEPT TO KNOW

Arrays are **mutable**. This means that the values can be changed in the elements. The syntax for accessing the elements of an array uses the bracket operator. The expression inside the brackets specifies the index of the element to be accessed. This index is the position number. Remember that the indices start at 0.

Here is an example of this expression:

```
class PetsArray {
  public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish"};
    System.out.println(pets[0]);
  }
}
```

Running this code produces this result:



```
› java PetsArray.java
dog
› 
```

TRY IT

Directions: Try it out yourself. Output the element in the array that has the value “fish”:

What would the index need to be set to? Click the plus (expand) icon on the right to see if you are correct. +

Were you correct? The index would need to be 2 to retrieve “fish” from the array.

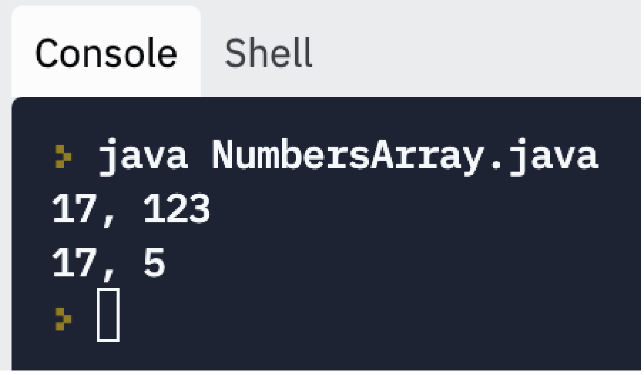
REFLECT

Arrays are mutable because the values of elements are changed. Or, they are assigned a new value to an element, as long as the data type is correct. When the bracket operator appears on the left side of the assignment operator, it identifies the element of the array to which the value will be assigned.

Here is an example of an array and the element to which a value will be assigned:

```
class NumbersArray {
  public static void main(String[] args) {
    int[] numbers = {17, 123};
    System.out.println(numbers[0] + ", " + numbers[1]);
    numbers[1] = 5;
    System.out.println(numbers[0] + ", " + numbers[1]);
  }
}
```

Here is what the output should look like:



```
> java NumbersArray.java
17, 123
17, 5
> 
```

THINK ABOUT IT

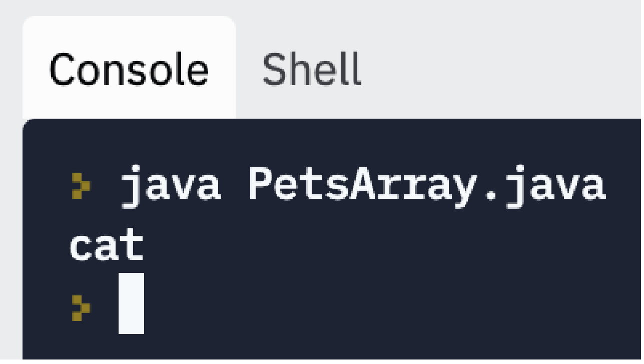
As this small program shows, the values of an array can have their values changed. The key points are to choose the correct index for the element to be updated and to make sure that the new value is of the correct data type. The element in index 1 (the second number) of the numbers array, which used to be 123, is now 5.

An array has a relationship between indices (index) and elements. This relationship is called a mapping. In this scenario, each index “maps to” one of the elements. Any integer expression can be used as an index.

Here is an example of an unusual (but still functional) integer expression:

```
class PetsArray {
  public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish"};
    System.out.println(pets[5-4]);
  }
}
```

The output should look like this:



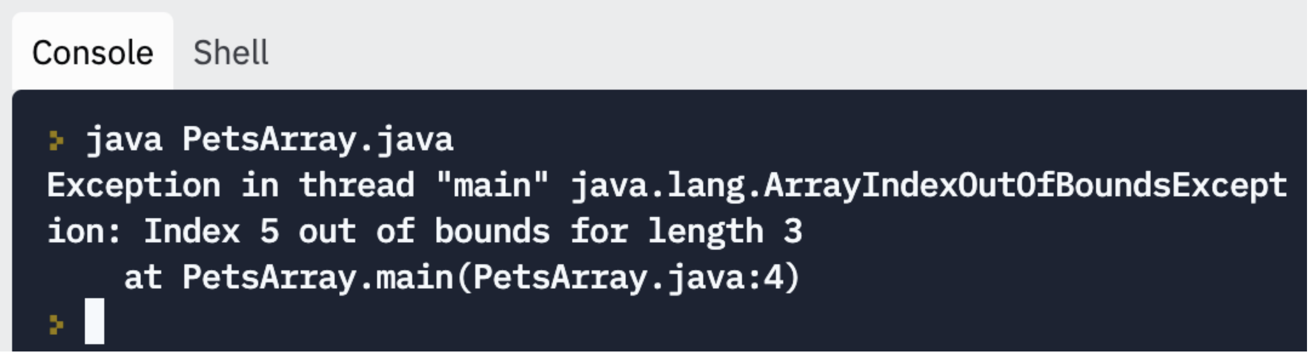
```
> java PetsArray.java
cat
> 
```

In the code above, 5 minus 4 is the same as if we had 1, so it would print out index 1, or cat.

Attempting to print an element that does not exist will cause anIndexError like this one:

```
class PetsArray {
  public static void main(String[] args) {
    String[] pets = {"dog", "cat", "fish"};
    System.out.println(pets[5]);
  }
}
```

The output should look like this:



```
> java PetsArray.java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsExcept
ion: Index 5 out of bounds for length 3
	at PetsArray.main(PetsArray.java:4)
> 
```

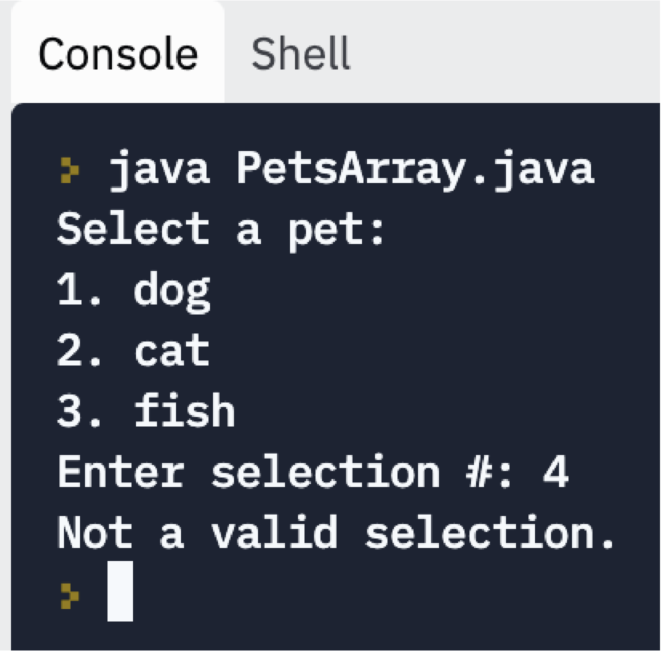
In the example above, the index values extend from 0 to 2, so index 5 produces an exception.

In a previous challenge, you learned how Java uses try and catch blocks to deal with exceptions. A try block could be used here when accessing the array.

A corresponding catch block can be used to handle the ArrayIndexOutOfBoundsException like the one below:

```
import java.util.Scanner;

class PetsArray {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String[] petList = {"dog", "cat", "fish"};
    System.out.println("Select a pet: ");
    System.out.println("1. " + petList[0]);
    System.out.println("2. " + petList[1]);
    System.out.println("3. " + petList[2]);
    System.out.print("Enter selection #: ");
    int choice = input.nextInt();
    // Subtract 1 from choice
    choice--;
    // try to access element in array
    try {
      System.out.println("You selected a " + petList[choice]);
    }
    catch(ArrayIndexOutOfBoundsException ex) {
      System.out.println("Not a valid selection.");
    }
  }
}
```

Running this program and entering an out-of-range exception looks like this:



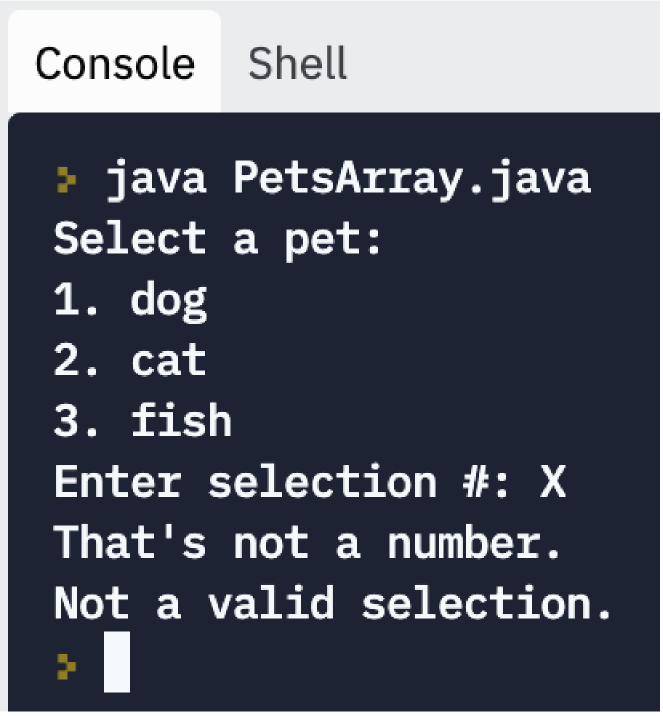
```
> java PetsArray.java
Select a pet:
1. dog
2. cat
3. fish
Enter selection #: 4
Not a valid selection.
> 
```

As discussed in the previous tutorial about handling errors, when using a Scanner to read input, users can use try and catch to react to invalid input. An example of this situation includes incidents where the entry isn't a number.

The full program then looks like this:

```
import java.util.Scanner;

class PetsArray {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String[] petList = {"dog", "cat", "fish"};
    System.out.println("Select a pet: ");
    System.out.println("1. " + petList[0]);
    System.out.println("2. " + petList[1]);
    System.out.println("3. " + petList[2]);
    System.out.print("Enter selection #: ");
    // Declare the variable choice before try block so it can
    // be used later in the program.
    int choice = 0;
    try {
      choice = input.nextInt();
    }
    catch(Exception ex) { // If user doesn't enter a number
      System.out.println("That's not a number.");
    }
    // Subtract 1 from choice
    choice--;
    // try to access element in array
    try {
      System.out.println("You selected a " + petList[choice]);
    }
    catch(ArrayIndexOutOfBoundsException ex) { // Number out of range
      System.out.println("Not a valid selection.");
    }
  }
}
```

If this program is run and a letter is entered rather than a number, the program produces this result:



```
> java PetsArray.java
Select a pet:
1. dog
2. cat
3. fish
Enter selection #: X
That's not a number.
Not a valid selection.
>
```

TERM TO KNOW

Mutable

Mutable means that we’re able to change the value of items in an array.

SUMMARY

In this lesson, you learned about **arrays**. You discovered that an array is the simplest data structure in

Java that can store multiple values. These values are called elements in a single variable. You also learned how to create an array, and that **accessing an array** element can be done using an index; remember that array indexes start at 0 for the first element. Finally, you learned that arrays are mutable and that you can change the order of elements in an array or reassign an element in an array.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from "Python for Everybody" By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Array

An array is a named collection of contiguous storage locations—storage locations that are next to each other—that contain data items of the same type. An array also has a fixed size.

Element

An element is one of the values in an array.

Index

The index (also known as the subscript or position) is an integer value that indicates an element in an array.

Mutable

Mutable means that we're able to change the value of items in an array.

Manipulating Arrays

by Sophia

WHAT'S COVERED

In this lesson, you will learn about various functions and methods that can be used to manipulate lists. You will also learn about the Arrays.toString() and Arrays.sort() methods. Specifically, this lesson covers:

1. Operators With Arrays

Values of single elements can be changed using the assignment operator (=).

1a. Assignment Operator (=)

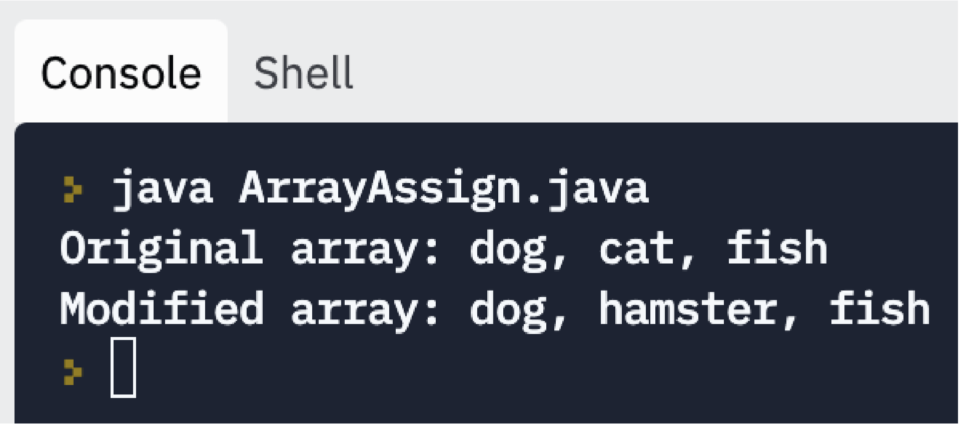
Recall, the = operator (equal sign) is an **assignment operator** that is used to assign values to variables. Values can be assigned to an array as easily as it was to assign a value to a variable. Individual elements can be replaced in the array using the same operator as well. You must ensure to include the index number in square brackets. This will indicate which element is to be changed. Once an array has been declared, it is not possible to replace the whole array using only literal values in curly brackets.

TRY IT

Directions: Enter the following code and run it to see the element replacement.

```
class ArrayAssign {
  public static void main(String[] args) {
    String[] petList = {"dog", "cat", "fish"};
    System.out.print("Original array: ");
    System.out.println(petList[0] + ", " + petList[1] + ", " + petList[2]);
    petList[1] = "hamster";
    System.out.print("Modified array: ");
    System.out.println(petList[0] + ", " + petList[1] + ", " + petList[2]);
  }
}
```

The output should look like this:



```
> java ArrayAssign.java
Original array: dog, cat, fish
Modified array: dog, hamster, fish
> 
```

REFLECT

As this small program shows, the assignment operator (=) works with individual elements in an array just like it does with any variable. Don't forget that the single equal sign is the assignment operator, while the double equal sign (==) is used to compare two values for equality.

In the example above, the element in index 1 was changed to hamster. This was the second element. Remember, the index starts at 0. To switch out dog for hamster, you would have needed to use petsList[0] with the index 0.

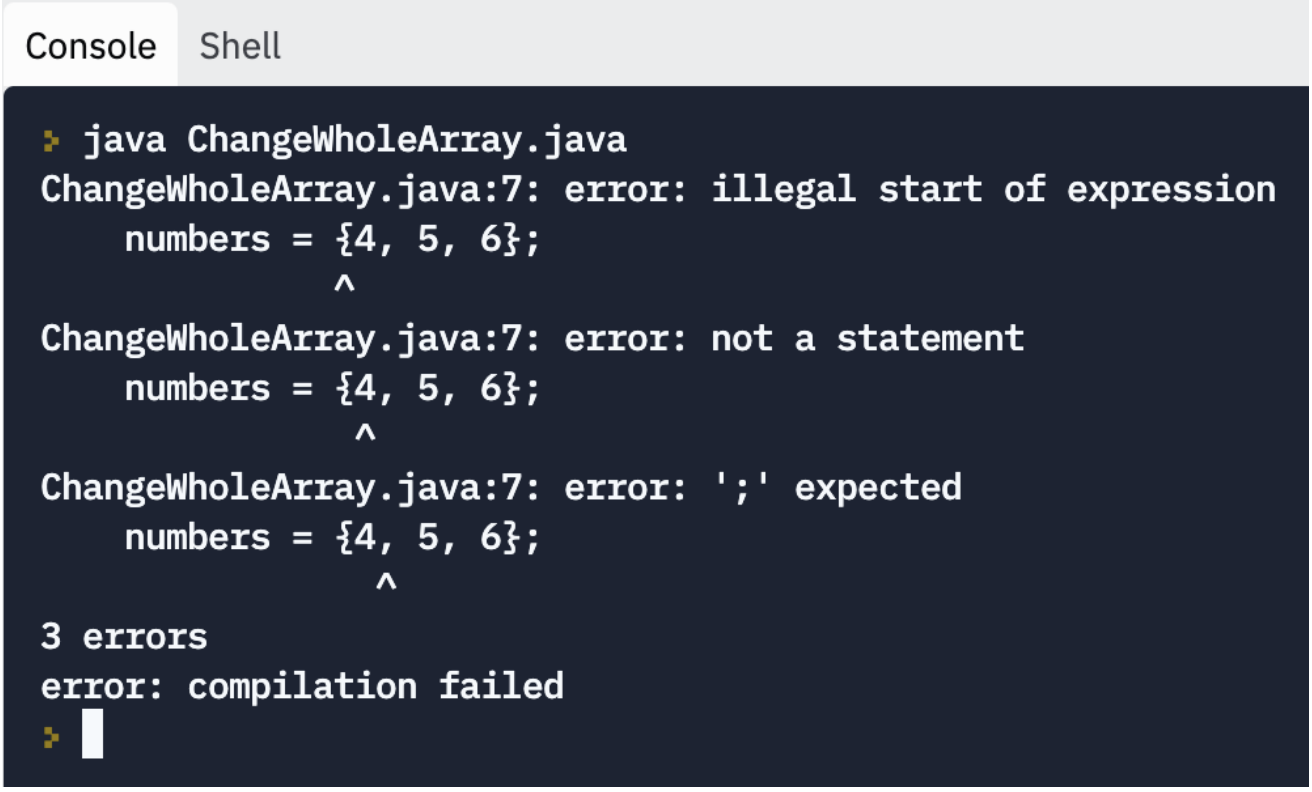
Next, consider changing the whole array using the assignment operator (=). As noted above, once an array has been declared, individual elements in the array can have their values changed using the single equal sign, but not the whole array. If you try to replace the whole array, it will generate an error.

TRY IT

Directions: Type the following code into Replit in a file named ChangeWholeArray.java:

```
import java.util.Arrays;

class ChangeWholeArray {
  public static void main(String[] args) {
    // Original array contents
    int[] numbers = {1, 2, 3};
    System.out.println("Array contents: " + Arrays.toString(numbers));
    // Try to change whole array
    numbers = {4, 5, 6};
    System.out.println("New array contents: " + Arrays.toString(numbers));
  }
}
```

Trying to run this code will produce a number of problems, as seen below:



```
> java ChangeWholeArray.java
ChangeWholeArray.java:7: error: illegal start of expression
    numbers = {4, 5, 6};
              ^
ChangeWholeArray.java:7: error: not a statement
    numbers = {4, 5, 6};
               ^
ChangeWholeArray.java:7: error: ';' expected
    numbers = {4, 5, 6};
                ^
3 errors
error: compilation failed
> 
```

REFLECT

The details of the error messages may seem confusing, but this small program makes clear that an entire array can't be modified just using an equal sign and appropriate values in curly brackets. Individual elements in an array can be changed, but not the whole array.

TRY IT

To change the contents of the array completely, use the new keyword to construct a whole new array object. Then, assign it using the single equal sign:

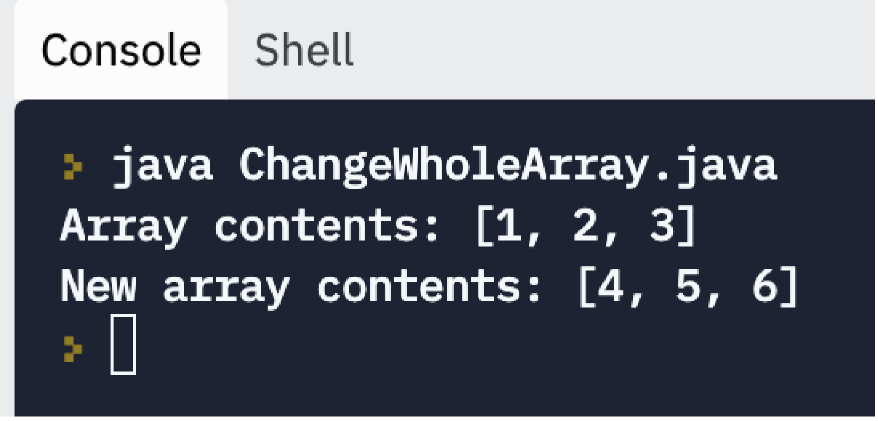
```
import java.util.Arrays;

class ChangeWholeArray {
  public static void main(String[] args) {
    // Original array
    int[] numbers = {1, 2, 3};
    System.out.println("Array contents: " + Arrays.toString(numbers));
    // Assign whole new array using new and =
    numbers = new int[]{4, 5, 6};
    System.out.println("New array contents: " + Arrays.toString(numbers));
  }
}
```

Directions: Now use the following line:

EXAMPLE

```
numbers = new int[]{4, 5, 6};
```

Doing so successfully changes the contents of the complete array to look like the output screen below:



```
> java ChangeWholeArray.java
Array contents: [1, 2, 3]
New array contents: [4, 5, 6]
>
```

When discussing variables earlier in the course, it was noted that the keyword final can be placed before the data type in a declaration. Doing this will result in a constant value. Unlike a variable, which can have its value changed, a **constant** declared with final cannot be changed from its initial value. The final keyword can be used when declaring a Java array, but it will not have the effect that one might expect. Since the individual elements in an array will always be mutable, using final will keep the whole array from being reassigned.

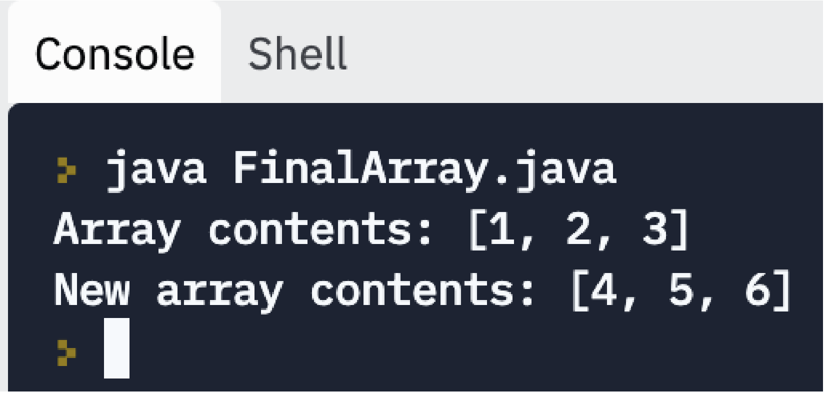
REFLECT

As we have seen, the assignment operator (=) by itself can't be used to change a whole array. Used in conjunction with the new keyword, though, the name of an existing array can be made to refer to a new array (unless the array has been declared final). We will learn more about the new keyword in the unit on Java classes.

Here is an example of how individual elements in an array declared as final can still be changed:

```
import java.util.Arrays;

class FinalArray {
  public static void main(String[] args) {
    // Original array - declared final (constant)
    final int[] numbers = {1, 2, 3};
    System.out.println("Array contents: " + Arrays.toString(numbers));
    // Assign new values to individual elements
    numbers[0] = 4;
    numbers[1] = 5;
    numbers[2] = 6;
    System.out.println("New array contents: " + Arrays.toString(numbers));
  }
}
```

The results will show that the individual array elements have been changed, as seen below (where they were changed separately):



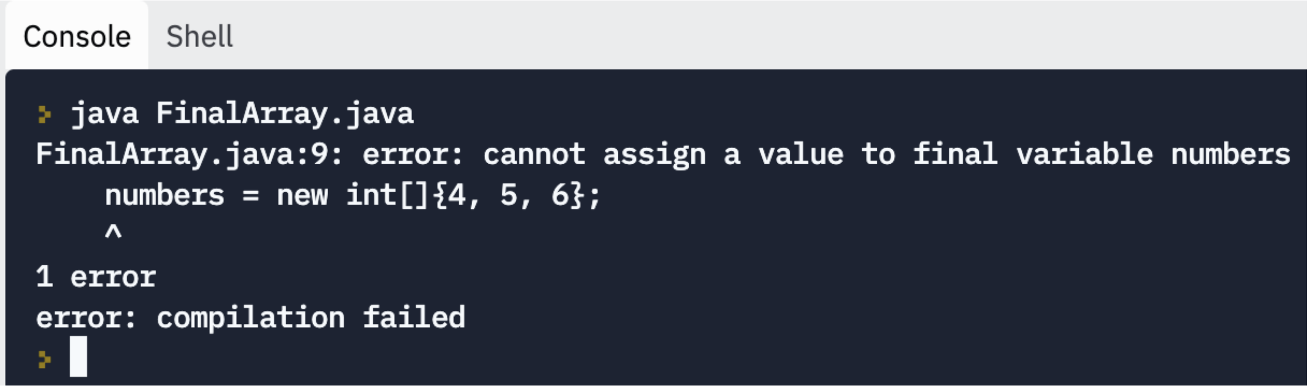
```
> java FinalArray.java
Array contents: [1, 2, 3]
New array contents: [4, 5, 6]
>
```

TRY IT

Directions: Now, try to run the following code:

```
import java.util.Arrays;

class FinalArray {
  public static void main(String[] args) {
    // Original array - declared final (constant)
    final int[] numbers = {1, 2, 3};
    System.out.println("Array contents: " + Arrays.toString(numbers));
    // Assign new array
    numbers = new int[]{4, 5, 6};
    System.out.println("New array contents: " + Arrays.toString(numbers));
  }
}
```

Java attempts to assign a whole new array using = and new. However, this approach will not work with a final array. It will generate an error message, which indicates that the array is immutable because it was declared final, as seen below:



```
> java FinalArray.java
FinalArray.java:9: error: cannot assign a value to final variable numbers
    numbers = new int[]{4, 5, 6};
    ^
1 error
error: compilation failed
>
```

REFLECT

As the result produced by this code shows, it is important to keep in mind that elements in a Java array can always be modified (at least individually), even if the array has been declared final.

When an array is declared final, the array itself (the whole array as an object) is immutable. However, the individual elements in an array are always mutable.

TERMS TO KNOW

Assignment Operator
The assignment operator is a single equal sign (=) that is used to assign a value on the right side of the equal sign to an array element (or other variable) on the left of the equal sign.

Constant
A named value of a specific data type stored in memory that cannot be changed after it has been given an initial value.

2. Methods

Java includes a number of useful methods for working with arrays, in the Arrays utility class. In this section, we will discuss two of these methods. They include Arrays.toString() and Arrays.sort().

CONCEPT TO KNOW

Keep in mind that when using these methods, that the Arrays class must be specified. Also consider that the name of the array to be converted to a String value or sorted is passed as an argument inside the parentheses.

2a. Arrays.toString() Method

The **Arrays.toString()** method is a useful utility for displaying the contents of an array.

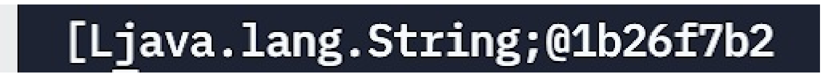
TRY IT

Directions: Try to print an array just using the name of the array (this leads to an odd result):

↪ EXAMPLE

```
System.out.println(petList);
```

Doing so produces the unexpected result below:



```
[Ljava.lang.String;@1b26f7b2
```

This is internal code used by the JVM and not intended for human consumption. The Array.toString() method produces a representation of the array that is easier to read.

↪ EXAMPLE A call to the code below produces a readable list of items in the array:

```
System.out.println(Arrays.toString(petList));
```

↪ EXAMPLE Use of Arrays.toString() requires the following import statement near the top of the .java file:

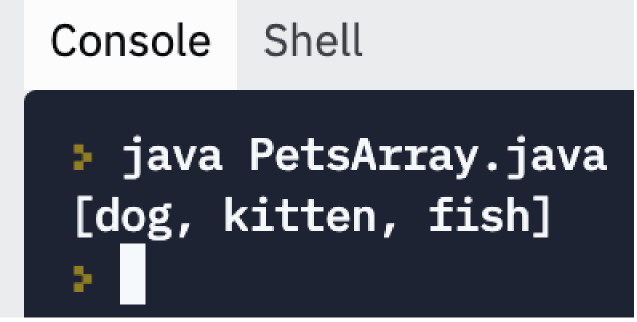
```
import java.util.Arrays;
```

TRY IT

Directions: Practice using Array.toString() in Replit, as follows:

```
import java.util.Arrays;

class PetsArray {
  public static void main(String[] args) {
    String[] petList = {"dog", "kitten", "fish"};
    petList[1] = "kitten";
    System.out.println(Arrays.toString(petList));
  }
}
```

Running this code appears to have worked much better:



```
> java PetsArray.java
[dog, kitten, fish]
>
```

REFLECT

Displaying the contents of an array to the user in square brackets might seem a bit unexpected in many cases, but Arrays.toString() is very valuable when developing and debugging code.

This method will be used often when discussing arrays in further tutorials.

2b. Arrays.sort() Method

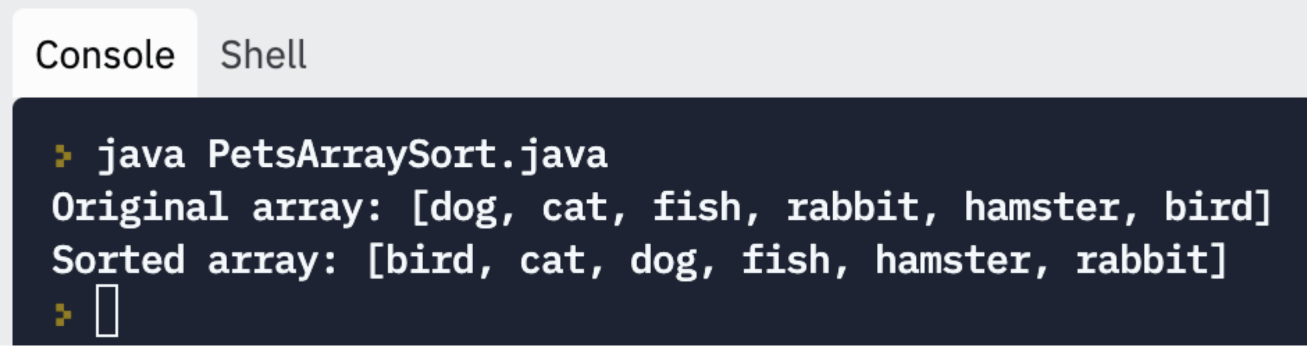
The **Arrays.sort() method** arranges all of the elements in the array from the lowest to highest. If the values are strings, the result will be a list of strings in alphabetical order.

TRY IT

Directions: Try using the .sort() method on the petsList:

```
import java.util.Arrays;

class PetsArraySort {
  public static void main(String[] args) {
    String[] petsArray = {"dog", "cat", "fish", "rabbit", "hamster", "bird"};
    System.out.println("Original array: " + Arrays.toString(petsArray));
    Arrays.sort(petsArray);
    System.out.println("Sorted array: " + Arrays.toString(petsArray));
  }
}
```

The result should be as follows:



```
> java PetsArraySort.java
Original array: [dog, cat, fish, rabbit, hamster, bird]
Sorted array: [bird, cat, dog, fish, hamster, rabbit]
> 
```

REFLECT

The output above shows that the order of the contents of the array has been changed, but the contents of the overall array haven't changed. The Arrays.sort() method will prove to be helpful in a number of contexts.

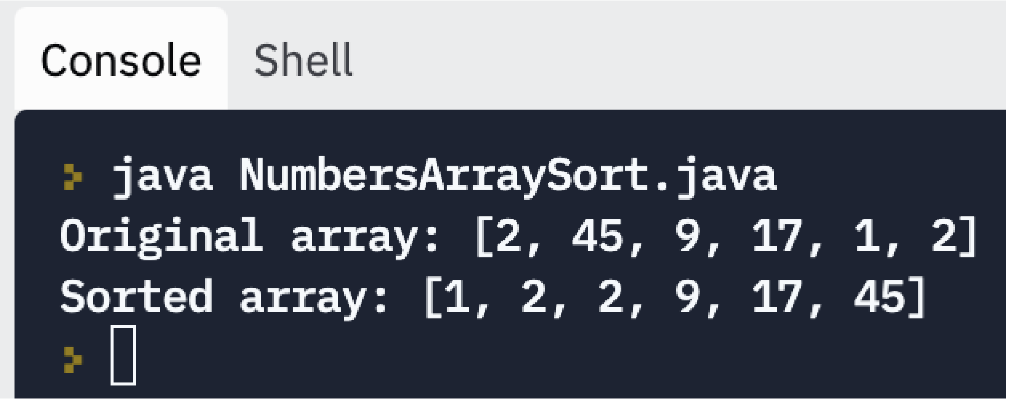
Bird to rabbit are all seen in alphabetical order. If the list contains numerical values, they are ordered from smallest to largest.

TRY IT

Directions: Try using the .sort() method again on a numerical list:

```
import java.util.Arrays;

class NumbersArraySort {
  public static void main(String[] args) {
    int[] numArray = {2, 45, 9, 17, 1, 2};
    System.out.println("Original array: " + Arrays.toString(numArray));
    Arrays.sort(numArray);
    System.out.println("Sorted array: " + Arrays.toString(numArray));
  }
}
```

The result should be as follows:



```
> java NumbersArraySort.java
Original array: [2, 45, 9, 17, 1, 2]
Sorted array: [1, 2, 2, 9, 17, 45]
> 
```

TERMS TO KNOW

Arrays.toString()

The Arrays.toString() method converts the contents of an array to String values that can be displayed on the screen.

Arrays.sort()
The Arrays.sort() method arranges all of the elements in an array from the lowest to highest. If the values are strings, the result will be a list of strings in alphabetical order.

SUMMARY

In this lesson, you learned about the **operators** and methods that can be used to manipulate arrays. Using these tools, we have learned various **methods and functions** that you can use to assign values to **arrays to string**, modify elements, print, and **sort arrays**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Arrays.sort()
The Arrays.sort() method arranges all of the elements in an array from the lowest to highest. If the values are strings, the result will be a list of strings in alphabetical order.

Arrays.toString()
The Arrays.toString() method converts the contents of an array to String values that can be displayed on the screen.

Assignment Operator
The assignment operator is a single equal sign (=) that is used to assign a value on the right side of the equal sign to an array element (or other variable) on the left of the equal sign.

Constant
A named value of a specific data type stored in memory that cannot be changed after it has been given an initial value.

Array Algorithms

by Sophia

WHAT'S COVERED

In this lesson, you will learn about how to process data using array algorithms in Java. This includes finding high and low values, search features, and copying code in arrays. Specifically, this lesson covers:

1. Finding the Highest and Lowest Values in an Array

In the previous lesson, you learned about the Arrays.sort() method. This method sorts the data in an array in numeric or alphabetic order. This functionality can be used to find the highest and lowest values in an array without looping through the array. While loops are used in this tutorial, they will be covered later in the course.

TRY IT

Directions: Start with an array of integers declared like this:

EXAMPLE

```
int[] scores = {77, 89, 100, 68, 95};
```

Using Arrays.sort() will then render the array in this order:

EXAMPLE

```
{68, 77, 89, 95, 100}
```

After the values have been sorted, you will see that the first value in the array is the lowest value. Keep in mind that the first element in the array has the index 0.

It can be seen as the lowest value and can be accessed like this:

EXAMPLE

```
int lowest = scores[0];
```

The last value in the array is the highest value in the array, after the data has been sorted. Calculating the last index is a bit more complex than finding the first index.

The **length (or size) of the array** can be determined when using the length properly, as seen below:

EXAMPLE

```
int arrayLength = scores.length;
```

Since the first index is 0, the highest index is always one less than the size of the array. It looks like this:

EXAMPLE

```
int highestIndex = scores.length - 1;
```

The highest value (in the last element in the array) can be accessed using highestIndex, which displays like this:

EXAMPLE

```
int highest = scores[highestIndex];
```

As you have learned, an expression can be used in the square brackets for the array index.

Therefore, the highest value in the array could also be accessed like this:

EXAMPLE

```
int highest = scores[scores.length - 1]
```

REFLECT

It is hard to overemphasize the point that the highest index in an array is always one less than the length of the array. Off-by-one errors are a common bug in code, so the programmer needs to be alert.

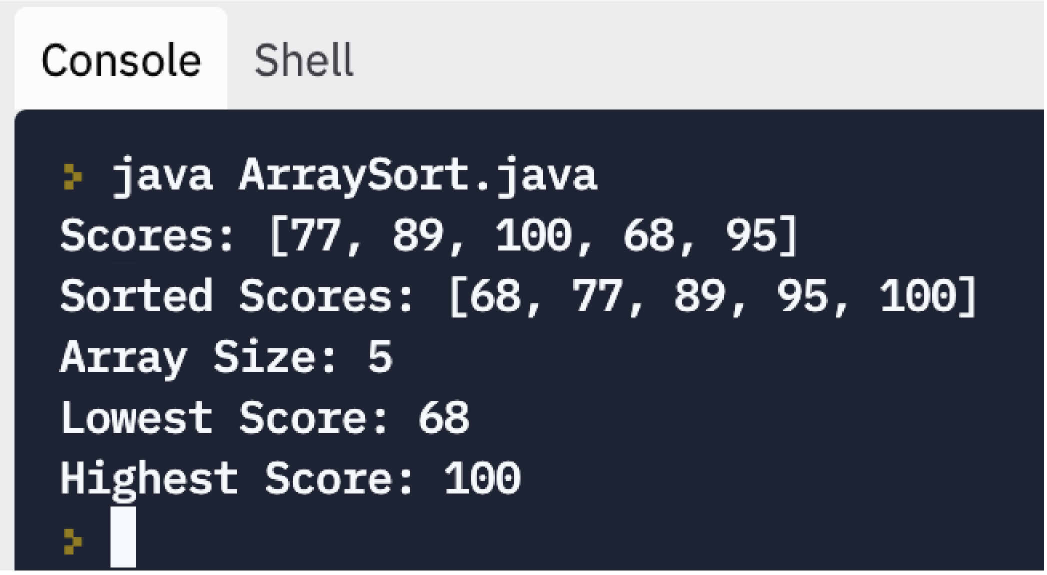
TRY IT

Directions: Try typing in the complete program in Replit to see the results:

```
import java.util.Arrays;

class ArraySort {
  public static void main(String[] args) {
    // Declare and initialize array with values
    int[] scores = {77, 89, 100, 68, 95};
    //System.out.println("Incorrect way to print: " + scores);
    System.out.println("Scores: " + Arrays.toString(scores));
    Arrays.sort(scores);
    System.out.println("Sorted Scores: " + Arrays.toString(scores));
    // Use the length property (or attribute) to get size of array
    int size = scores.length;
    System.out.println("Array Size: " + size);
    // Element 0 contains the lowest value
    System.out.println("Lowest Score: " + scores[0]);
    // Last index is 1 less than the size
    int lastIndex = size - 1;
    System.out.println("Highest Score: " + scores[lastIndex]);
  }
}
```

REFLECT

The output from running this program should look like this:



```
› java ArraySort.java
Scores: [77, 89, 100, 68, 95]
Sorted Scores: [68, 77, 89, 95, 100]
Array Size: 5
Lowest Score: 68
Highest Score: 100
›
```

TERM TO KNOW

Length (or Size) of an Array
The length or size of an array is the number of elements in the array. The .length property is used to get the length or size of an array.

---

2. Searching an Array

You have explored how to use Arrays.sort() to find the highest and lowest values in an array. The Arrays.sort() method also plays a role in searching an array. TheArrays utility class includes a method called **Arrays.binarySearch()**. This method returns the index of the sought value, or -1 if the value is not found in the array.

There is an important stipulation, though. The array being searched must be sorted first. If it is not sorted, the behavior and result of the search will be undefined and unreliable.

CONCEPT TO KNOW

In a small set of values like the example below, the binary search may seem to work, but it's important to remember that undefined behavior doesn't mean it won't work, just that it may fail—and perhaps at the worst possible moment.

After using Arrays.sort() to put the array elements in order, theArrays.binarySearch() method can be called like this (to search for the value 100):

↱ EXAMPLE

```
Arrays.sort(scores);
int location = Arrays.binarySearch(scores, 100);
```

HINT

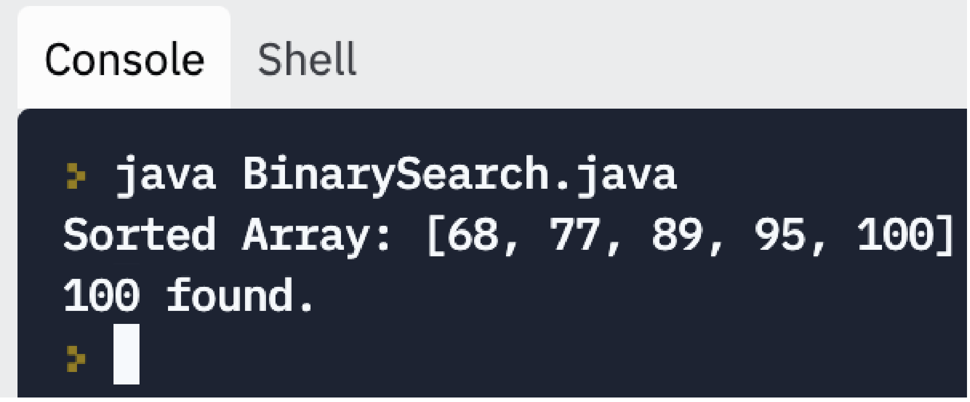
The data type of the value being searched for must be the same as the data type of the array.

TRY IT

Directions: Type this code into Replit to see how the process works:

```
import java.util.Arrays;

class BinarySearch {
  public static void main(String[] args) {
    int[] scores = {77, 89, 100, 68, 95};
    int searchValue = 100;
    Arrays.sort(scores);
    System.out.println("Sorted Array: " + Arrays.toString(scores));
    int location = Arrays.binarySearch(scores, searchValue);
    if(location > -1) {
      System.out.println(searchValue + " found.");
    }
    else {
      System.out.println(searchValue + " not found.");
    }
  }
}
```

The program's run should look like this:



```
> java BinarySearch.java
Sorted Array: [68, 77, 89, 95, 100]
100 found.
>
```

REFLECT

When using a binary search, it may seem that it works correctly when the array hasn't been sorted first, but it's important to remember that this is not a guarantee that it will work in a real-world production environment or with a larger array. Sometimes things may seem to be okay when tested in a small, simple program, but best practices are best practices, and all of the official documentation insists on sorting the array before using Arrays.binarySearch().

TERM TO KNOW

Arrays.binarySearch()
This method is used to search for the occurrence of a specified value in a sorted array.

3. Copying All or Part of an Array

The **Arrays.copyOf()** method allows users to copy an array from a source array and paste it to a destination array. The Arrays.copyOf() method uses two parameters.

These include the source array and the length or size of the copied array, as seen below:

EXAMPLE

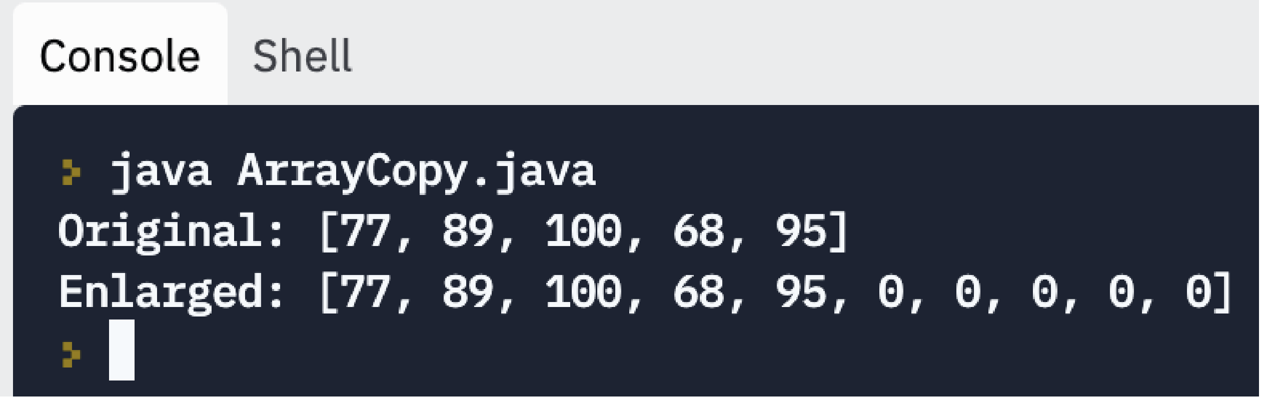
```
int[] scores = {77, 89, 100, 68, 95}; int[] scoresCopy = Arrays.copyOf(scores, 5);
```

Both arrays now contain the values {77, 89, 100, 68, 95}. If the value provided as the parameter for the size is larger than the original array, the new array will have the extra elements filled with 0, an empty char, or an empty String. Which one depends on the data type. This has the effect of allowing us to increase the size of an array, although it is not very efficient to do it this way.

