

Memory in Java

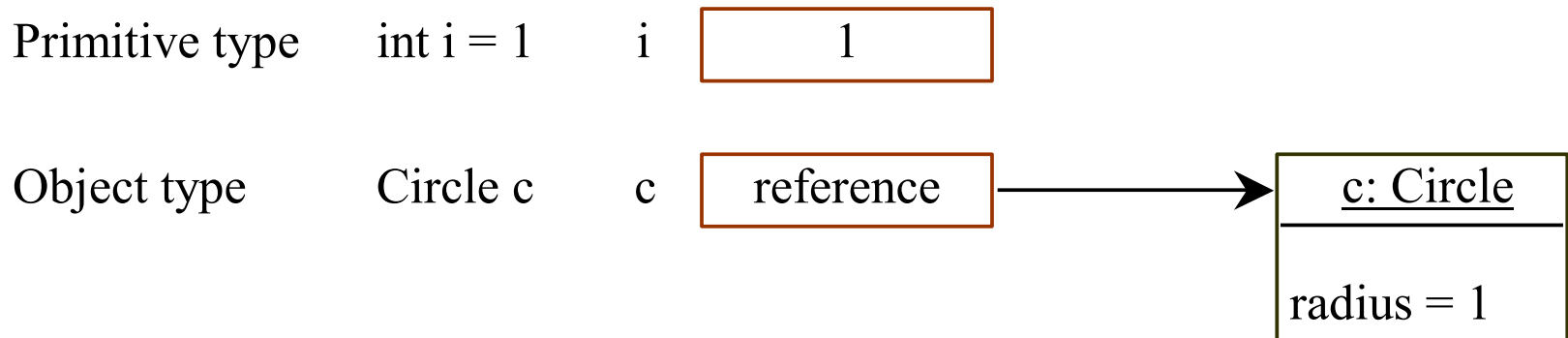
Introduction to Programming and
Computational Problem Solving:
Accelerated Pace

CSE 11

Lecture 6

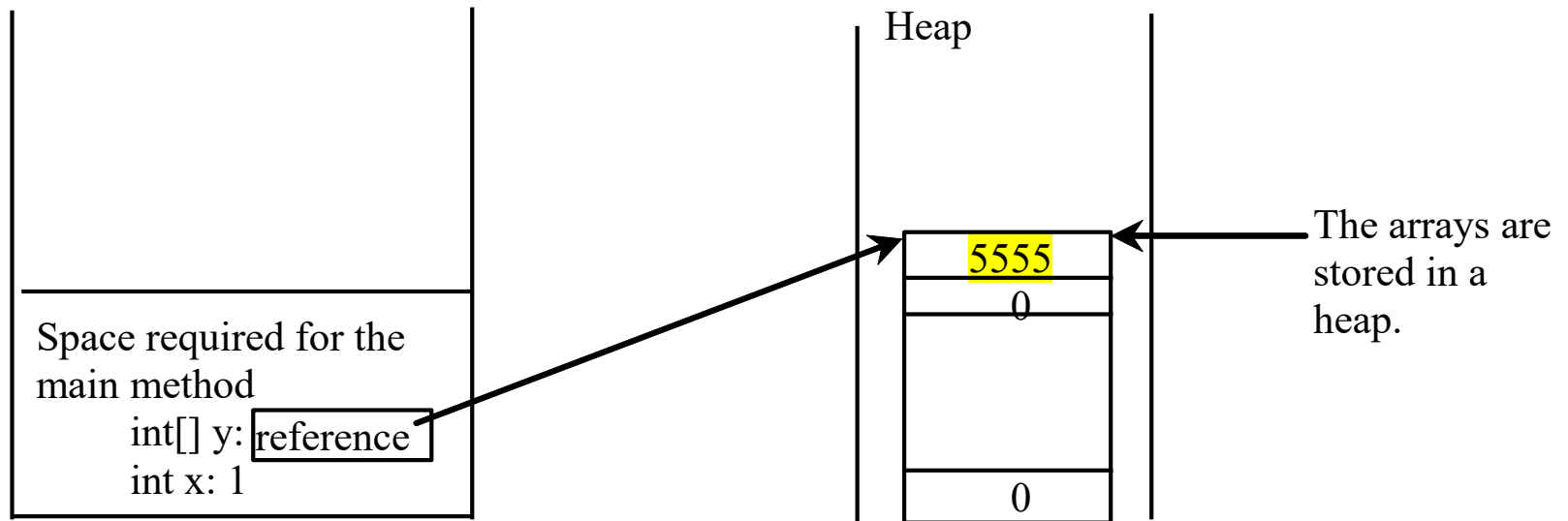
Differences between variables of primitive data types, and array or object types