Here is a sample program that increases the size of the scores array by copying a larger array onto the original:

```
import java.util.Arrays;

class ArrayCopy {
  public static void main(String[] args) {
    // Original array of 5 integers
    int[] scores = {77, 89, 100, 68, 95};
    System.out.println("Original: " + Arrays.toString(scores));
    // Copy larger array of 10 integers back to scores
    scores = Arrays.copyOf(scores, 10);
    System.out.println("Enlarged: " + Arrays.toString(scores));
  }
}
```

When run, this program produces the following output:



Console Shell

```
> java ArrayCopy.java
Original: [77, 89, 100, 68, 95]
Enlarged: [77, 89, 100, 68, 95, 0, 0, 0, 0, 0]
>
```

In addition to the Arrays.copyOf() method, there is also an **Arrays.copyOfRange()**. Arrays.copyOfRange() allows users to copy part of an array to a destination. This method is called with values passed for the source array, the starting index for the copy, and the ending index for the copy.

This method automatically handles the offset by 1:

↪ EXAMPLE

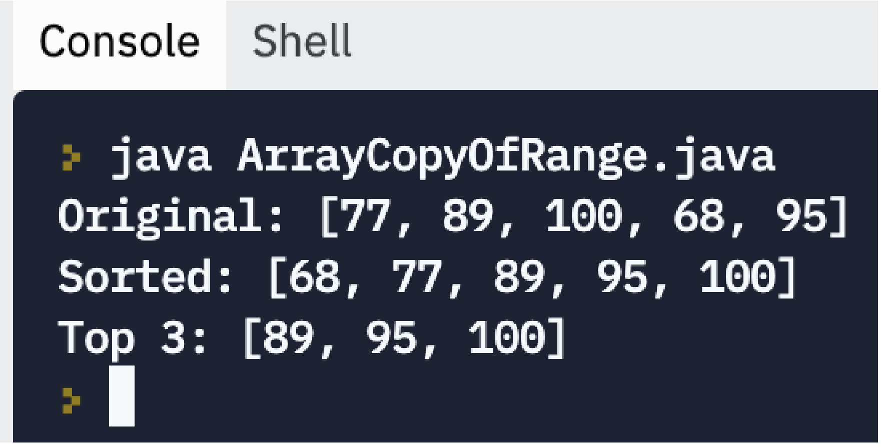
```
int[] scores = {77, 89, 100, 68, 95};
int[] lastThree = Arrays.copyOfRange(scores, scores.length - 3, scores.length);
```

The array lastThree will contain the values {100, 68, 95}.

If used with Arrays.sort(), Arrays.copyOfRange() allows for gathering the top or bottom values, as this example shows:

```
import java.util.Arrays;

class ArrayCopyOfRange {
  public static void main(String[] args) {
    // Original array of 5 integers
    int[] scores = {77, 89, 100, 68, 95};
    System.out.println("Original: " + Arrays.toString(scores));
    Arrays.sort(scores);
    System.out.println("Sorted: " + Arrays.toString(scores));
    // start is 3rd from end, runs until the end - 1
    int[] topThree = Arrays.copyOfRange(scores, scores.length - 3, scores.length);
    System.out.println("Top 3: " + Arrays.toString(topThree));
  }
}
```

The results from the program show the top 3 scores:



```
> java ArrayCopyOfRange.java
Original: [77, 89, 100, 68, 95]
Sorted: [68, 77, 89, 95, 100]
Top 3: [89, 95, 100]
> 
```

TERMS TO KNOW

Arrays.copyOf()
The Arrays.copyOf() method allows users to copy an array from a source array and paste it to a destination array. The Arrays.copyOf() method uses two parameters. These include the source array and the length or size of the copied array.

Arrays.copyOfRange()
This method allows copying part of an array to a destination. This method is called with values passed for the source array, the starting index for the copy, and the ending index for the copy.

SUMMARY

In this lesson, you have learned how to use a couple useful array processes. You have determined how to find the **highest and lowest values** in an array by using Arrays.sort(). You have also seen how Arrays.binarySearch() can be used to **search for a value in an array**, provided that the array has been sorted first. You have learned how to **copy all or part of an array**. Finally, you learned how methods provided by the Arrays class can help enlarge an array to assist in finding the top set of values.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Arrays.binarySearch()
This method is used to search for the occurrence of a specified value in a sorted array.

Arrays.copyOf()
The Arrays.copyOf() method allows users to copy an array from a source array and paste it to a destination array. The Arrays.copyOf() method uses two parameters. These include the source array and the length or size of the copied array.

Arrays.copyOfRange()
This method allows copying part of an array to a destination. This method is called with values passed

for the source array, the starting index for the copy, and the ending index for the copy.

Length (or Size) of an Array

The length or size of an array is the number of elements in the array. The .length property is used to get the length or size of an array.

Java Generics

by Sophia

WHAT'S COVERED

In this lesson, you will learn about how generic types and methods are used in Java. Specifically, this lesson covers:

1. Generic Types

As discussed in previous tutorials, the data type for the elements in the array has to be part of the declaration, when declaring an array. Additionally, all of the elements in the array must be of the same data type. This lesson serves as a bridge between these topics and introduces generic types in Java. Though certain Java collections that are a more flexible alternative to traditional arrays will be seen in this tutorial, they will be covered in more detail in a future tutorial.

From time to time, you may have a need to use code for various types of data. **Generic programming** allows programmers to use the same code for different types of data. This alleviates the need to write different versions of the same code when working with different data types. **Generics** are another example and can be used with Java classes, which will be covered in a later tutorial. For certain situations, generics can also be used with methods.

A generic method can work with data of any type. This will be called out in an example in the next section of the tutorial. A generic method returns the value at the midpoint in an array.

CONCEPT TO KNOW

Generics only work with objects (such as String), not primitive data types (such as int, char, double, etc...).

Fortunately, to address this, there are classes that wrap primitive data types so that they behave like objects.

Here is a list of the primitive types we have worked with so far and the corresponding wrapper classes:

Primitive Type	Wrapper Type	Description
boolean	Boolean	Stores true and false values
char	Character	Stores single letters, digits, and symbols
double	Double	Stores fractional numbers up to 15 decimal digits
float	Float	Stores fractional numbers up to 7 decimal digits
int	Integer	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	Long	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

THINK ABOUT IT

Note how the wrapper types, in the table above, all begin with capital letters. Remember that class names in Java always begin with capital letters. This identifies the wrapper types as classes.

TERM TO KNOW

Generic Programming (also Generic Types or Generics)
A type of programming that allows programmers to use the same code and structures for different data types without having to rewrite the code.

2. Generic Methods

Angle brackets (< >) are used when declaring a**generic method**. This designation indicates that the method is generic.

The variable in the angle brackets below, is a type variable that stands for a data type:

↪ EXAMPLE

(in the code below, <T>)

HINT

Note how T is then used to specify that the return data type from the method and the data type for the array are the same. Both include whatever T ends up being when the method is called.

When calling a generic method, the call looks like any other method call. The type of data is not specified in angle brackets. The data type is worked out by the compiler.

In the code below, note the following lines:

↪ EXAMPLE

```
String midPointName = getMidPointItem(names);
```

and

↪ EXAMPLE

```
char midPointLetter = getMidPointItem(letters);
```

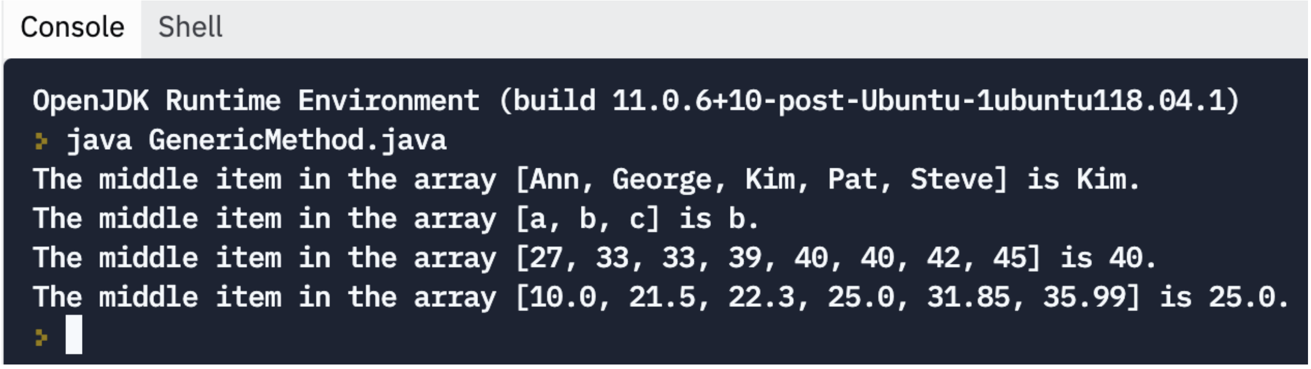
Among others that show how the generic `getMidPointItem()` method is called, as seen below:

```
import java.util.Arrays;
```

```
public class GenericMethod {

    // The angle brackets <> indicate a generic method.
    // T is a type variable that stands for the data type. The method
    // returns a value of the same data type as the data type for the array.
    public static <T> T getMidPointItem(T[] array) {
        // division of an int by an int results in an int quotient - no decimal.
        // Returns array element at index length / 2.
        return array[array.length / 2];
    }

    public static void main(String[] args) {
        String[] names = { "Ann", "George", "Kim", "Pat", "Steve"};
        String midPointName = getMidPointItem(names);
        System.out.print("The middle item in the array " + Arrays.toString(names));
        System.out.println(" is " + midPointName + ".");

        // Instead of primitive type char, use Character - note capital C
        Character[] letters = {'a', 'b', 'c'};
        char midPointLetter = getMidPointItem(letters);
        System.out.print("The middle item in the array " + Arrays.toString(letters));
        System.out.println(" is " + midPointLetter + ".");

        // Instead of primitive type int use Integer - note capital I
        Integer[] agesInYears = {27, 33, 33, 39, 40, 40, 42, 45};
        int midPointAge = getMidPointItem(agesInYears);
        System.out.print("The middle item in the array " +
          Arrays.toString(agesInYears));
        System.out.println(" is " + midPointAge + ".");

        // Instead of primitive type double use Double - note capital D
        Double[] temperatures = {10.0, 21.5, 22.3, 25.0, 31.85, 35.99};
        double midPointTemp = getMidPointItem(temperatures);
        System.out.print("The middle item in the array " +
          Arrays.toString(temperatures));
        System.out.println(" is " + midPointTemp + ".");
    }
}
```

TRY IT

Directions: Type in the code above in Replit using a file named GenericMethod.java. The output when the program is run should look like this:



```
OpenJDK Runtime Environment (build 11.0.6+10-post-Ubuntu-1ubuntu118.04.1)
> java GenericMethod.java
The middle item in the array [Ann, George, Kim, Pat, Steve] is Kim.
The middle item in the array [a, b, c] is b.
The middle item in the array [27, 33, 33, 39, 40, 40, 42, 45] is 40.
The middle item in the array [10.0, 21.5, 22.3, 25.0, 31.85, 35.99] is 25.0.
> 
```

REFLECT

Generic methods help programmers be more productive by allowing the writing of methods that can use the same code to work on a variety of data types without needing to write separate methods.

Generic types are used with Java collections and will be discussed in future tutorials.

TERM TO KNOW

Generic Method
A Java method written using generics that allow the same method to work on different data types.

SUMMARY

In this lesson, you learned about the uses of **generic types** in Java. You learned that **generics** are important because they allow the same code and data structures in Java to work with different data types. You learned that this approach helps to avoid repetitive code when a program needs to apply a similar process to different types of data. You learned that **generic methods** are written in Java using angle brackets. Finally, you learned that generic types will be important in future tutorials that include Java collection types.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Generic Method
A Java method written using generics that allow the same method to work on different data types.

Generic Programming (also Generic Types or Generics)
A type of programming that allows programmers to use the same code and structures for different data types without having to rewrite the code.

Collection Types

by Sophia

WHAT'S COVERED

In this lesson, you will learn about common collection types used in Java. These collection types are used for many purposes, which include tasks like data storage and retrieval. Specifically, this lesson covers:

1. Java Collections

Java **collections** are data structures, similar to arrays. They allow multiple values of the same data type to be stored in a container that is accessed by name. The main difference between an array and a collection is that the size of an array is fixed. However, collections can change size.

Java collections are generic classes that take type parameters in angle brackets, like generic methods, to indicate what type of data they contain. Primitive data types require the use of wrapper types, which are similar to those in generic methods.

Here is the list presented when we covered generic methods:

Primitive Type	Wrapper Type
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

When working with collections, keep in mind that the collection needs to be created before data can be added to the collection. Java features a number of different collection types. Commonly used collection types will be discussed in future tutorials.

TERM TO KNOW

Collection
A collection is a container that allows multiple values of the same data type to be stored. The main difference between an array and a collection is that the size of an array is fixed, but collections can change size.

2. ArrayList

As the name indicates, an **ArrayList** is a flexible list collection. It is similar to an array. Like an array, an ArrayList can contain duplicate values. The only constraint is that the values must all be of the same type (as with a plain array).

To use an ArrayList collection in a program, include the following import statement near the top of the file.

↪ EXAMPLE

```
import java.util.ArrayList;
```

When using an ArrayList, the collection needs to be created first and then values added to it.

To create an ArrayList, use statement like this:

↪ EXAMPLE

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

This constructs an ArrayList that will hold Integer (int) values.

As with a generic method, the data type is specified in angle brackets:

↪ EXAMPLE

```
ArrayList<Integer> and new ArrayList<Integer>();
```

CONCEPT TO KNOW

Since the <Integer> on the left side of the equal sign clearly indicates the data type, current versions of Java allow leaving out the type on the right side of the equal sign.

This statement does the same thing:

↪ EXAMPLE

```
ArrayList<Integer> list = new ArrayList<>();
```

To put values into the ArrayList, use the add() method like this:

↪ EXAMPLE

```
list.add(100);
list.add(99);
list.add(98);
```

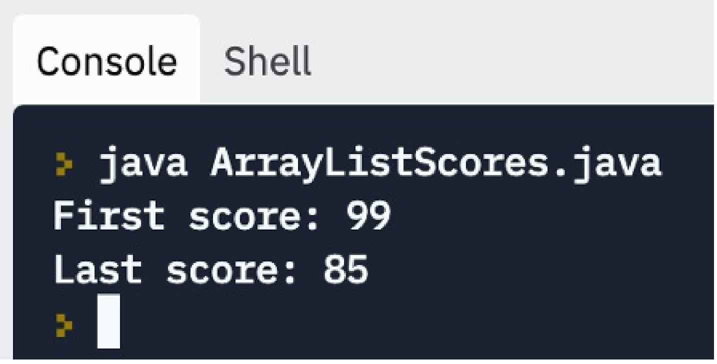
To access a value in an ArrayList, use the get() method. The get() method takes one parameter. This parameter includes the index of the desired list item. The first item in the list has the index 0, which is consistent with array indices. The last item in the list has an index that is one less than the size of the ArrayList. With the number of elements in the array provided by the length property, when working with an ArrayList, the size() method is used to get the number of items in the ArrayList.

TRY IT

Directions: To see how an ArrayList works, type the following code in Replit in a file named ArrayListScores.java:

```
import java.util.ArrayList;

class ArrayListScores {
  public static void main(String[] args) {
    // Construct an ArrayList named scores to hold Integer values
    ArrayList<Integer> scores = new ArrayList<>();
    // Add some scores
    scores.add(99);
    scores.add(88);
    scores.add(100);
    scores.add(85);
    System.out.println("First score: " + scores.get(0));
    int listLength = scores.size();
    System.out.println("Last score: " + scores.get(listLength - 1));
  }
}
```

The program you ran should produce this output:



```
> java ArrayListScores.java
First score: 99
Last score: 85
>
```

REFLECT

An ArrayList collection provides a more flexible kind of array, since the size of the collection is not fixed and can grow as items are added at runtime. Additional items automatically appear at the end of the list even if there are repeating items.

There are a number of other similarities between arrays andArrayLists. To print out the values in an ArrayList, you would use a toString() method.

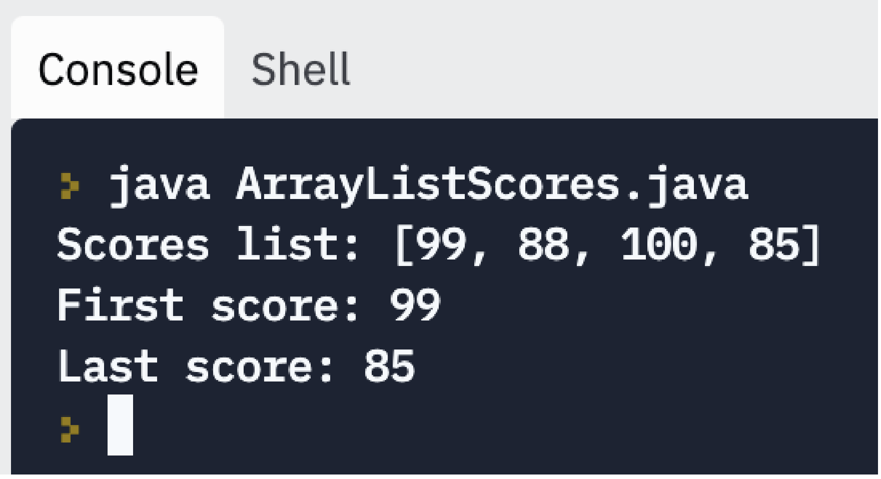
CONCEPT TO KNOW

Though it is part of the ArrayList class itself, it is not part of the Collections utilities.

The program demonstrates this:

```
import java.util.ArrayList;

class ArrayListScores {
  public static void main(String[] args) {
    // Construct an ArrayList named scores to hold Integer values
    ArrayList<Integer> scores = new ArrayList<>();
    // Add some scores
    scores.add(99);
    scores.add(88);
    scores.add(100);
    scores.add(85);
    System.out.println("Scores list: " + scores.toString());
    System.out.println("First score: " + scores.get(0));
    int listLength = scores.size();
    System.out.println("Last score: " + scores.get(listLength - 1));
  }
}
```

The results should look like this:



```
> java ArrayListScores.java
Scores list: [99, 88, 100, 85]
First score: 99
Last score: 85
>
```

Similar to the way an array can be sorted using the Arrays.sort() method, there is a corresponding Collections.sort() method that can be called like this:

↱ EXAMPLE

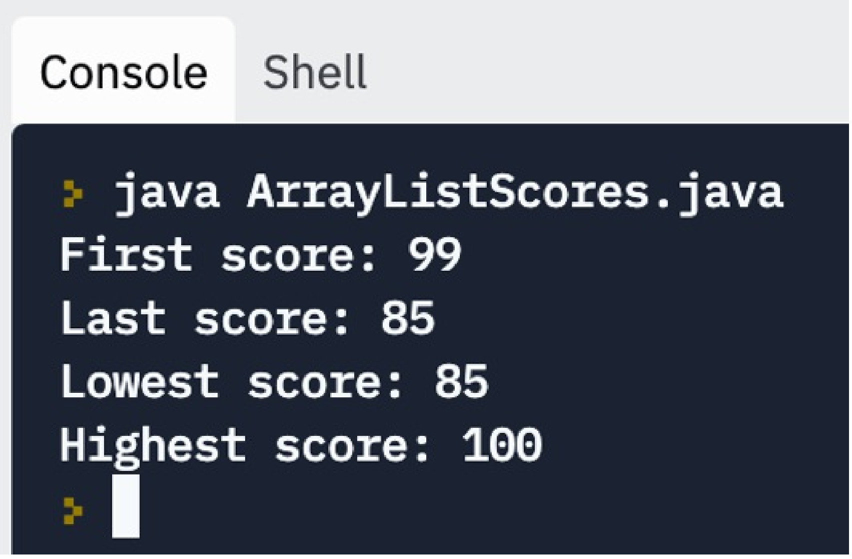
```
Collections.sort(scores);
```

The sort is done using the "natural order," which is similar to an array. It also uses the numeric order for numeric data types and alphabetic order for character and string data.

See how to use a sorted list to get the lowest and highest values in the ArrayList below:

```
import java.util.ArrayList;
import java.util.Collections;

class ArrayListScores {
  public static void main(String[] args) {
    // Construct an ArrayList named scores to hold Integer values
    ArrayList<Integer> scores = new ArrayList<>();
    // Add some scores
    scores.add(99);
    scores.add(88);
    scores.add(100);
    scores.add(85);
    System.out.println("First score: " + scores.get(0));
    int listLength = scores.size();
    System.out.println("Last score: " + scores.get(listLength - 1));
    Collections.sort(scores);
    System.out.println("Lowest score: " + scores.get(0));
    System.out.println("Highest score: " + scores.get(listLength - 1));
  }
}
```

Running this code will produce these results:



```
> java ArrayListScores.java
First score: 99
Last score: 85
Lowest score: 85
Highest score: 100
> 
```

TERM TO KNOW

ArrayList

An ArrayList is a flexible list collection that has similarities to an array, except that its size is not fixed.

3. HashSet

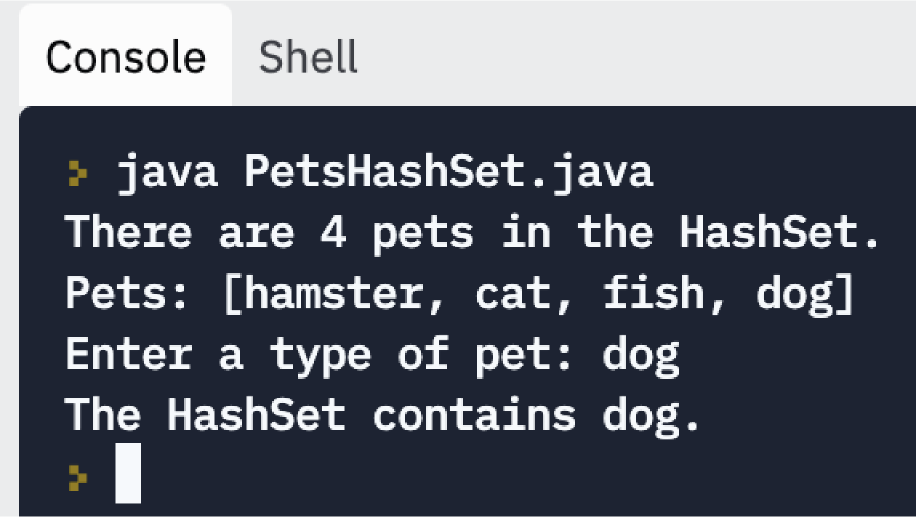
A **HashSet** is a Java collection that stores unique values. A HashSet never contains duplicate values. Adding a duplicate value to a HashSet has no effect. After a HashSet has been declared, values of the appropriate type are added using the add() method. There is also a remove() method that removes the value passed as a parameter.

The following code shows the use of the size() method to get the number of items in the HashSet. The program also demonstrates the use of the HashSet's contains() method.

This program will return a boolean value, or a true if the value passed in is present; otherwise, it will return false.

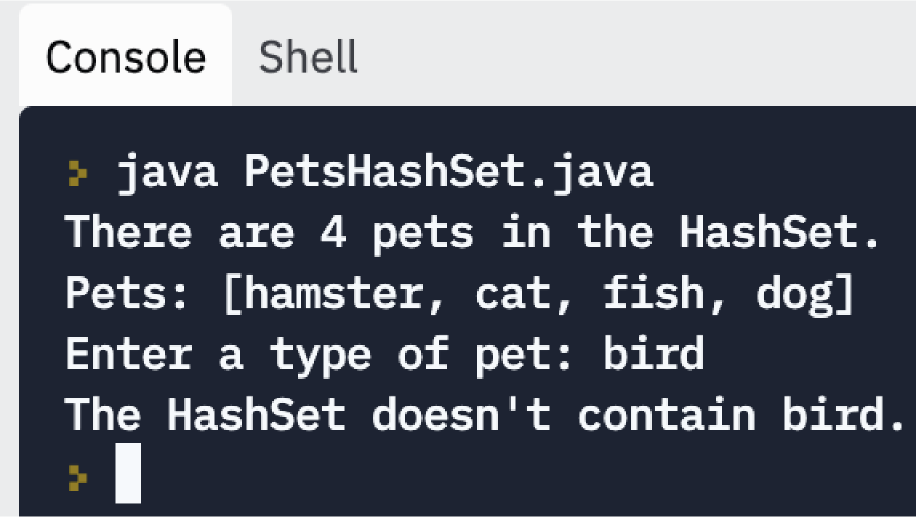
```
import java.util.HashSet;
import java.util.Scanner;

public class PetsHashSet {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    HashSet<String> petSet = new HashSet<>();
    petSet.add("hamster");
    petSet.add("cat");
    petSet.add("fish");
    petSet.add("dog");
    petSet.add("dog"); // duplicate accepted but not kept
    System.out.println("There are " + petSet.size() + " pets in the HashSet.");
    System.out.println("Pets: " + petSet.toString());

    System.out.print("Enter a type of pet: ");
    String pet = input.nextLine();
    // convert entry to lowercase
    pet = pet.toLowerCase();
    // contains() returns true if value is present in set, otherwise false
    if(petSet.contains(pet)) {
      System.out.println("The HashSet contains " + pet + ".");
    }
    else {
      System.out.println("The HashSet doesn't contain " + pet + ".");
    }
  }
}
```

The output for this program should look like the following output screens:



```
> java PetsHashSet.java
There are 4 pets in the HashSet.
Pets: [hamster, cat, fish, dog]
Enter a type of pet: dog
The HashSet contains dog.
> 
```

and



```
> java PetsHashSet.java
There are 4 pets in the HashSet.
Pets: [hamster, cat, fish, dog]
Enter a type of pet: bird
The HashSet doesn't contain bird.
> 
```

CONCEPT TO KNOW

It is important to note that a HashSet does not provide any mechanism to access individual items in the set. It can only determine if a HashSet contains a given value.

TERM TO KNOW

HashSet
A HashSet is a Java collection that stores unique values (i.e., there are no duplicate values).

---

4. HashMap

Java includes many collection types, but the last specific type that will be considered in this tutorial is the **HashMap**. Each element in a HashMap contains two values. These include the key value that identifies a unique element. The second is the value that is associated with the key. Since each element contains two values and two data types, they need to be specified when declaring a HashMap. The first identifies the key and the second identifies the value.

The code below declares the HashMap like this:

↪ EXAMPLE

```
HashMap<String, Integer> scores = new HashMap<>();
```

In this case, the key is a student or user ID that is a string value and the value is an integer value representing the score for the associated ID.

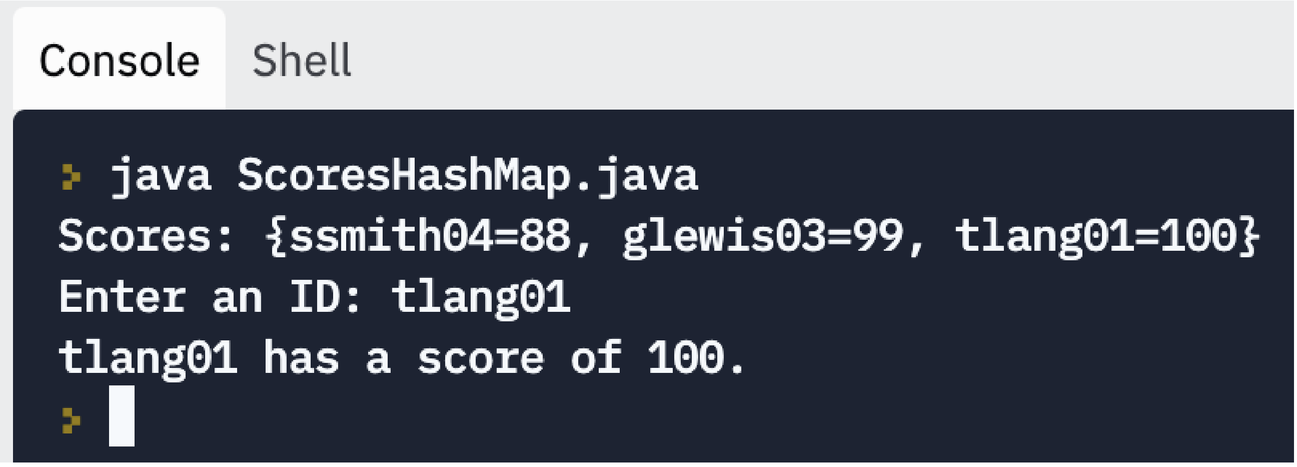
After a HashMap has been declared, the key-value pairs are added using theput() method. To access an item in the HashMap, the get() method is used. This method takes a parameter for passing in the key value being sought. The get() method returns null (no value) if the key is not present in theHashMap. The code below also shows the use of the contains() method, which returns true or false, and allows the programmer to check if a key is found in the HashMap.

TRY IT

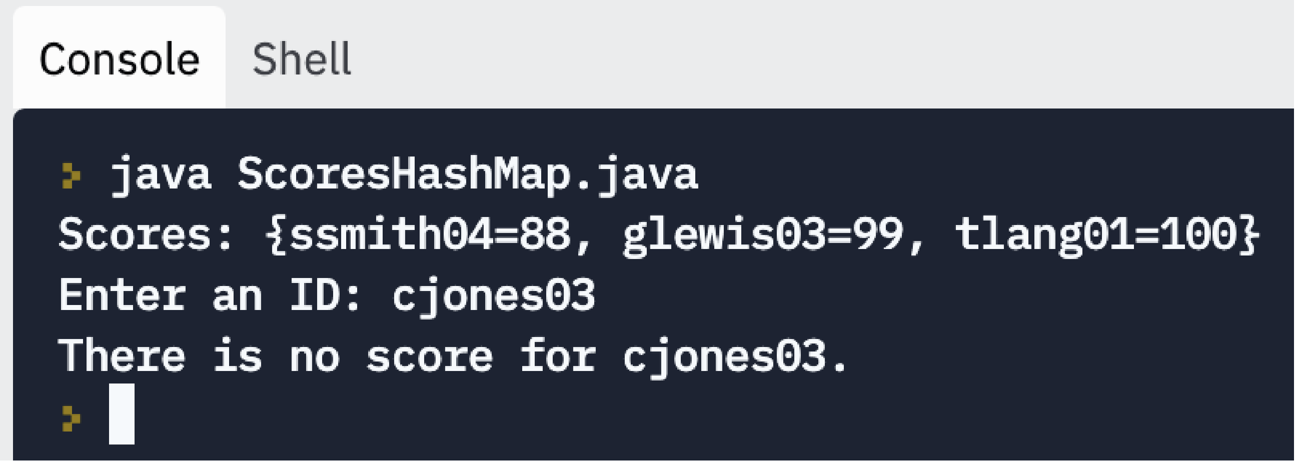
Directions: To see how a HashMap collection works, type in the following code in Replit in a file named ScoresHashMap.java:

```
import java.util.HashMap;
import java.util.Scanner;

public class ScoresHashMap {

  public static void main(String[] args) {
    // HashMap holds key-value pairs.
    // The key (user ID) is a String (case sensitive).
    // The value (score) is an Integer (int)
    HashMap<String, Integer> scores = new HashMap<>();
    scores.put("ssmith04", 88);
    scores.put("tlang01", 100);
    scores.put("glewis03", 99);
    System.out.println("Scores: " + scores.toString());

    Scanner input = new Scanner(System.in);

    System.out.print("Enter an ID: ");
    String id = input.nextLine();
    // Check if the HashMap contains the key (id)
    if(scores.containsKey(id)) {
      // Only safe to use get() to retrieve value if key exists in HashMap
      int score = scores.get(id);
      System.out.println(id + " has a score of " + score + ".");
    }
    else {
      System.out.println("There is no score for " + id + ".");
    }
  }
}
```

The output tables below are the return of a couple sample runs of the program:



```
> java ScoresHashMap.java
Scores: {ssmith04=88, glewis03=99, tlang01=100}
Enter an ID: tlang01
tlang01 has a score of 100.
>
```



```
> java ScoresHashMap.java
Scores: {ssmith04=88, glewis03=99, tlang01=100}
Enter an ID: cjones03
There is no score for cjones03.
>
```

REFLECT

This example has shown how to use a Java HashMap collection to store data that is organized into key-value pairs. As with any collection, a HashMap doesn't have a fixed size (though it cannot contain duplicate keys).

TERM TO KNOW

HashMap
A HashMap is a Java collection that stores key-value pairs. The key is a unique value that identifies the pair, and the value is the data associated with the key.

SUMMARY

In this lesson, you have learned about Java collection types. You learned that **Java collections** are an important data construct in Java, since they can contain a flexible range of data. You discovered that, unlike arrays, collections do not have fixed sizes. You also learned that there are a large number of collection types in Java that are useful for storing data in a wide range of programming constructs. Finally, you learned that a few of the most commonly used types are **ArrayList**, **HashSet**, and **HashMap**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

ArrayList
An ArrayList is a flexible list collection that has similarities to an array, except that its size is not fixed.

Collection
A collection is a container that allows multiple values of the same data type to be stored. The main difference between an array and a collection is that the size of an array is fixed, but collections can change size.

HashMap

A HashMap is a Java collection that stores key-value pairs. The key is a unique value that identifies the pair, and the value is the data associated with the key.

HashSet

A HashSet is a Java collection that stores unique values (i.e., there are no duplicate values).

Multiple Dimensions

by Sophia

WHAT'S COVERED

In this lesson, you will learn about multiple **dimension** arrays. A dimension of an array is an axis along which the elements are organized. The data in an array can be organized along 1, 2, or more axes or dimensions. Specifically, this lesson covers:

1. Multidimensional Arrays

Java multidimensional arrays are arranged as an array of arrays. The elements of multi-dimensional arrays are seen in rows and columns. Before discussing multiple dimensions in Java arrays, It is important to explore what multiple dimensions look like.

1a. One Dimensional Arrays

Recall what you learned about one dimension arrays in a previous tutorial. It resembles a list data collection type.

It appears as a straight line on a flat plane as demonstrated in this diagram:

PLACEHOLDER JAV 152 Alt Text: A double sided arrow.

A one dimensional listing would look like this on paper:

element value	"sun"	"fun"	run	bun	nun	pun
Index	0	1	2	3	4	5

The index position is one integer. To locate “run” from this listing you can use the single index position.

CONCEPT TO KNOW

Remember that list elements start at 0.

EXAMPLE

“run” is at index value 2.

TERM TO KNOW

Dimension

A dimension of an array is an axis along which the elements are organized.

1b. Two Dimensional Arrays

When nesting an array within an array you are, in a sense, adding a second dimension. A nested list is a two dimensional list; or 2D for short.

Here is what it looks like graphically:

Placeholder JAV 153 Alt Text: An X/Y chart.

You have likely seen this previously in an X/Y chart. It has two dimensions. One dimension is running diagonally and the other vertically; though they are still on a flat plane.

A two dimensional listing would look like this on paper:

X/Y indexes		X →		
		0	1	2
Y ↓	0	fun	sun	run
	1	bun	nun	pun
	2	fan	van	man
	3	can	tan	ran

There are two index positions for each element. To locate “fan” from this two dimensional listing identify both index positions, starting with the row then column.

EXAMPLE

“fan” is at position 2, 0 (Y,X or Row/Column

There can be more than one dimension when it comes to an array in Java. Consider how teachers store their grades for a class in a grade book. They have a list of students and within the list of students and they have a list of assignments. For each of those assignments, they will have a grade assigned.

The scores in Row 0 represent the scores for one student. This person's scores, for the different assignments, are seen in Column 0, Column 1, Column 2, and Column 3. The scores in Row 1 and Row 2 represent scores for other students. Since all of the values in an array have to be of the same data type, you can not mix names or letter grades in with the integer scores.

	Column 0	Column 1	Column 2	Column3
Row 0	100	92	99	85
Row 1	100	95	88	91
Row 2	99	100	100	100

Placeholder JAV 154 Alt Text: Four inner lists within an outer list making this a 2D list.

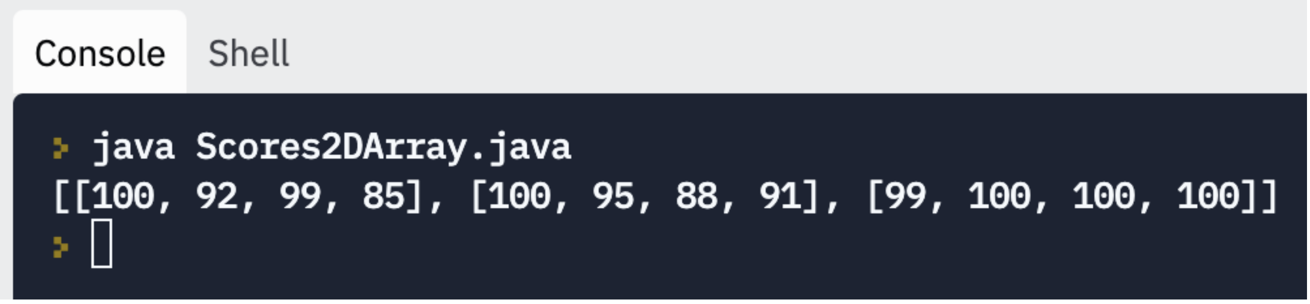
Let’s see this 2D array in Java code. To print out a 2-dimensional array, use theArrays.deepToString() method. This method has to "go deep" because it needs to work down through both dimensions in the array to display the values, not just 1.

Here is what a 2D array in Java code would look like:

```
import java.util.Arrays;
```

```
class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = {
      {100, 92, 99, 85},
      {100, 95, 88, 91},
      {99, 100, 100, 100}
    };
    System.out.println(Arrays.deepToString(scores));
  }
}
```

The output should look like this:



```
> java Scores2DArray.java
[[100, 92, 99, 85], [100, 95, 88, 91], [99, 100, 100, 100]]
> 
```

As demonstrated above, this 2D list is a long row of elements separated by the square brackets. Keep in mind that a 2-dimensional array in Java can be thought of as an array of arrays.

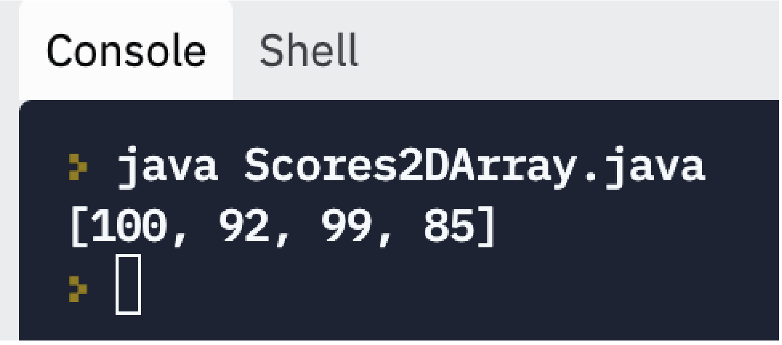
The array scores could have been declared like this:

```
import java.util.Arrays;

class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = new int[3][4];
    // Assign a 1D array to each row in the 2D array
    scores[0] = new int[] {100, 92, 99, 85};
    scores[1] = new int[] {100, 95, 88, 91};
    scores[2] = new int[] {99, 100, 100, 100};
    // Print out just the first row as a 1-dimensional array
    System.out.println(Arrays.toString(scores[0]));
  }
}
```

Either way, the first list can be accessed by using the index for the first row (index 0):

```
import java.util.Arrays;

class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = {
      {100, 92, 99, 85},
      {100, 95, 88, 91},
      {99, 100, 100, 100}
    };
    // Print out just the first row as a 1-dimensional array
    System.out.println(Arrays.toString(scores[0]));
  }
}
```

The output is just the first row from the 2D array as a 1D array as seen below:



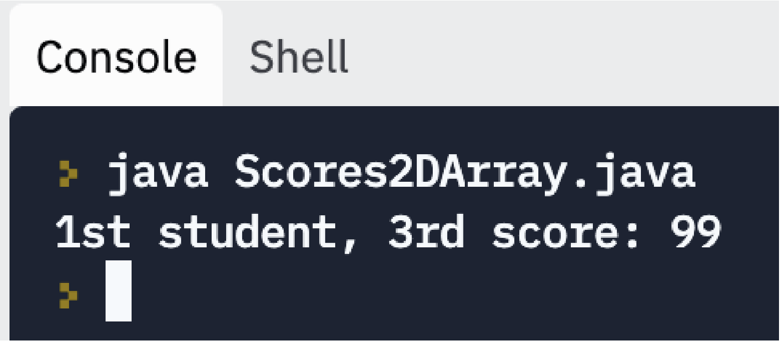
```
> java Scores2DArray.java
[100, 92, 99, 85]
> 
```

To access a specific element within that first row, use an index in a second pair square brackets after the first set. To select the third element from the first row, use the 2 as the column index. Each row in the 2D array represents the grades for one student.

The following code shows how to access the 3rd score for the 1st student, since each column (the 2nd dimension or index) represents a different assignment score:

```
class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 column
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = new int[3][4];
    // Assign a 1D array to each row in the 2D array
    scores[0] = new int[] {100, 92, 99, 85};
    scores[1] = new int[] {100, 95, 88, 91};
    scores[2] = new int[] {99, 100, 100, 100};
    // Print out just the element in the first row,
    // third column as int. No toString() needed.
    System.out.println("1st student, 3rd score: " + scores[0][2]);
  }
}
```

The output should look like this:



```
> java Scores2DArray.java
1st student, 3rd score: 99
> 
```

TERMS TO KNOW

Two-Dimensional (or 2D) Array
A two-dimensional array is an array of arrays with data laid out in a grid-like pattern of rows and columns.

Arrays.deepToString()
A Java method that converts data in a two-dimensional array to a format that can be displayed on the screen. [Text]

1c. Three Dimensional Arrays

Using **three-dimensional arrays** can be complicated. Here, you will consider X,Y, and Z index positions. At this point it becomes difficult to see a representation of it on "paper" or a flat plane. Utilizing three dimensional collections of data is rarely used unless it is for 3D object modeling or mapping.

When defining a 3D array representing some sort of values in points in 3-dimensional space, the declaration would look like this:

```
int[][][] temperatures3D = new int[100][100][100];
```

The values in the elements of this array could be assigned and accessed using 3 indices. The first index is the "layer" in the 3-dimensional structure. The second index is the row within that layer, and the third index is the column within that row.

TERM TO KNOW

Three-Dimensional (or 3D) Array
A three-dimensional array adds a 3rd index or axis to the organization of the array elements.

SUMMARY

In this lesson, you learned about using arrays with **multiple dimensions**. You learned about the difference between **one, two, and three dimensions**. Finally, you learned that three-dimensional arrays are complex and not often used in basic programming in Java.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Arrays.deepToString()

A Java method that converts data in a two-dimensional array to a format that can be displayed on the screen.

Dimension

A dimension of an array is an axis along which the elements are organized.

Three-Dimensional (or 3D) Array

A three-dimensional array adds a 3rd index or axis to the organization of the array elements.

Two-Dimensional (or 2D) Array

A two-dimensional array is an array of arrays with data laid out in a grid-like pattern of rows and columns.

by Sophia

WHAT'S COVERED

In this lesson, you will learn to recognize that arrays have more than one dimension as you explore the difference between one, two, and three dimensional arrays. Specifically, this lesson covers:

1. Multidimensional Arrays

A **dimension** of an array is an axis along which the elements are organized. The data in an array can be organized along one, two, or more axes or dimensions.

CONCEPT TO KNOW

Java multidimensional arrays are arranged as an array of arrays.

The elements of multidimensional arrays are seen in rows and columns. Before discussing multiple dimensions in Java arrays, it is important to explore what multiple dimensions look like.

TERM TO KNOW

Dimension
A dimension of an array is an axis along which the elements are organized.

1a. One-Dimensional Arrays

Recall what you learned about one-dimensional arrays in a previous tutorial. A one-dimensional array resembles a list data collection type.

It appears as a straight line on a flat plane, as demonstrated in this diagram:

A one-dimensional listing would look like this on paper:

element value	“sun”	“fun”	“run”	“bun”	“nun”	“pun”
Index	0	1	2	3	4	5

The index position is one integer. To locate “run” from this listing, you can use the single index position. Remember that list elements start at 0.