- A variable of a **primitive type** holds a value of the primitive type
- A variable of a **reference type** holds a reference to where an array or object is stored in memory



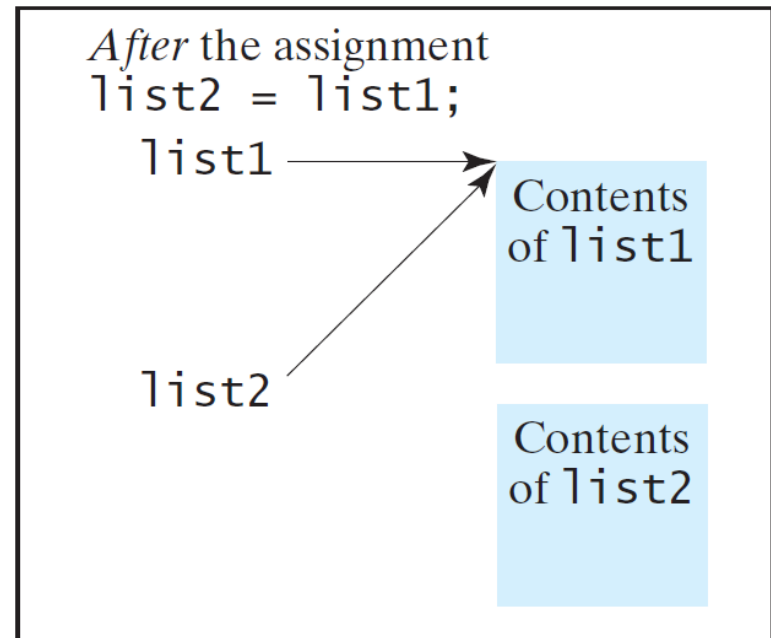
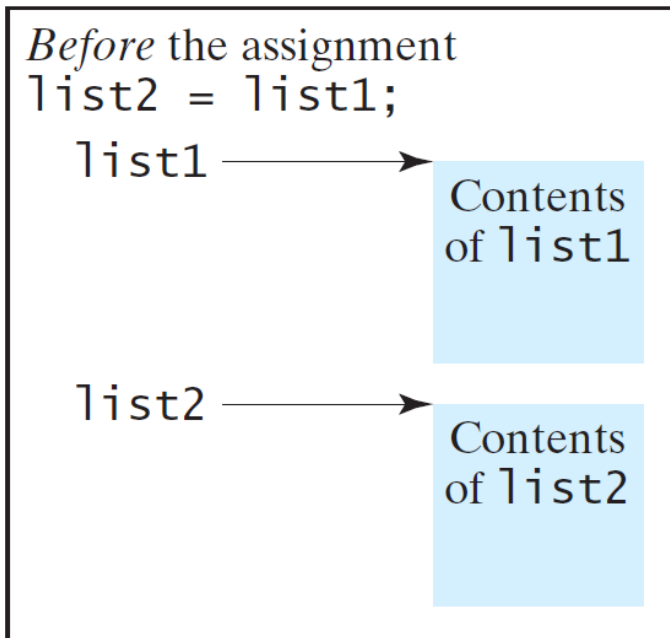
Heap

- The Java Virtual Machine (JVM) stores arrays and objects in an area of memory called the *heap*, which is used for dynamic memory allocation



Assigning arrays

- **The assignment statement does not copy the contents, it only copies the *reference value***
`list2 = list1;`