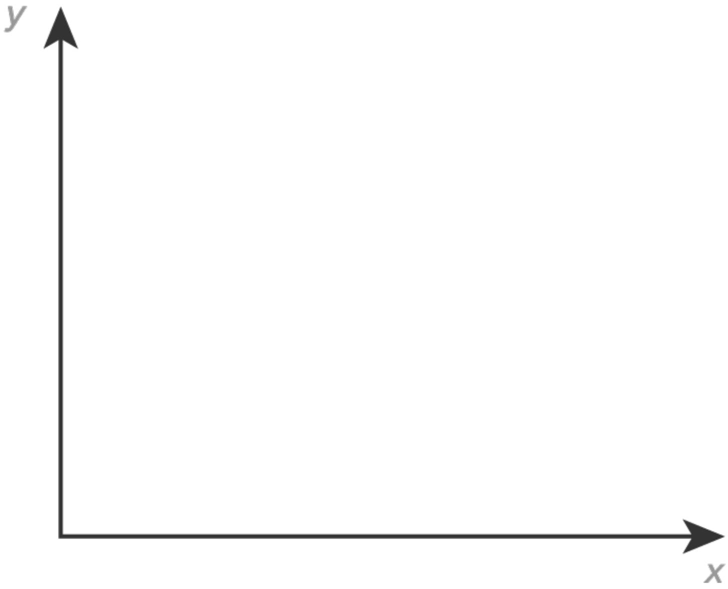
EXAMPLE The element “run” is at index value 2.

1b. Two-Dimensional Arrays

When nesting an array within an array, you are, in a sense, adding a second dimension. A nested list is a **two-**

dimensional list, or 2D, for short.

Here is what it looks like graphically:



You have likely seen this previously in an X/Y chart. It has two dimensions. One dimension is running diagonally and the other vertically, though they are still on a flat plane.

A two-dimensional listing would look like this on paper:

X/Y indexes		X →		
		0	1	2
Y ↓	0	fun	sun	run
	1	bun	nun	pun
	2	fan	van	man
	3	can	tan	ran

There are two index positions for each element. To locate "fan" from this two-dimensional listing, identify both index positions, starting with the row, then column.

↱ EXAMPLE The element "fan" is at position 2, 0 (Y, X or Row/Column)

There can be more than one dimension when it comes to an array in Java. Consider how teachers store their grades for a class in a grade book. They have a list of students and within the list of students, they have a list of assignments. For each of those assignments, they will have a grade assigned.

The scores in Row 0 represent the scores for one student. This person's scores, for the different assignments, are seen in Column 0, Column 1, Column 2, and Column 3. The scores in Row 1 and Row 2 represent scores for other students. Since all of the values in an array have to be of the same data type, you cannot mix names or letter grades in with the integer scores. The following table demonstrates four inner lists within an outer list, making this a 2D list:

	Column 0	Column 1	Column 2	Column3

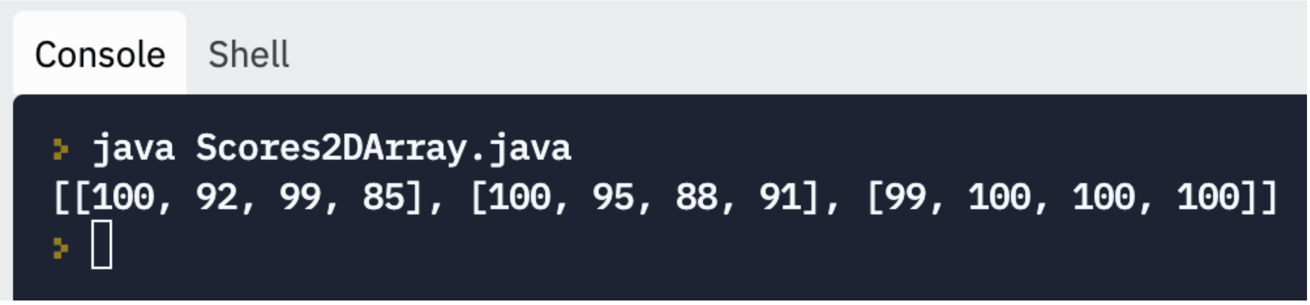
Row 0	100	92	99	85
Row 1	100	95	88	91
Row 2	99	100	100	100

Let’s see this 2D array in Java code. To print out a two-dimensional array, use the **Arrays.deepToString()** method. This method has to "go deep" because it needs to work down through both dimensions in the array to display the values, not just one.

Here is what a 2D array in Java code would look like:

```
import java.util.Arrays;

class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = {
      {100, 92, 99, 85},
      {100, 95, 88, 91},
      {99, 100, 100, 100}
    };
    System.out.println(Arrays.deepToString(scores));
  }
}
```

The output should look like this:



```
> java Scores2DArray.java
[[100, 92, 99, 85], [100, 95, 88, 91], [99, 100, 100, 100]]
> 
```

As demonstrated above, this 2D list is a long row of elements separated by the square brackets. Keep in mind that a two-dimensional array in Java can be thought of as an array of arrays.

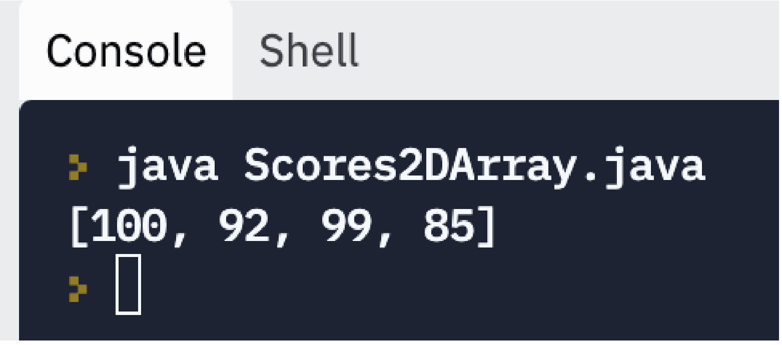
The array scores could have been declared like this:

```
import java.util.Arrays;

class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = new int[3][4];
    // Assign a 1D array to each row in the 2D array
    scores[0] = new int[] {100, 92, 99, 85};
    scores[1] = new int[] {100, 95, 88, 91};
    scores[2] = new int[] {99, 100, 100, 100};
    // Print out just the first row as a 1-dimensional array
    System.out.println(Arrays.toString(scores[0]));
  }
}
```

Either way, the first list can be accessed by using the index for the first row (index 0):

```
import java.util.Arrays;

class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 columns
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = {
      {100, 92, 99, 85},
      {100, 95, 88, 91},
      {99, 100, 100, 100}
    };
    // Print out just the first row as a 1-dimensional array
    System.out.println(Arrays.toString(scores[0]));
  }
}
```

The output is just the first row from the 2D array as a 1D array, as seen below:



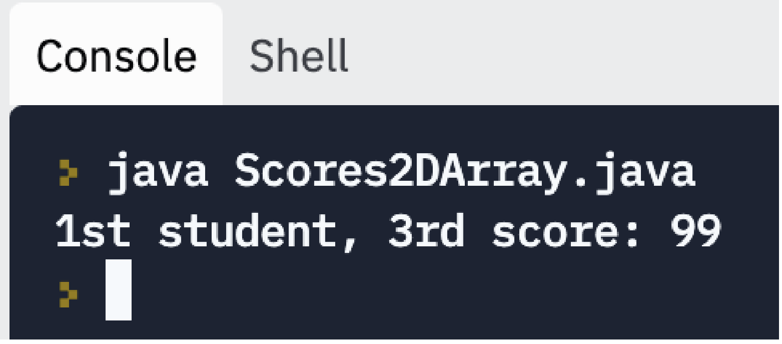
```
> java Scores2DArray.java
[100, 92, 99, 85]
> 
```

To access a specific element within that first row, use an index in a second pair of square brackets after the first set. To select the third element from the first row, use the 2 as the column index. Each row in the 2D array represents the grades for one student.

The following code shows how to access the 3rd score for the 1st student, since each column (the 2nd dimension or index) represents a different assignment score:

```
class Scores2DArray {
  public static void main(String[] args) {
    // Array of 3 rows & 4 column
    // 2 pairs of square brackets declare 2D array of int
    int[][] scores = new int[3][4];
    // Assign a 1D array to each row in the 2D array
    scores[0] = new int[] {100, 92, 99, 85};
    scores[1] = new int[] {100, 95, 88, 91};
    scores[2] = new int[] {99, 100, 100, 100};
    // Print out just the element in the first row,
    // third column as int. No toString() needed.
    System.out.println("1st student, 3rd score: " + scores[0][2]);
  }
}
```

The output should look like this:



```
> java Scores2DArray.java
1st student, 3rd score: 99
> 
```

TERMS TO KNOW

Two-Dimensional (or 2D) Array
A two-dimensional array is an array of arrays with data laid out in a grid-like pattern of rows and columns.

Arrays.deepToString()
A Java method that converts data in a two-dimensional array to a format that can be displayed on the screen.

1c. Three-Dimensional Arrays

Using **three-dimensional arrays** can be complicated. Here, you will consider X, Y, and Z index positions. At this point, it becomes difficult to see a representation of it on “paper” or a flat plane. Utilizing three-dimensional collections of data is rarely done unless it is for 3D object modeling or mapping.

When defining a 3D array representing some sort of values in points in three-dimensional space, the declaration would look like this:

↪ EXAMPLE

```
int[][][] temperatures3D = new int[100][100][100];
```

The values in the elements of this array could be assigned and accessed using three indices. The first index is the "layer" in the three-dimensional structure. The second index is the row within that layer, and the third index is the column within that row.

TERM TO KNOW

Three-Dimensional (or 3D) Array
A three-dimensional array adds a 3rd index or axis to the organization of the array elements.

SUMMARY

In this lesson, you learned about using arrays with **multiple dimensions**. You learned about the difference between **one, two, and three dimensions**. Finally, you learned that three-dimensional arrays are complex and not often used in basic programming in Java.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Arrays.deepToString()
A Java method that converts data in a two-dimensional array to a format that can be displayed on the screen.

Dimension
A dimension of an array is an axis along which the elements are organized.

Three-Dimensional (or 3D) Array
A three-dimensional array adds a 3rd index or axis to the organization of the array elements.

Two-Dimensional (or 2D) Array
A two-dimensional array is an array of arrays with data laid out in a grid-like pattern of rows and columns.

Debugging Arrays and other Collection Types

by Sophia

WHAT'S COVERED

In this lesson, you will learn about common problems that arise when working with arrays and collection types used in Java. Specifically, this lesson covers:

1. Array Index Errors and Debugging

When working with arrays and accessing items in an ArrayList, out-of-range indices are one of the most common **array index errors** that can occur. It is important to remember that the first element in an array always has the index 0.

CONCEPT TO KNOW

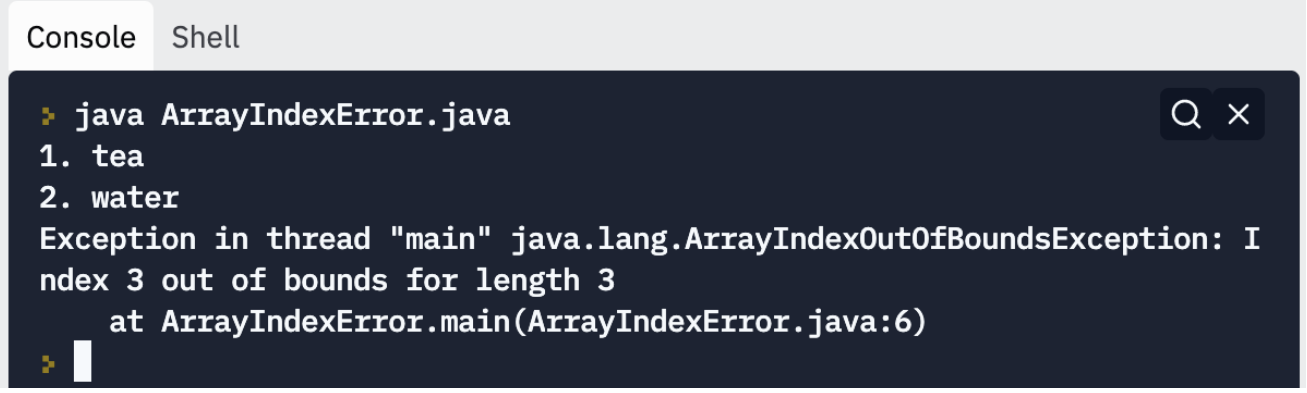
Array indices are never negative.

The "off-by-one" error is another common error in arrays. This error is generated when forgetting that the last index in an array is always equal to the size of the array minus 1. If an array has the length 5, the indices in the array are 0, 1, 2, 3, and 4 (not 1, 2, 3, 4, and 5). Java will produce errors rather than try to access an invalid memory location, as can happen with languages like C and C++, if the programmer is not careful.

Consider the following sample:

```
class ArrayIndexError {
  public static void main(String[] args) {
    String[] choices = {"coffee", "tea", "water"};
    System.out.println("1. " + choices[1]);
    System.out.println("2. " + choices[2]);
    System.out.println("3. " + choices[3]);
  }
}
```

The results should look like this:



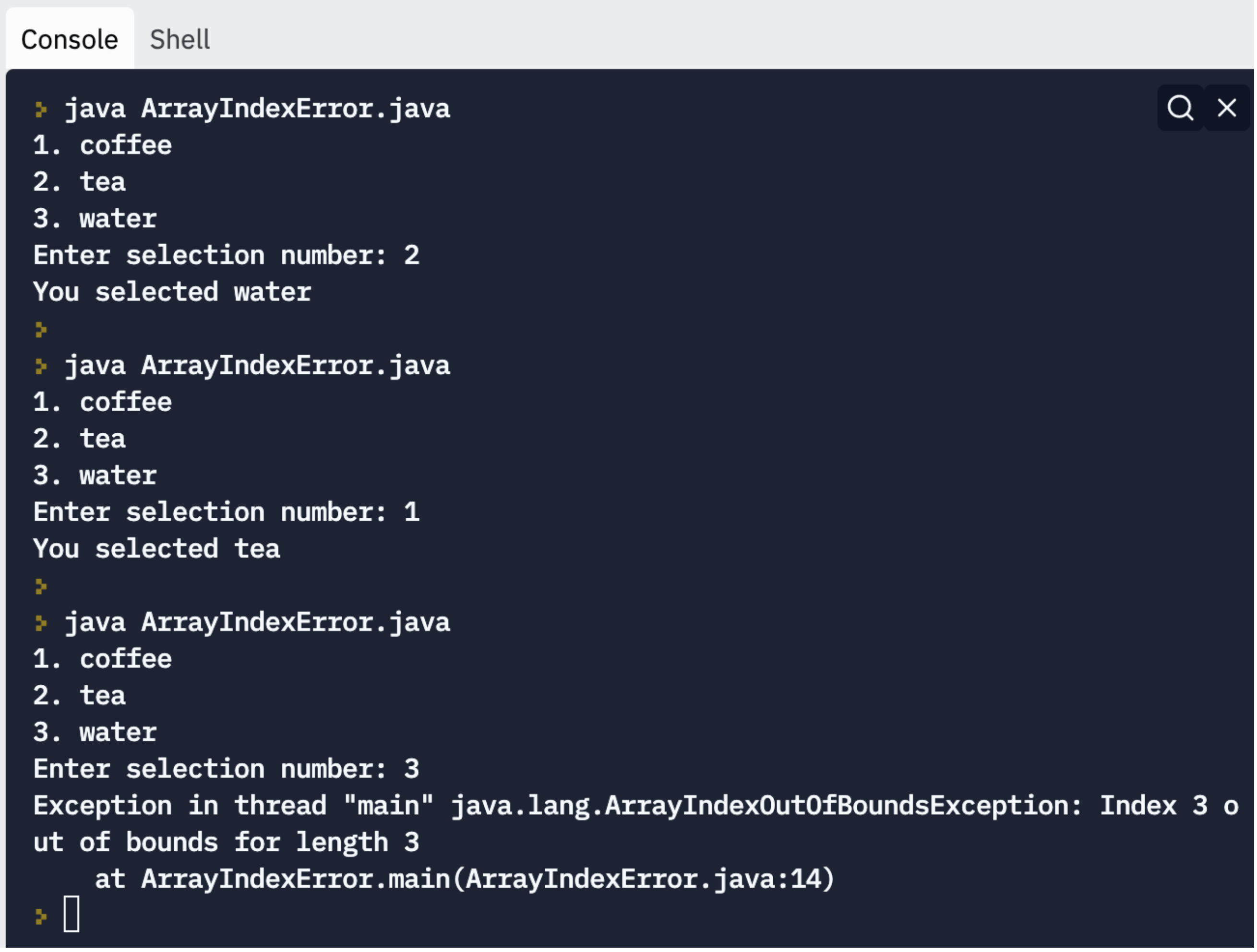
```
> java ArrayIndexError.java
1. tea
2. water
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: I
ndex 3 out of bounds for length 3
    at ArrayIndexError.main(ArrayIndexError.java:6)
>
```

In the introduction to arrays tutorial, you learned that try and catch blocks can be used to deal with out-of-bounds indices. Although, this case is a programming error. The code should be corrected rather than relying on try and catch blocks. The list in the output above starts with the second element in the array because the code accesses choices[1] for the first item rather than choices[0], so the code produces an inaccurate list in addition to causing a runtime error. The error, an ArrayIndexOutOfBounds exception, is caused by trying to read past the end of the array, choices[3], when the last element in the array is choices[2].

Here is what it looks like if the array is accessed correctly when the list of choices is printed but the input is used incorrectly:

```
import java.util.Scanner;

class ArrayIndexError {
  public static void main(String[] args) {
    String[] choices = {"coffee", "tea", "water"};
    Scanner input = new Scanner(System.in);
    System.out.println("1. " + choices[0]);
    System.out.println("2. " + choices[1]);
    System.out.println("3. " + choices[2]);
    System.out.print("Enter selection number: ");
    int selection = input.nextInt();
    // Next line produces the wrong result. It should be use
    // choices[selection - 1]
    System.out.println("You selected " + choices[selection]);
  }
}
```

This screenshot shows three runs of the program. Two of the trials do not produce any error messages; however, the outcome is wrong. The third run results in an exception, as seen in the output below:



```
› java ArrayIndexError.java
1. coffee
2. tea
3. water
Enter selection number: 2
You selected water
›
› java ArrayIndexError.java
1. coffee
2. tea
3. water
Enter selection number: 1
You selected tea
›
› java ArrayIndexError.java
1. coffee
2. tea
3. water
Enter selection number: 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 o
ut of bounds for length 3
    at ArrayIndexError.main(ArrayIndexError.java:14)
› 
```

CONCEPT TO KNOW

You must always be mindful of such "off-by-one" errors and code carefully to avoid them.

- The first index in an array always has the value 0.
- The last index in an array always has the value length minus 1 (where length is the number of elements in the array).

When assessing user input to determine which element in an array needs to be accessed, it is important to always verify that the input is in the correct range. A numbered menu list almost always begins with 1; therefore, you will need to remember to subtract 1 when checking that the selection is in bounds, and when accessing the array element via its index.
When using a two-dimensional array, remember that the first index refers to the row, and the second index refers to the column. Keep in mind that a program relying on user input may not have the inputs in this order.

HINT

When running into problems with an array in your code, it can be helpful to add some temporary output statements that use Arrays.toString() (for 1D arrays) and Arrays.deepToString() (for 2D and 3D arrays) to display the contents of an array on the screen. This will help you see what data is where in the array, at a particular point in the program. Do not forget to remove this extra output when you are finished debugging.

TERM TO KNOW

Array Index Error

An error that occurs when a programmer accesses the wrong element in an array (or ArrayList collection) or tries to access an element that does not exist.

2. Debugging Other Collection Types

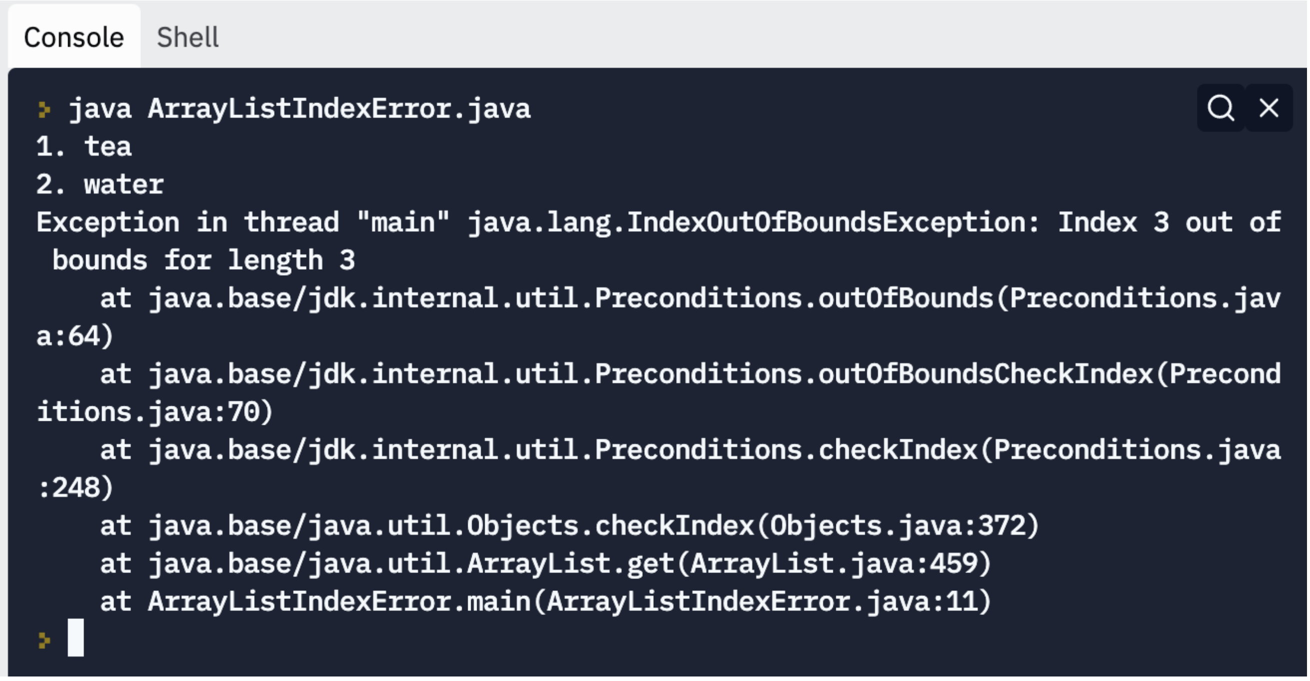
There are similarities between a plain, single-dimensional array and anArrayList. This also means that some similar problems can arise.

While the size of an ArrayList can grow and be reduced, as items are added with theadd() method and removed with the remove() method), the get() method needs a valid index to access an item in the ArrayList. The first item in an ArrayList always has the index 0, and the last item index is equal to the ArrayList's size minus 1.

Below are problems similar to the array example above. This code prints out the list of beverages using 1 to try to access the first item and 3 to access the last, which doesn't work well:

```
import java.util.ArrayList;

class ArrayListIndexError {
  public static void main(String[] args) {
    ArrayList<String> choices = new ArrayList<>();
    choices.add("coffee");
    choices.add("tea");
    choices.add("water");
    System.out.println("1. " + choices.get(1));
    System.out.println("2. " + choices.get(2));
    System.out.println("3. " + choices.get(3));

  }
}
```

Running this code in a file named ArrayListIndexError.java produces this output:



```
> java ArrayListIndexError.java
1. tea
2. water
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of
 bounds for length 3
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.jav
a:64)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Precond
itions.java:70)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java
:248)
    at java.base/java.util.Objects.checkIndex(Objects.java:372)
    at java.base/java.util.ArrayList.get(ArrayList.java:459)
    at ArrayListIndexError.main(ArrayListIndexError.java:11)
> 
```

As with the array example, we can see that the first item in the list is "tea" rather than the correct value "coffee," since this line actually accesses the second item in the ArrayList rather than the first:

EXAMPLE

```
System.out.println("1. " + choices.get(1));
```

This line needs to get the item at index 0 instead:

EXAMPLE

```
System.out.println("1. " + choices.get(0));
```

The output shows "water" next, since the code that produces this output reads:

EXAMPLE

```
System.out.println("2. " + choices.get(2));
```

In an ArrayList of three items, though, index 2 points to the last item of the three, not the second. Finally, consider the following statement:

EXAMPLE

```
System.out.println("3. " + choices.get(3));
```

This example results in an **IndexOutOfBoundsException**, since the last index in an ArrayList of three items is 2. When checking the size of an ArrayList, remember that you need to use the size() method, not the array's length property.

If the program relies on user input to access an item in the ArrayList, it is important to carry out appropriate checking and translation. This is important, since the menu item will be 1 greater than the actual index in the ArrayList, before trying to get a value from the ArrayList.

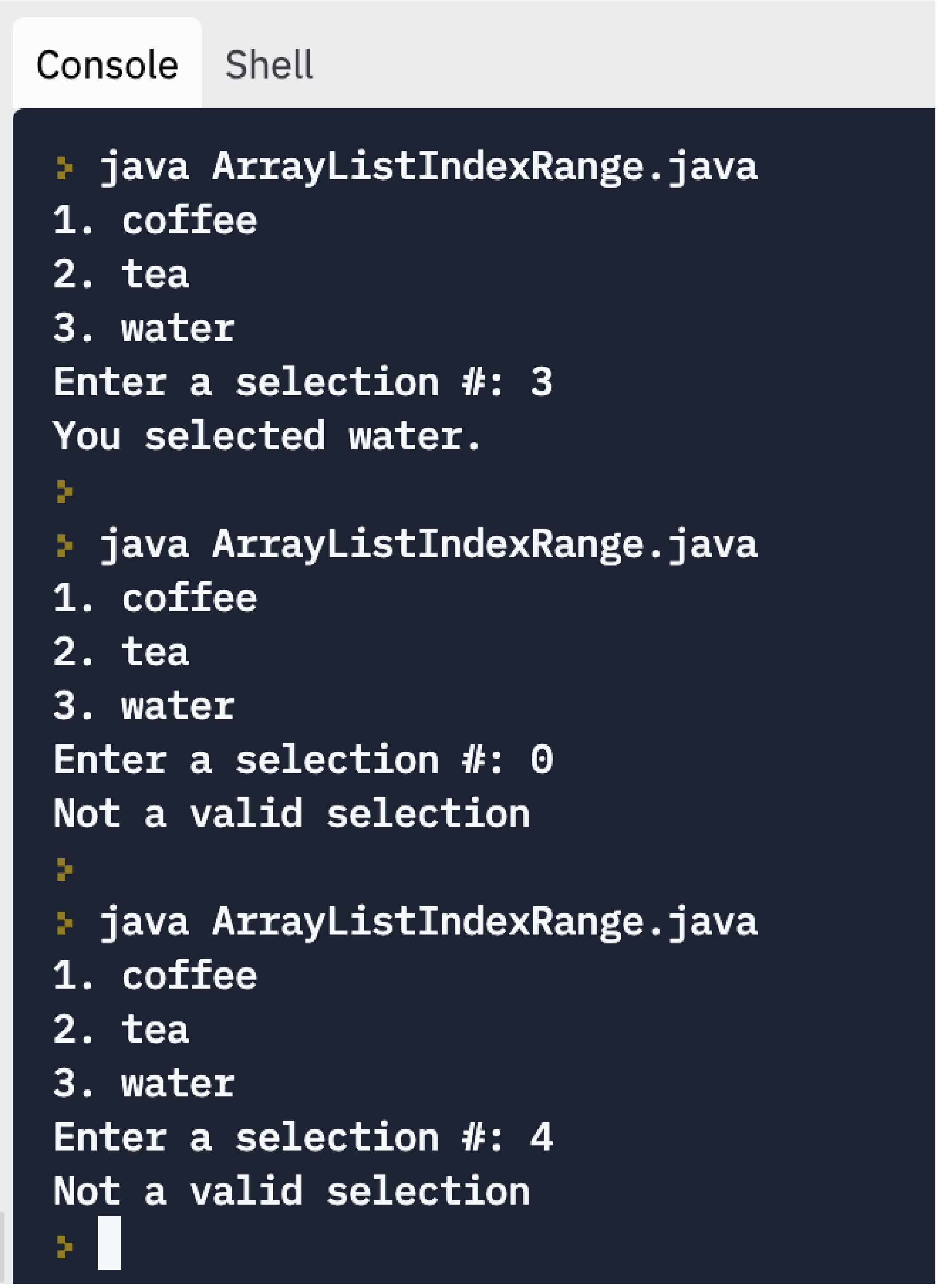
Here is a version of the code that accesses the items in theArrayList correctly and checks to make sure the selection is within the correct range. While it would be possible to use try and then catch an IndexOutOfBoundsException, it is usually best to use selection statements to deal with an out-of-bounds selection.

TRY IT

Directions: Type this code into a file named ArrayListIndexRange.java in Replit and try running it:

```
import java.util.ArrayList;
import java.util.Scanner;

class ArrayListIndexRange {
  public static void main(String[] args) {
    ArrayList<String> choices = new ArrayList<>();
    // Add as many choices as needed, ArrayList grows to fit.
    choices.add("coffee");
    choices.add("tea");
    choices.add("water");
    System.out.println("1. " + choices.get(0));
    System.out.println("2. " + choices.get(1));
    System.out.println("3. " + choices.get(2));

    Scanner input = new Scanner(System.in);
    System.out.print("Enter a selection #: ");
    int selection = input.nextInt();
    // Subtract 1 from menu choice to get index
    int index = selection - 1;
    if(index >= 0 && index < choices.size()) {
      System.out.println("You selected " + choices.get(selection - 1) + ".");
    }
    else {
      System.out.println("Not a valid selection");
    }

  }
}
```

Here is a screenshot showing three sample runs of the program:



```
> java ArrayListIndexRange.java
1. coffee
2. tea
3. water
Enter a selection #: 3
You selected water.
>
> java ArrayListIndexRange.java
1. coffee
2. tea
3. water
Enter a selection #: 0
Not a valid selection
>
> java ArrayListIndexRange.java
1. coffee
2. tea
3. water
Enter a selection #: 4
Not a valid selection
>
```

REFLECT

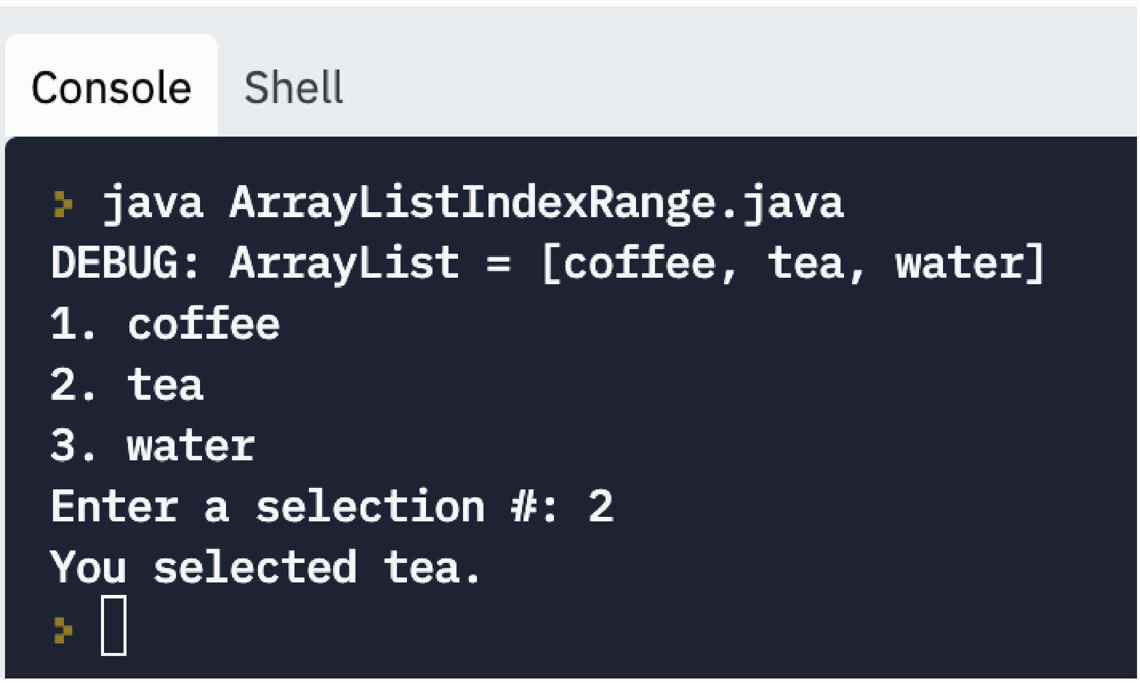
This program shows how a combination of selection statements can be used when working with a collection to avoid exceptions and program crashes if the user makes an incorrect selection. A message about the problem is a better user experience than a crash caused by an unhandled exception. Using selection statements like this is better than using try and catch blocks. Why do you think this might be the case?

TRY IT

Directions: When working with ArrayLists, you can also add temporary output statements that use the toString() method to show the current contents:

```
import java.util.ArrayList;
import java.util.Scanner;

class ArrayListIndexRange {
  public static void main(String[] args) {
    ArrayList<String> choices = new ArrayList<>();
    // Add as many choices as needed, ArrayList grows to fit.
    choices.add("coffee");
    choices.add("tea");
    choices.add("water");
    // A temporary output statement for debugging
    System.out.println("DEBUG: ArrayList = " + choices.toString());
    System.out.println("1. " + choices.get(0));
    System.out.println("2. " + choices.get(1));
    System.out.println("3. " + choices.get(2));

    Scanner input = new Scanner(System.in);
    System.out.print("Enter a selection #: ");
    int selection = input.nextInt();
    // Subtract 1 from menu choice to get index
    int index = selection - 1;
    if(index >= 0 && index < choices.size()) {
      System.out.println("You selected " + choices.get(selection - 1) + ".");
    }
    else {
      System.out.println("Not a valid selection");
    }
  }
}
```

Note the "Debug" line in the output that allows the programmer to confirm that the collection contains the expected values:



```
> java ArrayListIndexRange.java
DEBUG: ArrayList = [coffee, tea, water]
1. coffee
2. tea
3. water
Enter a selection #: 2
You selected tea.
> 
```

REFLECT

This code shows how a temporary output statement and a collection's toString() method can be used to confirm the contents of a collection when developing and debugging an application. Of course, such temporary output statements should be removed when development is done, since they are likely to confuse someone using the program.

HINT

Be sure to remove such extra output statements when you're done with the debugging process.

When working with HashSet and HashMap collections, such index-out-of-bounds errors will not occur, since these collections don't allow access to individual elements by index. Almost all collection types use temporary output statements that make use of toString(). They can be helpful for checking the contents of the collection and making sure that they match the programmer's expectations.

TERM TO KNOW

IndexOutOfBoundsException
An exception (runtime error) thrown when code tries to access an element that is outside the bounds of an array.

SUMMARY

In this lesson, you learned about common problems that can arise when working with arrays and collections in Java. These include **array index errors**. "Off-by-one" errors are a common problem. You learned that it is important to keep in mind that while people generally start counting at 1, computers and programming languages usually start with 0. Finally, you learned that when accessing data in arrays and collections and **when debugging other collections**, it is important to check that the index is in bounds. This includes cases where the index is based on user input.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Array Index Error
An error that occurs when a programmer accesses the wrong element in an array (or ArrayList collection) or tries to access an element that does not exist.

IndexOutOfBoundsException
An exception (runtime error) thrown when code tries to access an element that is outside the bounds of an array.

Tic-Tac-Toe Program

by Sophia

WHAT'S COVERED

In this lesson, you will learn about and how to create a program that uses arrays and applies earlier concepts. Specifically, this lesson covers:

1. Review of Arrays

As you learned in tutorial 2.1.1, an array is a special type of variable that represents multiple values all of the same data type. You learned about working with single-dimensional arrays and two-dimensional arrays. This includes acknowledging the existence of three-dimensional arrays, though we are not working with them. Remember that arrays, unlike collections, have fixed sizes. Arrays are often useful for modeling simple structures in the real world that are characterized by sets of values, as long as the values are the same type of data.

1a. One-Dimensional Arrays

A single-dimensional array is like a row of numbered boxes or slots, where each space allows the storing one value of the data type declared for the array. These spaces (or boxes or slots) in the array are called elements. Each element has an index, an integer value, that identifies it in the array and indicates its position in the array. The first element in an array has the index 0, and the last element in the array has an index equal to the declared size of the array minus 1.

HINT

Remember that unlike a collection, the size of an array is fixed and cannot easily change. The values in the elements in the array can, however, be changed.

1b. Two-Dimensional Arrays

While a single-dimensional array can be thought of as a single row of elements, a two-dimensional array can be pictured as a grid or checkerboard with rows and columns. A two-dimensional array is specified using the number of rows and the number of columns. As with a single-dimensional array, the size of the two-dimensional array (the number of rows and the number of columns) is fixed when the array is declared. Elements in a two-dimensional array are specified using two indices: the index for the row and the index for the column. The value of a single-dimensional array can be changed, and the value of any element can be changed as well.

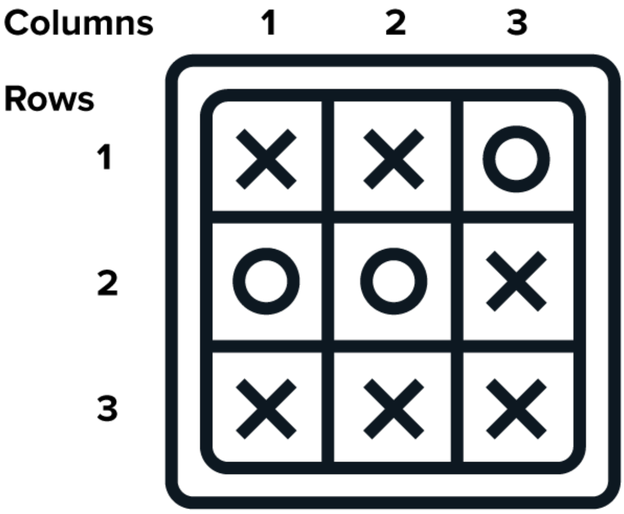
HINT

When programming in Java, it is helpful to think of a two-dimensional array as being an array of arrays. Unlike a single-dimensional array which consists of an integer or string values, think of a two-dimensional array as a series of elements in an array. Each array contains a one-dimensional array. Each nested array is a row in the two-dimensional array.

2. The Start of the Tic-Tac-Toe Game

Picture a Tic-Tac-Toe board. As you look at it, you will see that it has three rows, and each row has three squares (columns).

The size of the board is fixed and does not vary:



Each of the squares can be empty, or contain an X or an O. The position of each square never changes, so the order does matter. The contents of each of the squares can change from empty to either an X or an O, but then change no further. Given the data structures you have worked with, consider which is the best type to represent a row in a Tic-Tac-Toe board. Keep in mind that a row has exactly three squares. Each square contains information that can change and the order matters.

Since each row consists of three squares, it can be viewed as a single-dimensional array. There are three rows, so the board can be seen as three single-dimensional arrays joined into a two-dimensional array.

TRY IT

Directions: Model a row like this:

↪ EXAMPLE

```
String[] row0 = {" - ", " - ", " - "};
```

The - indicates that the cell is empty (has been filled with an X or an O).

Directions: Given that a Tic-Tac-Toe board has three rows, we can now define the rows like this:

↪ EXAMPLE

```
String[] row0 = {" - ", " - ", " - "};
String[] row1 = {" - ", " - ", " - "};
String[] row2 = {" - ", " - ", " - "};
```

Directions: Model our board like this using the previous rows and join them together into a two-dimensional array (an array of single-dimensional arrays):

↪ EXAMPLE

```
String[][] board = new String[3][3];
board[0] = row0;
board[1] = row1;
board[2] = row2;
```

REFLECT

This code is not a complete program, but it is a first attempt at showing how a Tic-Tac-Toe board can be represented as a two-dimensional array. Each row in the game is a one-dimensional array.

STEP BY STEP

1. The first step is Board Creation. Instead of this structure, we can also establish a two-dimensional array to define it. This approach would have the same result, and it could be declared like the following:

Enter the following code into Replit to start the build process for the Tic-Tac-Toe board:

```
String[][] board = {{" - ", " - ", " - "},
                    {" - ", " - ", " - "},
                    {" - ", " - ", " - "}};
```

Use this nested approach, since it makes the data easier to access and reflects the real-world behavior of the Tic-Tac-Toe board. There are three columns with three elements in each row in this list, as it resembles the board.

For now, make this game a bit easier by allowing players to alternate and choose the position. Once they have selected the position, they will be able to place the X or O in that position as long as the position still has a “-” (hyphen). Use the hyphen as an empty position.

HINT

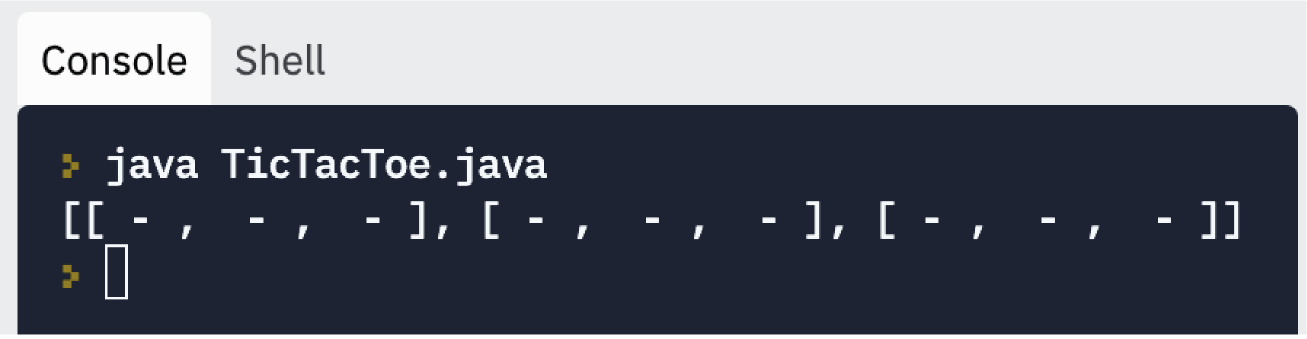
At this point, do not introduce a checking function to determine if anyone has won. For now, this board is just meant to be a replacement for drawing out the board on paper. As you work through this, think about ways to improve this process going forward.

TRY IT

Directions: Start by setting up the board and printing it out. Use the println() and Arrays.deepToString() methods to see what this board looks like on screen:

```
import java.util.Arrays;

public class TicTacToe {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};
    System.out.println(Arrays.deepToString(board));
  }
}
```

The results should look like this:



```
› java TicTacToe.java
[[ - ,  - ,  - ], [ - ,  - ,  - ], [ - ,  - ,  - ]]
› 
```

REFLECT

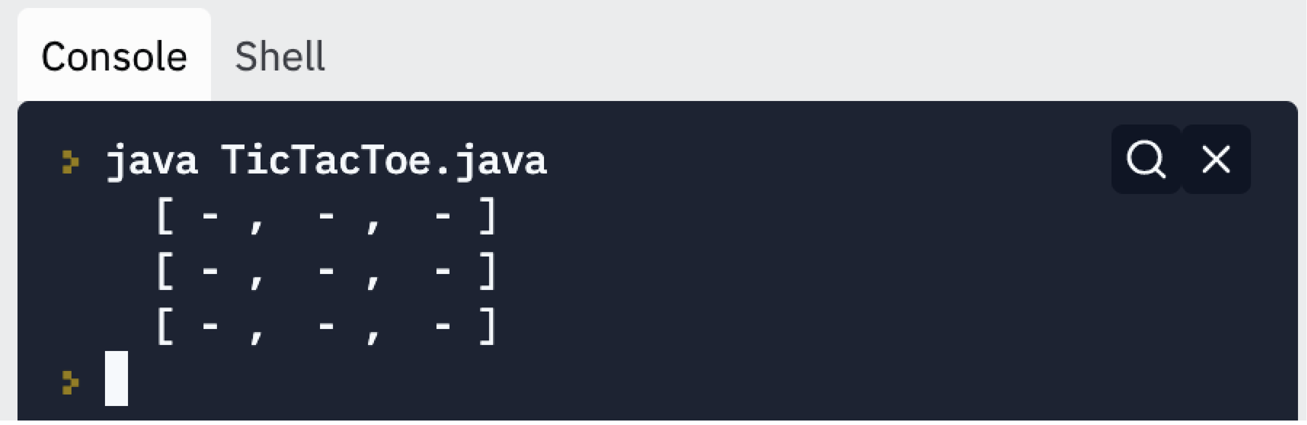
Does that look like a Tic-Tac-Toe board? Not a traditional board, though. The deepToString() method doesn't keep the formatting from the definition. Although this is our board, it would be a bit difficult to tell who won visually. To help with that, we want to output each row separately.

TRY IT

Directions: Now try the code below with each row being separately outputted to the screen:

```
import java.util.Arrays;

public class TicTacToe {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]));
  }
}
```

The results should look like this:



```
> java TicTacToe.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
>
```

REFLECT

That’s a bit better. However, what other ways could this be improved as you move into the next step? This small but complete program shows an evolution in design over the previous version by declaring the game board as a single two-dimensional array. Look at both versions of the code again to appreciate how the code has been refined.

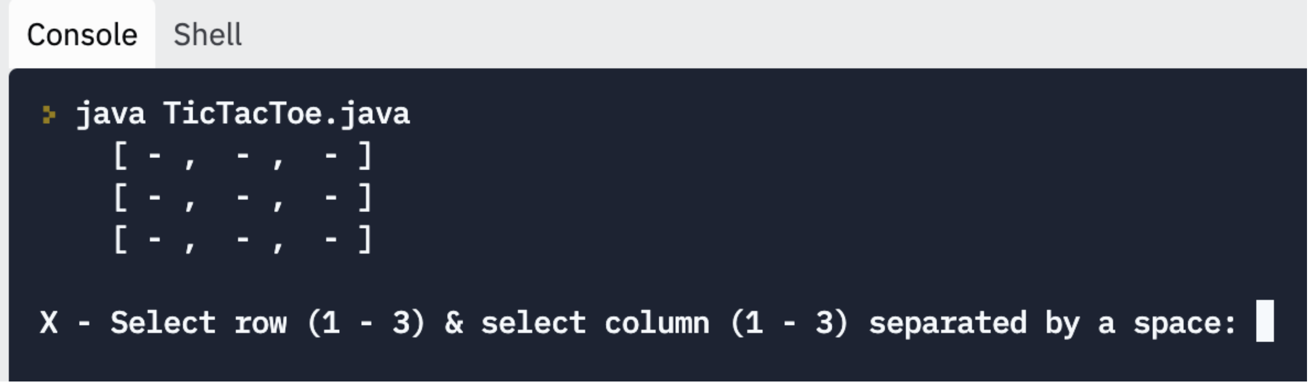
2. The second step is to prompt the user to enter in a value between 1-3 for the column and then again (1-3) for the row. Although the array starts with an index of 0 and goes to 2 (three elements), that would be confusing to the end user, which is why we will let them select between 1 and 3.

TRY IT

Directions: Enter the code to output directions to the X user for their selection:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToe {

  public static void main(String[] args) {
    int col;
    int row;
    Scanner input = new Scanner(System.in);
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};
    // The leading tab chars (\t) indent the board
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    // The \n adds a blank line below the board
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    // Display prompt text
    System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
    // Space separates row & column
    System.out.print("separated by a space: ");
    col = input.nextInt();
    row = input.nextInt();
  }
}
```

Running the code above should produce output like this:



```
› java TicTacToe.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space:
```

REFLECT

An X can now be placed in that position; output the board again. Remember that the user has entered values in the range from 1 to 3, but the indices of the array run from 0 to 2. The code will need to subtract 1 from the row and column to update the correct square on the board.

TRY IT

Directions: Add the following code to place the X in the row and column that has been identified:

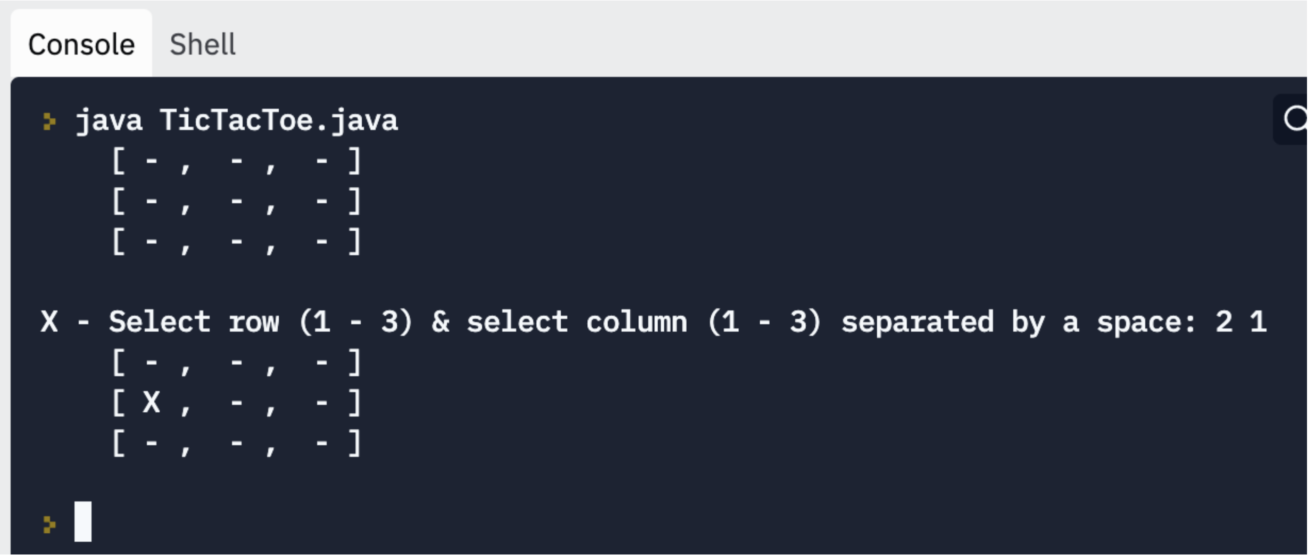
```
board[row - 1][col - 1] = " X ";
System.out.println("\nTic-Tac-Toe Board:\n");
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");
```

Now test what you have so far. Make sure your code looks like the code snippet below, then run the program:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToe {

  public static void main(String[] args) {
    int col;
    int row;
    Scanner input = new Scanner(System.in);

    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");
```

```
 // Display prompt text
  System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
 // Space separates row & column
  System.out.print("separated by a space: ");
  row = input.nextInt();
  col = input.nextInt();
  // Mark the requested square. Subtract 1 to get correct square
  board[row - 1][col - 1] = " X ";
  // Print updated board
  System.out.println("\t" + Arrays.toString(board[0]));
  System.out.println("\t" + Arrays.toString(board[1]));
  System.out.println("\t" + Arrays.toString(board[2]) + "\n");
 }
}
```

The results should look like this:



```
> java TicTacToe.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 1
    [ - ,  - ,  - ]
    [ X ,  - ,  - ]
    [ - ,  - ,  - ]

> 
```

REFLECT

Did you (as player X) get your X in the column/row you selected?

Next, prompt the player playing O to go next. Use the same approach with a couple substitutions:

TRY IT

Directions: Try adding the code for the second player to enter their position:

```
System.out.print("O - Select row (1-3) & select column (1-3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
// Mark the requested square. Subtract 1 to get correct square
board[row - 1][col - 1] = " O ";
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");
```

REFLECT

While entering this code, you may have noticed a potential problem. What if player O enters a position that player X has already chosen?

You do not want to overwrite that position. Rather, you will just say that the user ends up forfeiting their turn. The game then moves onto the next player. This is certainly not ideal. It is important to think about what you ideally want as you move forward regarding if you want the user to keep selecting a spot until they choose one that is available. Look for that type of correction in future tutorials. For now, check if the spot is empty by looking for the “–” (hyphen) character. If the selection is the “-” character, we’ll replace it with an O, and if it wasn’t, we’ll let the user know that spot was already taken, and the other player selects.

TRY IT

Directions: Enter the code to check if the space has been filled or not:

```
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
```

3. The last step is to repeat this structure seven more times since you have two entries complete and have nine total boxes to fill. This can get a bit busy with the number of times you repeat it. It would be beneficial to add comments to indicate each move.

TRY IT

Directions: Enter the next seven attempts to complete the Tic-Tac-Toe game. See the comments to separate each turn:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToe {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    int col;
    int row;
```

```
// X's 1st turn
System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// O's 1st turn
System.out.print("O - Select row (1-3) & select column (1-3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 2nd turn
System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");


// O's 2nd turn
System.out.print("O - Select row (1-3) & select column (1-3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
```

```
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 3rd turn
System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// O's 3rd turn
System.out.print("O - Select row (1-3) & select column (1-3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 4th turn
System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
      System.out.println("Sorry, that spot is taken.");
    }
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    // O's 4th turn
    System.out.print("O - Select row (1-3) & select column (1-3) ");
    System.out.print("separated by a space: ");
    row = input.nextInt();
    col = input.nextInt();
    if(board[row - 1][col - 1].equals(" - ")) {
      board[row - 1][col - 1] = " O ";
    }
    else {
      System.out.println("Sorry, that spot is taken.");
    }
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    // X's 5th turn
    System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
    System.out.print("separated by a space: ");
    row = input.nextInt();
    col = input.nextInt();
    if(board[row - 1][col - 1].equals(" - ")) {
      board[row - 1][col - 1] = " X ";
    }
    else {
      System.out.println("Sorry, that spot is taken.");
    }
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");
  }
}
```

REFLECT

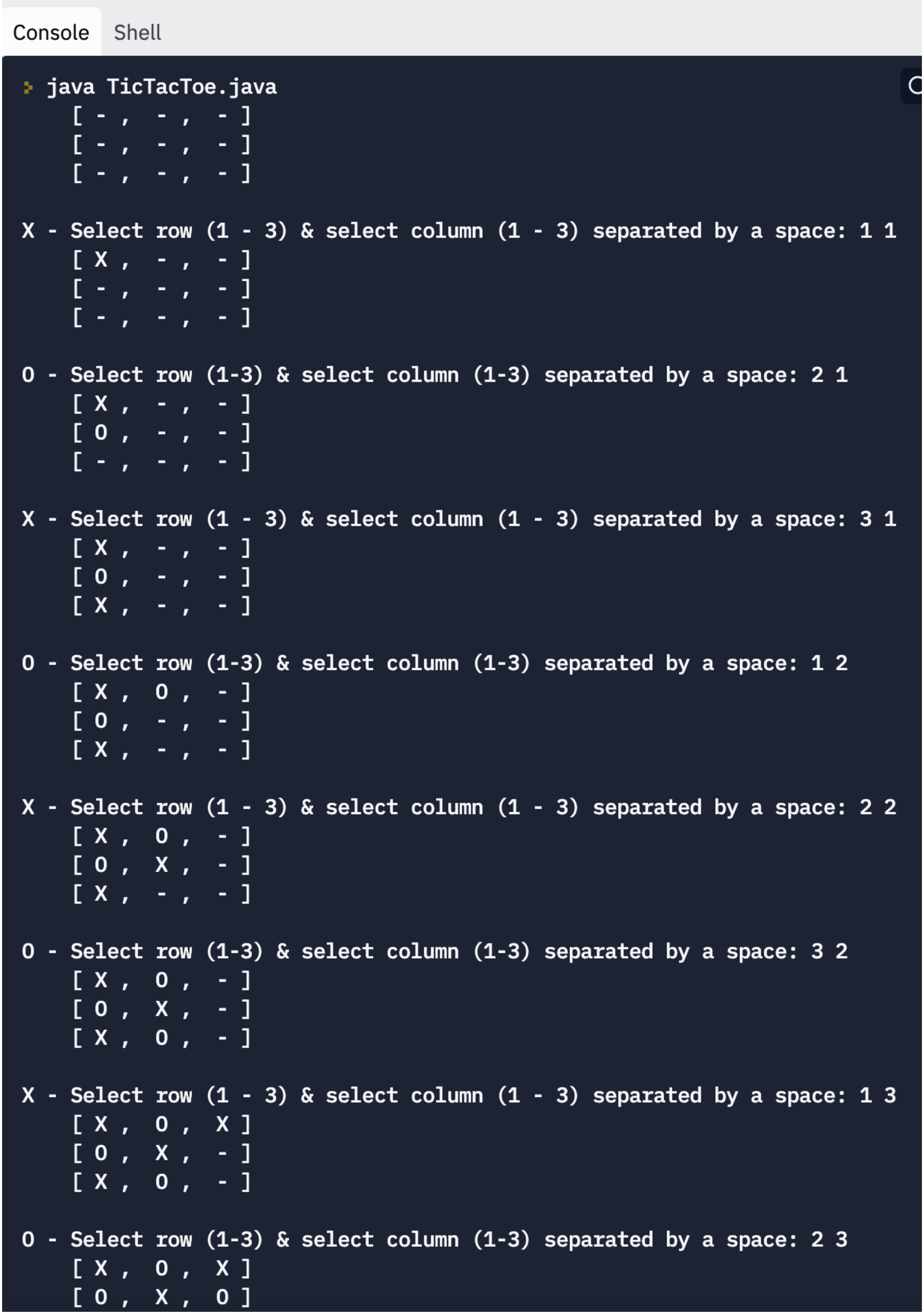
This version of the program includes code to handle all of the turns in the game. To get ready for the next version of the Tic-Tac-Toe game, think about the common steps involved in each turn.

We’re now ready to test it to see how we did!

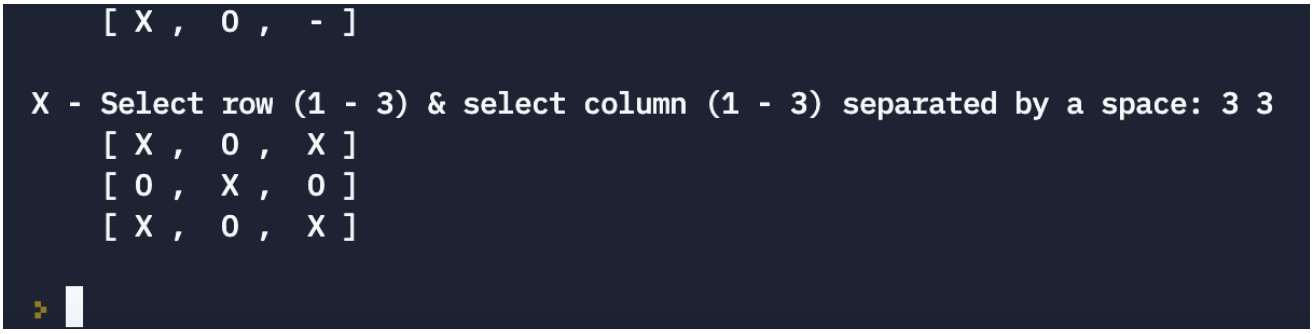
TRY IT

Directions: Run the Tic-Tac-Toe Program and enter choices for players X and O:

The result should look like this:



```
> java TicTacToe.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 1
    [ X ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

O - Select row (1-3) & select column (1-3) separated by a space: 2 1
    [ X ,  - ,  - ]
    [ O ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 3 1
    [ X ,  - ,  - ]
    [ O ,  - ,  - ]
    [ X ,  - ,  - ]

O - Select row (1-3) & select column (1-3) separated by a space: 1 2
    [ X ,  O ,  - ]
    [ O ,  - ,  - ]
    [ X ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 2
    [ X ,  O ,  - ]
    [ O ,  X ,  - ]
    [ X ,  - ,  - ]

O - Select row (1-3) & select column (1-3) separated by a space: 3 2
    [ X ,  O ,  - ]
    [ O ,  X ,  - ]
    [ X ,  O ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 3
    [ X ,  O ,  X ]
    [ O ,  X ,  - ]
    [ X ,  O ,  - ]

O - Select row (1-3) & select column (1-3) separated by a space: 2 3
    [ X ,  O ,  X ]
    [ O ,  X ,  O ]
```



```
    [ X ,  O ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 3 3
    [ X ,  O ,  X ]
    [ O ,  X ,  O ]
    [ X ,  O ,  X ]

 
```

REFLECT

As noted, this version of the program handles all of the turns. In addition to thinking about the common elements in each turn, what do you think the next steps in the development process will be?

You have now replaced the paper version with a computerized version. However, you may notice that there's a lot of repeating elements. It is missing other features, including things like checking if a player has already won. Additionally, it is missing a check to see if the player is entering a value that is within the correct range. During future tutorials on loops, you will learn how much we can trim this program down and optimize the code even further.

SUMMARY

You **reviewed the use of arrays**. This included the use of **one-dimensional arrays**. Based on the properties of the data, you learned that a **two-dimensional array** was the best match for our**Tic-Tac-Toe program**, since it was known that the number of elements is fixed, but the values in the elements will need to change. You learned that the order of the elements is important to represent the state of the game. You designed and implemented the board of the program using nested lists and structured it to resemble a real-world Tic-Tac-Toe board. Finally, you programmed both players' input and sent the results to the screen. This program does not handle incorrect player input; the player chooses an element that is already taken. This will be covered in future tutorials to optimize the code.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

Introduction to Loops

by Sophia

WHAT'S COVERED

In this lesson, you will learn about the basics of loops. Specifically, this lesson covers:

1. Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well, and people do poorly. Iteration is one of the most important concepts in looping.

CONCEPT TO KNOW

Iteration is the repetition of a sequence of steps.

Iteration is so common that Java provides several language features to make iteration easier to use. Loops allow us to repeat code multiple times efficiently and without unduly repetitious code. A programming feature that implements iteration well is called a loop.

In programming, there are two types of iteration with loops, indefinite and definite iteration.

1a. Indefinite Iteration

When using **indefinite iteration**, users do not specify the number of times that the loop is meant to run in advance. Instead, the loop would run repeatedly, as long as a condition is met. This is very similar to a selection statement that is checked each time. Think about a video game menu that keeps repeating the selections until the user quits the game by choosing the option to exit. In Java, indefinite loops are created using any loop. This tutorial will focus on the use of the while loop.

TERM TO KNOW

Indefinite Iteration
A loop that does not specify, in advance, how many times it is to be run. It repeats as long as a condition is met. In Java, indefinite loops are typically created using the while loop.

1b. Definite Iteration

When using **definite iteration**, the number of times the block of code runs and the maximum number of times that the block of code is allowed to run should be explicitly defined when the loop starts. This is typically implemented using the for loop. The for loop has a specific start and endpoint.

CONCEPT TO KNOW

When working with any kind of loop, it is important the determine the following:

- Which steps in the process are not repeated and need to happen before the loop starts repeating a sequence of steps. This sets up the context in which the loop will run.
- Which steps need to be repeated, and thus should be in the body of the loop.
- Which steps in the process happen only once after the end of the loop (i.e., after the loop is done repeating).

TERM TO KNOW

Definite Iteration
Definite iteration identifies the number of times, or the maximum number of times, the block of code runs. This should be explicitly defined when the loop actually starts. Typically, this is implemented using the for loop. The for loop has a specific start and endpoint.

2. Basics of a while Loop

The **while loop** is one of the most commonly used loops. A while loop continually evaluates a condition looking for a true or false value. It keeps looping as long as the evaluated condition is true. A while loop is used when the number of times the loop will run, or the maximum number of times the loop will run, is not known at programming time. This is an example of indefinite iteration.

Let's first look at the structure of the while loop. The format of a while loop looks like the following:

EXAMPLE

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

As you learned in the tutorial about selection statements, if the body of the loop consists of just one statement, the curly brackets are not required. However, if the code is modified so that the body of the loop consists of more than one statement, and the curly brackets are not added, the code will behave in unexpected ways. As a general rule, it is best to put in the curly brackets in all cases, since using them is never wrong, and they just take up a small amount of space on the screen. They make the structure of the program clearer (especially for beginners).

In the example above, the <expression> and <statement(s)> are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of the parts of a while loop.

The <statements(s)> term represents the block of code that should be executed repeatedly. This is also called the **body of the loop**. This is notated in the same way as a conditional statement is, with an indent. All of the iteration features like the while and for loop (for loop coming up later) use the same style of indentation as used previously, with the if/else/elif conditional statements by Java to define the code blocks.

The <expression> term typically is based on one or more variables that are initialized outside of the loop and then modified within the body of the loop. The <expression> represents the evaluated condition. Java is looking for a Boolean True or False value from this condition.

2a. How Does a while Loop Work?

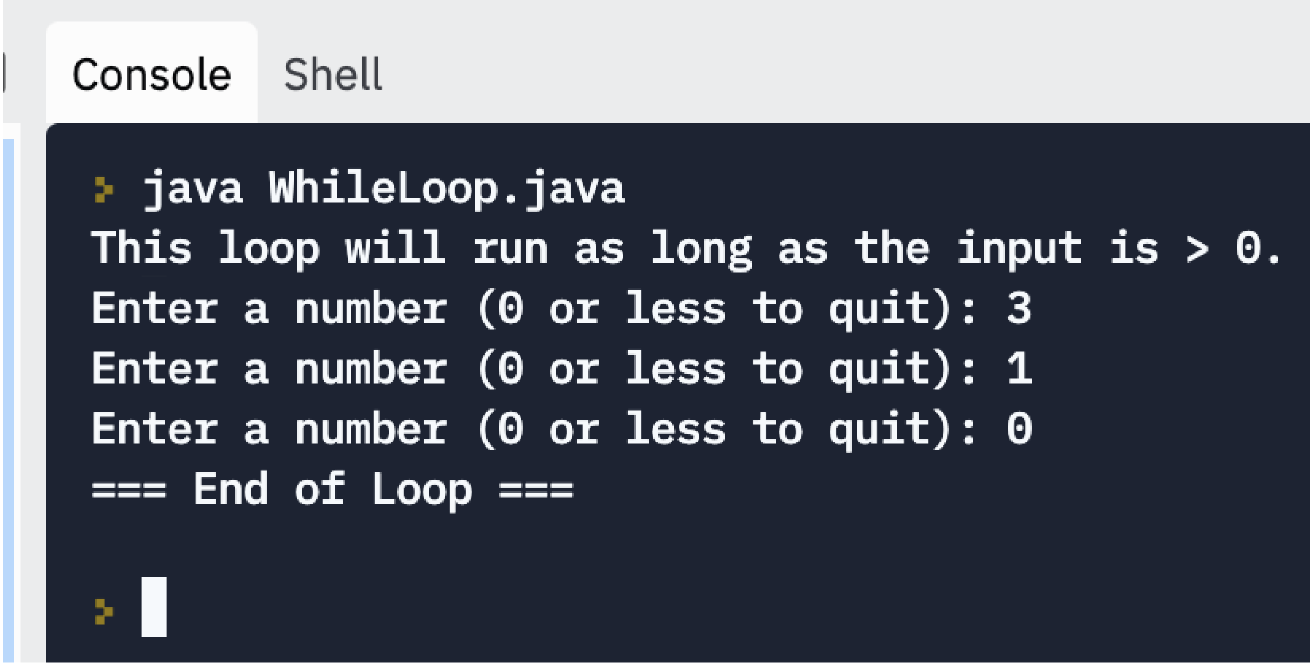
During execution, or runtime, when a while loop is reached first, the program will evaluate the<expression>. This includes determining what conditions are in the expression. If the result is true, the body of the loop will execute. Once all of the statements in the body of the loop are executed, the <expression> is checked again. If it is still true, the body of the loop executes again. This is where the term “loop” comes from, because the last statement in the body of the loop, loops back around to the top. We call each time we execute the body of the loop an iteration. This process keeps running until the <expression> becomes false. When that occurs, the program moves on to the first statement that’s outside of the body of the loop.

Let’s look at a simple example to demonstrate how this works. This program keeps prompting the user to enter a number as long as the entry is greater than 0.

When the user enters a 0 (or a negative number), the program ends as seen below:

```
import java.util.Scanner;

class WhileLoop {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("This loop will run as long as the input is > 0.");
    // Initialize loop variable so the loop will run (> 0)
    int entry = 1;
    while(entry > 0) {
      System.out.print("Enter a number (0 or less to quit): ");
      entry = input.nextInt();
    }
    System.out.println("=== End of Loop ===\n");
  }
}
```

Here is the output from a sample run of the program:



```
> java WhileLoop.java
This loop will run as long as the input is > 0.
Enter a number (0 or less to quit): 3
Enter a number (0 or less to quit): 1
Enter a number (0 or less to quit): 0
=== End of Loop ===

> 
```

The following table describes the iterations in further detail:

Iteration	Description
First Iteration	Because the variable entry is initialized to 1, the loop's condition (entry > 0) evaluates to true, so the loop begins the first iteration. During this iteration, the user is prompted to enter a number, and 3 is entered. The flow of the program returns to the top of the loop.
Second Iteration	In the sample run, the value of entry is 3, which is greater than 0, so the loop's condition evaluates to true. Once again, the code in the body of the loop prompts the user to enter a number, and the user enters 1, which now becomes the value of entry. The program returns to the top of the loop.
Third Iteration	Since the value of entry is 1, which is greater than 0, the loop is executed again. The code in the body of the loop prompts the user for the input of a number, and the user enters 0. This value is assigned to entry. After the third iteration, the value of entry is 0, so the loop's condition evaluates to false. The loop does not run again. Program flow resumes after the end of the body of the loop (the closing curly bracket). The "End of Loop" message is displayed.

Since the expression in the while loop is tested first, it’s possible that it could have been false to begin with, which means the body of the loop would never have run at all. To avoid this problem, the variable entry was initialized to a value greater than 0 so the condition would be true.

CONCEPT TO KNOW

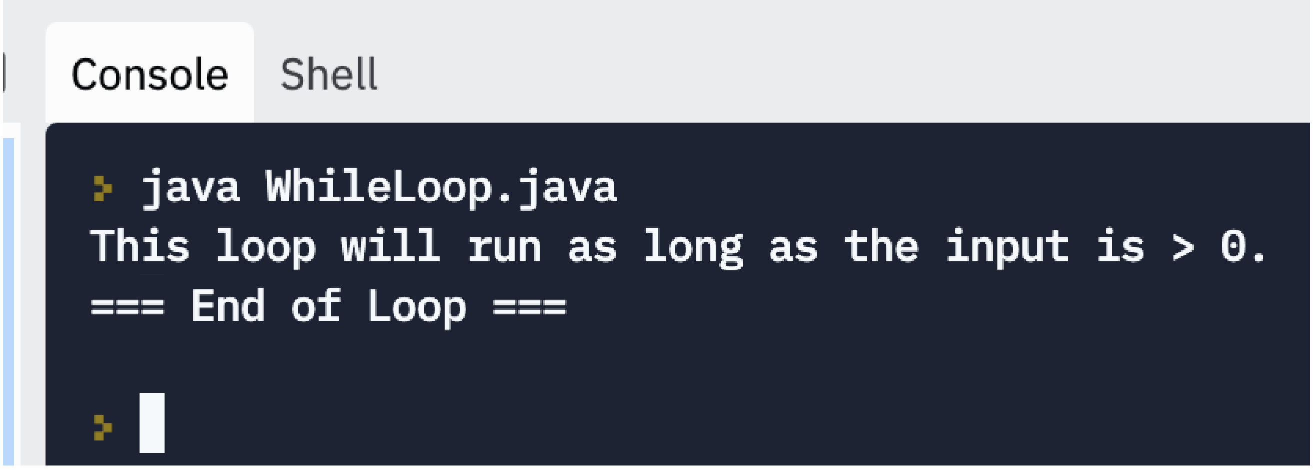
The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. The variable that changes each time the loop executes and controls when the loop finishes is referred to as the **iteration variable**. If there is no iteration variable, the loop will repeat forever, and results in an **infinite loop**. An infinite loop is a loop in which the terminating condition is never satisfied or for which there is no terminating condition.

In the previous example, entry acted as the iteration variable. Each iteration, its value was changed. This was based on user input. This ensured that at some point, the condition would be evaluated as false, thus ending the loop.

In the example below, we’re using the same code, but the variable entry is first set to 0. Since 0 > 0 returns false, the body of the loop never executes:

```
import java.util.Scanner;

class WhileLoop {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("This loop will run as long as the input is > 0.");
    // Initialize loop variable so the loop will run (> 0)
    int entry = 0;
    while(entry > 0) {
      System.out.print("Enter a number (0 or less to quit): ");
      entry = input.nextInt();
    }
    System.out.println("=== End of Loop ===\n");
  }
}
```

The results should look like this:



```
> java WhileLoop.java
This loop will run as long as the input is > 0.
=== End of Loop ===

> 
```

THINK ABOUT IT

During runtime, the while loop condition was checked, found to be false, and so program flow passed out of the loop to the next line of code, which happened to be the final println() method.

TERMS TO KNOW

while Loop
The while loop is one of the most commonly used loops. It keeps going as long as some condition is true.

Body of the Loop
The body of the loop represents the indented block of code that should be executed repeatedly within the loop during each loop iteration.

Iteration Variable
The variable that changes each time the loop executes and controls when the loop finishes.

Infinite Loop
An infinite loop is a loop in which the terminating condition is never satisfied or for which there is no terminating condition.

3. Basics of a for Loop

In Java, definite iteration loops are generally implemented using **for loops**. There are a couple types of for loops. This tutorial is focused on the basic for loop. "Enhanced" for loops will be addressed in a future tutorial.

The pattern for a for loop is:

↪ EXAMPLE

```
for(<initialization>; <condition>; <increment/decrement>) {
   <statement(s)>
 }
```

In the example above, the <initialization>, <condition>, <increment/decrement> and <statement(s)> are terms with angle brackets that are just for information purposes. These are not keywords or actual code. These explain what goes into each of these parts of a for loop.

The <initialization> may declare a numeric variable (int or long) that is used as the loop counter and sets its initial value. If the variable has been declared before the loop, the <initialization> just sets the initial value. If the variable is declared and initialized here, the variable is only accessible in the body of the loop. If the variable is declared before the loop, its value is accessible outside of the loop (as well as inside the body of the loop). It is important to remember that this initialization happens only once at the start of the loop.

The <condition> is a boolean expression. It evaluates true or false based on the result of comparison. The loop executes as long as this expression evaluates as true. The loop stops when this expression becomes false.

HINT

There is a semicolon after the <initialization> and the <condition>.

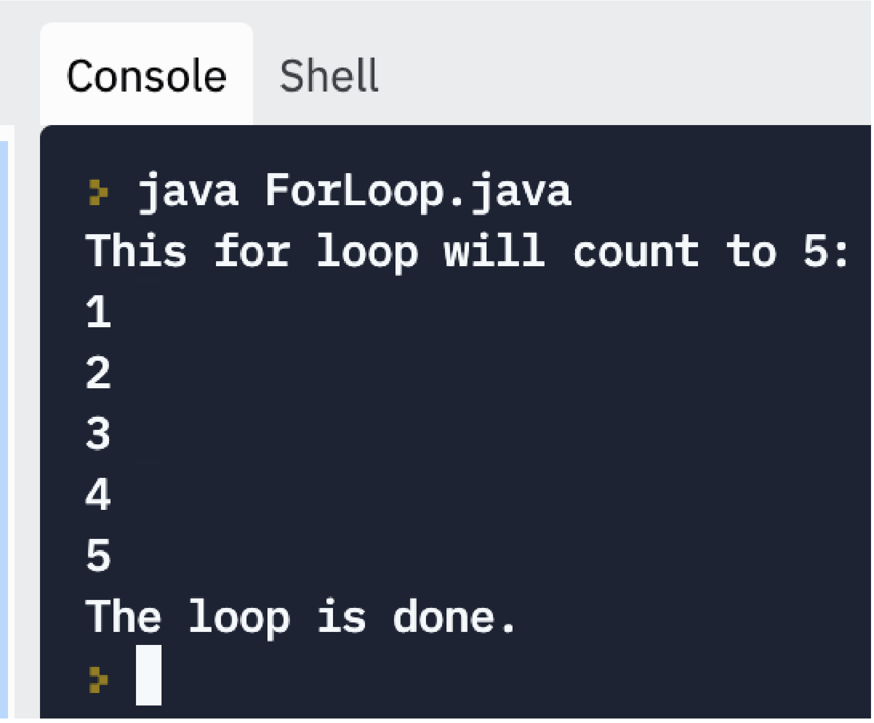
The <increment/decrement> is an expression that defines how the value of the loop variable is modified at the end of each iteration. The most common expression used is the loop variable followed by the increment operator (++), so the loop variable's value is incremented by one after each iteration. It is also possible to change the variable by other values or decrement the variable, but incrementing by one is the most common.

HINT

The <statement(s)> are the statements that will execute each time the loop runs. Consider the following sample program:

```
class ForLoop {
  public static void main(String[] args) {
    System.out.println("This for loop will count to 5: ");
    /* The loop variable, count, starts with a value of 1.
       The loop keeps running as long as count <= 5.
       After each iteration - before returning to the top of the loop
       count is incremented by 1.
    */
    for(int count = 1; count <= 5; count++) {
      System.out.println(count);
    }
    System.out.println("The loop is done.");
  }
}
```

The results should look like this:



```
> java ForLoop.java
This for loop will count to 5:
1
2
3
4
5
The loop is done.
>
```

TERM TO KNOW

for Loop
In Java, the definite iteration loops are generally called for loops.

SUMMARY

In this lesson, you learned about the basics of **iteration**. You learned that **indefinite iteration** is a loop that runs as long as a condition is true and will exit from the loop when that condition becomes false. You also learned that **definite iteration** relies on defined start and stop values. You explored the **basic properties of a while loop** and that these loops are generally used for indefinite conditions where the number of times the loop should be run is not known. Finally, you learned about the **basics of a for loop** and that they are used for definite iterations where we know ahead of time how many times the loop should run or the maximum number of times the loop should run.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Body of the Loop
The body of the loop represents the indented block of code that should be executed repeatedly within the loop during each loop iteration.

Definite Iteration
Definite iteration identifies the number of times, or the maximum number of times, the block of code runs. This should be explicitly defined when the loop actually starts. Typically, this is implemented using

the for loop. The for loop has a specific start and endpoint.

Indefinite Iteration

A loop that does not specify, in advance, how many times it is to be run. It repeats as long as a condition is met. In Java, indefinite loops are typically created using the while loop.

Infinite Loop

An infinite loop is a loop in which the terminating condition is never satisfied or for which there is no terminating condition.

Iteration Variable

The variable that changes each time the loop executes and controls when the loop finishes.

Loop

In Java, the definite iteration loops are generally called for loops.

while Loop

The while loop is one of the most commonly used loops. It keeps going as long as some condition is true.

Loops Using While & Do...While

by Sophia

WHAT'S COVERED

In this lesson, you will learn about patterns that generate while loops when coding an algorithm. Specifically, this lesson covers:

1. Getting Started With While

In the previous tutorial, you briefly learned about the while loop.

As you recall, the structure of the while loop looks like this:

EXAMPLE

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

HINT

In the example above, the <expression> and <statement(s)> terms with outside arrows are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of the parts of a while loop.

The <statements(s)> represents the block of code that should be executed repeatedly. This is also called the body of the loop. The <expression> typically is based on one or more variables that are initialized outside of the loop and then modified within the body of the loop. The while loop continually evaluates the <expression> (the condition) looking for a True or False value. It keeps going (looping) as long as the evaluated condition is True. Once it is False, it exits the loop.

CONCEPT TO KNOW

You also learned about the use of iteration variables. The body of the loop should change the value of one or more variables (in the <expression>) so that eventually the condition becomes false and the loop terminates. The iteration variable is what changes each time the loop executes and controls when the loop finishes. Without an iteration variable to change the condition, the loop would never finish.

Here is a simple program that keeps prompting the user to enter an even number. If the entry is not even, the loop ends:

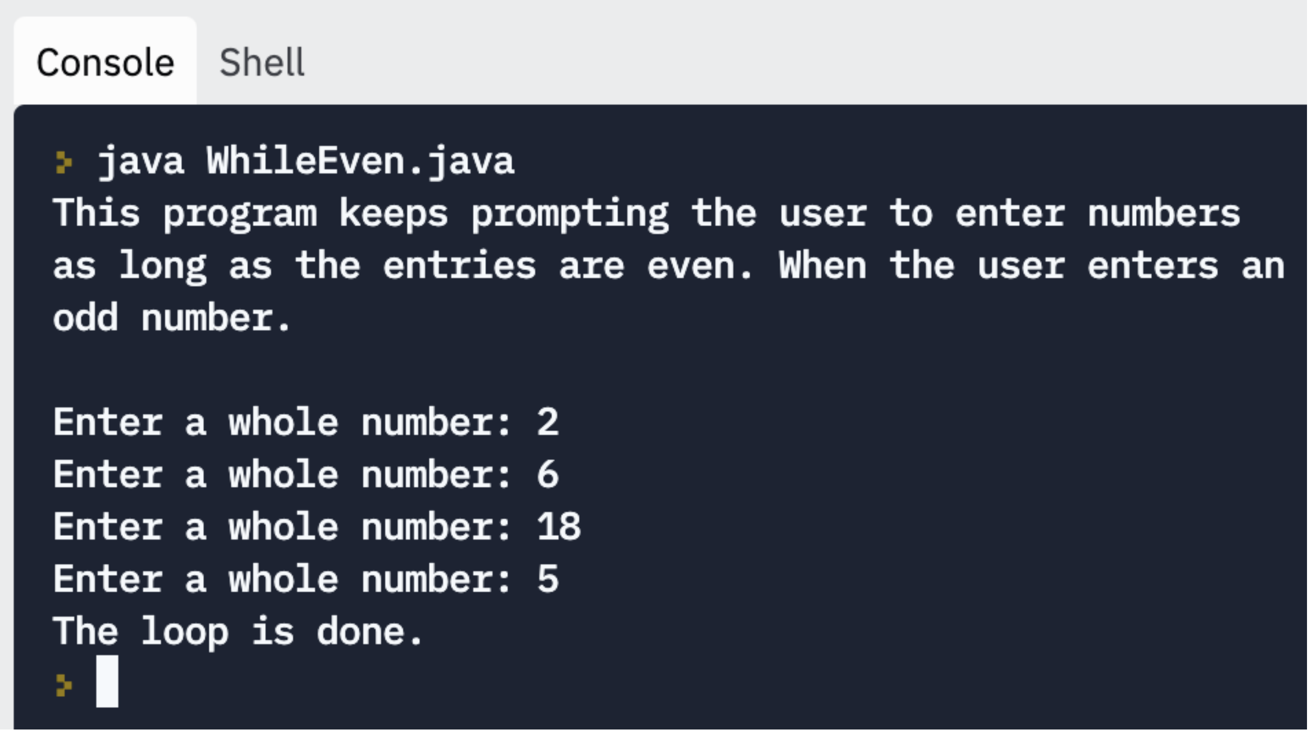
```
import java.util.Scanner;
```

```
class WhileEven {
  public static void main(String[] args) {
    System.out.println("This program keeps prompting the user to enter numbers ");
    System.out.println("as long as the entries are even. When the user enters an ");
    System.out.println("odd number.\n");
    Scanner input = new Scanner(System.in);

    int number = 0;
    while(number % 2 == 0) {
      System.out.print("Enter a whole number: ");
      number = input.nextInt();
    }
    System.out.println("The loop is done.");
  }
}
```

THINK ABOUT IT

The variable number is initialized with a value of 0. Java treats 0 as an even number, so the loop starts running. The variable number is the iteration variable (or loop variable). This variable will change with each loop (iteration) to the latest entry by the user.

Here is a sample run of the program:



```
> java WhileEven.java
This program keeps prompting the user to enter numbers
as long as the entries are even. When the user enters an
odd number.

Enter a whole number: 2
Enter a whole number: 6
Enter a whole number: 18
Enter a whole number: 5
The loop is done.
>
```

CONCEPT TO KNOW

More formally, here is the flow of execution for a while loop:

1. Evaluate the expression, yielding a boolean value of true or false.
2. If the condition is false, exit the while loop and continue execution at the next statement after the loop.
3. If the condition is true, execute the body of the loop and then go and check the condition again.

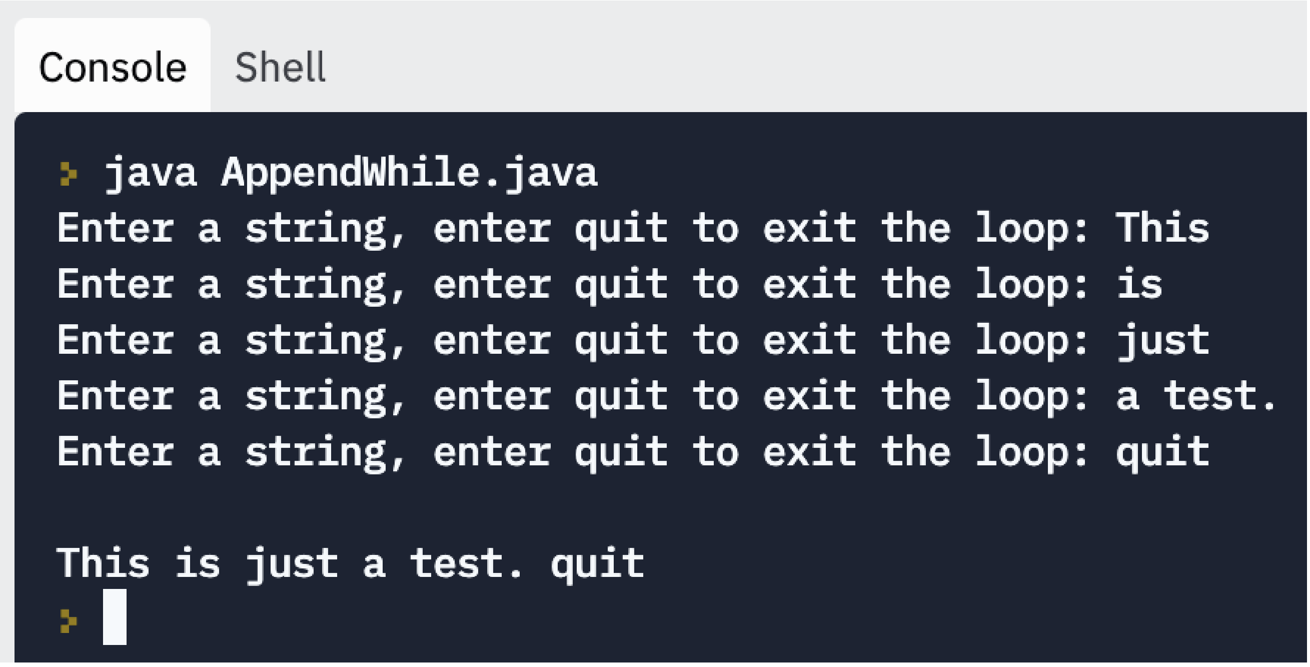
THINK ABOUT IT

Think back to a past tutorial. Remember that each time the body of the loop is executed is an iteration. For the above while loop, we would say, “It had four iterations,” which means that the body of the loop was executed four times.

Here is an example where a text entry is required to exit the loop:

```
import java.util.Scanner;

class AppendWhile {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String textToAdd = "";
    String text = "";

    // Can't use == to check for equality of Strings. Need to use
    // the equals() or equalsIgnoreCase() to compare Strings.
    // Remember that ! means "not"
    while(!textToAdd.equalsIgnoreCase("quit")) {
      System.out.print("Enter a string, enter quit to exit the loop: ");
      textToAdd = input.nextLine();
      // Append input to text with space after it.
      text += textToAdd + " ";
    }
    System.out.println("\n" + text);
  }
}
```

Now, consider the while loop. It loops as long as the entry is not set to the word “quit.” Start by prompting the user for a string and then store it in textEntered. Check if the textEntered is not equal to “quit.” If it isn’t, then the stringBuilder adds the textEntered with a space. Once the user enters the word “quit,” the loop ends, and the text is output to the screen.