Differences between variables of primitive data types, and array or object types

- Variable assignment

Primitive type assignment $i = j$

Before:

i 1

j 2

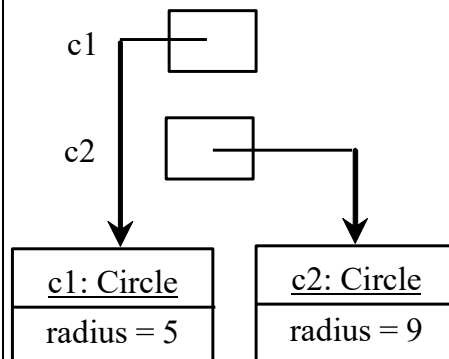
After:

i 2

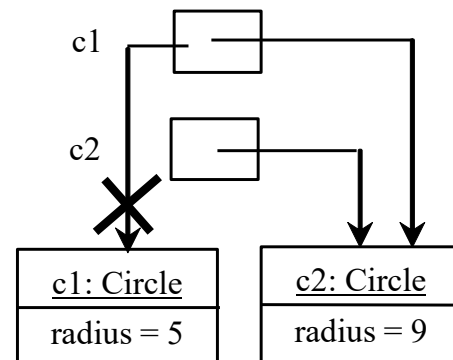
j 2

Object type assignment $c1 = c2$

Before:



After:



Passing arrays or objects to methods

- When passing an array or object to a method, the **reference value** of the array or object is passed to the method

```
public static void printArray(int[] array) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

Invoke the method, example 1:

```
int[] list = {3, 1, 2, 6, 4, 2};  
printArray(list);
```

Invoke the method, example 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

Pass by value

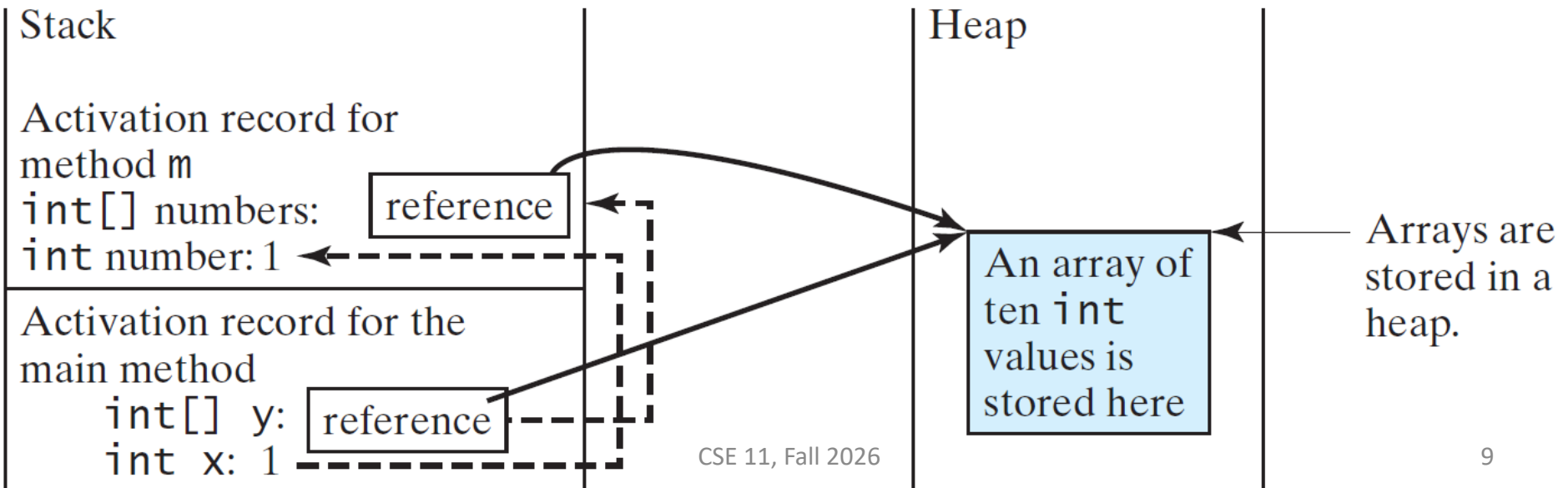
- Remember, Java uses **pass by value** to pass arguments to a method
- For a parameter of a **primitive type**, the **actual value** is passed
 - Changing the value of the local parameter inside the method **does not affect** the value of the variable outside the method
- For a parameter of an **array or object type**, the **reference value** is passed
 - Any changes to the array or object that occur inside the method body **will affect** the original array or object that was passed as the argument

Pass by value