```
> java AppendWhile.java
Enter a string, enter quit to exit the loop: This
Enter a string, enter quit to exit the loop: is
Enter a string, enter quit to exit the loop: just
Enter a string, enter quit to exit the loop: a test.
Enter a string, enter quit to exit the loop: quit

This is just a test. quit
> 
```

2. Break and Continue

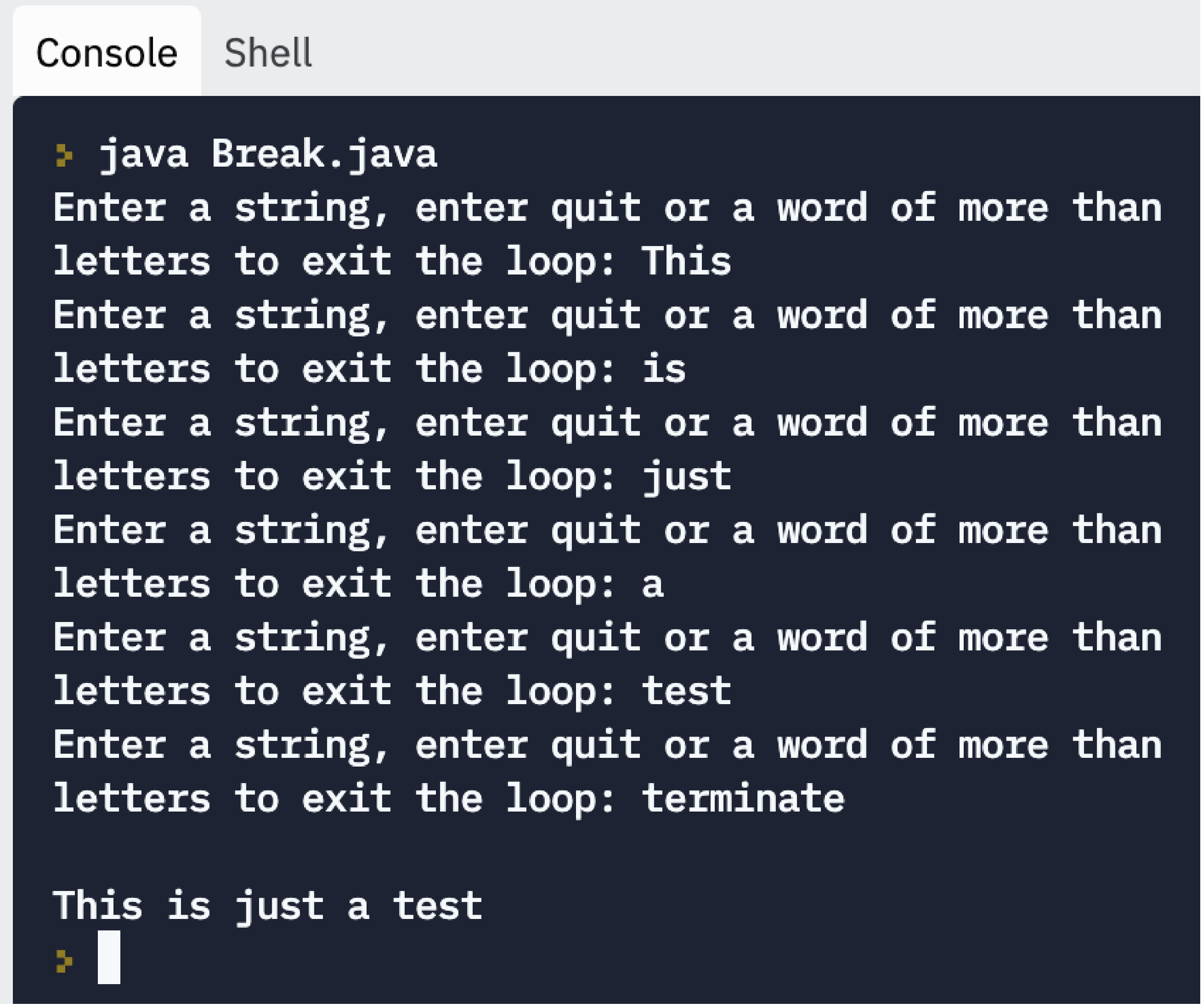
When using while loops so far, the entire body of the loop was executed on each iteration. Java has two reserved keywords that allow a loop to end execution early. The **break statement** can immediately terminate a loop. The program goes to the first statement after the loop. There is also the **continue statement** which ends the current loop iteration and returns to the top of the loop (starting a new iteration). The expression is then evaluated to determine if the loop will execute again, or end there.

Let's see how the break statement works. The next version of the program adds words up to four letters in length until the user enters quit. However, if the user enters a word longer than four letters, the loop ends (using break) and the text is displayed.

Enter the following code in Replit in a file named Break.java:

```
import java.util.Scanner;

class Break {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String textToAdd = "";
    String text = "";

    // Can't use == to check for equality of Strings. Need to use
    // the equals() or equalsIgnoreCase() to compare Strings.
    // Remember that ! means "not"
    // If the user enters a word longer than 4 letters, the loop ends.
    while(!textToAdd.equalsIgnoreCase("quit")) {
      System.out.println("Enter a string, enter quit or a word of more than ");
      System.out.print("letters to exit the loop: ");
      textToAdd = input.nextLine();
      if(textToAdd.length() > 4) {
        break;
      }
      // Append input to text with space after it.
      text += textToAdd + " ";
    }
    System.out.println("\n" + text);
  }
}
```

The results should look like this:



```
› java Break.java
Enter a string, enter quit or a word of more than
letters to exit the loop: This
Enter a string, enter quit or a word of more than
letters to exit the loop: is
Enter a string, enter quit or a word of more than
letters to exit the loop: just
Enter a string, enter quit or a word of more than
letters to exit the loop: a
Enter a string, enter quit or a word of more than
letters to exit the loop: test
Enter a string, enter quit or a word of more than
letters to exit the loop: terminate

This is just a test
›
```

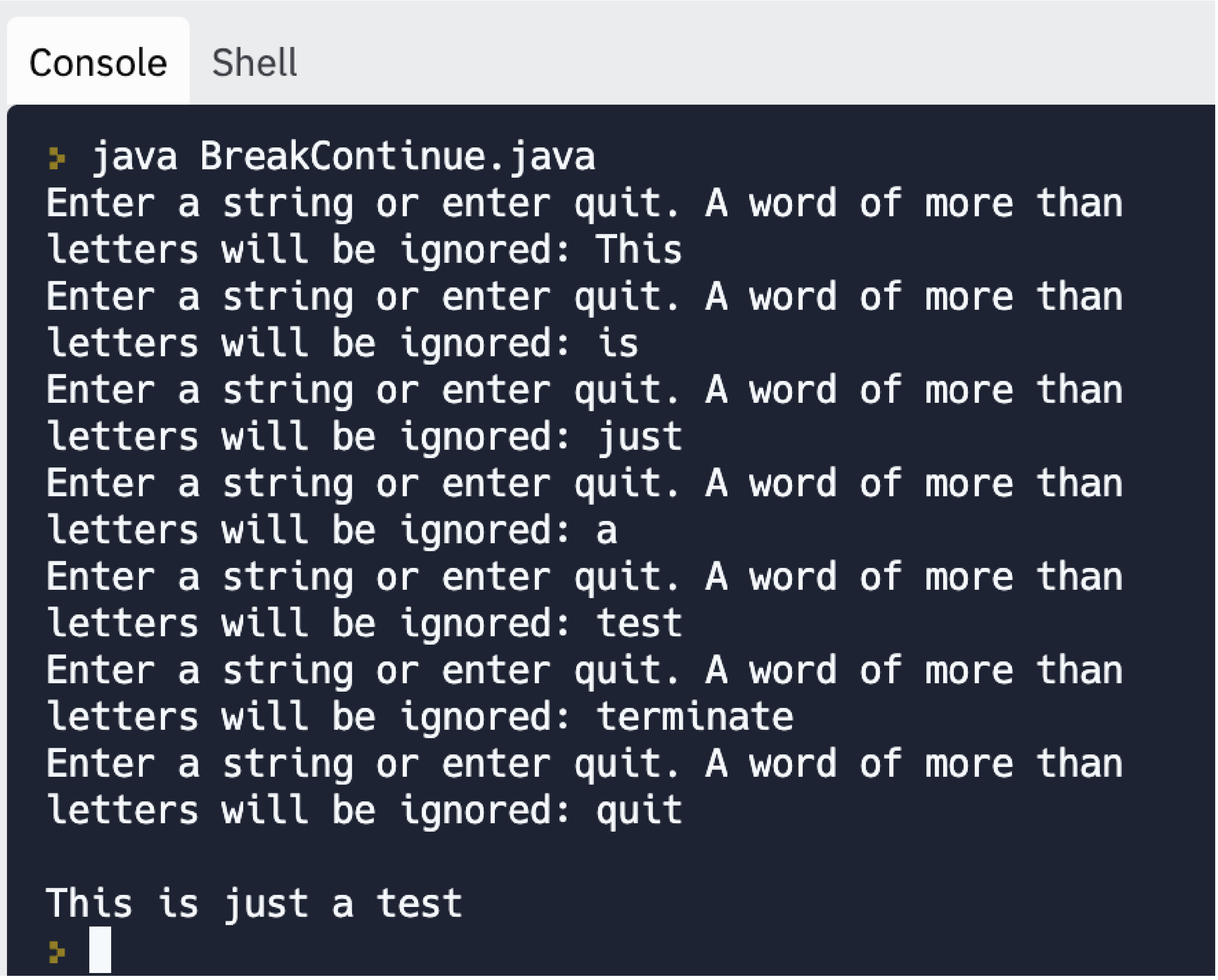
Let’s try the same program, but this time, we'll add thecontinue keyword to skip a word of more than four letters but not quit until the user enters quit.

TRY IT

Directions: Enter the code in Replit in a file named BreakContinue.java:

```
import java.util.Scanner;

class BreakContinue {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String textToAdd = "";
    String text = "";

    // Can't use == to check for equality of Strings. Need to use
    // the equals() or equalsIgnoreCase() to compare Strings.
    // Remember that ! means "not"
    // If the user enters a word longer than 4 letters, the loop ends.
    while(!textToAdd.equalsIgnoreCase("quit")) {
      System.out.println("Enter a string or enter quit. A word of more than ");
      System.out.print("letters will be ignored: ");
      textToAdd = input.nextLine();
      if(textToAdd.length() > 4) {
        continue;
      }
      if(textToAdd.equalsIgnoreCase("quit")) {
        break;
      }
      // Append input to text with space after it.
      text += textToAdd + " ";
    }
    System.out.println("\n" + text);
  }
}
```

The results should look like this:



```
› java BreakContinue.java
Enter a string or enter quit. A word of more than
letters will be ignored: This
Enter a string or enter quit. A word of more than
letters will be ignored: is
Enter a string or enter quit. A word of more than
letters will be ignored: just
Enter a string or enter quit. A word of more than
letters will be ignored: a
Enter a string or enter quit. A word of more than
letters will be ignored: test
Enter a string or enter quit. A word of more than
letters will be ignored: terminate
Enter a string or enter quit. A word of more than
letters will be ignored: quit

This is just a test
›
```

REFLECT

Looking at the code above, think about how the keywords break and continue are used to modify how the steps in the loop are carried out. The statements in the loop are always the same, but break and continue can modify the flow through the loop when needed.

TERMS TO KNOW

break
This is a reserved keyword that creates a break statement for loops. The break statement can immediately terminate a loop entirely and disregard the execution of the loop. When this occurs, the program goes to the first statement/line of code after the loop.

continue
This is a reserved keyword that creates a continue statement for loops. The continue statement will end the current loop iteration, meaning that the execution jumps back to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.

3. do...while Loops

In addition to the basic while loop that has been covered, there is an alternative loop for indefinite iteration. It is referred to as the **do...while loop**. A plain while loop has the condition for the loop at the top of the loop, and the condition is evaluated before each iteration, including the first. Depending on how the variable has been declared and initialized, there is a possibility that the loop may not run at all. The do...while loop puts the condition at the button of the loop, so the loop is guaranteed to run at least once before exiting the loop.

The basic pattern for the do...while loop is:

↱ EXAMPLE

```
do {
  <statement(s)>
} while( <expression>)
```

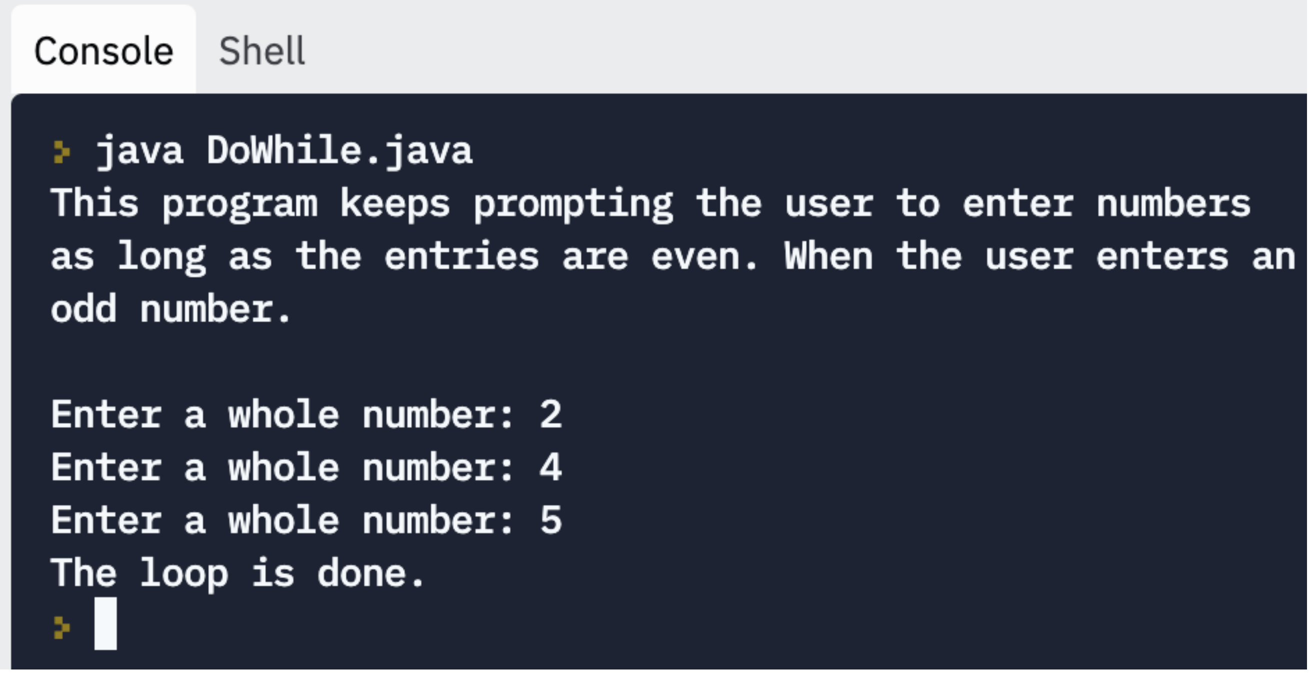
Here is a revised version of the first program in this lesson. Instead of initializing the loop variable before the while loop, this version uses do...while to assign the user's input to the variable. Since the condition is evaluated after the input, the loop is guaranteed to run at least once.

TRY IT

Directions: Try this version of the code in Replit, saved in a file named DoWhile.java:

```
import java.util.Scanner;

class DoWhile {
  public static void main(String[] args) {
    System.out.println("This program keeps prompting the user to enter numbers ");
    System.out.println("as long as the entries are even. When the user enters an ");
    System.out.println("odd number.\n");
    Scanner input = new Scanner(System.in);
    // number not initialized since it will get a value in the body of the loop
    int number;
    do{
      System.out.print("Enter a whole number: ");
      number = input.nextInt();
    }
    while(number % 2 == 0);

    System.out.println("The loop is done.");
  }
}
```

A sample run of the program works like this:



```
> java DoWhile.java
This program keeps prompting the user to enter numbers
as long as the entries are even. When the user enters an
odd number.

Enter a whole number: 2
Enter a whole number: 4
Enter a whole number: 5
The loop is done.
> 
```

REFLECT

When thinking about using a do...while loop, it is important to note how its structure differs from other loop types. With other types of loops, the definition of the condition that controls the loop is in the first line, but where is the key condition for a do...while loop?

As the first code in this lesson shows, it's possible to do the same thing using a plain while loop with a properly initialized variable. A plain while loop also has the advantage of placing the condition at the top of the loop, where it's easily visible, rather than at the bottom of the loop.

TERM TO KNOW

do...while Loop
A do...while loop is a specialized type of while loop where the condition is evaluated at the bottom of the loop rather than the top of the loop, so the loop will always run at least one time.

SUMMARY

In this lesson, you learned about the **while loop** in more detail. You also learned about the **break statement**, which allows us to exit out of a loop, and the**continue statement**, which allows us to jump past the end of the loop of the current iteration and continues back at the loop's conditional check. Finally, you learned about the **do...while version of the loop** as an alternative that guarantees the body of the loop will run at least once.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

break

This is a reserved keyword that creates a break statement for loops. The break statement can immediately terminate a loop entirely and disregard the execution of the loop. When this occurs, the program goes to the first statement/line of code after the loop.

continue

This is a reserved keyword that creates a continue statement for loops. The continue statement will end the current loop iteration, meaning that the execution jumps back to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.

do...while Loop

A do...while loop is a specialized type of while loop where the condition is evaluated at the bottom of the loop rather than the top of the loop, so the loop will always run at least one time.

Loops Using For

by Sophia

WHAT'S COVERED

In this lesson, you will learn about patterns that generate for loops when coding an algorithm. This lesson focuses on the basic or traditional version of the for loop. Specifically, this lesson covers:

1. Reviewing the for Loop

You learned that a for loop is a repetitive control structure. A for loop allows use of a loop that is executed a specific number of times

If you recall, the structure of the for loop looks like this:

EXAMPLE

```
for(<initialization>; <condition>; <increment/decrement>) {
  <statement(s)>
}
```

HINT

In the example above, the <initialization>, <condition>, <increment/decrement> and <statement(s)> terms with outside arrows are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of these parts of a for loop.

The <initialization> may declare a numeric variable (int or long) that is used as the loop counter and sets its initial value. If the variable has been declared before the loop, the <initialization> just sets the initial value. If the variable is declared and initialized here, the variable is only accessible in the body of the loop. If the variable is declared before the loop, its value is accessible outside of the loop (as well as inside the body of the loop).

CONCEPT TO KNOW

It is important to remember that this initialization happens only once at the start of the loop; the variable is initially declared and used to refer to each of the elements in the iterable object (such as a list).

The <condition> is a boolean expression (evaluating to true or false based on the result of comparison). The loop executes as long as this expression evaluates as true. The loop stops when this expression becomes false.

There is a semicolon after the <initialization> and the <condition>.

The <increment/decrement> is an expression that defines how the value of the loop variable is modified at the end of each iteration. The most common expression used is the loop variable followed by the **increment operator (++)**, so the loop variable's value is incremented by one after each iteration. It is also possible to change the variable by other values or show an example of the variable, but incrementing by one is the most common.

CONCEPT TO KNOW

The syntax of a for loop is similar to the while loop in that there is a for loop statement and then the body of the loop.

TERM TO KNOW

Increment Operator (++)
The ++ operator increases the value in the variable immediately to the left of the operator by one.

2. Uses of the for Loop

There are many scenarios where the for loop is valuable when programming in Java. These include, but are not limited to:

- Counting elements within a list,
- Computing totals based on a list, or
- Finding the largest and smallest elements within the list.

Any time that each element in an array needs to be reviewed, it is a perfect time to use a for loop.

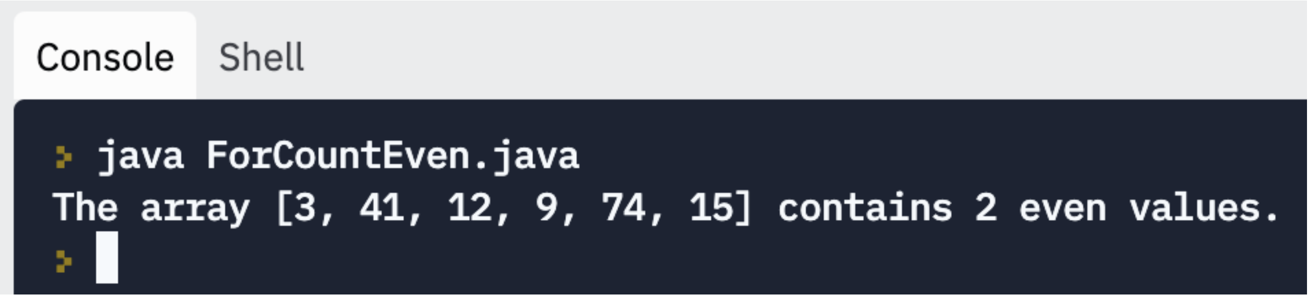
2a. Counting Certain Elements in an Array

To count the number of even values in an array, write the following code saved in a file named ForCountEven.java.

Note the setup of the for loop:

```
import java.util.Arrays;

class ForCountEven {
  public static void main(String[] args) {
    int[] numbers = {3, 41, 12, 9, 74, 15};
    // Counter initialized to 0
    int evenCount = 0;

    // Loop variable starts at 0, since first element index in array is 0.
    // Remember the last index is 1 less than length of array.
    for(int index = 0; index < numbers.length; index++) {
      if(numbers[index] % 2 == 0) {
        evenCount++;
      }
    }
    // Display result
    System.out.print("The array " + Arrays.toString(numbers) + " ");
    System.out.println("contains " + evenCount + " even values.");
  }
}
```

Running this program should produce the following output:



```
> java ForCountEven.java
The array [3, 41, 12, 9, 74, 15] contains 2 even values.
> 
```

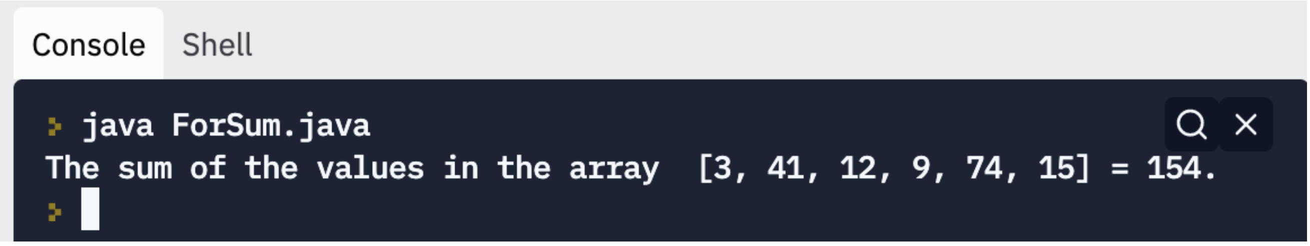
2b. Computing Totals

For loops are also useful in adding up the values in an array. This can be an important step in carrying more complex calculations, such as finding an average.

Another similar loop that computes the total of an array of numbers is as follows:

```
import java.util.Arrays;

class ForSum {
  public static void main(String[] args) {
    int[] numbers = {3, 41, 12, 9, 74, 15};
    // Sum initialized to 0
    int sum = 0;
    // Loop variable starts at 0, since the first element index in array is 0.
    // Remember that last index is 1 less than length of array.
    for(int index = 0; index < numbers.length; index++) {
        sum += numbers[index];
    }
    // Display result
    System.out.print("The sum of the values in the array  ");
    System.out.print(Arrays.toString(numbers));
    System.out.println(" = " + sum + ".");
  }
}
```

The output from this program should look like this:



```
> java ForSum.java
The sum of the values in the array  [3, 41, 12, 9, 74, 15] = 154.
> 
```

☆ BIG IDEA

The variable sum was set to zero before the loop starts. In this loop, the iteration variable was used in the body of the loop to access each item in the array in turn and add it cumulatively to the sum variable. As the loop executes, the `sum` variable accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

Similar to the while loop, the `break` and `continue` statements can be used in for loops, and they will work in the same way. The break statement will immediately end the for loop and execute the first statement after the end of the loop. The `continue` statement will end the current iteration of the loop and go back to the start of the loop again to evaluate the expression's condition.

SUMMARY

In this lesson, you reviewed the **for loop and its uses** in more detail. This included its use as a **definite loop**. You used the for loop to both count and total elements of a list. You learned how a for loop can be used to **count the largest or smallest element in an array** as well. You learned that a for loop can also use the continue and break statements to change the flow of the loop. Finally, you learned about **computing totals**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Increment Operator (++)
The ++ operator increases the value in the variable immediately to the left of the operator by one.

Loops Using Enhanced For (for each iteration)

by Sophia

WHAT'S COVERED

In this lesson, you will learn about patterns that generate for loops when coding an algorithm. Specifically, this lesson covers:

1. Enhanced for Loops

The **enhanced for loop**, also known as a **for-each loop** or **range-based** for loop in other programming languages, is a special version of the for loop that is designed to work with Java arrays and collections. It provides a simpler and safer way of iterating over these data structures than using a plain for loop.

Java's enhanced for loop uses the for keyword, but the contents of the parentheses are different.

The basic syntactic pattern is:

EXAMPLE

```
for(<variable> : <array/collection>) {
 <statements>
}
```

This style of for loop iterates the values in <array/collection>. Each value is accessible in turn via the <variable>. The statements in the body of the loop have access to the <variable> and can even change it in the loop, but changes made in the body of the loop to <variable> don't affect the original array or collection.

TERM TO KNOW

Enhanced for Loop (also called a for-each Loop or Range-Based for Loop)
A special version of the Java for loop designed to work with arrays and collections that provides a simpler way to set up a for loop.

---

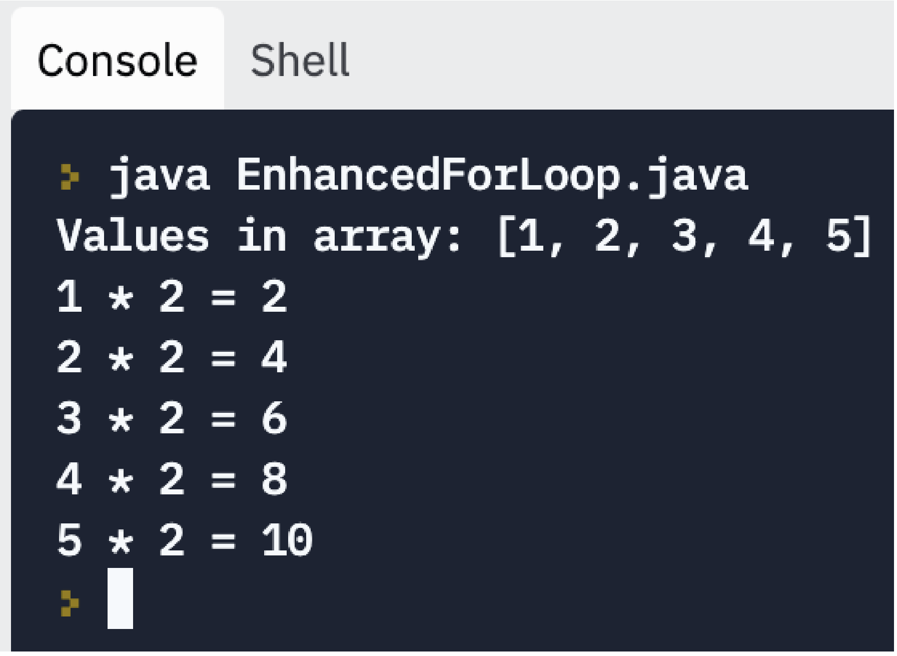
2. Enhanced for Loops With Arrays

The enhanced for loop works well with Java arrays because when working with this type of loop, the compiler works out the size of the array. This means the loop will run automatically once for each element in the array without needing to write code that checks the size of the array (the middle term in the three-part definition of a standard for loop).

Here is a sample program that uses an enhanced for loop to loop over the values in an array and carry out calculations using the values:

```
import java.util.Arrays;

class EnhancedForLoop {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    System.out.println("Values in array: " + Arrays.toString(numbers));
    for(int number : numbers) {
      /* value can be used in statements in loop body */
      System.out.println(number + " * 2 = " + number * 2);
    }
  }
}
```

The program above should produce the following output:



```
> java EnhancedForLoop.java
Values in array: [1, 2, 3, 4, 5]
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
> 
```

To see how the enhanced for loop makes the code a bit simpler, compare the code above with this version that uses a plain for loop:

```
import java.util.Arrays;

class EnhancedForLoop {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    System.out.println("Values in array: " + Arrays.toString(numbers));
    for(int number = 0; number < numbers.length; number++) {
      System.out.println(numbers[number] + " * 2 = " + numbers[number] * 2);
    }
  }
}
```

Here is the start of the enhanced loop:

EXAMPLE

```
for(int number : numbers) {
```

Now compare it with the start of the plain for loop:

EXAMPLE

```
for(int number = 0; number < numbers.length; number++) {
```

Note, the following line from the enhanced for loop version simplifies access to the array contents:

EXAMPLE

```
System.out.println(number + " * 2 = " + number * 2);
```

Compare this to the corresponding line in the code using a plain for loop:

EXAMPLE

```
System.out.println(numbers[number] + " * 2 = " + numbers[number] * 2);
```

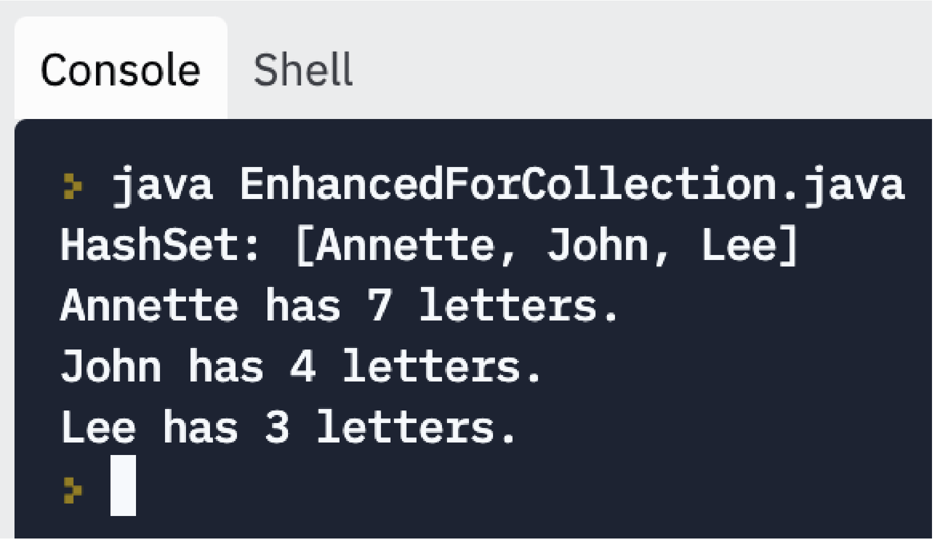
3. Enhanced for Loops With ArrayLists and HashSets

An enhanced for loop can also be used with some collections. A collection has to be**iterable** to be used with an enhanced for loop. A collection is iterable if the collection provides a mechanism for the items in the collection to be accessed in a fixed order.

An ArrayList is iterable, and here is some sample code using anArrayList:

```
import java.util.ArrayList;

class EnhancedForCollection {
  public static void main(String[] args) {
    ArrayList<String> names = new ArrayList<>();
    names.add("Annette");
    names.add("John");
    names.add("Lee");
    System.out.println("ArrayList: " + names.toString());
    for(String name : names) {
      System.out.println(name + " has " + name.length() + " letters.");
    }
  }
}
```

When run, this program produces results like this:

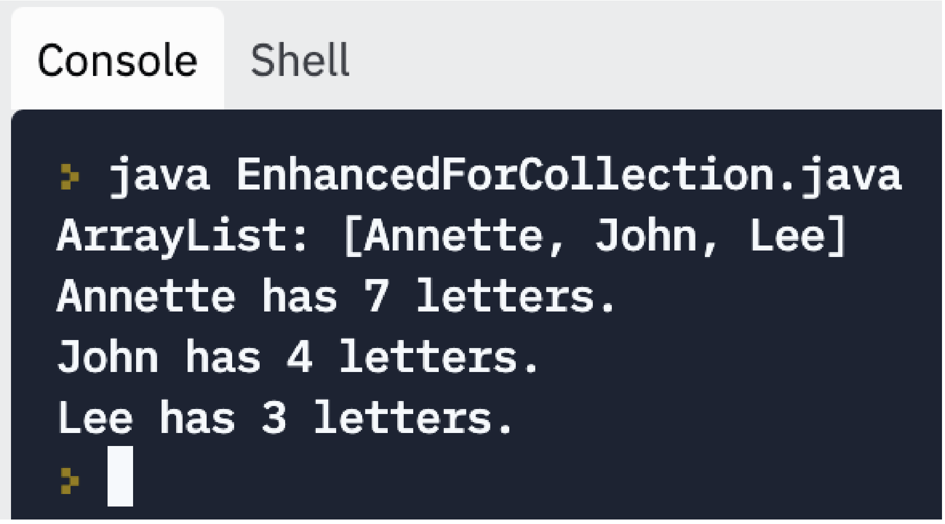


```
> java EnhancedForCollection.java
HashSet: [Annette, John, Lee]
Annette has 7 letters.
John has 4 letters.
Lee has 3 letters.
>
```

The enhanced for loop works with other collection types, too. Here is a version of the previous program using a HashSet:

```
import java.util.HashSet;

class EnhancedForCollection {
  public static void main(String[] args) {
    HashSet<String> names = new HashSet<>();
    names.add("Annette");
    names.add("John");
    names.add("Lee");
    System.out.println("HashSet: " + names.toString());
    for(String name : names) {
      System.out.println(name + " has " + name.length() + " letters.");
    }
  }
}
```

The output from this program should be nearly the same as the previous one:



```
> java EnhancedForCollection.java
ArrayList: [Annette, John, Lee]
Annette has 7 letters.
John has 4 letters.
Lee has 3 letters.
>
```

TERM TO KNOW

Iterable
A collection is iterable if the collection provides a mechanism for the items in the collection to be accessed in a fixed order.

---

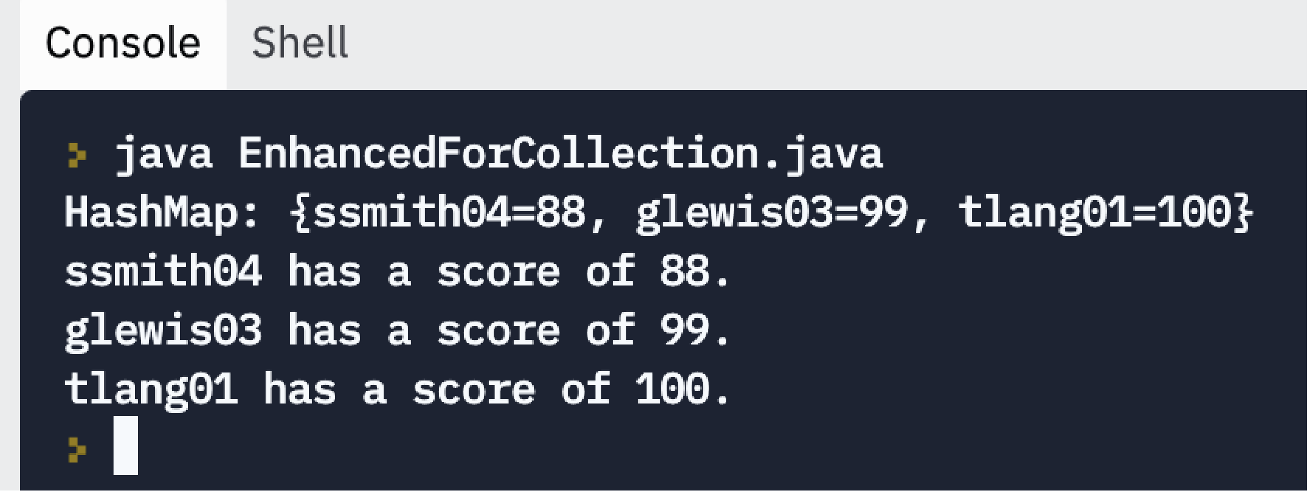
4. Enhanced for Loops With HashMaps

The HashMap collection type is not iterable. It cannot be directly looped over using an enhanced for loop. However, there is a workaround. The HashMap class has a **.keySet()** method that retrieves the Set of keys from the HashMap. This Set of keys is iterable, so it can be used with an enhanced for loop.

With a key value, the HashMap's get() method can be used to get the value corresponding to the key:

```
import java.util.HashMap;

public class EnhancedForCollection {

  public static void main(String[] args) {
    // HashMap holds key-value pairs.
    // The key (user ID) is a String (case sensitive).
    // The value (score) is an Integer (int)
    HashMap<String, Integer> scores = new HashMap<>();
    scores.put("ssmith04", 88);
    scores.put("tlang01", 100);
    scores.put("glewis03", 99);
    System.out.println("HashMap: " + scores.toString());
    /* A HashMap isn't iterable, but the keySet() method returns
       a Set with the keys in the HashMap, which can be interated
       over. The value can be retrieved using the key value
    */
    for(var key : scores.keySet()) {
      // For each key retrieve the value (score) & print
      System.out.println(key + " has a score of " + scores.get(key) + ".");
    }
  }
}
```

The code above should produce this output:



```
> java EnhancedForCollection.java
HashMap: {ssmith04=88, glewis03=99, tlang01=100}
ssmith04 has a score of 88.
glewis03 has a score of 99.
tlang01 has a score of 100.
> 
```

TERM TO KNOW

keySet()

The keySet() method returns a Java Set collection containing the keys used in a HashMap. This Set of keys is iterable, unlike the HashSet itself, which is not.

SUMMARY

In this lesson, you learned about Java's **enhanced for loop**. You learned about the use of **enhanced for loops with arrays**. You also learned about using enhanced for loops to iterate over Java collections such as **ArrayLists, HashSets, and HashMaps**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from "Python for Everybody" By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Enhanced for Loop (also called a For - Each loop or Range-Based for Loop)

A special version of the Java for loop designed to work with arrays and collections that provides a simpler way to set up a for loop.

Iterable

A collection is iterable if the collection provides a mechanism for the items in the collection to be accessed in a fixed order.

keySet()

The keySet() method returns a Java Set collection containing the keys used in a HashMap. This Set of keys is iterable, unlike the HashSet itself, which is not.

Nested Loops

by Sophia

WHAT'S COVERED

In this lesson, you will learn about loops created within another loop, known as nested loops. Specifically, this lesson covers:

1. Nested Loops

At a high level, a **nested loop** is just a loop within another loop. This is very similar to nested conditional statements, which was discussed in the previous Challenge 1.3. In that challenge, you used nested if statements nested in other if statements. The same nesting idea can be used on loops as well.

CONCEPT TO KNOW

A nested loop can be quite useful when there is a series of statements that includes a loop that you want to have repeated.

The functionality is no different than using a single loop except that the**“outer” loop** has one or more **“inner” loops** within it. There are many situations in which using nested loops are beneficial—for example, looping through two-dimensional arrays, which will be covered later in this tutorial. Another great example could be prompting a teacher for numeric grades to calculate the average grade for a student. However, instead of prompting a single student’s grades, it could prompt the entire class. To handle this, you would add an outer loop that loops across all of the students, while the inner loop prompts the grades for each student.

The format of nested while loops would look something like this:

EXAMPLE

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

And the format of nested for loops would look something like this:

EXAMPLE

```
for(<initialization>; <condition>; <increment/decrement>) {
  for(<initialization>; <condition>; <increment/decrement>) {
    <statement(s)>
  }
  <statement(s)>
}
```

In the examples above (as when these were first presented covering the different types of loops), the `<expression>`, `<variable>`, `<iterable>`, and `<statement(s)>` terms in angle brackets are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of the parts for the nested loop examples. If you notice in the nested examples, the inner loop would be executed one time for each iteration of the outer loop.

THINK ABOUT IT

What do you think would happen if the curly brackets were not paired correctly or not nested correctly?

If the curly brackets are not paired correctly, there could be syntax errors that keep the code from compiling. If the loops and their brackets are not nested correctly, it is possible that the code may compile but not run as expected. The placement of the curly brackets determines which loop a line of code is part of. Using correct indentation will help make sure the loops and brackets are organized correctly. This is especially true as the number of loops increases and selection statements with their blocks of code come into play.

There is no limitation on the use of nested loops or their iterations. In a nested loop, the number of total iterations of all loops included will be the total of the number of iterations in the outer loop multiplied by the iterations of the inner loop.

EXAMPLE

100 x 100 = 10,000

In each iteration of the outer loop, the inner loop will execute all of its iterations. Then, for each iteration of an outer loop, the inner loop will restart and complete its execution before the outer loop can continue to its next iteration. These nested loops are especially useful when it comes to working with multidimensional arrays.

TERMS TO KNOW

Nested Loop
A nested loop is any loop that occurs inside the body of another loop.

Outer Loop
A loop that contains one or more other loops in its body.

Inner Loop
A loop that is contained in the body of another loop.

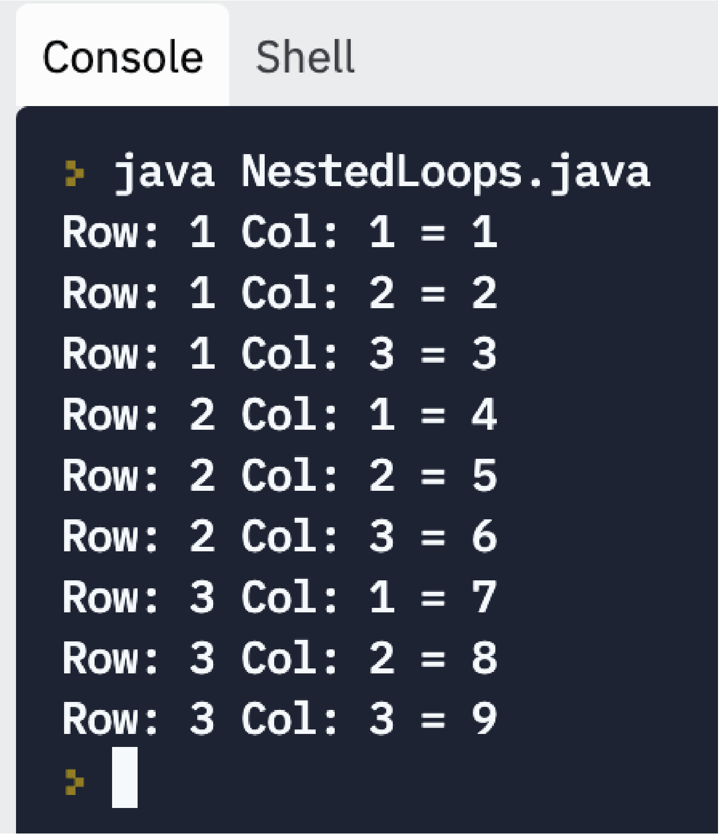
2. Nested for Loops

A two-dimensional array can be conceived of as an array of arrays, with each row in the array being a one-dimensional array. This is done by looping over the contents of an array, by first looping over the rows in the array. The array that makes up a given row then needs to be looped, or iterated over. This means that for each iteration of the loop that steps through the rows, there is a nested loop that goes through the elements in a given row. Since the number of rows and columns in a two-dimensional array is known, for loops are the appropriate choice for the outer and the inner loop.

Let's look at a simple example of a two-dimensional array and how to use nested for loops to print it out:

```
class NestedLoops {
  public static void main(String[] args) {
    int[][] numbers = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
    };
    // Outer loop iterates over rows
    for(int row = 0; row < numbers.length; row++ ) {
      // Inner loop iterates over columns in each row
      for(int col = 0;  col < numbers[row].length; col++) {
        /* (row + 1) displays row # starting with 1 rather than 0.
           (col + 1) displays column # starting with 1 rather than 0.
           Parentheses needed so expression is evaluated before printing.
           These expressions do not change the values of row and col so
           array access works as expected. */
        System.out.println("Row: " + (row + 1) + " Col: " + (col + 1) +
          " = " + numbers[row][col]);
      }
    }
  }
}
```

The outer loop iterates over the three rows in the array. As each row is accessed, the inner loop iterates over the three columns like this:



```
> java NestedLoops.java
Row: 1 Col: 1 = 1
Row: 1 Col: 2 = 2
Row: 1 Col: 3 = 3
Row: 2 Col: 1 = 4
Row: 2 Col: 2 = 5
Row: 2 Col: 3 = 6
Row: 3 Col: 1 = 7
Row: 3 Col: 2 = 8
Row: 3 Col: 3 = 9
> 
```

Note how the outer loop gets the number of rows using this expression:

↪ EXAMPLE

```
numbers.length
```

Since the first dimension of a two-dimensional array is the number of rows, getting the length of the first dimension returns the number of rows.

The inner loop gets the number of columns in each row using this expression:

↪ EXAMPLE