```
public class Test {  
    public static void main(String[] args) {  
        int x = 1; // x represents an int value  
        int[] y = new int[10]; // y represents an array of int values  
  
        m(x, y); // Invoke m with arguments x and y  
  
        System.out.println("x is " + x);  
        System.out.println("y[0] is " + y[0]);  
    }  
  
    public static void m(int number, int[] numbers) {  
        number = 1001; // Assign a new value to number  
        numbers[0] = 5555; // Assign a new value to numbers[0]  
    }  
}
```

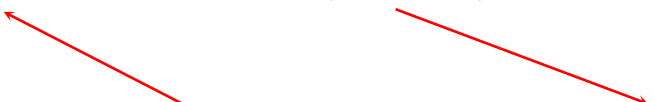
Pass by value

- When invoking $m(x, y)$, the values of x and y are passed to `number` and `numbers`
- Since y contains the reference value to the array, `numbers` now contains the same reference value to the same array



Returning an array from a method

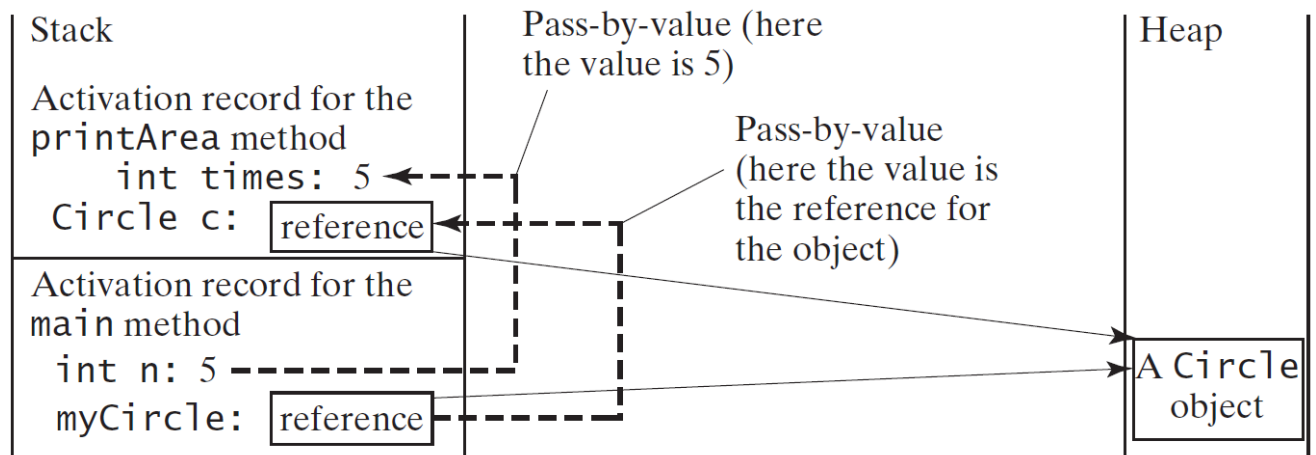
```
int[] list1 = {1, 2, 3, 4, 5, 6};  
int[] list2 = reverse(list1);  
  
public static int[] reverse(int[] list) {  
    int[] result = new int[list.length];  
  
    int j = result.length - 1;  
    for (int i = 0; i < list.length; i++) {  
        result[j] = list[i];  
        j--;  
    }  
  
    return result;  
}
```

Two red arrows originate from the 'reverse' method call in the second line of code. One arrow points to the 'reverse' method name, and the other points to the 'list1' argument. Both arrows converge towards the 'int[]' type declaration of the 'reverse' method signature in the line below, illustrating that the method returns an array of integers.

Passing objects to methods

```
public static void main(String[] args) {  
    Circle myCircle = new Circle(1, "red");  
    int n = 5;  
    printRadii(myCircle, n);  
}
```