```
numbers[row].length
```

Remember that the second dimension is the number of columns in each row. Using rows as the first dimension, we can then use length to get the number of columns in each row.

3. Nested Enhanced for Loops

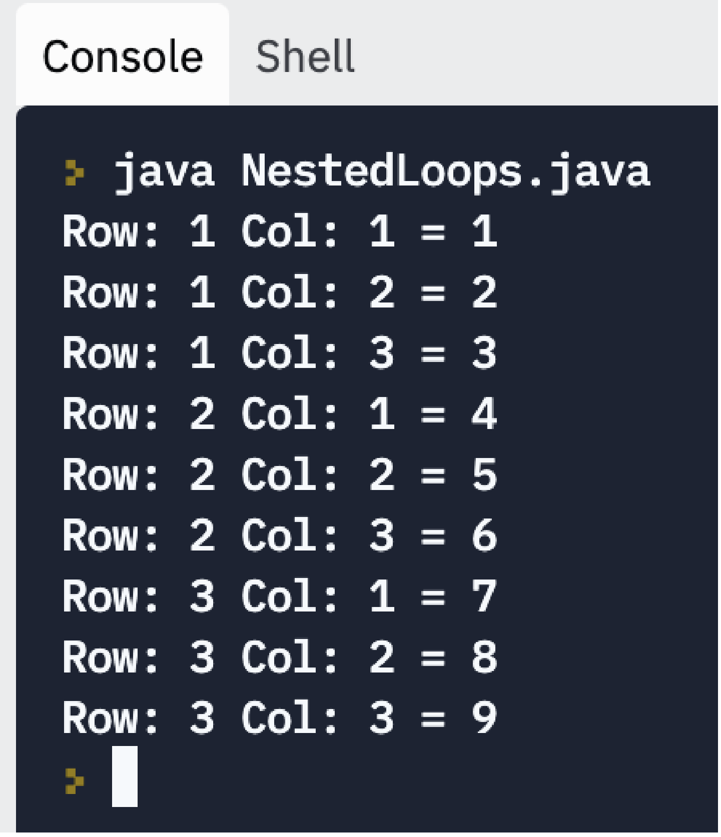
Compared to "plain" for loops, we have seen how enhanced for loops can simplify the code for iterating over an array or collection. Nested enhanced for loops can be used to iterate over a two-dimensional array, as this sample code shows.

TRY IT

Directions: Try typing in the following code in Replit in a file named NestedLoops.java:

```
class NestedLoops {
  public static void main(String[] args) {
    int[][] numbers = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
    };
    // Without counters in loops, need rowNumber, colNumber
    int rowNumber = 1;
    int colNumber = 1;
    // Outer loop iterates over rows. Each row is a single-dimensional array
    for(int[] row : numbers) {
      // Inner loop iterates over columns in each row
      for(int value : row) {
        // Note space between colNumber++ and following +
        // Remember that ++ is increment operator, + is concatenation operator
        System.out.println("Row: " + rowNumber + " Col: " + colNumber++ +
          " = " + value);
      }
      colNumber = 1; // Reset colNumber after down with columns in row
      rowNumber++; // increment rowNumber after done with row
    }
  }
}
```

Running the code should produce the following output:



```
> java NestedLoops.java
Row: 1 Col: 1 = 1
Row: 1 Col: 2 = 2
Row: 1 Col: 3 = 3
Row: 2 Col: 1 = 4
Row: 2 Col: 2 = 5
Row: 2 Col: 3 = 6
Row: 3 Col: 1 = 7
Row: 3 Col: 2 = 8
Row: 3 Col: 3 = 9
> 
```

REFLECT

Nested loops may seem more complex at first, but how does using a pair of nested for loops simplify the processing of the Tic-Tac-Toe board in this code?

Since the enhanced for loops don't have counter variables, note the need to declare and initialize rowNumber and colNumber before the start of the loop. After the inner loop runs each time, the value of colNumber needs to be reset and the value of rowNumber needs to be incremented.

There is also an important detail to note in this statement:

EXAMPLE

```
System.out.println("Row: " + rowNumber + " Col: " + colNumber++ +
  " = " + value);
```

The ++ is the increment operator that increases the value of colNumber by one after it has been accessed. The ++ is then followed by the concatenation operator (+).

BIG IDEA

Java doesn't recognize a triple plus sign as an operator, and the increment operator needs to be appended to the end of the variable name, since it is essential to include the space between the ++ and the + for the line to be syntactically valid and work as expected.

SUMMARY

In this lesson, you learned about using loops within other loops, also known as **nested loops**. You also learned that using loops within other loops, or **nested for loops**, is similar to the nested conditional statements. You learned that **nested enhanced for loops** can be quite useful when you have a series of statements that includes a loop that you want to have repeated. You discovered that this can be useful when working with multidimensional arrays. Finally, you created and ran a few programs using nested loops.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Inner Loop
A loop that is contained in the body of another loop.

Nested Loop
A nested loop is any loop that occurs inside the body of another loop.

Outer Loop

A loop that contains one or more other loops in its body.

Debugging Loops

by Sophia

WHAT'S COVERED

This lesson focuses on debugging issues that can occur with creating and using loops. Specifically, this lesson covers:

1. Infinite Loop

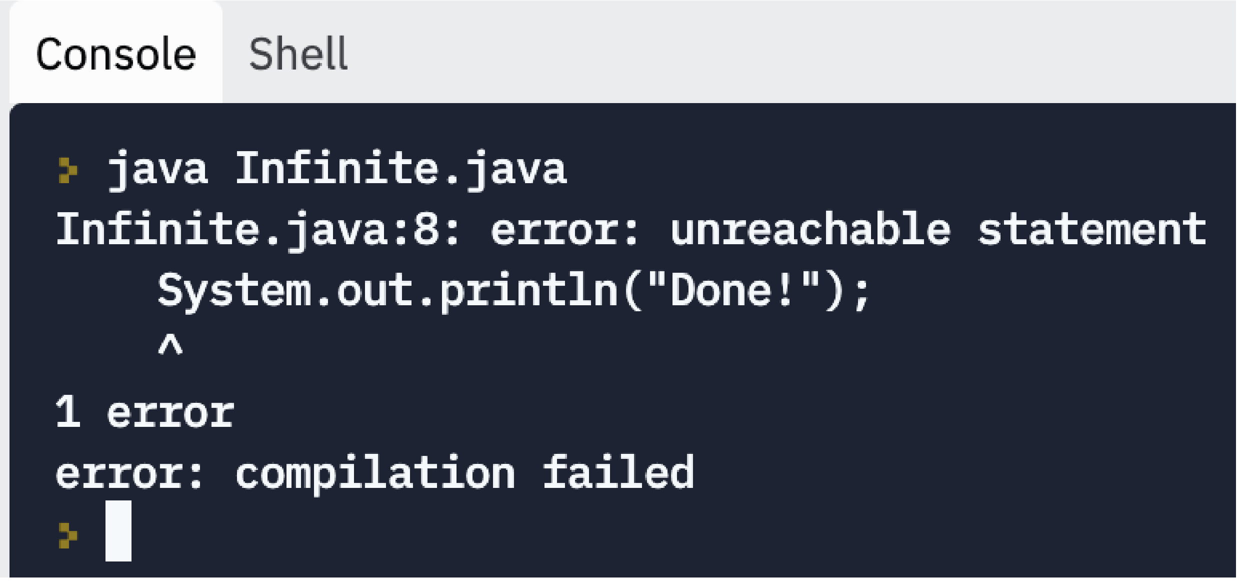
An endless source of amusement for programmers is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an **infinite loop**. In this example, there is no iteration variable telling you how many times to execute the loop. In programming, an infinite loop is a loop that includes no mechanism for ending the loop.

TRY IT

Directions: Sometimes Java will catch an infinite and treat it as an error. Run the following code:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(true) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    System.out.println("Done!");
  }
}
```

Trying to run this code produces a compiler error:



```
> java Infinite.java
Infinite.java:8: error: unreachable statement
    System.out.println("Done!");
    ^
1 error
error: compilation failed
>
```

REFLECT

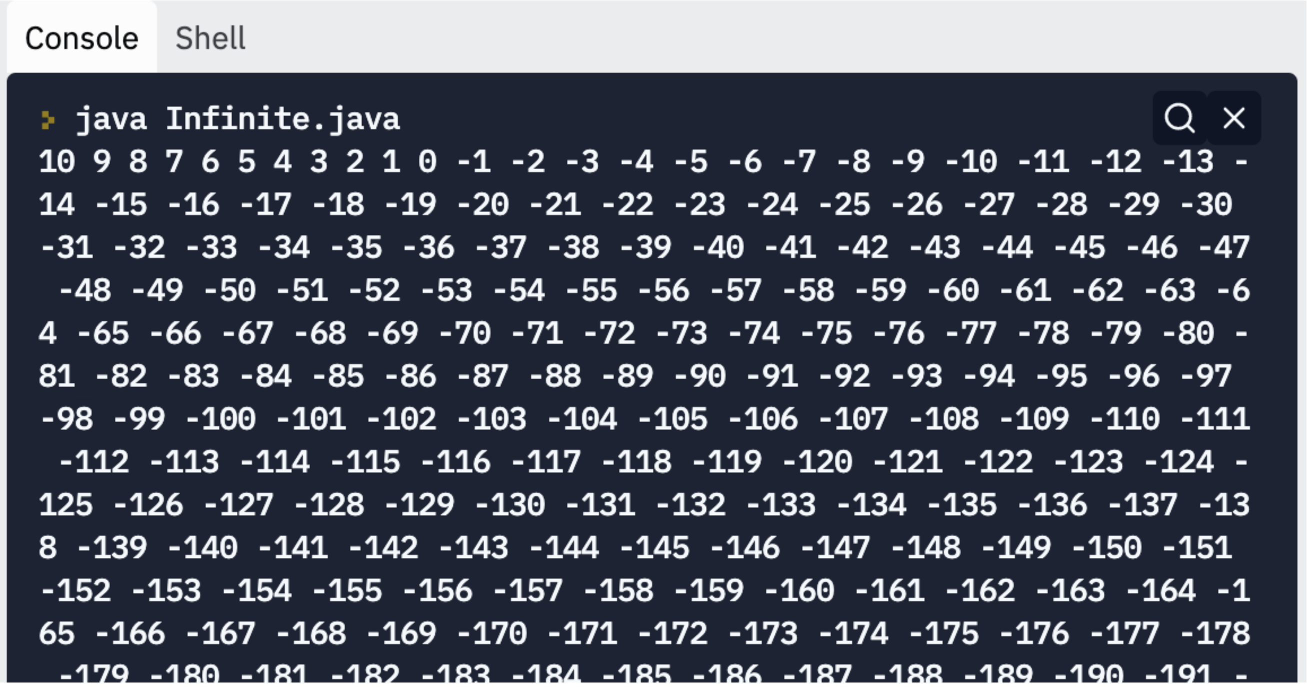
The detection of a line of code that can never be reached is a good catch by the compiler that calls attention to the problem of an infinite loop.

TRY IT

Directions: If the println() statement is removed or commented out, though, the compiler will no longer object and the loop will run forever. Run the code below:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(true) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    //System.out.println("Done!");
  }
}
```

As this screen shot shows, the loop will now run indefinitely:



```
› java Infinite.java
10 9 8 7 6 5 4 3 2 1 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 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -6
4 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -
81 -82 -83 -84 -85 -86 -87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97
-98 -99 -100 -101 -102 -103 -104 -105 -106 -107 -108 -109 -110 -111
 -112 -113 -114 -115 -116 -117 -118 -119 -120 -121 -122 -123 -124 -
125 -126 -127 -128 -129 -130 -131 -132 -133 -134 -135 -136 -137 -13
8 -139 -140 -141 -142 -143 -144 -145 -146 -147 -148 -149 -150 -151
-152 -153 -154 -155 -156 -157 -158 -159 -160 -161 -162 -163 -164 -1
65 -166 -167 -168 -169 -170 -171 -172 -173 -174 -175 -176 -177 -178
 -179 -180 -181 -182 -183 -184 -185 -186 -187 -188 -189 -190 -191 -
```

REFLECT

If you make the mistake and run this code, you will learn quickly how to stop a runaway Java process on your system or find where the power-off button is on your computer. In Replit, you can press Ctrl - C in the console to end the program.

We could have also done the same thing by creating a condition where it is always true without using the true value. Since n is starting at 10 and the value is decrementing, in the example below, n will always be < 20, so it will loop infinitely in the same way that a true boolean constant would.

TRY IT

Directions: Here is an example of an infinite loop being run the same as a true boolean constant. Run the code below:

```
class Infinite {
  public static void main(String[] args) {
    int n = 10;
    while(n < 20) {
      // Print out n & then decement it
      System.out.print(n-- + " ");
    }
    System.out.println("Done!");
  }
}
```

While this is a dysfunctional infinite loop, you can still use this pattern to build useful loops. This requires you to carefully add code to the body of the loop. This code should explicitly exit the loop using a break statement when we have reached the exit condition where the loop condition evaluates to false.

REFLECT

An incorrectly constructed loop in one program or context may be the correct loop to use in different code. Can you think of a situation where the loop above could be useful?

TRY IT

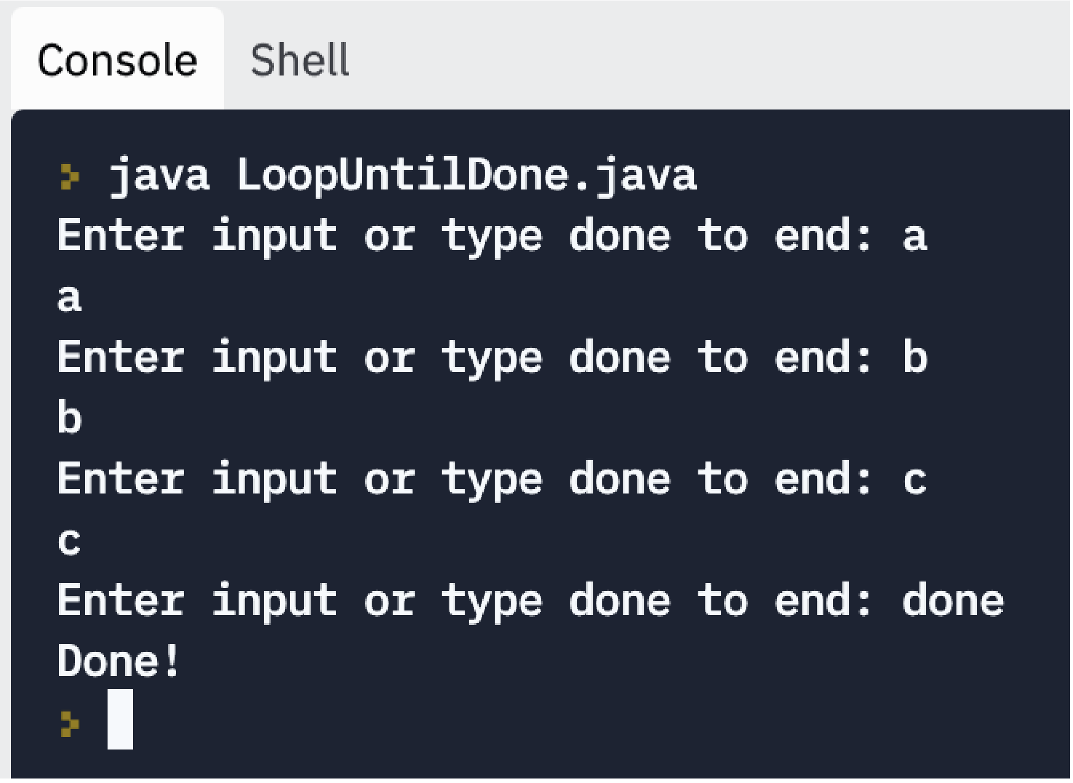
Directions: For example, suppose you want to take input from the user until they type “done.” Write the following:

```
import java.util.Scanner;

class LoopUntilDone {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    while(true) {
      System.out.print("Enter input or type done to end: ");
      String line = input.nextLine();
      // Check if line is "done" (ignoring case)
      if(line.equalsIgnoreCase("done")) {
        break; // end loop when done has been entered
      }
      else {
        // Print out entry is not equal to "done"
        System.out.println(line);
      }
    }
    System.out.println("Done!");
  }
}
```

REFLECT

The loop condition is true, which is always true, so the loop runs repeatedly until it hits the break statement. The compiler does not generate an error message in this case because it can see that there is a way to end the loop and allow the code after the loop to run.

Each time through, the body of the loop prompts the user for input. If the user types “done,” the break statement exits the loop. If not done, the program prints out whatever the user types and goes back to the top of the loop. Here’s a sample run:



```
> java LoopUntilDone.java
Enter input or type done to end: a
a
Enter input or type done to end: b
b
Enter input or type done to end: c
c
Enter input or type done to end: done
Done!
> 
```

TRY IT

Directions: Try typing the code above (in a .java file with a name that matches the name of the class in spelling and capitalization) and run the program a few times, testing out the word “done” to exit.

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top), and you can express the stop condition affirmatively (“stop when this happens”) rather than negatively (“keep going until that happens”).

HINT

Finish iterations with the break statement.

Sometimes when in an iteration of a loop, there is a need to finish the current iteration and immediately jump to the next iteration. In that case, use the continue statement to skip to the next iteration, without finishing the body of the loop for the current iteration.

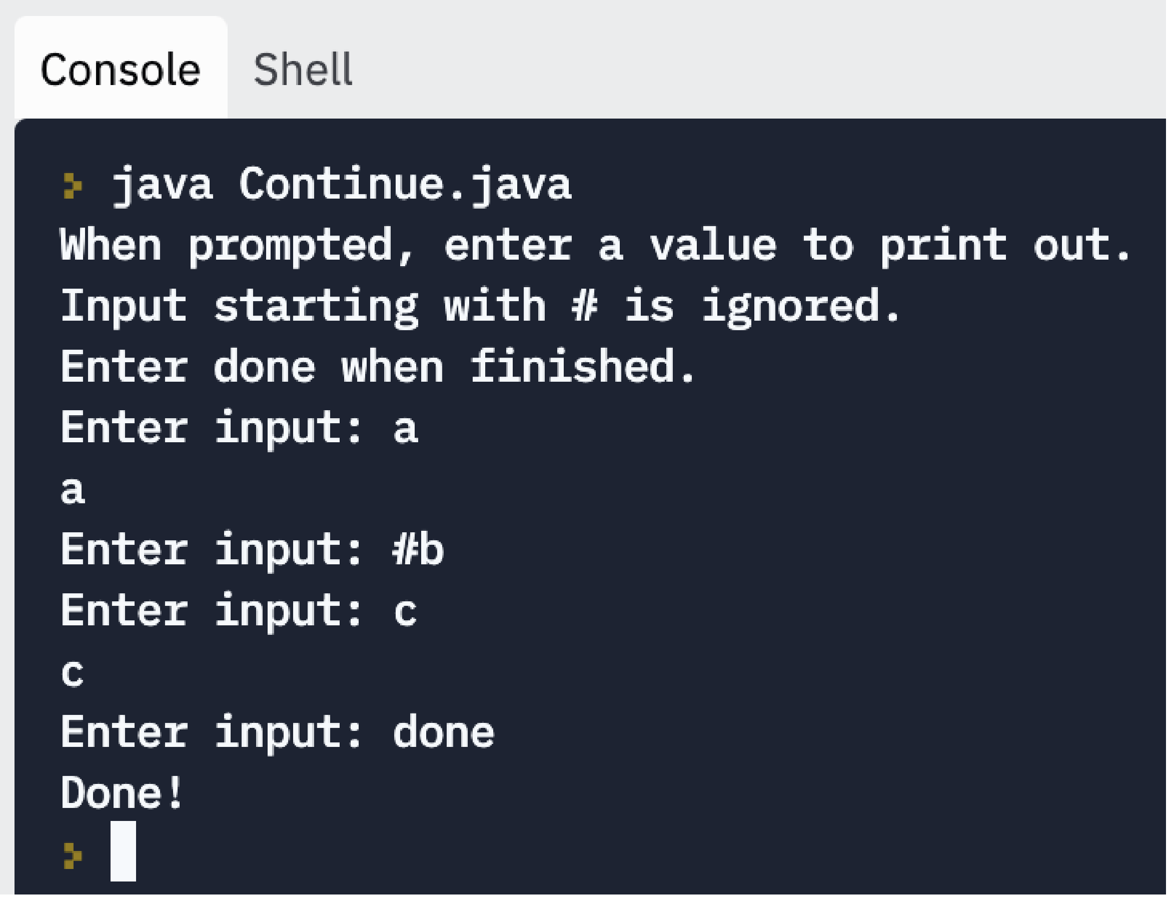
REFLECT

We have seen how the continue and break statements alter how a loop runs. Can you see how these statements play an especially important role in loops like the one above that are set up to run as infinite loops without such intervention?

Below you can see a loop that prints out its user input until the user types “done,” but treats lines that start with the hash character as lines not to be printed, kind of like Java comments:

```
while True:
    line = input('Enter input (# tag will not print). Type done when finished > ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

Here is a sample run of this new program with continue statement added:



```
> java Continue.java
When prompted, enter a value to print out.
Input starting with # is ignored.
Enter done when finished.
Enter input: a
a
Enter input: #b
Enter input: c
c
Enter input: done
Done!
> 
```

All the lines are printed except the one that starts with the hash sign (#). This happens when the continue statement is executed. It ends the current iteration and jumps back to the beginning of the while loop. This starts the next iteration, skipping the System.out.println() method call.

TRY IT

Directions: Go ahead and try the program with the continue statement. Notice that the program will only check if the user input starts with a hashtag. Using the symbol later in the sentence will not have the same effect.

REFLECT

When working with loops in Java code, the break statement is probably used more often than the continue statement. Think about why this would be the case.

BIG IDEA

As you write bigger programs, you may find yourself spending more time debugging. More code means more chances to make an error. There are more places for bugs to "hide." One way to cut your debugging time is "debugging by bisection."

For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

TERM TO KNOW

Infinite Loop
A loop that does not include a means for ending the loop, so the loop will run forever.

2. Other Common Issues

Debugging by bisection is the practice of breaking program code into "sections" for debugging and validation purposes. This will help speed up the debugging process. It may be able to find issues within a section rather than focusing on the full program. This shortens the time to debug, if no issues are present in earlier sections.

When starting, consider breaking the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a System.out.println() statement and run the program. If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice, it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

BIG IDEA

Debugging by bisection is especially important if you've made changes to an existing program. You most likely will not need to test the whole program prior to where the change is made but only on everything afterwards.

TERM TO KNOW

Debugging By Bisection
Debugging by bisection is the practice of breaking your program code into "sections" for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

SUMMARY

In this tutorial, you learned about **infinite loops**. You learned why they are an issue, since they will not stop running unless you have an option to stop the process, like the stop button in Replit. It is a best practice to always create a loop that has some means of being able to exit. In most cases, infinite loops are errors that should be resolved. You also learned about other **common issues** when using loops.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Debugging By Bisection

Debugging by bisection is the practice of breaking your program code into “sections” for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

Infinite Loop

A loop that does not include a means for ending the loop, so the loop will run forever.

Revisiting the Tic-Tac-Toe Program

by Sophia

WHAT'S COVERED

In this lesson, you will expand on the Tic-Tac-Toe program using loops. Specifically, this lesson covers:

1. Tic-Tac-Toe With Loops

This is a great time to revisit the Tic-Tac-Toe game that you worked on at the end of the last challenge. If you recall, there were many different instances where loops could have been used. Let’s first look at the code that we had previously ended with.

TRY IT

Directions: If you don’t have an existing Repl (file) in Replit (TicTacToe.java), enter in the following code to reflect where the Tic-Tac-Toe program ended in tutorial 2.1.8:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToe {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
               {" - ", " - ", " - "},
               {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    int col;
    int row;

    // X's 1st turn
    System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
    System.out.print("separated by a space: ");
    col = input.nextInt();
    row = input.nextInt();
    if(board[row - 1][col - 1].equals(" - ")) {
```

```
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// O's 1st turn
System.out.print("O - Select column (1-3) & select row (1-3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 2nd turn
System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");


// O's 2nd turn
System.out.print("O - Select column (1-3) & select row (1-3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
```

```
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 3rd turn
System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// O's 3rd turn
System.out.print("O - Select column (1-3) & select row (1-3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " O ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");

// X's 4th turn
System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
System.out.print("separated by a space: ");
col = input.nextInt();
row = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
System.out.println("\t" + Arrays.toString(board[0]));
System.out.println("\t" + Arrays.toString(board[1]));
System.out.println("\t" + Arrays.toString(board[2]) + "\n");
```

```
    // O's 4th turn
    System.out.print("O - Select column (1-3) & select row (1-3) ");
    System.out.print("separated by a space: ");
    col = input.nextInt();
    row = input.nextInt();
    if(board[row - 1][col - 1].equals(" - ")) {
      board[row - 1][col - 1] = " O ";
    }
    else {
      System.out.println("Sorry, that spot is taken.");
    }
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    // X's 5th turn
    System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
    System.out.print("separated by a space: ");
    col = input.nextInt();
    row = input.nextInt();
    if(board[row - 1][col - 1].equals(" - ")) {
      board[row - 1][col - 1] = " X ";
    }
    else {
      System.out.println("Sorry, that spot is taken.");
    }
    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");
  }
}
```

REFLECT

When you last saw this code, you were asked to reflect on common steps in the turns. What common steps did you come up with? Since we will next turn to adding loops to handle repetitive steps in the code, it's good to have specific ideas about what those steps are.

TRY IT

Directions: Now that you have the Tic-Tac-Toe program where you left off in the previous challenge, go ahead and run it a few times to get reacquainted with its process.

REFLECT

Now that you've had the chance to rerun the program, do you observe anything that would enhance its functionality?

In the current state of coding, you can quickly determine that in some situations, we have some repetitiveness. Notice that you are constantly prompting each user for their selected position (O's third move, X's fifth move, etc.). You could loop those inputs for both the X and O players as part of the body of the loop. You will keep X player's input at the start where we don't have to determine if the spot has been taken. That essentially is letting the X user go first before the looping begins.

TRY IT

Directions: Adding a new element (loop) to the program code, comments have been used to call it out. Add the looping for X and O player’s section for repeat player position input below. Then run the program to see the functionality:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeLoop {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    int col;
    int row;

    // for loop to provide 5 rounds of turns
    for(int turn = 0; turn < 5; turn++) {
      // X's turn
      System.out.print("X - Select column (1 - 3) & select row (1 - 3) ");
      System.out.print("separated by a space: ");
      col = input.nextInt();
      row = input.nextInt();
      if(board[row - 1][col - 1].equals(" - ")) {
        board[row - 1][col - 1] = " X ";
      }
      else {
        System.out.println("Sorry, that spot is taken.");
      }
      System.out.println("\t" + Arrays.toString(board[0]));
      System.out.println("\t" + Arrays.toString(board[1]));
      System.out.println("\t" + Arrays.toString(board[2]) + "\n");

      // If 5th turn, end loop after X's turn - only 9 spaces to fill
      if(turn == 4) {
        break;
      }

      // O's turn
      System.out.print("O - Select column (1-3) & select row (1-3) ");
      System.out.print("separated by a space: ");
      col = input.nextInt();
      row = input.nextInt();
      if(board[row - 1][col - 1].equals(" - ")) {
        board[row - 1][col - 1] = " O ";
      }
      else {
        System.out.println("Sorry, that spot is taken.");
      }
      System.out.println("\t" + Arrays.toString(board[0]));
      System.out.println("\t" + Arrays.toString(board[1]));
      System.out.println("\t" + Arrays.toString(board[2]) + "\n");
    }
  }
}
```

REFLECT

Did you notice any change to the functionality of the program? Probably not, but using a for loop has brought the code down from 150+ lines to 57 lines of code and has made it much clearer.

TRY IT

Directions: Change the loop initializing row to include nine iterations instead of five. Change the following:

↪ EXAMPLE

From:

```
for(int turn = 0; turn < 5; turn++) {
```

To:

```
for(int turn = 0; turn < 9; turn++) {
```

REFLECT

While this change increases the number of times the loop will run, how could this change help simplify the code?

TRY IT

Directions: Using a char variable (player), we can use the rest of the original body of the loop within the original “O move” section for both players X and O now. Before the start of the loop, declare the player variable like this:

```
char player;
```

For iteration 0 and the following even iterations, it is X's turn. The odd-numbered iterations are O's turn. We can use this code in the for loop (at the start or top of the loop) to set the player variable to X or O, as appropriate. This code should be the first statements in the body of the for loop.

HINT

There is a full listing of the code after the discussion of the sections to be added or changed, if you have questions about where the changes should be made.

EXAMPLE

```
if(turn % 2 == 0) {
  player = 'X';
}
else {
  player = 'O';
}
```

We will now include the player variable in the body of the loop to indicate whether it's X's or O's turn.

TRY IT

Directions: Change these following rows:

EXAMPLE

```
From:
System.out.print("X - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " X ";
}

To:
System.out.print(player + " - Select row (1 - 3) & select column (1 - 3) ");
System.out.print("separated by a space: ");
row = input.nextInt();
col = input.nextInt();
if(board[row - 1][col - 1].equals(" - ")) {
  board[row - 1][col - 1] = " " + player + " ";
}
```

Since this code handles the prompt and input for both X's and O's turns, remove the previous code that handled O's turn.

REFLECT

Using the same lines of code to handle both players' turns has increased the level of abstraction in the code by using a single variable to prompt the correct player. How does such abstraction help programmers write code more concisely and efficiently?

TRY IT

Directions: Make sure you have made the changes discussed above so your program will look like the code below. We have added some comments to the lines of code that were added or changed. Once you have verified the changes, go ahead and run the program. From the user's perspective, the program should function the same as the previous versions:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeLoop {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                       {" - ", " - ", " - "},
                       {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    int col;
    int row;
    // Variable to keep track of current player
    char player;

    // for loop to provide total of 9 turns
    for(int turn = 0; turn < 9; turn++) {
      // If an even turn number, player is X
      if(turn % 2 == 0) {
        player = 'X';
      }
      // Otherwise, odd turns are O's turns
      else {
        player = 'O';
      }

      System.out.print(player + " - Select row (1 - 3) & select column (1 - 3) ");
      System.out.print("separated by a space: ");
      row = input.nextInt();
      col = input.nextInt();
      if(board[row - 1][col - 1].equals(" - ")) {
        // Use player to provide character (X or O)
        board[row - 1][col - 1] = " " + player + " ";
      }
      else {
        System.out.println("Sorry, that spot is taken.");
      }
      System.out.println("\t" + Arrays.toString(board[0]));
      System.out.println("\t" + Arrays.toString(board[1]));
      System.out.println("\t" + Arrays.toString(board[2]) + "\n");
    }
  }
}
```

REFLECT

When changes are made in code to improve the structure or performance of the program that are not visible to the end user, the process of making such changes is called "refactoring." How does such refactoring fit into the process of creating an application?

2. Adding Validation

These changes in the code have really helped simplify the code. However, there is more that can be done to make the game even better. The next thing that you can do is do some error checking or **validation** based on the row and column that was entered by the player.

You will want to check two separate things:

1. Ensure that the value being entered by the player is between 1 and 3. Remember, our two-dimensional list (2D) is three columns by three rows.
2. Ensure that if the position in the 2D list has an entry already there, we want to inform the player to choose again. Right now if a player selects a position that is already taken, they lose their turn.

2a. Solution for Validating the User Input Value

Item #1 can be solved above using a while loop to check for positions that are not valid. For the loop to work correctly, be sure to initialize the row and col variables when they are declared:

EXAMPLE

```
int col = 0;
int row = 0;
```

TRY IT

Directions: Set up a while loop like this:

EXAMPLE

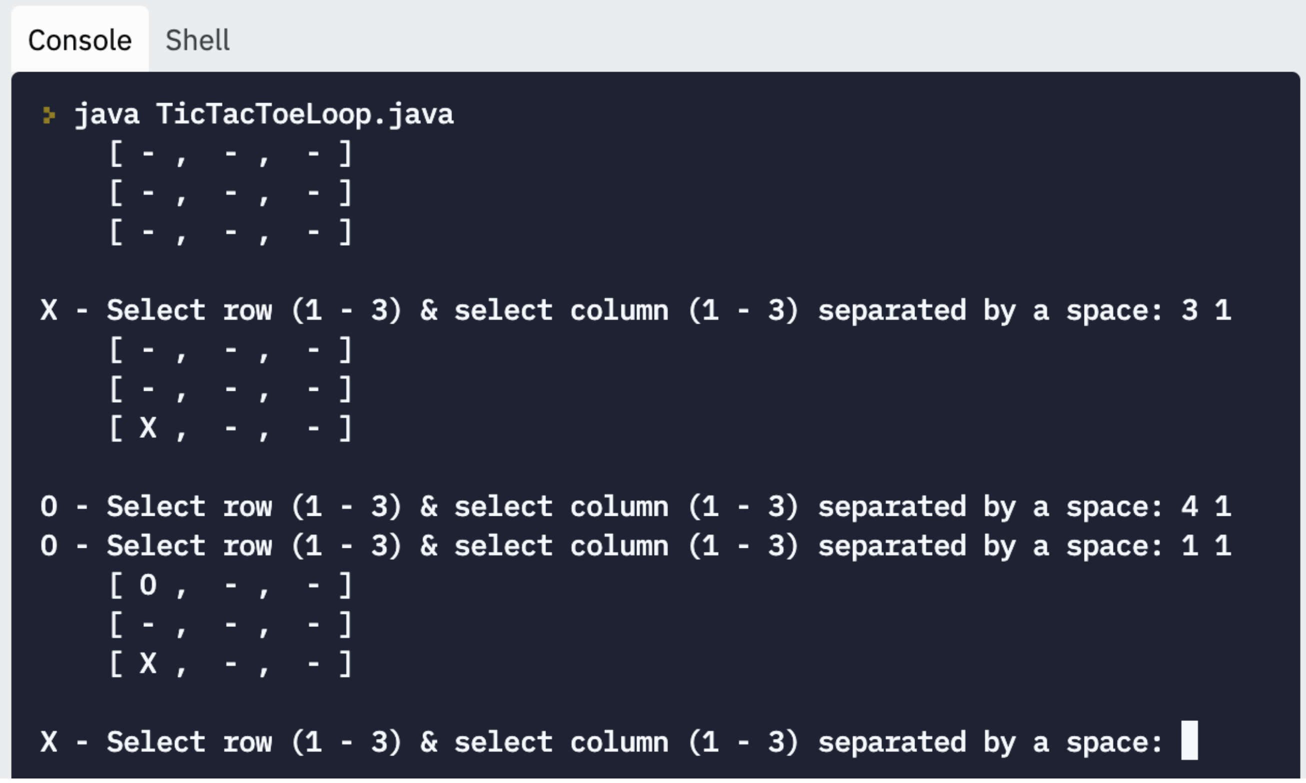
```
while(col < 1 || col > 3 || row < 1 || row > 3){
    System.out.print(player + " - Select row (1 - 3) & select column (1 - 3) ");
    System.out.print("separated by a space: ");
    row = input.nextInt();
    col = input.nextInt();
  }
```

Directions: At the bottom of the loop, set the values of col and row back to 0 so that the while loop will run and prompt the user for input (and validate the new input):

↪ EXAMPLE

```
col = 0;
row = 0;
```

In the code above, the while loop’s condition states that as long as the variable col’s value is less than 1 or greater than 3, the statements in the body of the loop would be repeated. If the entry from the user for col was less than 1 or larger than 3, you will output to the user that the column has to be between 1 and 3.



```
> java TicTacToeLoop.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 3 1
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ X ,  - ,  - ]

O - Select row (1 - 3) & select column (1 - 3) separated by a space: 4 1
O - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 1
    [ O ,  - ,  - ]
    [ - ,  - ,  - ]
    [ X ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 
```

REFLECT

The loop that handles the turns is a for loop because the maximum number of turns is a known value and does not change. The validation loop needs to be a while, though. Why is this the case? (Why wouldn't a for loop be the right choice for the validation loop?)

Here is the full code at this point; the solutions for item 1 (while loops from column and row input) are commented below.

Notice that we set the col and row to 0 outside of the while loop so with each iteration, the col and row will be reset to ensure that the user enters in an updated value each time:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeLoop {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                         {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    // Initialize col & row so that the input validation works
    int col = 0;
    int row = 0;
    // Variable to keep track of current player
    char player;

    // for loop to provide total of 9 turns
    for(int turn = 0; turn < 9; turn++) {
      // If an even turn number, player is X
      if(turn % 2 == 0) {
        player = 'X';
      }
      // Otherwise, odd turns are O's turns
      else {
        player = 'O';
      }
      // Check if col or row is less than 1 or greater than 3
      while(col < 1 || col > 3 || row < 1 || row > 3){
        System.out.print(player + " - Select row (1 - 3) & select column (1 - 3) ");
        System.out.print("separated by a space: ");
        row = input.nextInt();
        col = input.nextInt();
      }
      if(board[row - 1][col - 1].equals(" - ")) {
        // Use player to provide character (X or O)
        board[row - 1][col - 1] = " " + player + " ";
      }
      else {
        System.out.println("Sorry, that spot is taken.");
      }
      // Reset col and row to 0 so the validation loop runs for next turn
      col = 0;
      row = 0;
      System.out.println("\t" + Arrays.toString(board[0]));
      System.out.println("\t" + Arrays.toString(board[1]));
      System.out.println("\t" + Arrays.toString(board[2]) + "\n");
    }
  }
}
```

TRY IT

Directions: Try running this program with the changes above and test for outside positions (1< or >3). Did the program produce the intended results?

2b. Solution for Validating the Position in the 2D List Has an Entry

There is already a selection statement that checks if the position on the board is open or already taken. If it is taken, the code prints the appropriate message:

```
if(board[row - 1][col - 1].equals(" - ")) {
  // Use player to provide character (X or O)
  board[row - 1][col - 1] = " " + player + " ";
}
else {
  System.out.println("Sorry, that spot is taken.");
}
```

To keep the player from losing a turn if the position is already taken, we'll add a statement to the else block to roll the for loop's variable back by one so that the player's turn repeats:

EXAMPLE

```
turn--;
```

The board should not be printed out again unless the selection was valid and the position was open. Input the code for printing out the board in the if() block that checks if the value at the position is equal to " - ".

This section of the code will end up like this:

EXAMPLE

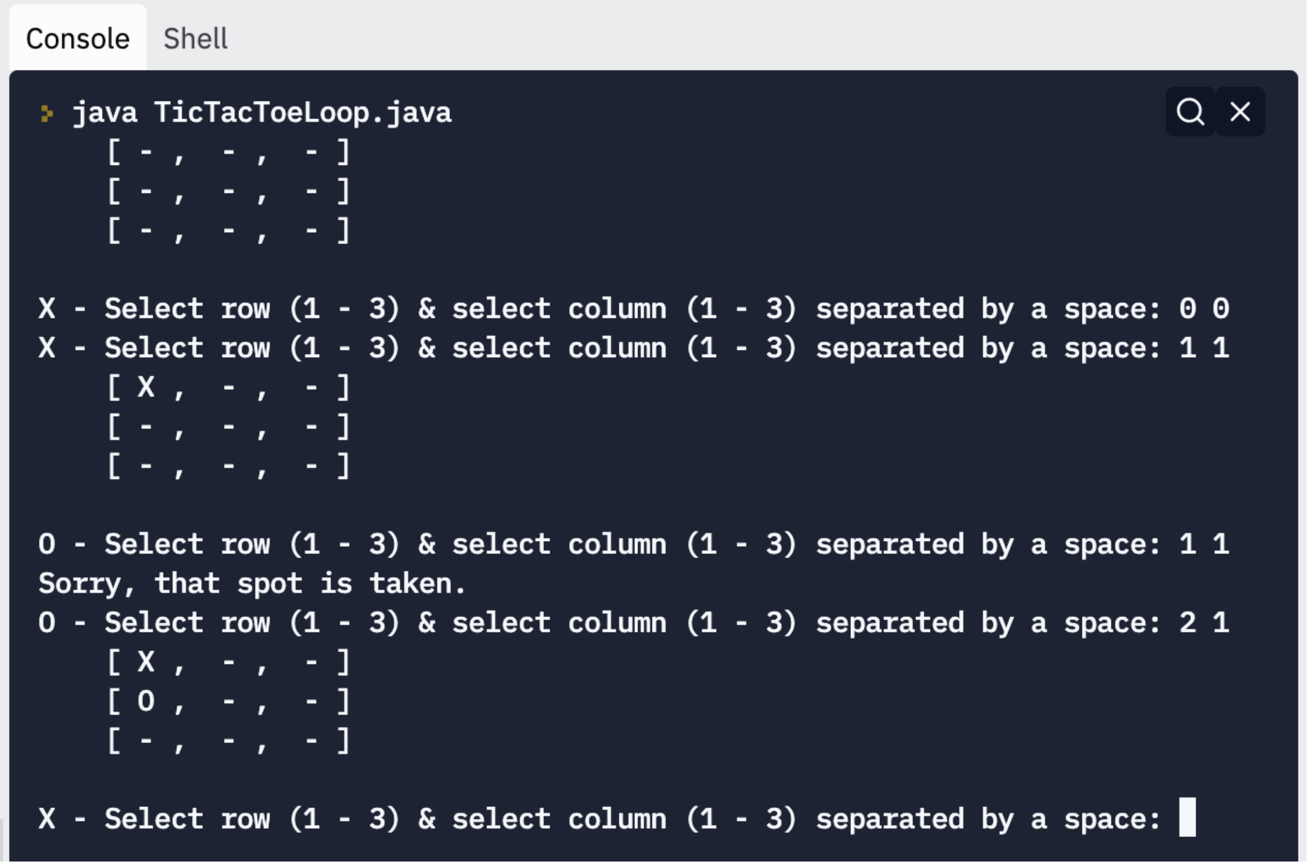
```
if(board[row][col].equals(" - ")) {
  // Use player to provide character (X or O)
  board[row][col] = " " + player + " ";
  System.out.println("\t" + Arrays.toString(board[0]));
  System.out.println("\t" + Arrays.toString(board[1]));
  System.out.println("\t" + Arrays.toString(board[2]) + "\n");
}
else {
  // If position already taken, print message
  System.out.println("Sorry, that spot is taken.");
  // Roll loop var back by 1 so that the turn repeats
  turn--;
}
```

TRY IT

Directions: The complete program should now look like the following. Go ahead and make sure that your code matches this (or type the code into a file named TicTacToeLoop.java now):

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeLoop {

  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};

    System.out.println("\t" + Arrays.toString(board[0]));
    System.out.println("\t" + Arrays.toString(board[1]));
    System.out.println("\t" + Arrays.toString(board[2]) + "\n");

    Scanner input = new Scanner(System.in);
    // Initialize col & row so that the input validation works
    int col = 0;
    int row = 0;
    // Variable to keep track of current player
    char player;

    // for loop to provide total of 9 turns
    for(int turn = 0; turn < 9; turn++) {
      // If an even turn number, player is X
      if(turn % 2 == 0) {
        player = 'X';
      }
      // Otherwise, odd turns are O's turns
      else {
        player = 'O';
      }
      // Check if col or row is less than 1 or greater than 3
      while(col < 1 || col > 3 || row < 1 || row > 3){
        System.out.print(player + " - Select row (1 - 3) & select column (1 - 3) ");
        System.out.print("separated by a space: ");
        row = input.nextInt();
        col = input.nextInt();
      }
      if(board[row - 1][col - 1].equals(" - ")) {
        // Use player to provide character (X or O)
        board[row - 1][col - 1] = " " + player + " ";
        System.out.println("\t" + Arrays.toString(board[0]));
        System.out.println("\t" + Arrays.toString(board[1]));
        System.out.println("\t" + Arrays.toString(board[2]) + "\n");
      }
      else {
        // If position already taken, print message
        System.out.println("Sorry, that spot is taken.");
        // Roll loop var back by 1 so that the turn repeats
        turn--;
      }
      // Reset col and row so that validation loop runs correctly
      col = 0;
      row = 0;
    }
  }
}
```

Running this code should produce results like this:



```
> java TicTacToeLoop.java
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 0 0
X - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 1
    [ X ,  - ,  - ]
    [ - ,  - ,  - ]
    [ - ,  - ,  - ]

O - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 1
Sorry, that spot is taken.
O - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 1
    [ X ,  - ,  - ]
    [ O ,  - ,  - ]
    [ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 
```

REFLECT

When running the program, think about the relationship between the "outer" for loop that controls the turns and the while loop that handles the validation. Which parts of the output are controlled by which loop?

BRAINSTORM

Go ahead and try running this program again with the additional logic to check the selected spot and test some cases of positions that are already taken. There is still room for improvement, such as checking if the player has won or not. We'll get further into that in the next challenge with functions and methods.

TERM TO KNOW

Validation

The process in a program of checking that input is in a valid range and appropriate in the current state of the program.

SUMMARY

In this lesson, you improved the **Tic-Tac-Toe program by using loops**for player input. Using loops, you greatly reduced the large blocks of repetitive code from our first version of the game. You then created nested loops to check for input **position validation** within the game board and whether or not the chosen position was already taken. If either validation was invalid, an output message was sent to the player, and they were asked to try again.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Validation

The process in a program of checking that input is in a valid range and appropriate in the current state of the program.

Method Parameters and Arguments

by Sophia

WHAT'S COVERED

In this lesson, you will learn about method parameters and arguments that are used when determining needed values. In tutorial 1.3.1, you briefly learned about method parameters and arguments. In this tutorial, you will look further and work with method parameters and arguments in greater detail. Specifically, this lesson covers:

1. Method Parameters

In the introduction to methods provided in tutorial 1.3.1, you saw examples of how methods in Java can include a special kind of variable called a **parameter**. You learned that parameters are used to pass data when calling a method. It is possible to have methods that do not take any parameters. This is because they do not require data from outside of the method to do their work. However, it is more common to create and use methods that take one or more parameters. Parameters for a method are placed in the parentheses that follow the name of the method. The earlier discussion included a simple Java method:

↱ EXAMPLE

```
static String sayHello(String name) {
  return "Hello, " + name;
}
```

The method in this case has the name sayHello. In the parentheses following the method name, you can see that there is one parameter for passing data to the method. String indicates that the parameter must be a Java String. The data type is followed by the parameter name, which in this case is name. The name of the method and the list of parameters make up the method's signature.

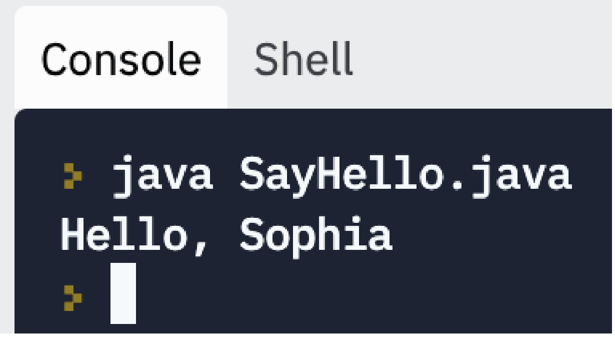
The parameters then serve as variables in the body of the method. This means that the parameters are **variables with local scope** (more commonly called **local variables**). Local variables are accessible from where they are declared to the end of the block. Parameters are "in scope" or accessible between the method signature and the end of the method.

As the code above shows, the name of the parameter can then be used as a variable in the body of the method. The value in the variable name is concatenated at the end of the greeting text.

The method can be called from the main() method. A complete sample program looks like this:

```
class SayHello {
  static String sayHello(String name) {
    return "Hello, " + name;
  }

  public static void main(String[] args) {
    String userName = "Sophia";
    // Pass variable userName as parameter to sayHello()
    String greeting = sayHello(userName);
    // Pass variable greeting to as parameter println()
    System.out.println(greeting);
  }
}
```

The output from running this code looks like this:



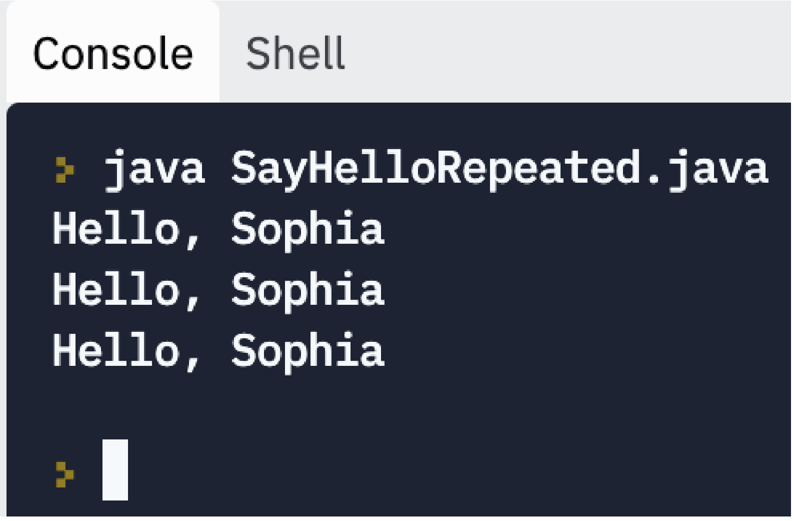
```
> java SayHello.java
Hello, Sophia
>
```

This method's signature includes only one parameter, but the method can be defined to have more. You can add an int parameter to indicate the number of repetitions, to repeat the greeting.

Adding an int parameter to indicate repetition:

```
public class SayHelloRepeated {
  static String sayHello(String name, int count) {
    // Local variable to assemble greeting
    String greeting = "";
    for(int i = 0; i < count; i++) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }

  public static void main(String[] args) {
    String userName = "Sophia";
    // Pass variable userName as parameter to sayHello()
    String greetingOutput = sayHello(userName, 3);
    // Pass variable greetingOutput to as parameter println()
    System.out.println(greetingOutput);
  }
}
```

The output from this version of the program should look like this:

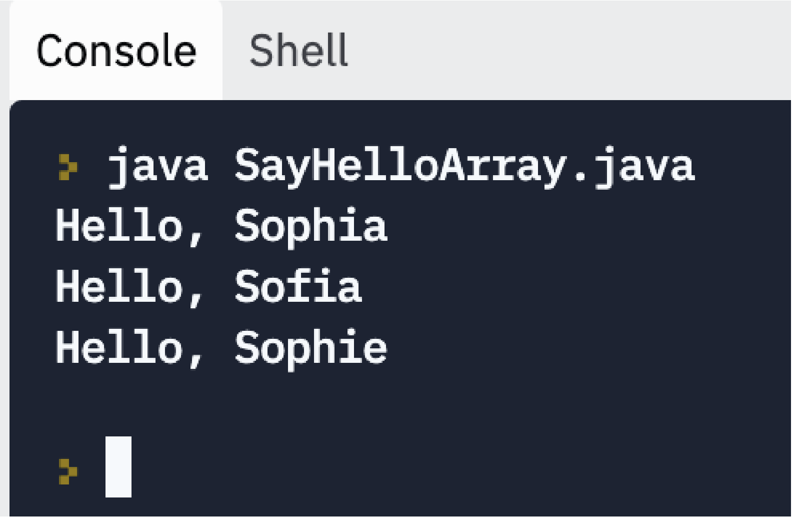


```
> java SayHelloRepeated.java
Hello, Sophia
Hello, Sophia
Hello, Sophia

> 
```

Rather than printing the same name multiple times, you can use an array as a parameter to pass multiple names:

```
public class SayHelloArray {
  // Pass array of names to method
  static String sayHello(String[] names) {
    // Local variable to assemble greeting
    String greeting = "";
    for(String name : names) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }

  public static void main(String[] args) {
    String[] userNames = {"Sophia", "Sofia", "Sophie"};
    // Pass variable userNames as parameter to sayHello()
    String greetingOutput = sayHello(userNames);
    // Pass variable greetingOutput to as parameter println()
    System.out.println(greetingOutput);
  }
}
```

The output should look like this:



```
> java SayHelloArray.java
Hello, Sophia
Hello, Sofia
Hello, Sophie

> 
```

The signature for the sayHello() method, sayHello(String[] names), includes square brackets to indicate that the parameter is an array of String values:

↱ EXAMPLE

```
static String sayHello(String[] names) {
```

When the method is called in main(), though, note that the array is passed just by name without the square brackets:

↱ EXAMPLE

```
String greetingOutput = sayHello(userNames);
```

Java methods can also take collections as parameters. Remember that collections can be generic:

```
import java.util.ArrayList;

public class SayHelloCollection {
  // Pass ArrayList of names to method
  // Remember that collections are generic. T is placeholder
  // for the data type
  static <T> String sayHello(ArrayList<T> names) {
    // Local variable to assemble greeting
    String greeting = "";
    // T is the data type
    for(T name : names) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }

  public static void main(String[] args) {
    ArrayList<String> userNames = new ArrayList<>();
    userNames.add("Sophia");
    userNames.add("Sophie");
    userNames.add("Sophie");
    // Pass variable userNames as parameter to sayHello()
    String greetingOutput = sayHello(userNames);
    // Pass variable greetingOutput to as parameter println()
    System.out.println(greetingOutput);
  }
}
```

In this case, String is the most likely data type, but the method could work with the Character type (perhaps representing an initial):

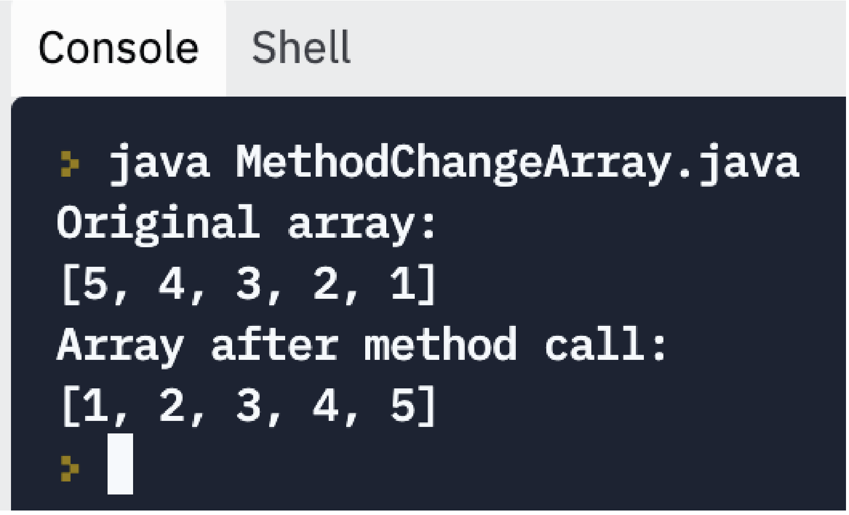
```
import java.util.ArrayList;

public class SayHelloCollection {
  // Pass ArrayList of names to method
  // Remember that collections are generic. T is placeholder
  // for the data type
  static <T> String sayHello(ArrayList<T> names) {
    // Local variable to assemble greeting
    String greeting = "";
    // T is the data type
    for(T name : names) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }

  public static void main(String[] args) {
    ArrayList<Character> userInitials = new ArrayList<>();
    userInitials.add('A');
    userInitials.add('B');
    userInitials.add('C');
    // Pass variable userInitials as parameter to sayHello()
    String greetingOutput = sayHello(userInitials);
    // Pass variable greetingOutput to as parameter println()
    System.out.println(greetingOutput);
  }
}
```

When passing a "plain" variable to a method, the code in the method works with a local copy of the data. The original value, in main() or wherever the method was called from, is not changed if the copy passed to the method is changed. As noted, parameters serve as local variables in the method to which they pass data. That means that the variables are only accessible in the block of code that makes up the body of the method. This is not exactly the case with arrays, though.

Changes made to an array in a method actually change the "original" array outside of the method, as seen below:

```
import java.util.Arrays;

class MethodChangeArray {
  // This method does not return anything but still
  // changes the order of items in the array passed
  // to it.
  public static void sortArray(int[] values) {
    Arrays.sort(values);
  }

  public static void main(String[] args) {
    int[] numbers = {5, 4, 3, 2, 1};
    // Display array contents before method call
    System.out.println("Original array:");
    System.out.println(Arrays.toString(numbers));
    // Call method
    sortArray(numbers);
    System.out.println("Array after method call:");
    // Display array contents after method call
    System.out.println(Arrays.toString(numbers));
  }
}
```

The results from running this program show that the array has been sorted outside of the method:



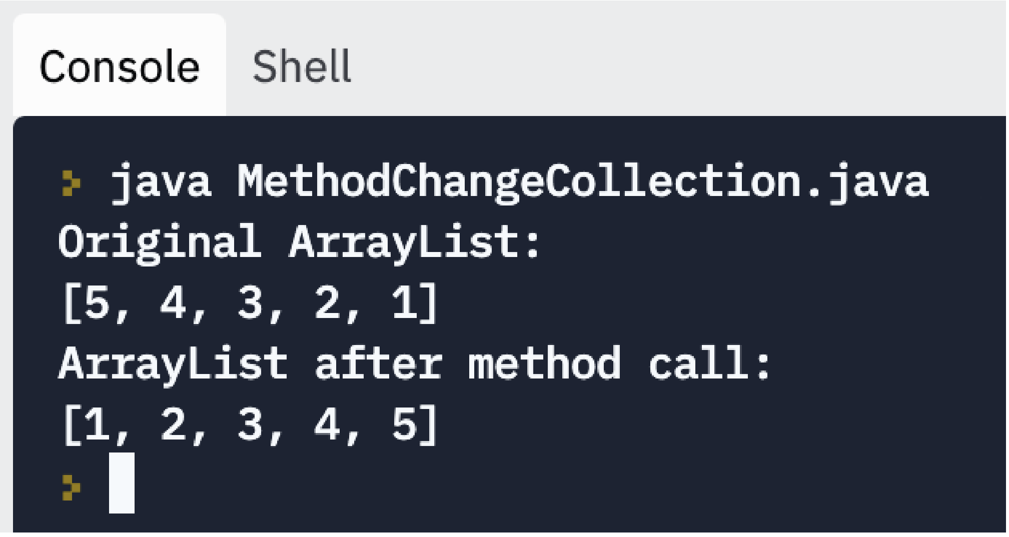
```
› java MethodChangeArray.java
Original array:
[5, 4, 3, 2, 1]
Array after method call:
[1, 2, 3, 4, 5]
›
```

A similar situation can arise when passing a collection as a parameter.

Here is a version of the same code that uses an ArrayList collection rather than a plain array:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class MethodChangeCollection {
  // This method does not retun anything but still
  // changes the order of items in the ArrayList passed
  // to it.
  public static void sortArrayList(ArrayList<Integer> values) {
    Collections.sort(values);
  }

  public static void main(String[] args) {
    // ArrayList collection of Integer values
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(5);
    numbers.add(4);
    numbers.add(3);
    numbers.add(2);
    numbers.add(1);
    // Display array contents before method call
    System.out.println("Original ArrayList:");
    System.out.println(numbers.toString());
    // Call method
    sortArrayList(numbers);
    System.out.println("ArrayList after method call:");
    // Display array contents after method call
    System.out.println(numbers.toString());
  }
}
```

Here is the output from this code:



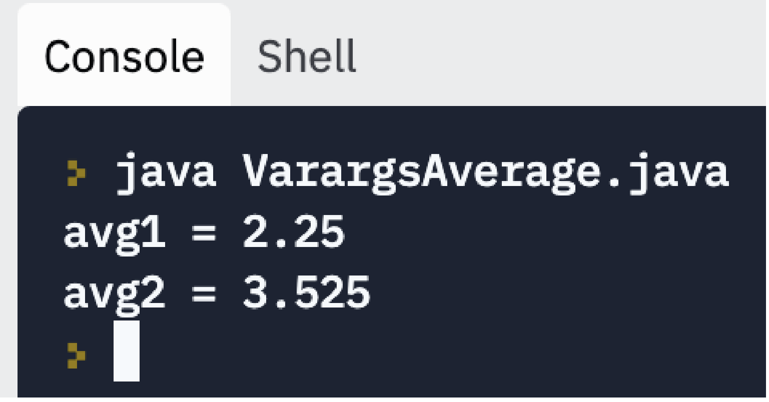
```
> java MethodChangeCollection.java
Original ArrayList:
[5, 4, 3, 2, 1]
ArrayList after method call:
[1, 2, 3, 4, 5]
> 
```

In addition to passing multiple values via array or collection parameters, Java also has another mechanism for passing a variable number of values to a method using a single named parameter. The parameter's data type is followed (without a space) by 3 dots (...). The sample below shows how to design a getAverage() method that calculates the average of one or more double values passed via a parameter named values.

To access the values passed, the code in the method used an enhanced for loop (for-each loop) to process each in turn:

```
class VarargsAverage {
  // The ... indicates that the method can take a variable
  // number of values as its parameter (in this case, doubles)
  public static double getAverage(double... values) {
    double sum = 0;
    int count = 0;
    // Use an enhanced for loop to iterate through values
    for(double number : values) {
      sum += number;
      count++;
    }
    return sum / count;
  }

  public static void main(String[] args) {
    double avg1 = getAverage(1.2, 3.3);
    System.out.println("avg1 = " + avg1);
    double avg2 = getAverage(1.2, 3.3, 4.5, 5.1);
    System.out.println("avg2 = " + avg2);
  }
}
```

Running this code produces this output:



```
java VarargsAverage.java
avg1 = 2.25
avg2 = 3.525
```

The next section covers arguments passed when a method is called in greater detail. In this section of the lesson, you have learned about the parameters associated with methods that provide a means for passing data into a method. The names and data types for the parameters are set as part of the method's signature.

TERMS TO KNOW

Parameter (also called a Formal Parameter)
A kind of variable that is used as a channel to pass data to a method.

Variables with Local Scope (more commonly called Local Variables)
Variables that can only be accessed between their point of declaration and the end of the block of code.

2. Arguments

The parameters that a method takes are the channels which are used to pass information into a method. This allows the code in the method to do its work. The parameters are the mechanism to pass information.

CONCEPT TO KNOW

Parameters are not the information itself. The actual data is passed as arguments when the method is called.

The **arguments** passed to a method must match the parameters specified in the method's signature. The order and data type would need to match as well. For instance, the first method we looked at, sayHello(), has the following signature:

↱ EXAMPLE

```
sayHello(String name)
```

This means that when the sayHello() method is called, it has to have a String value passed in, representing the name of the person to be greeted.

In the sample program above, a String variable was declared with a value like this:

↱ EXAMPLE

```
String userName = "Sophia";
```

And in the sample program above, the variable userName was then passed to the method like this:

↱ EXAMPLE

```
String greeting = sayHello(userName);
```

Rather than using a variable, it would have also been possible to call thesayHello() method using a literal String value:

↱ EXAMPLE

```
String greeting = sayHello("Sophia");
```

CONCEPT TO KNOW

When a variable is used to pass an argument, the data type is not specified in the parentheses, though it needs to be part of the variable's declaration. This is in contrast to the parameter in the method signature that has to include the data type.

If a method takes an array or collection as a parameter, just the name of the array or collection is passed. This is done without square brackets or accessing a specific element, since the whole array or collection is being passed.

Note the version of the sayHello() method that takes an array with this signature:

↱ EXAMPLE

```
sayHello(String[] names)
```

The method call looks like this, with just the name of the array in the parentheses:

↪ EXAMPLE

```
String greetingOutput = sayHello(userNames);
```

Recall that the userNames array was declared like this:

↪ EXAMPLE

```
String[] userNames = {"Sophia", "Sofia", "Sophie"};
```

THINK ABOUT IT

Review the version of the code with the `sayHello()` method that takes an `ArrayList` as its parameter. Note how the version using a collection passes the name of the collection (similar to how an array argument is passed by name).

TERM TO KNOW

Arguments
The actual variable or literal value passed when a method is called.

SUMMARY

In this lesson, you learned about **method parameters** and **arguments**. You also learned that they are used when determining needed values. Finally, you looked into method parameters and arguments in greater detail.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Arguments
The actual variable or literal value passed when a method is called.

Parameter (also called a Formal Parameter)
A kind of variable that is used as a channel to pass data to a method.

Variables with Local Scope (more commonly called Local Variables)
Variables that can only be accessed between their point of declaration and the end of the block of code.

The Return Statement

by Sophia

WHAT'S COVERED

In this lesson, you will learn about the return statement in methods. Specifically, this lesson covers:

1. Return Statements

When a method is done doing its work, it needs to return control of the program flow to the portion of the called, that called it. To understand how methods return the program flow to the caller, you will need to discern methods that do their work and don't need to send back a value. Next, you'll need to consider methods that are designed to do their work in order to return the results of their work. The Java data type that specifies the kind of value returned by a method is called the method's **return type**.

It is important to keep in mind that a method may return no value, or only one value or object. When the method does not return a value, the return type is said to be void. In cases where the method returns a value, the return type is the data type of the value returned.

↪ EXAMPLE

int, double, char, String or data type.

TERMS TO KNOW

return
The reserved keyword return is used to exit a method and return a value, if the method returns a value.

Return Type
The data type of the value returned by a method.

---

2. Return void

Some methods print out a greeting without needing to return anything.

A Simple Method:

↪ EXAMPLE

```
// A simple method that doesn't need to return anything
public static void printGreeting(String name) {
  System.out.println("Hello, " + name);
  return;
}
```

This method calls `System.out.println()` itself, so there is no further work to be done when the method is finished. The keyword `void` to the left of the method's name indicates that the method does not return a value.

The return statement without a value indicates that the method returns to the caller after printing out the greeting, but it also makes clear that no value is returned. In a method that returns nothing, it has the return type `void`.

When a method has the return type `void`, there is no need to provide an explicit return statement. In such cases, the control flow in the program returns to the caller when the code in the method is done executing.

TERM TO KNOW

void
The return type void indicates that a method does not return a value.

3. Return a Value

Many times, a method will need to return a value to the caller, with the result of running the code in the method. For instance, the simplest version of the `sayHello()` method from the last tutorial was defined like this because it returned a `String` with the greeting assembled by the method.

CONCEPT TO KNOW

The return type is the data type to the left of the name of the method. Keep in mind that `static` is an access modifier that we will discuss later in the course, so look past it for now.

A static String:

↱ EXAMPLE

```
static String sayHello(String name) {
  return "Hello, " + name;
}
```

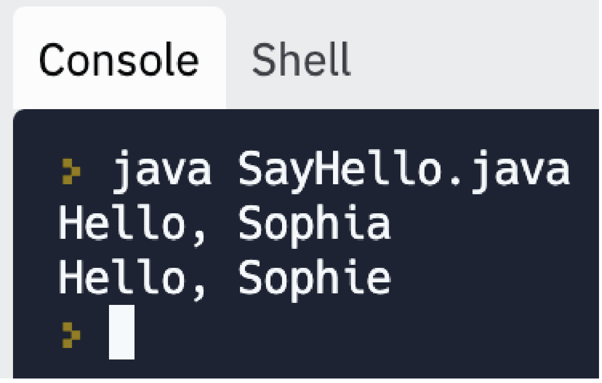
This method returns a `String` with the greeting concatenated with the desired name appended to it.

TRY IT

Directions: When calling a method with a returned value, the caller provides a variable to hold the result that is returned or passes the value returned to another method, placing one method call inside of another:

```
class SayHello {
  public static void main(String[] args) {
    // The variable to the left of the equal sign holds
    // the value returned from the method (a String)
    String greeting = sayHello("Sophia");
    System.out.println(greeting);
    // In this call, the String returned from the method
    // is passed directly to println() for display
    System.out.println(sayHello("Sophie"));
  }

  static String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

The output for this should look like this:



```
> java SayHello.java
Hello, Sophia
Hello, Sophie
> 
```

REFLECT

This small method returns the String with the greeting text rather than printing it out directly. Why might a programmer prefer a method that returns a value for display rather than printing it out directly in the method itself?

The first line of output is the result of running these two statements on two separate lines of code:

EXAMPLE

```
String greeting = sayHello("Sophia");
System.out.println(greeting);
```

The second line of output is the result of running this one line of code:

EXAMPLE

```
System.out.println(sayHello("Sophie"));
```

Note that it is also possible to make a call like this where there is no variable to capture the value returned.

The method call is not embedded in another method call:

EXAMPLE

```
sayHello("Sofia");
```

HINT

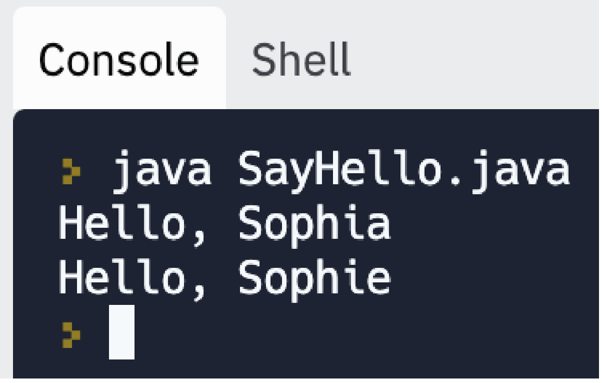
There is little point in doing so, since the String returned by sayHello() is lost.

TRY IT

Directions: Add the statement to the code. When you run it, you should notice that it has no visible effect:

```
class SayHello {
  public static void main(String[] args) {
    // The variable to the left of the equal sign holds
    // the value returned from the method (a String)
    String greeting = sayHello("Sophia");
    System.out.println(greeting);
    // In this call, the String returned from the method
    // is passed directly to println() for display
    System.out.println(sayHello("Sophie"));
    // Since the following does not save the returned String
    // to a String variable and since the call is not embedded in
    // another method call, the String is lost
    sayHello("Sofia");
  }

  static String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

This output shows the same two lines as before, with no sign of the third call to sayHello():



```
> java SayHello.java
Hello, Sophia
Hello, Sophie
>
```

REFLECT

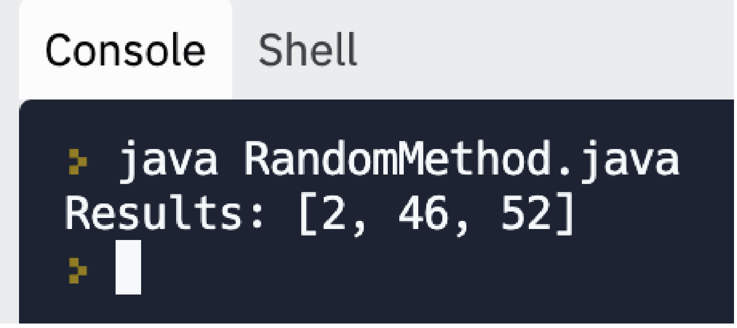
When calling a method that returns a value, the code should normally assign the return value to an appropriate variable. Why would overlooking the returned value be unwise?

In addition to returning a single, primitive value, a method can also return multiple values using an array or a collection. Keep in mind that all of the values in the array or collection have to be of the same data type.

Here is an example of a program that includes a method that returns an array of random integers:

```
import java.util.Random;
import java.util.Arrays;

class RandomMethod {
  // Method returns an array of random integers
  // Parameters are maximum of the range (minimum is 1)
  // count is number of random numbers to return
  static int[] getRandomIntegers(int max, int count) {
    // Create random number generator - see the guessing game
    Random randomGenerator = new Random();
    int[] randomNumbers = new int[count];
    for(int i = 0; i < count; i++) {
      // Get a random number in the range from 1 to max
      randomNumbers[i] = randomGenerator.nextInt(max) + 1;
    }
    // Return array to caller
    return randomNumbers;
  }

  public static void main(String[] args) {
    // Get an array of 3 integers in the range from 1 to 100
    int[] results = getRandomIntegers(100, 3);
    System.out.println("Results: " + Arrays.toString(results));
  }
}
```

Here is the output from a sample run of the program:



```
› java RandomMethod.java
Results: [2, 46, 52]
›
```

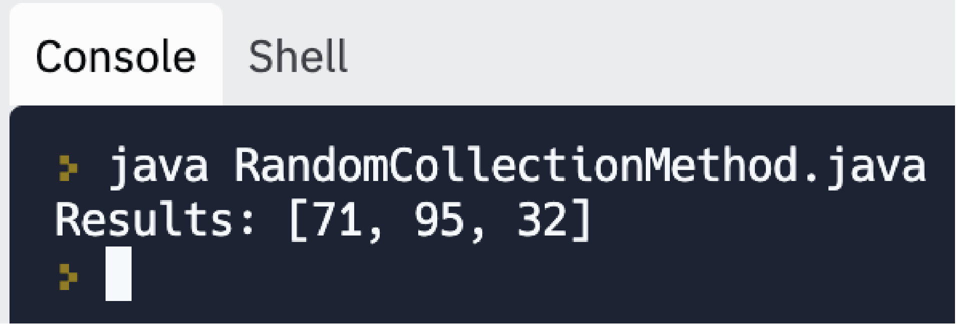
It is possible to create a similar method that returns a collection (an ArrayList) instead of an array.

TRY IT

Directions: Type in and run this version of the code:

```
import java.util.Random;
import java.util.ArrayList;

class RandomCollectionMethod {
  static ArrayList<Integer> getRandomIntegers(int max, int count) {
    // Create random number generator - see the guessing game
    Random randomGenerator = new Random();
    ArrayList<Integer> randomNumbers = new ArrayList<>();
    for(int i = 0; i < count; i++) {
      // Get a random number in the range from 1 to max
      randomNumbers.add(randomGenerator.nextInt(max) + 1);
    }
    // Return array to caller
    return randomNumbers;
  }

  public static void main(String[] args) {
    // Get an array of 3 integers in the range from 1 to 100
    ArrayList<Integer> results = getRandomIntegers(100, 3);
    System.out.println("Results: " + results.toString());
  }
}
```

A sample run of this version produces similar results (but keep in mind that the numbers are random, so the output will vary):



```
> java RandomCollectionMethod.java
Results: [71, 95, 32]
>
```

REFLECT

Returning an array or collection of values is a good way to return more than one value from a method, but array values must all be of the same data type. What effect does this fact have on the usefulness of returning an array from a method?

TRY IT

Directions: Go ahead and add the example above into Replit and see if you get the same type of data in the output.

BRAINSTORM

In a method like the one above, do you see advantages to using an array or a collection?

SUMMARY

This tutorial has covered the process of returning methods. Some methods do their work without **returning a value**. You learned that some methods have the**return void** with a type void. You also learned that other methods send back the result of the method's work using a **return statement**. Finally, you learned that a return statement can only return one item, but an array or a collection can be used to return multiple values of the same data type.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Return Type
The data type of the value returned by a method.

return
The reserved keyword return is used to exit a method and return a value, if the method returns a value.

void
The return type void indicates that a method does not return a value.

Overloading Methods

by Sophia

WHAT'S COVERED

In this lesson, you will learn about overriding methods when using loops. Specifically, this lesson covers:

1. Overloading Methods

It is useful to have different versions of a method that have the same name but take different numbers and types of parameters, when designing methods in Java. When there is more than one version of a method with the same name that take different parameters, including different numbers or types of parameters, the method is said to be **overloaded**. It is important to note that only the method's signature, or the method name and parameters, play a role in method overloading. The return type is not considered.

CONCEPT TO KNOW

It is not possible to have two versions of a method with the same name that differ only in the return type.

We have previously encountered this simple method that takes a single `String` parameter:

EXAMPLE

```
static String sayHello(String name) {
  return "Hello, " + name;
}
```

We have also seen a different version of the `sayHello()` method that takes two parameters, a `String` for the name and an int for the number of repetitions:

EXAMPLE

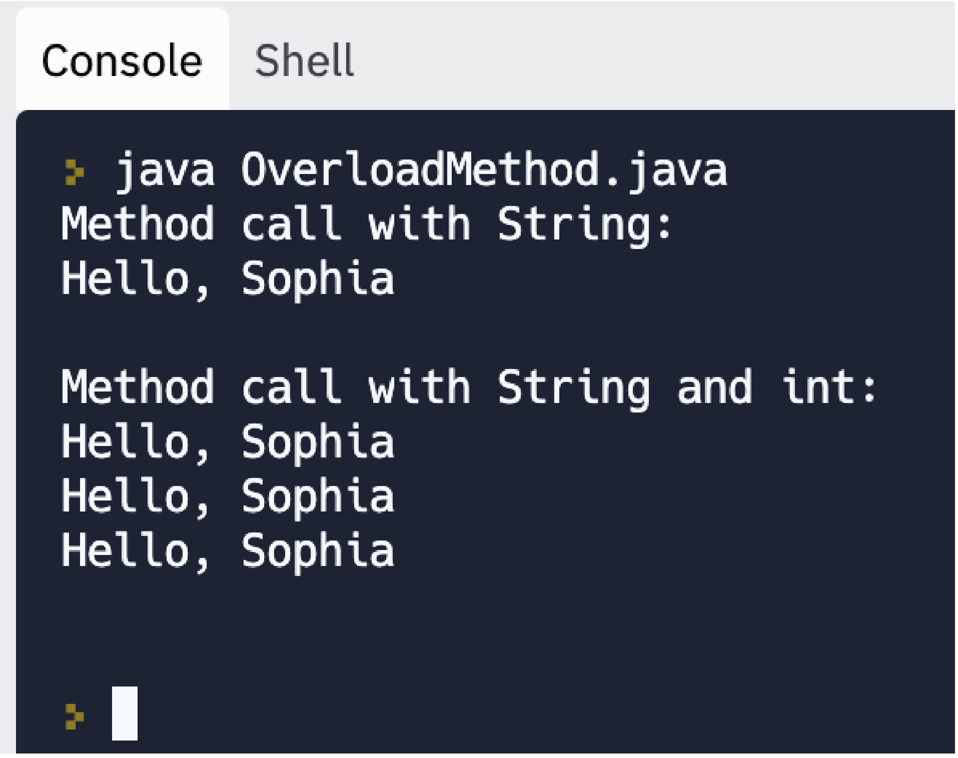
```
static String sayHello(String name, int count) {
  // Local variable to assemble greeting
  String greeting = "";
  for(int i = 0; i < count; i++) {
    greeting += "Hello, " + name + "\n";
  }
  return greeting;
}
```

When working with these methods in lesson 2.3.1, you found that they were in separate programs. Since they have different method signatures, or different numbers of parameters, they can be included in the same program.

The Java compiler determines which version of the method intended to be called based on the number and types of the arguments passed:

```
class OverloadMethod {
  public static void main(String[] args) {
    System.out.println("Method call with String: ");
    String result = sayHello("Sophia");
    System.out.println(result + "\n");

    System.out.println("Method call with String and int: ");
    result = sayHello("Sophia", 3);
    System.out.println(result + "\n");
  }

  // Method with 1 String parameter

}
```

If you type in the code above (in a file named OverloadMethod.java) and run it in Replit, the results should look like this:



```
> java OverloadMethod.java
Method call with String:
Hello, Sophia

Method call with String and int:
Hello, Sophia
Hello, Sophia
Hello, Sophia


>
```

The compiler looks for a version of the sayHello() method that has the parameters that match the arguments passed when the method is called.

Notice the call on this line:

↪ EXAMPLE

```
String result = sayHello("Sophia");
```

It invokes the version of sayHello() with this signature:

EXAMPLE

```
sayHello(String name)
```

Notice the call to sayHello() on this line:

EXAMPLE

```
result = sayHello("Sophia", 3);
```

It matches the overload of the sayHello() method with this signature:

EXAMPLE

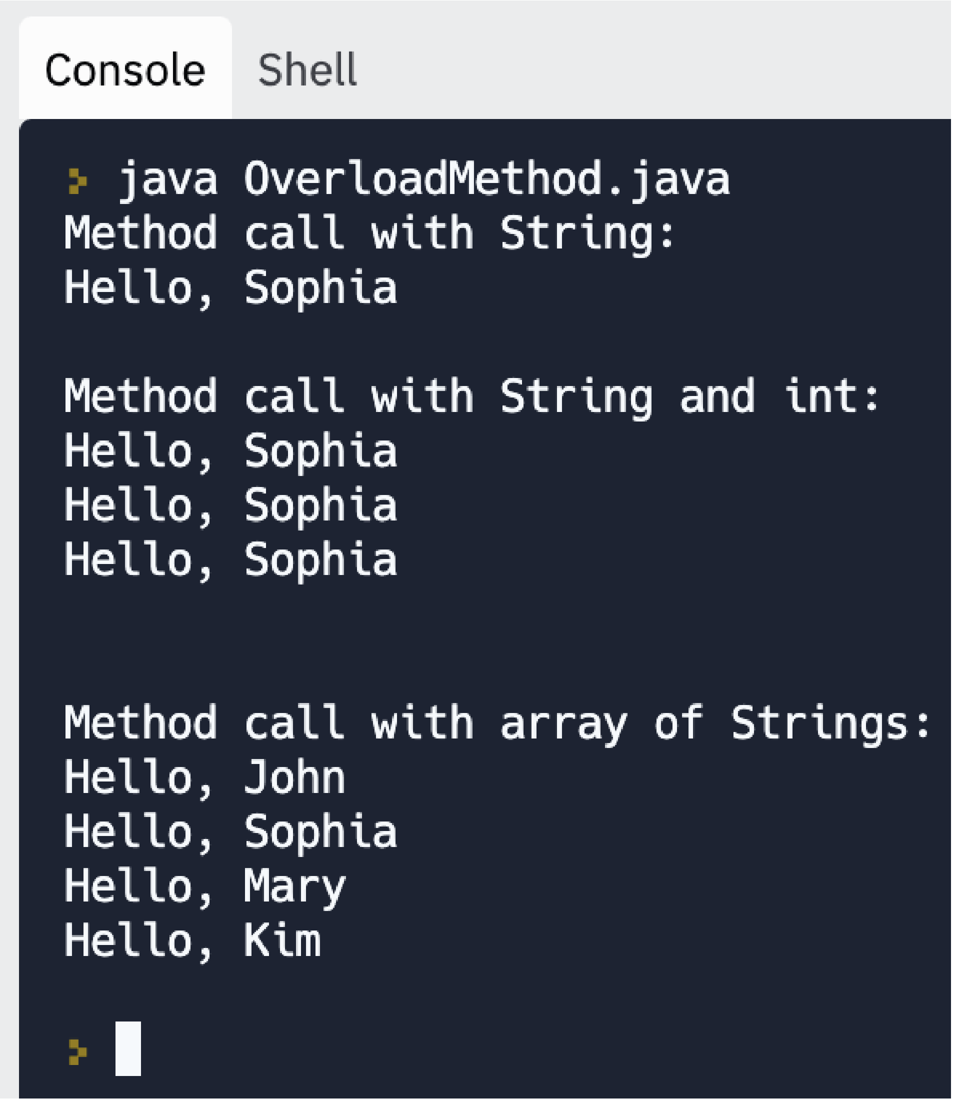
```
sayHello(String name, int count)
```

These two overloads of the sayHello() method differ in the number of parameters. An overload can be written of the method that takes an array of String values, rather than a plain String. While this version has only one parameter, the parameter is a String[] rather than a single String. The parameter is of a different type, which is an array of Strings rather than a String. We can call this overload of the method by passing an array of Strings as the argument.

This version of the program demonstrates the use of these three overloads in a single program:

```
class OverloadMethod {
  public static void main(String[] args) {
    System.out.println("Method call with String: ");
    String result = sayHello("Sophia");
    System.out.println(result + "\n");

    System.out.println("Method call with String and int: ");
    result = sayHello("Sophia", 3);
    System.out.println(result + "\n");

    System.out.println("Method call with array of Strings: ");
    String[] firstNames = {"John", "Sophia", "Mary", "Kim"};
    result = sayHello(firstNames);
    System.out.println(result);
  }

  // Method with 1 String parameter
  static String sayHello(String name) {
    return "Hello, " + name;
  }

  // Method with 2 parameters (String & int)
  static String sayHello(String name, int count) {
    // Local variable to assemble greeting
    String greeting = "";
    for(int i = 0; i < count; i++) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }

  // Method with 1 array of Strings parameter
  static String sayHello(String[] names) {
    String greeting = "";
    for(String name : names) {
      greeting += "Hello, " + name + "\n";
    }
    return greeting;
  }
}
```

This program with three overloads of the `sayHello()` method produces these results (assuming the code is saved in a file named OverloadMethod.java:



```
java OverloadMethod.java
Method call with String:
Hello, Sophia

Method call with String and int:
Hello, Sophia
Hello, Sophia
Hello, Sophia


Method call with array of Strings:
Hello, John
Hello, Sophia
Hello, Mary
Hello, Kim
```

CONCEPT TO KNOW

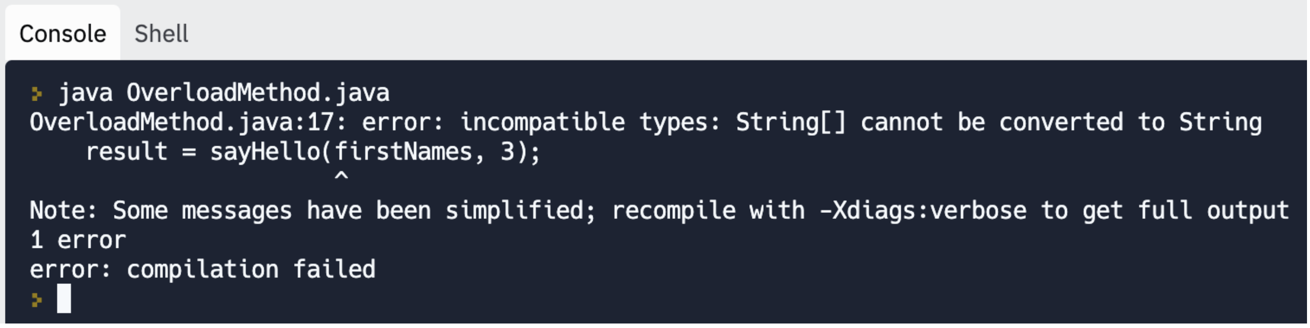
The programmer is responsible for providing the overloads of the method. If the program tries to call a method with parameters that don't correspond to one of the overloads, the program ends with an exception error.

Consider if the code above included a call that passes an array and an int using a call like this:

↪ EXAMPLE

```
result = sayHello(firstNames, 3);
```

The result is this:



```
java OverloadMethod.java
OverloadMethod.java:17: error: incompatible types: String[] cannot be converted to String
    result = sayHello(firstNames, 3);
                      ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
error: compilation failed
```

While it may be difficult to understand what this error is trying to communicate, the first line of the output indicates that the compiler can't convert an array of Strings to a plain String. The compiler "sees" that there is an overload of the method with two parameters, but it can't convert an array of Strings to a String. There is no appropriate match.

The method definitions below were based on the code above:

↪ EXAMPLE

```
static String sayHello(String name)
static String sayHello(String name, int count)
static String sayHello(String[] names)
```

TERM TO KNOW

Overloaded
A method is overloaded if different versions of the method exist that differ in the number or type of parameters in the method's signature.

SUMMARY

In this lesson, you learned about **overloading methods**. You also learned that there are different versions of the same method that can be called based on the signature of the method. This means that the overloads can have different types and numbers of parameters. You learned that when the return type is not part of a method's signature, it is not possible to have overloads that differ only by their return types. Finally, you learned that methods that need to return different data types need to have different names.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Overloaded
A method is overloaded if different versions of the method exist that differ in the number or type of parameters in the method's signature.

Debugging Complex Methods

by Sophia

WHAT'S COVERED

In this lesson, you will learn about debugging complex methods when using loops. Specifically, this lesson covers:

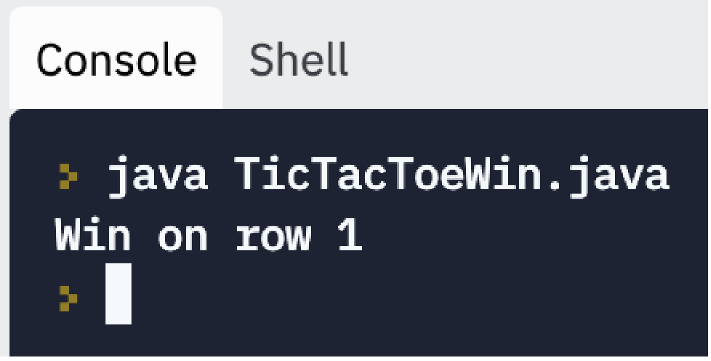
1. A Method to Detect a Tic-Tac-Toe Win in a Row

Consider this method intended to determine if there is a win on one of the rows on the Tic-Tac-Toe board.

Here is an attempt at a method called checkForRowWin() to detect if a player has marked all three spaces in a given row. To keep things simpler while working through the method, the program relies on a hard-coded array for the board with X's and O's rather than relying on user input:

```
class TicTacToeWin {
  public static void main(String[] args) {
    // Board pattern to test
    // X has a win on the second row (array index 1 is row 2)
    String[][] board = {{" - ", " O ", " X "},
                        {" X ", " X ", " X "},
                        {" O ", " O ", " - "}};
    int winningRow = checkForRowWin(board, 'X');
    if(winningRow > 0) {
      System.out.println("Win on row " + winningRow);
    }
  }

  public static int checkForRowWin(String[][] board, char player) {
    // Variable to track row that holds a win
    int rowNumber = 0;
    for(int row = 0; row <= 3; row++) {
      if(board[row][0].equals(" " + player + " ") ||
        board[row][1].equals(" " + player + " ") ||
        board[row][2].equals(" " + player + " ")) {
        // Add 1 to row number to match human count (starting with 1)
        rowNumber = row + 1;
        break;
      }
    }
    // If rowNumber > 0, there is a winning row
    return rowNumber;
  }
}
```

If player X has marked all three spaces in a row, the method returns the number of the row (1 - 3) containing the win; otherwise, it returns 0. Running the code in Replit should show a win on row 2, but this code produces the following result:



```
> java TicTacToeWin.java
Win on row 1
>
```

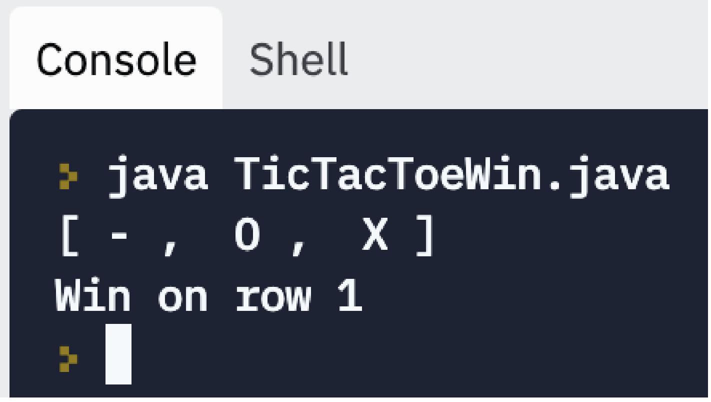
To assess the problem, a temporary statement is used. The temporary statement will be removed when the issue has been resolved. When the code detects a winning row, this version of the program will print out the contents of that row.