```
public static void printRadii(Circle c, int times) {  
    System.out.println("Radius");  
    while (times >= 1) {  
        System.out.println(c.radius() * times);  
        times--;  
    }  
}
```



Arrays of objects

- An array can hold objects as well as primitive type values
- An array of objects is actually an array of reference variables

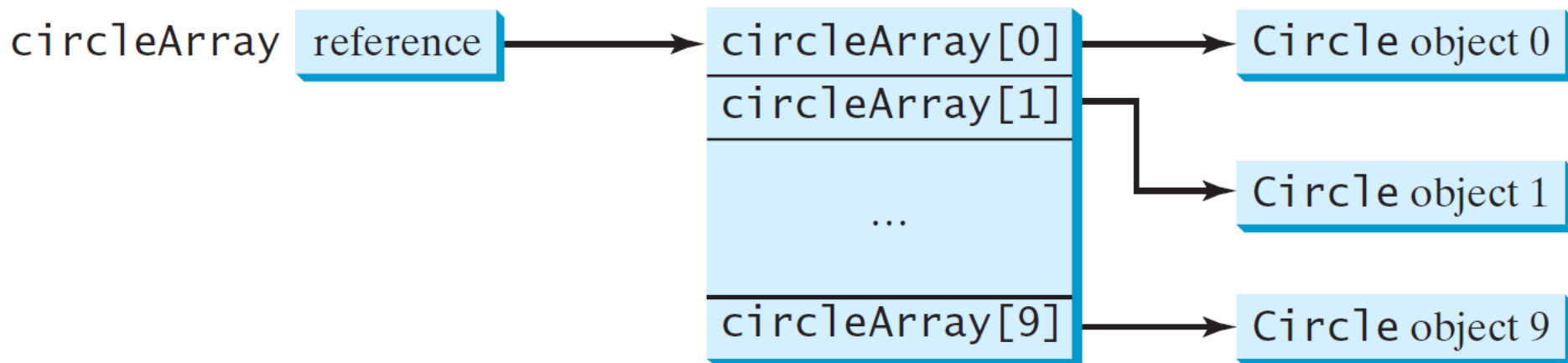
Arrays of objects

- Create an array **and** each object in it
- When creating an array using `new`, each element in the array is a reference variable with a default value of `null`

```
Circle[] circleArray = new Circle[10];  
for (int i = 0; i < circleArray.length; i++)  
{  
    circleArray[i] = new Circle(1, "red");  
}
```

Arrays of objects

- Invoking `circleArray[1].radius()` involves two levels of referencing
 - `circleArray` references to the entire array
 - `circleArray[1]` references to a `Circle` object



Garbage and its collection

- If an array or object is no longer referenced, then it is considered **garbage**
- Garbage occupies memory space
- Garbage collection
 - The Java Virtual Machine (JVM) automatically detects garbage and reclaims the space it occupies
- If you know an array or object is no longer needed, then you can explicitly assign `null` to the array or object reference variable

Constructing strings

- Create from a string literal
 - Syntax

```
String newString = new String(stringLiteral);
```
 - Example

```
String message = new String("Welcome to Java");
```
 - Since strings are used frequently, Java provides a shorthand initializer for creating a string

```
String message = "Welcome to Java";
```
- Create from an array of characters
 - Syntax

```
String newString = new String(charArray);
```

 - where, for example

```
char[] charArray = {'U', 'C', 'S', 'D', ' ', 'C', 'S', 'E'};
```

Strings are immutable

- A `String` object is immutable (i.e., its contents cannot be changed once the string is created)
- The following code does not change the contents of the string

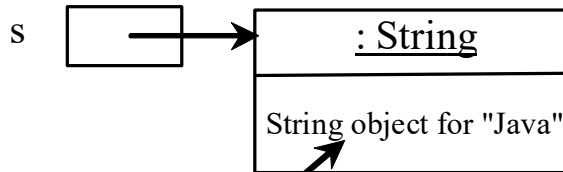
```
String s = "Java";  
s = "HTML";
```

Strings are immutable

```
String s = "Java";
```

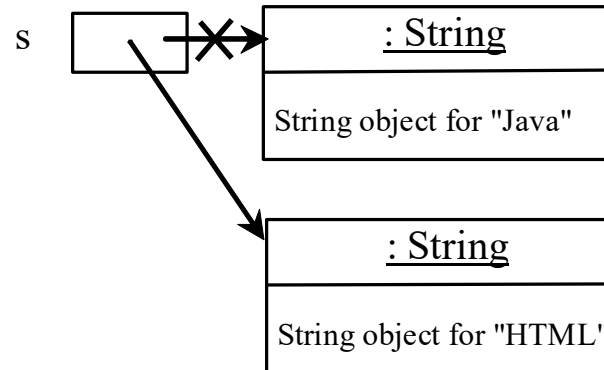
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



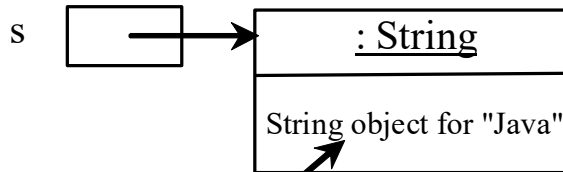
This string object is now unreferenced

Strings are immutable