The lines in bold are the new additions:

↪ EXAMPLE

```
import java.util.Arrays; // Added to print out row

class TicTacToeWin {  public static void main(String[] args) {
  // Board pattern to test
  String[][] board = {{" - ", " O ", " X "},
                              {" X ", " X ", " X "},
                              {" O ", " O ", " - "}};
  int winningRow = checkForRowWin(board, 'X');
  if(winningRow > 0) {
   System.out.println("Win on row " + winningRow);
  }
 }

 public static int checkForRowWin(String[][] board, char player) {
  // Variable to track row that holds a win
  int rowNumber = 0;
  for(int row = 0; row <= 3; row++) {
   if(board[row][0].equals(" " + player + " ") ||
    board[row][1].equals(" " + player + " ") ||
    board[row][2].equals(" " + player + " ")) {
    rowNumber = row + 1;
    // Temporary statement to print out winning row
    System.out.println(Arrays.toString(board[row]));
    break;
   }
  }
  // If rowNumber > 0, there is a winning row
  return rowNumber;
 }
}
```

The added information in the output shows that the code is declaring a win in a row that clearly is not:



```
> java TicTacToeWin.java
[ - ,  O ,  X ]
Win on row 1
>
```

THINK ABOUT IT

There is only one X in the "winning" row. What can be the cause?

TRY IT

Directions: Let's look at the selection statement that is falsely detecting the win:

↪ EXAMPLE

```
if(board[row][0].equals(" " + player + " ") ||
 board[row][1].equals(" " + player + " ") ||
 board[row][2].equals(" " + player + " ")) {
```

The logical or (||) means that a player is declared a winner if the player has at least one space marked in the row. Since a winner has to have all three spaces, the correct choice is a logical and operator (&&).

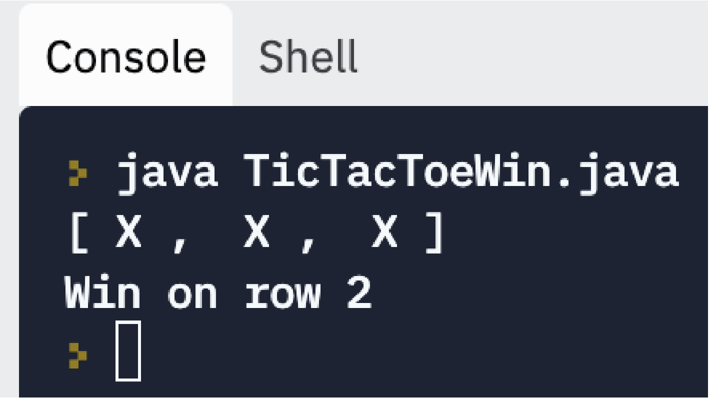
TRY IT

Try replacing the code above with the following:

↪ EXAMPLE

```
if(board[row][0].equals(" " + player + " ") &&
     board[row][1].equals(" " + player + " ") &&
     board[row][2].equals(" " + player + " ")) {
```

The code now produces the following output:



```
> java TicTacToeWin.java
[ X ,  X ,  X ]
Win on row 2
>
```

REFLECT

When using such a temporary output statement, it is important to display useful content. The code above displays the row when a win has been detected. Would it be useful to have a temporary statement print out the rows where a win is not found?

Now let's try testing with a board that does not contain a winning row.

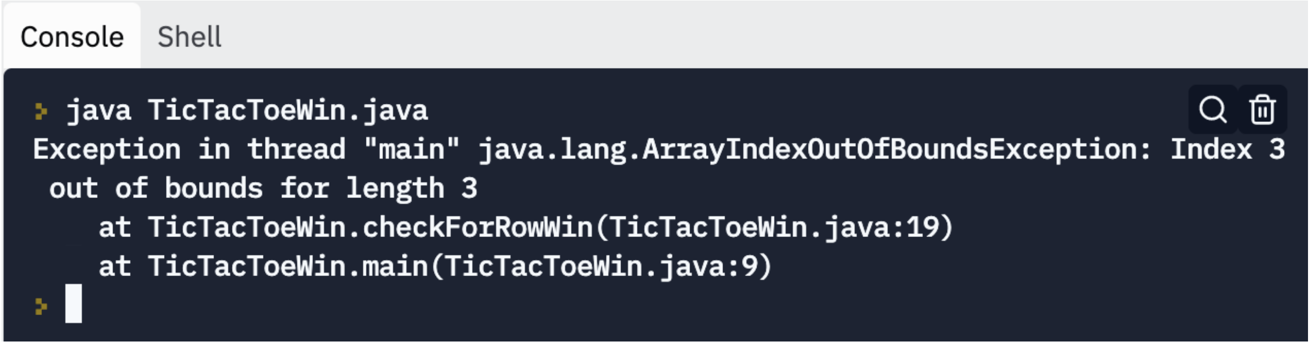
TRY IT

Directions: Change the declaration and initialization of the board array to look like this:

EXAMPLE

```
String[][] board = {{" - ", " O ", " X "},
                    {" X ", " - ", " X "},
                    {" O ", " O ", " - "}};
```

Running the code produces this result:



```
> java TicTacToeWin.java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3
 out of bounds for length 3
    at TicTacToeWin.checkForRowWin(TicTacToeWin.java:19)
    at TicTacToeWin.main(TicTacToeWin.java:9)
>
```

REFLECT

As the error message in the stack trace indicates, the array index 3 is out of range (an array with three rows has indexes 0, 1, and 2). But why is 3 even occurring in the first place?

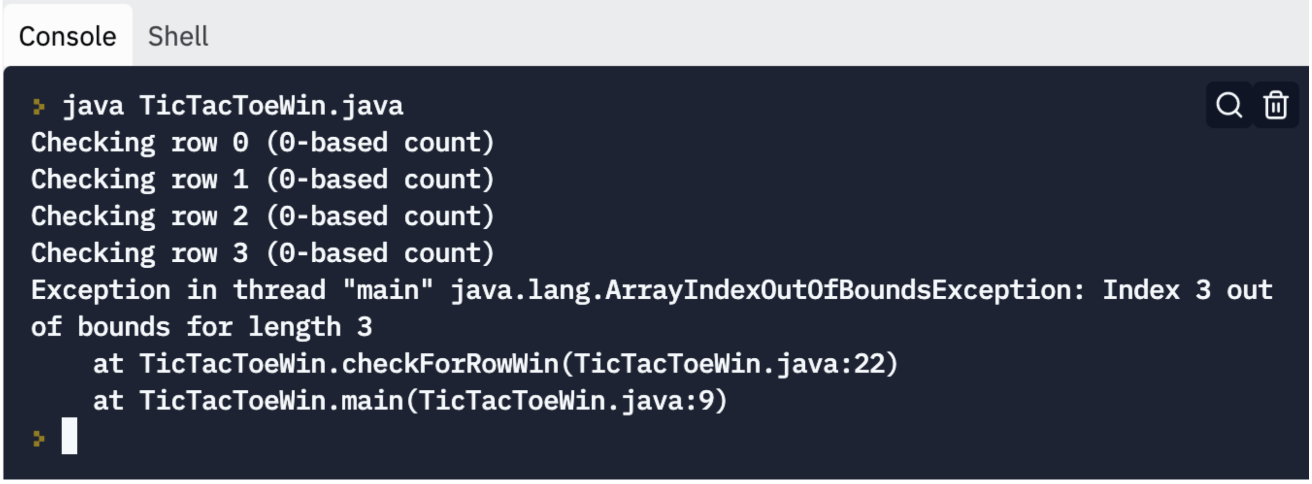
TRY IT

Directions: Adding another "temporary" output statement in the loop indicating the row being processed (zero-based count) can help us see how the loop is progressing:

EXAMPLE

```
public static int checkForRowWin(String[][] board, char player) {
 // Variable to track row that holds a win
 int rowNumber = 0;
 for(int row = 0; row <= 3; row++) {
  // Temporary statement to display row number as processed
  System.out.println("Checking row " + row + " (0-based count)");
  if(board[row][0].equals(" " + player + " ") ||
   board[row][1].equals(" " + player + " ") ||
   board[row][2].equals(" " + player + " ")) {
   rowNumber = row + 1;
   // Temporary statement to print out winning row
   System.out.println(Arrays.toString(board[row]));
   break;
  }
 }
 // If rowNumber > 0, there is a winning row
 return rowNumber;
}
```

The output now shows how the loop is progressing:



```
> java TicTacToeWin.java
Checking row 0 (0-based count)
Checking row 1 (0-based count)
Checking row 2 (0-based count)
Checking row 3 (0-based count)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out
of bounds for length 3
    at TicTacToeWin.checkForRowWin(TicTacToeWin.java:22)
    at TicTacToeWin.main(TicTacToeWin.java:9)
>
```

REFLECT

Since the board consists of three rows, the zero-based indexes for the rows should be 0, 1, and 2, but the loop is clearly trying to run four times and access a row with the index 3.

TRY IT

Directions: Look at the line that starts the loop; we can find the problem:

↪ EXAMPLE

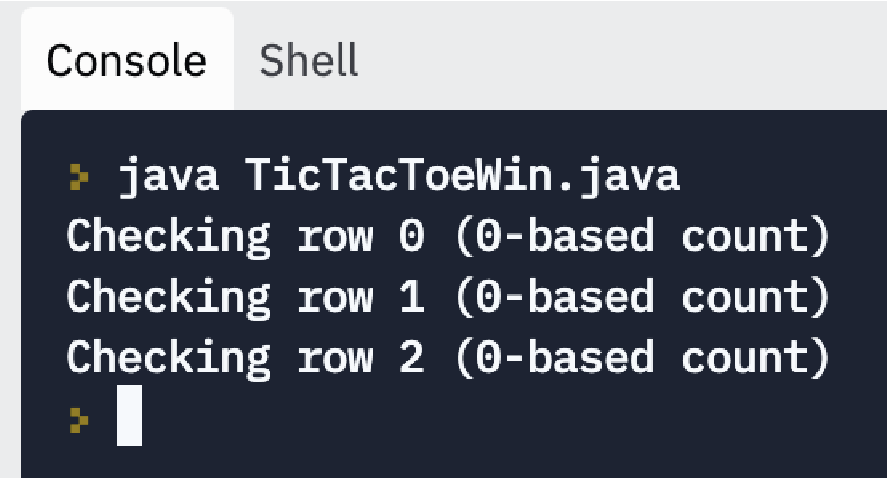
```
for(int row = 0; row <= 3; row++) {
```

The value of row should not reach 3, but the incorrect comparison operator <= (less than or equal to) causes row to run through the values 0, 1, 2, and 3. Changing the loop's definition to

↱ EXAMPLE

```
for(int row = 0; row < 3; row++) {
```

should fix the problem. If you make the change and run the code, you should get the following results:



```
> java TicTacToeWin.java
Checking row 0 (0-based count)
Checking row 1 (0-based count)
Checking row 2 (0-based count)
>
```

REFLECT

The loop is now running through the correct set of lines.

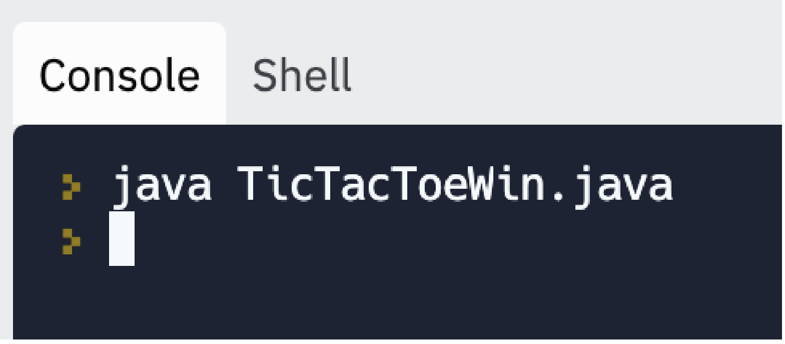
2. A Method to Detect a Tic-Tac-Toe Win in a Column

It is also important to be able to draft a method that detects a win in one of the columns.

TRY IT

Directions: The code below includes some temporary output to show the contents of the column that is found to contain a win:

```
import java.util.Arrays; // Added to print out row

class TicTacToeWin {
  public static void main(String[] args) {
    // Board pattern to test
    String[][] board = {{" X ", " O ", " X "},
                        {" X ", " O ", " X "},
                        {" O ", " O ", " - "}};
    int winningCol = checkForColumnWin(board, 'O');
    if(winningCol > 0) {
      System.out.println("Win on column " + winningCol);
    }
  }

  public static int checkForColumnWin(String[][] board, char player) {
    // Variable to track row that holds a win
    int colNumber = 0;
    for(int col = 0; col < 3; col++) {
      if(board[col][0].equals(" " + player + " ") &&
        board[col][1].equals(" " + player + " ") &&
        board[col][2].equals(" " + player + " ")) {
        colNumber = col + 1;
        // Temporary statement to print out winning row
        System.out.println("Col:\n"+ board[0][col] + "\n" +
          board[1][col] + "\n" + board[2][col]);
        break;
      }
    }
    // If colNumber > 0, there is a winning row
    return colNumber;
  }
}
```



```
> java TicTacToeWin.java
> 
```

REFLECT

Running the code in this form doesn't produce any results. How can we figure out what is going on?

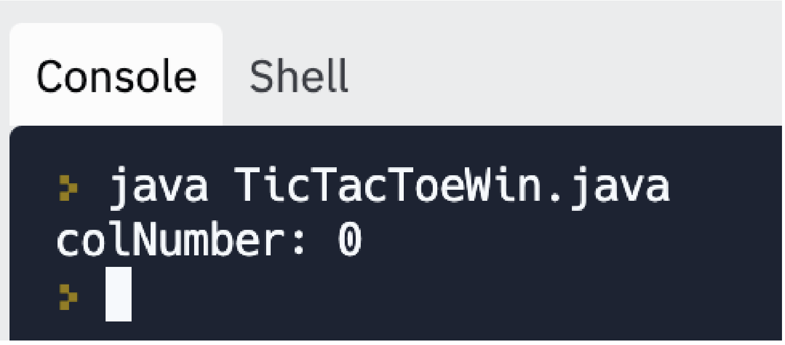
TRY IT

Directions: Add a temporary output line to show the value of colNumber before the method's return statement. Add the following lines:

EXAMPLE

```
// Temporary output
System.out.println("colNumber: " + colNumber);
return colNumber;
```

Now the result of running the code looks like this:



```
> java TicTacToeWin.java
colNumber: 0
> 
```

The added code shows that the value colNumber is 0 when it is returned. Since the code adds 1 to colNumber when a winning column is detected to convert it to a column number as a person would count them, the fact that 0 is returned indicates that no win is found in the columns.

To get a better idea of what is happening as the loop runs through the columns, it is helpful to have a couple added temporary output lines to display the contents of each column as the loop runs.

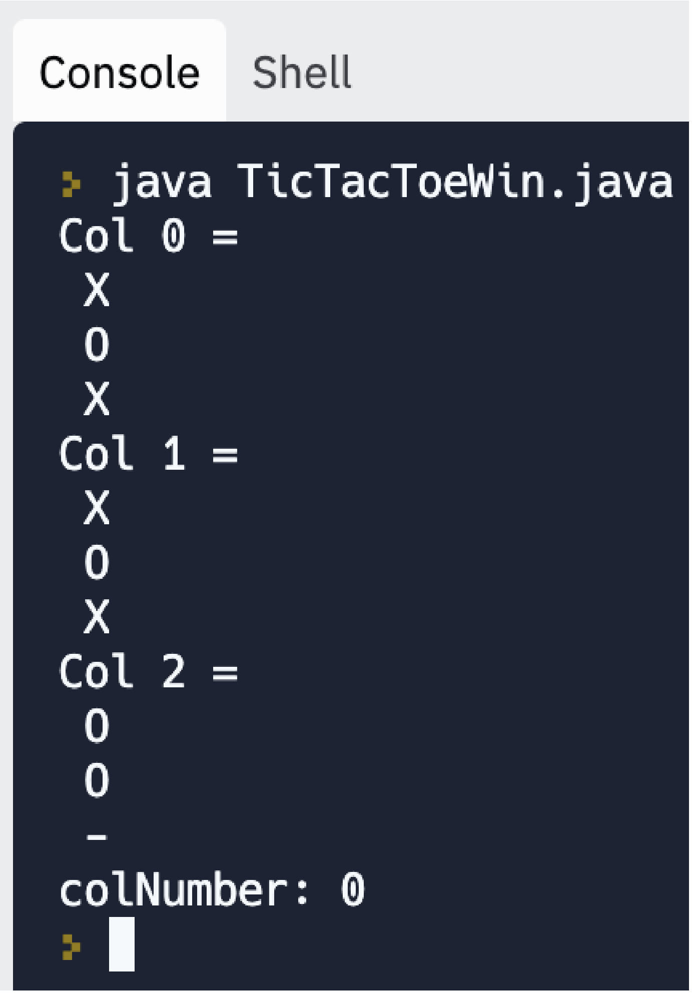
TRY IT

Directions: Add these lines before the if() statement in the body of the loop:

↪ EXAMPLE

```
// 2 lines to display contents of each column
System.out.println("Col " + col + " = ");
System.out.println(board[col][0] + "\n" + board[col][1] + "\n" + board[col][2]);
```

The added output produces these results:



```
> java TicTacToeWin.java
Col 0 =
 X
 O
 X
Col 1 =
 X
 O
 X
Col 2 =
 O
 O
 -
colNumber: 0
> 
```

REFLECT

When considering this output, it's important to remember that the array representing the board is declared and initialized like this:

EXAMPLE

```
String[][] board = {{" X ", " O ", " X "},
                    {" X ", " O ", " X "},
                    {" O ", " O ", " - "}};
```

The output lists the contents of the first column (index 0) as X O X, but we can see from the declaration of the board that the first column holds the value X X O. The sequence X O X is found in the first row, not the first column. The output shows the contents of the second column (index 1) as X O X, but the declaration shows that the second column contains O O O. Again, in this case, the X O X is found in a row (the second as well as the first), not a column. The output of the third column (index 2) also shows that the contents in the output don't correspond to the declaration, and the contents displayed for the third column actually correspond to the third row in the declaration.

Given these facts, it seems clear that the code has somehow confused columns and rows.

Let's look at the selection statement that checks each column for a win:

↪ EXAMPLE

```
if(board[col][0].equals(" " + player + " ") &&
  board[col][1].equals(" " + player + " ") &&
  board[col][2].equals(" " + player + " ")) {
  colNumber = col + 1;
  // Temporary statement to print out winning row
  System.out.println("Col:\n"+ board[0][col] + "\n" +
    board[1][col] + "\n" + board[2][col]);
   break;
}
```

REFLECT

It's important to consider that when working with a two-dimensional array, the first index designates the row, and the second index indicates the column. Clearly, though, this code that checks for a win has mixed up the columns and rows.

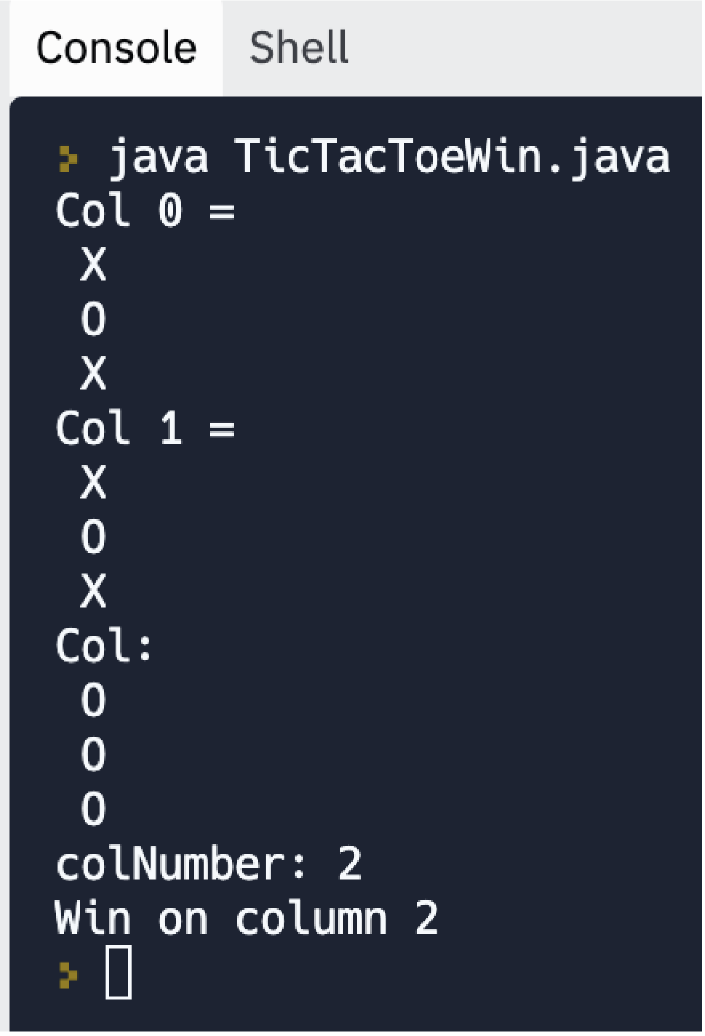
TRY IT

Directions: Next, redo this section of the code so that the first index is the row and the second is the column:

↪ EXAMPLE

```
if(board[0][col].equals(" " + player + " ") &&
  board[1][col].equals(" " + player + " ") &&
  board[2][col].equals(" " + player + " ")) {
  colNumber = col + 1;
   // Temporary statement to print out winning row
  System.out.println("Col:\n"+ board[0][col] + "\n" +
    board[1][col] + "\n" + board[2][col]);
  break;
}
```

With the correction to the order of the indices made, the program (which still includes the extra debugging statements) displays the following output when run:



```
> java TicTacToeWin.java
Col 0 =
 X
 O
 X
Col 1 =
 X
 O
 X
Col:
 O
 O
 O
colNumber: 2
Win on column 2
> 
```

REFLECT

The third column (index 2) is indeed the winning column with three O's.

TRY IT

Directions: Remove the extra temporary output statements; the correct version of the draft method looks like this:

```
public static int checkForColumnWin(String[][] board, char player) {
  // Variable to track row that holds a win
  int colNumber = 0;
  for(int col = 0; col < 3; col++) {
    if(board[0][col].equals(" " + player + " ") &&
      board[1][col].equals(" " + player + " ") &&
      board[2][col].equals(" " + player + " ")) {
      // Correct for human count (starting with 1, not 0)
      colNumber = col + 1;
      break;
    }
  }
  // If colNumber > 0, there is a winning row
  return colNumber;
}
```

REFLECT

Removing the temporary output statements gets rid of the extra output that was for the developer but not the end user. After removing the extra output, it is important to make sure that the program compiles and runs correctly.

SUMMARY

In this lesson, we have looked at a couple drafts of **methods to detect a win in a row** or **methods to detect a win in a column** in the Tic-Tac-Toe game. You learned that the initial draft versions contained errors, and we have seen how temporary output statements can be used to show how the process of checking the rows or columns is progressing. You also learned that while working on these methods in isolation forms, the rest of the code has made it possible to work out how to handle win detection. In the final lesson for this challenge, we will get these methods worked into the complete game and add the other necessary methods to get the game working correctly.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

Finishing the Tic-Tac-Toe Program

by Sophia

WHAT'S COVERED

In this lesson, you will finish our Tic-Tac-Toe program to make use of methods. Specifically, this lesson covers:

1. Methods to Improve the Board

Over the course of the game, the board will need to be redrawn many times to reflect the current state. This task is best encapsulated in a separate method. As with previous versions of the game, the method can use a loop to iterate over the array. It can also print out the positions marked by each player.

The method needs data about the board, so the two-dimensional array will need to be passed as a parameter. Since this method will print out the current state of the game to the console, there is no need for a return value.

Here is a reasonable design for the method:

EXAMPLE

```
public static void drawGame(String[][] board) {
 for(int row = 0; row < board.length; row++) {
  System.out.println(Arrays.toString(board[row]));
 }
 System.out.println(); // Add a blank line after drawing the board
}
```

Since this is a static method, no variable is needed to capture the value returned.

The board is initially declared like this in the main() method:

EXAMPLE

```
String[][] board = {{" - ", " - ", " - "},
          {" - ", " - ", " - "},
          {" - ", " - ", " - "}};
```

The drawGame() method can be called like this (also from within the body of the main() method) drawGame(board), by checking that the selection is a valid row and column.

Another aspect of working with the Tic-Tac-Toe board is checking that a selected position is valid. The size of the board (three rows and three columns) is a fixed quantity, so the method that checks if a selected position is a valid position on the board just needs the requested row and column. This method needs to determine if the position is valid or invalid (a two-way distinction). The method can indicate this by returning a boolean value (true or false).

A method that returns true if the selected position is not out of the valid range of rows and columns:

EXAMPLE

```
public static boolean isValidSelection(int row, int column) {
  // ! needed to return true if selection is valid position
  return !(row < 1 || row > 3 || column < 1 || column > 3);
}
```

Another approach to checking that the player has chosen a valid row and column is to turn the logic around and check if the selection is within the valid range of rows and within the valid range of columns.

The if() statement in this case looks like this:

EXAMPLE

```
return (row >= 1 && row <= 3 && column >= 1 && column <= 3);
```

CONCEPT TO KNOW

Note that this second version needs to use logical && (and) rather than || (or) since it tests that all of the tests pass (rather than checking if any of the tests fail).

There is another common functionality related to the board that needs to be considered as well. When the user selects a space on the board, the game needs to check and make sure that the space isn't already taken. For this method to do its work, it needs the array with the data about the current state of the board and the integers for the row and column. This method, too, can return a boolean to indicate if the space is already taken. A position on the board is open if its current value is the String " - ".

Here is a potential version of such a method:

EXAMPLE

```
public static boolean isPositionOpen(String[][] board, int row, int column) {
  return board[row - 1][column - 1].equals(" - ");
}
```

2. Methods to Check for a Winner

The next important piece of the game's functionality is determining if the game has been won. This is a more complicated process than checking that the selected position is a valid, open space. However, the process can be broken down to make it manageable.

HINT

In the previous lesson on debugging methods, you learned that methods can check the rows for a win and check the columns for a win. You also learned that a method can be used to check for a win on a diagonal. Here are the methods that the previous tutorial arrived at after making the needed corrections.

The checkForRowWin() method ended up like this:

EXAMPLE

```
public static int checkForRowWin(String[][] board, char player) {
 // Variable to track row that holds a win
 int rowNumber = 0;
 for(int row = 0; row <= 3; row++) {
  if(board[row][0].equals(" " + player + " ") &&
   board[row][1].equals(" " + player + " ") &&
   board[row][2].equals(" " + player + " ")) {
   // Add 1 to row number to match human count (starting with 1)
   rowNumber = row + 1;
   break;
  }
 }
 // If rowNumber > 0, there is a winning row
 return rowNumber;
}
```

The final version of the method for checking the columns for a win looks like this:

EXAMPLE

```
public static int checkForColumnWin(String[][] board, char player) {
 // Variable to track row that holds a win
 int colNumber = 0;
 for(int col = 0; col < 3; col++) {
  if(board[0][col].equals(" " + player + " ") &&
   board[1][col].equals(" " + player + " ") &&
   board[2][col].equals(" " + player + " ")) {
   // Correct for human count (starting with 1, not 0)
   colNumber = col + 1;
   break;
  }
 }
 // If colNumber > 0, there is a winning row
 return colNumber;
}
```

Now the game needs a method to check for a diagonal win. While there are three rows to check for a win and three columns, there are only two ways to win diagonally. The first is a run of three of the same character from the top left corner to the lower right corner, and the second is a run of three of the same character running from the top right corner to the lower left. In either case, a win is only possible if the winner has taken the middle square (the intersection of the second row and the second column).

Like the other two methods that check for a win, this method takes two parameters. These include the array containing the current state of the board and a character representing the player for whom the method is checking for a win.

The resulting method should be set up like this:

EXAMPLE

```
public static int checkForDiagonalWin(String[][] board, char player) {
  int diagonalNumber = 0;
  if(board[1][1].equals(" " + player + " ")) {
    if(board[0][0].equals(" " + player + " ") &&
       &nbsp;board[2][2].equals(" " + player + " ")) {
      diagonalNumber = 1;
    }
    else if(board[0][2].equals(" " + player + " ") &&
        board[2][0].equals(" " + player + " ")) {
      diagonalNumber = 2;
    }
  }
  return diagonalNumber;
}
```

With these methods added to the code for the program, the next step is to work out how to call them to check for a win. Keep in mind that each of the methods that detects a win returns a value greater than 0 if the player whose positions are being checked has won.

Next, you would call the three methods in turn and retain the result for each check. Assume that these statements are in the body of a for loop that limits the game to nine turns using the loop variable turn.

The player array consists of 'X' and 'O', so the modulus operator gets the index of the current player's mark:

EXAMPLE

```
int winningRow = checkForRowWin(board, player[turn % 2]);
int winningColumn = checkForColumnWin(board, player[turn % 2]);
int winningDiagonal = checkForDiagonalWin(board, player[turn % 2]);
```

Since the winner has to have taken one of the rows, one of the columns, or one of the diagonals, the overall process for checking for a win involves using the Boolean || (or) operator.

A selection statement like this will compare the values:

↪ EXAMPLE

```
if(winningRow > 0 || winningColumn > 0 || winningDiagonal > 0) {
  System.out.println("\n" + player[turn % 2] + " wins!\n");
  break;
}
```

It is assumed that this code is in the body of the for loop that handles the turns. The break statement is needed, since there is no point in continuing the game after a player has won.

Since there is no way for either player to win before the fifth iteration of the loop (when X has had three turns and O has had two), there is no need to check for a win until the fifth iteration.

TRY IT

Directions: To wrap up your work for this unit, start a new file in Replit named TicTacToeMethods.java and assemble the pieces (methods), along with the code in main(), to produce a complete game.

The overall process that is needed is to define the methods covered, with the appropriate parameters and return types:

1. drawGame()
2. isValidSelection()
3. isPositionOpen()
4. checkForRowWin()
5. checkForColumnWin()
6. checkForDiagonalWin()

With these methods set up, the next task is to get the code in place for the main() method:

STEP BY STEP

1. Declare and initialize the two-dimensional array for the board.
2. Declare and initialize the one-dimensional char array with the symbols for the players.
3. Declare and initialize integers for col and row (with initial value 0).
4. Declare a validSelection boolean with an initial value of false (so the user is prompted for #input until the entry is valid).
5. Declare a Scanner for reading the user input.
6. Draw the initial empty board.
7. Set up a for loop to run up to nine times to handle the turns. (Even turns are X's turns and odd turns are O's turns).
8. Inside the for loop that handles the turns, set up a while loop that runs until the user makes a valid move.
9. In the while loop, prompt for the chosen row and column and read the input.
10. Check if the selected position is valid.
11. If the selected position is valid, check if the selected space is open.

12. If the position is valid and open, set the character for the select square and set variable so that the while loop ends.
13. Redraw the board.
14. Check for a win if the turn is the 5th iteration of the for loop.
15. At the bottom of the for loop, set the variable for the while loop so that the loop will run during the next iteration of the for loop.

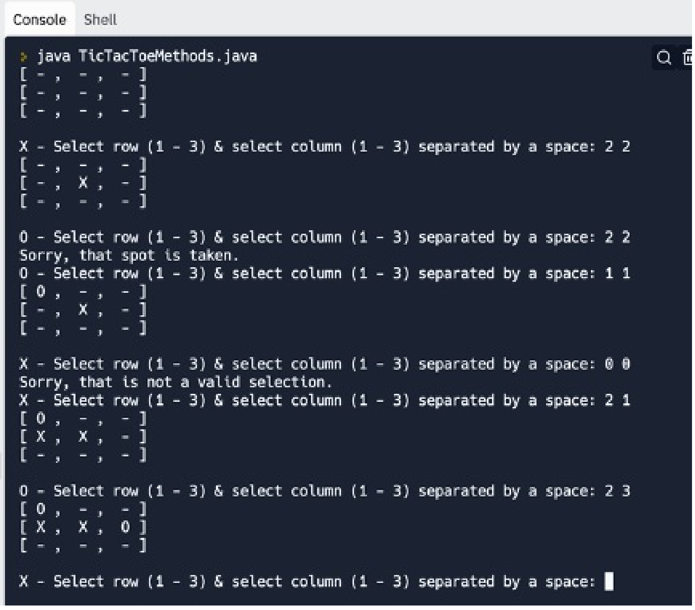
There is some room for variation, but here is a working implementation of this process:

```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeMethods {
 public static void main(String[] args) {
  String[][] board = {{" - ", " - ", " - "},
            {" - ", " - ", " - "},
            {" - ", " - ", " - "}};
  // Array of player marks
  char[] player = {'X', 'O'};

  int col = 0;
  int row = 0;
  // Initialized so input while loop runs until entry is valid
  boolean validSelection = false;

  Scanner input = new Scanner(System.in);

  // Draw initial empty board
  drawGame(board);

    // Start turns. Even turns are X, odd turns are O
  for(int turn = 0; turn < 9; turn++) {
   // Prompt for input until the user makes a valid move
   while(!validSelection) {
    System.out.print(player[turn % 2] + " - Select row (1 - 3) & " +
     "select column (1 - 3) separated by a space: ");
    row = input.nextInt();
    col = input.nextInt();
    if(isValidSelection(row, col)) {
     if(isPositionOpen(board, row, col)) {
      // Set player character at position
      board[row - 1][col - 1] = " " + player[turn % 2] + " ";
      // while loop will end when selected move is valid
      validSelection = true;
     }
     else { // Position is not open
      System.out.println("Sorry, that spot is taken.");
     }
    }
```

```
      // Not a valid position
      else {
        System.out.println("Sorry, that is not a valid selection.");
      }

    }
    drawGame(board);
    // Check if this is 5th or later turn (turn starts at 0)
    if(turn >= 4) {
      int winningRow = checkForRowWin(board, player[turn % 2]);
      int winningColumn = checkForColumnWin(board, player[turn % 2]);
      int winningDiagonal = checkForDiagonalWin(board, player[turn % 2]);
      if(winningRow > 0 || winningColumn > 0 || winningDiagonal > 0) {
        System.out.println("\n" + player[turn % 2] + " wins!\n");
        break;
      }
    }
    // Set so that while loop repeats until next player makes valid move
    validSelection = false;
  } // end of for loop for turns
} // end of main()

// Method to draw board. Parameter is a 2-d array with board data.
public static void drawGame(String[][] board) {
  for(int row = 0; row < board.length; row++) {
    System.out.println(Arrays.toString(board[row]));
  }
  System.out.println();
}

// Method to check if player has selected row & column within bounds
// Takes 2 int parameters for selected row & column.
// Returns true if selection is in bounds, false if not
public static boolean isValidSelection(int row, int column) {
  // ! needed to return true if selection is valid position
  return !(row < 1 || row > 3 || column < 1 || column > 3);
}

// Method to check if selected position is open.
// Parameters: 2-d array with board data, ints for row & column
// Returns true is selected position is open, otherwise false
public static boolean isPositionOpen(String[][] board, int row, int column)
{
  // Open squares marked with " - "
  return board[row - 1][column - 1].equals(" - ");
}

// Check if a player has a win in 1 of the rows.
// Parameters: 2-d array with board data, char ('X' or 'O') for player
// Returns 0 if no winning row, or 1, 2, or 3 if winning row
```

```
  public static int checkForRowWin(String[][] board, char player) {
    int rowNumber = 0;
    for(int row = 0; row < 3; row++) {
      if(board[row][0].equals(" " + player + " ") &&
          board[row][1].equals(" " + player + " ") &&
          board[row][2].equals(" " + player + " ")) {
        rowNumber = row + 1;
        break;
      }
    }
    return rowNumber;
  }

  // Check if a player has a win in 1 of the columns.
  // Parameters: 2-d array with board data, char ('X' or 'O') for player
  // Returns 0 if no winning col, or 1, 2, or 3 if winning col
  public static int checkForColumnWin(String[][] board, char player) {
    int columnNumber = 0;
    for(int column = 0; column < 3; column++) {
      if(board[0][columnNumber].equals(" " + player + " ") &&
          board[1][columnNumber].equals(" " + player + " ") &&
          board[2][columnNumber].equals(" " + player + " ")) {
        columnNumber = column + 1;
      }
    }
    return columnNumber;
  }

  // Check if a player has a win in 1 of the diagonals.
  // Parameters: 2-d array with board data, char ('X' or 'O') for player
  // Returns 0 if no winning diagonal, or 1 or 2 if winning diagonal

  public static int checkForDiagonalWin(String[][] board, char player) {
    int diagonalNumber = 0;
    // Player must have central square to have winning diagonal
    if(board[1][1].equals(" " + player + " ")) {
      if(board[0][0].equals(" " + player + " ") &&
          board[2][2].equals(" " + player + " ")) {
        diagonalNumber = 1;
      }
      else if(board[0][2].equals(" " + player + " ") &&
          board[2][0].equals(" " + player + " ")) {
        diagonalNumber = 2;
      }
    }
    return diagonalNumber;
  }

} // end of TicTacToeMethods class
```

Here is a screen shot of the first few turns in a game to show how this program works:



```
> java TicTacToeMethods.java
[ - ,  - ,  - ]
[ - ,  - ,  - ]
[ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 2
[ - ,  - ,  - ]
[ - ,  X ,  - ]
[ - ,  - ,  - ]

O - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 2
Sorry, that spot is taken.
O - Select row (1 - 3) & select column (1 - 3) separated by a space: 1 1
[ O ,  - ,  - ]
[ - ,  X ,  - ]
[ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: 0 0
Sorry, that is not a valid selection.
X - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 1
[ O ,  - ,  - ]
[ X ,  X ,  - ]
[ - ,  - ,  - ]

O - Select row (1 - 3) & select column (1 - 3) separated by a space: 2 3
[ O ,  - ,  - ]
[ X ,  X ,  O ]
[ - ,  - ,  - ]

X - Select row (1 - 3) & select column (1 - 3) separated by a space: █
```

SUMMARY

In this lesson, you have completed the Tic-Tac-Toe game. Building on work in previous lessons, you learned how to put together **methods to improve the board** and handle key parts of the functionality for the game. You learned that these methods draw the board with the current state of the game, validate a player's move, and **check for a win**. You have seen how to pass the relevant data to each method using appropriate parameters to get the result from the method via an appropriate return type.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

Terms to Know

Arguments

The actual variable or literal value passed when a method is called.

Array

An array is a named collection of contiguous storage locations—storage locations that are next to each other—that contain data items of the same type. An array also has a fixed size.

Array Index Error

An error that occurs when a programmer accesses the wrong element in an array (or ArrayList collection) or tries to access an element that does not exist.

ArrayList

An ArrayList is a flexible list collection that has similarities to an array, except that its size is not fixed.

Arrays.binarySearch()

This method is used to search for the occurrence of a specified value in a sorted array.

Arrays.copyOf()

The Arrays.copyOf() method allows users to copy an array from a source array and paste it to a destination array. The Arrays.copyOf() method uses two parameters. These include the source array and the length or size of the copied array.

Arrays.copyOfRange()

This method allows copying part of an array to a destination. This method is called with values passed for the source array, the starting index for the copy, and the ending index for the copy.

Arrays.deepToString()

A Java method that converts data in a two-dimensional array to a format that can be displayed on the screen.

Arrays.sort()

The Arrays.sort() method arranges all of the elements in an array from the lowest to highest. If the values are strings, the result will be a list of strings in alphabetical order.

Arrays.toString()

The Arrays.toString() method converts the contents of an array to String values that can be displayed on the screen.

Assignment Operator

The assignment operator is a single equal sign (=) that is used to assign a value on the right side of the equal sign to an array element (or other variable) on the left of the equal sign.

Body of the Loop

The body of the loop represents the indented block of code that should be executed repeatedly within the loop during each loop iteration.

Collection

A collection is a container that allows multiple values of the same data type to be stored. The main difference between an array and a collection is that the size of an array is fixed, but collections can change size.

Constant

A named value of a specific data type stored in memory that cannot be changed after it has been given an initial value.

Debugging By Bisection

Debugging by bisection is the practice of breaking your program code into “sections” for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

Definite Iteration

Definite iteration identifies the number of times, or the maximum number of times, the block of code runs. This should be explicitly defined when the loop actually starts. Typically, this is implemented using the for loop. The for loop has a specific start and endpoint.

Dimension

A dimension of an array is an axis along which the elements are organized.

Element

An element is one of the values in an array.

Enhanced for Loop (also called a For - Each loop or Range-Based for Loop)

A special version of the Java for loop designed to work with arrays and collections that provides a simpler way to set up a for loop.

Generic Method

A Java method written using generics that allow the same method to work on different data types.

Generic Programming (also Generic Types or Generics)

A type of programming that allows programmers to use the same code and structures for different data types without having to rewrite the code.

HashMap

A HashMap is a Java collection that stores key-value pairs. The key is a unique value that identifies the pair, and the value is the data associated with the key.

HashSet

A HashSet is a Java collection that stores unique values (i.e., there are no duplicate values).

Increment Operator (++)

The ++ operator increases the value in the variable immediately to the left of the operator by one.

Indefinite Iteration

A loop that does not specify, in advance, how many times it is to be run. It repeats as long as a condition is met. In Java, indefinite loops are typically created using the while loop.

Index

The index (also known as the subscript or position) is an integer value that indicates an element in an array.

IndexOutOfBoundsException

An exception (runtime error) thrown when code tries to access an element that is outside the bounds of an array.

Infinite Loop

An infinite loop is a loop in which the terminating condition is never satisfied or for which there is no terminating condition.

Inner Loop

A loop that is contained in the body of another loop.

Iterable

A collection is iterable if the collection provides a mechanism for the items in the collection to be accessed in a fixed order.

Iteration Variable

The variable that changes each time the loop executes and controls when the loop finishes.

Length (or Size) of an Array

The length or size of an array is the number of elements in the array. The .length property is used to get the length or size of an array.

Loop

In Java, the definite iteration loops are generally called for loops.

Mutable

Mutable means that we're able to change the value of items in an array.

Nested Loop

A nested loop is any loop that occurs inside the body of another loop.

Outer Loop

A loop that contains one or more other loops in its body.

Overloaded

A method is overloaded if different versions of the method exist that differ in the number or type of parameters in the method's signature.

Parameter (also called a Formal Parameter)

A kind of variable that is used as a channel to pass data to a method.

Return Type

The data type of the value returned by a method.

Three-Dimensional (or 3D) Array

A three-dimensional array adds a 3rd index or axis to the organization of the array elements.

Two-Dimensional (or 2D) Array

A two-dimensional array is an array of arrays with data laid out in a grid-like pattern of rows and columns.

Validation

The process in a program of checking that input is in a valid range and appropriate in the current state of the program.

Variables with Local Scope (more commonly called Local Variables)

Variables that can only be accessed between their point of declaration and the end of the block of code.

break

This is a reserved keyword that creates a break statement for loops. The break statement can immediately terminate a loop entirely and disregard the execution of the loop. When this occurs, the program goes to the first statement/line of code after the loop.

continue

This is a reserved keyword that creates a continue statement for loops. The continue statement will end the current loop iteration, meaning that the execution jumps back to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.

do...while Loop

A do...while loop is a specialized type of while loop where the condition is evaluated at the bottom of the loop rather than the top of the loop, so the loop will always run at least one time.

keySet()

The keySet() method returns a Java Set collection containing the keys used in a HashMap. This Set of keys is iterable, unlike the HashSet itself, which is not.

return

The reserved keyword return is used to exit a method and return a value, if the method returns a value.

void

The return type void indicates that a method does not return a value.

while Loop

The while loop is one of the most commonly used loops. It keeps going as long as some condition is true.