```
String s = "Java";
```

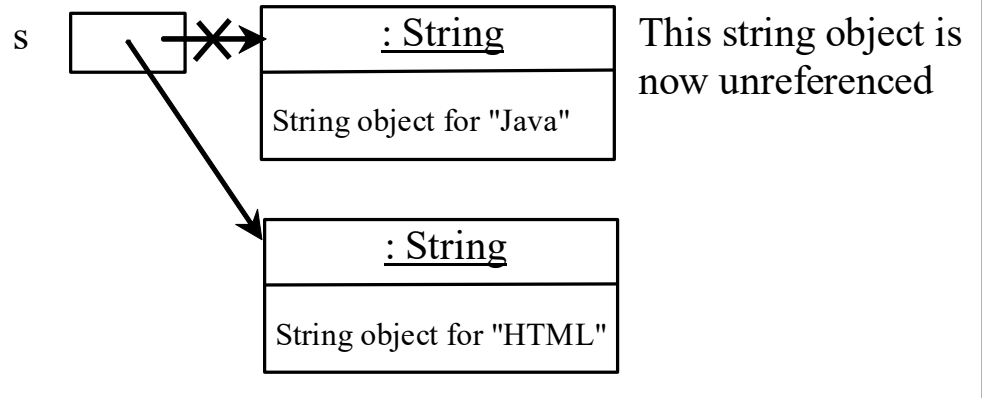
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



Interned strings

- Since strings are immutable and are frequently used, to improve efficiency and save memory, the Java Virtual Machine (JVM) uses a unique instance for string literals with the same character sequence
- Such an instance is called *interned*

Interned strings

- A new object is created if you use the new operator
- If you use the string initializer, no new object is created if the interned object is already created

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

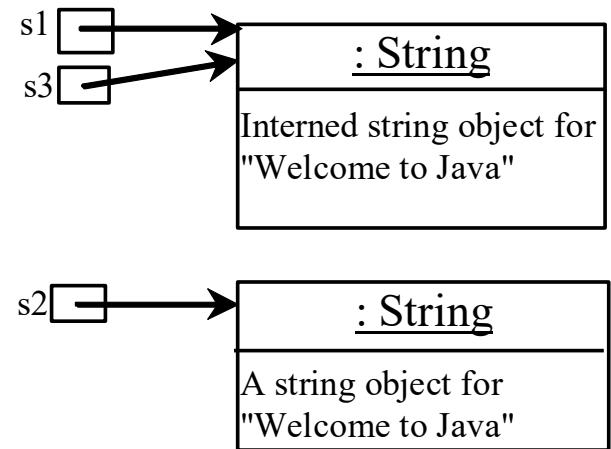
```
String s3 = "Welcome to Java";
```

```
System.out.println("s1 == s2 is " + (s1 == s2));
```

```
System.out.println("s1 == s3 is " + (s1 == s3));
```

s1 == s2 is false

s1 == s3 is true



.equals is different than ==

- Remember, the String class has **the method equals(s1)**, which returns true if and only if the argument is not null and s1 represents **the same sequence of characters** as the current String object
- Remember, for records the compiler automatically generates private final fields, public accessor methods, a canonical constructor (i.e., a constructor that accepts all components of a record in the exact order and type they are declared), and **standard implementations of equals()**, hashCode(), and toString() (we'll cover these later in the quarter)
- The java.util.Arrays class contains useful methods for common array operations

<https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html>

- java.util.Arrays.equals checks whether two arrays are strictly equal
 - Returns true if the two specified arrays are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are null.
 - Two **primitive data types** v1 and v2 are considered **equal** if v1 == v2 results in true
 - Two **objects** o1 and o2 are considered **equal** if o1.equals(o2) returns true

Next Lecture

- Nesting objects