From Python to Java: Primitives, Objects, and References and the Java Memory Model

Simple Java Program

```
import java.util.*;

public class Play {

   public static void main( String [][] args ) {

      Scanner scan = new Scanner( System.in );

      int num1 = scan.nextInt();
      int num2 = scan.nextInt();
      int num3 = scan.nextInt();

      System.out.print("The numbers entered are: " );
      System.out.prinln( num + " " + num2 + " " + num3 );
              .
              .
              .
              .
   }
}
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
double area = 125.5;
```

- `boolean` - either `true` or `false`

```
boolean isPrime = false;
```

- `char` – a single character stored using 2 bytes

```
char c = 'C';
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
double area = 125.5;
```

- `boolean` - either `true` or `false`

```
boolean isPrime = false;
```

- `char` – a single character stored using 2 bytes

```
char c = 'C'; // Unicode decimal equivalent (43)
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes
- `d` ... `imal)`
- `b`

We have seen another data type in Java (and used it's objects as well). Any ideas?

- `char` – a single character stored using 2 bytes

```
char c = 'C';
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

-

-

-

```
boolean isPrime = false;
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

A *class* is a **custom** data type!

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

-

-

-

As you will see, strings are objects of their own class: The Java String class!

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
double area = 125.5;
```

- `boolean` - either `true` or `false`

```
boolean isPrime = false;
```

Primitive types

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
double area = 125.5;
```

- `boolean` - either `true` or `false`

```
boolean isPrime = false;
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

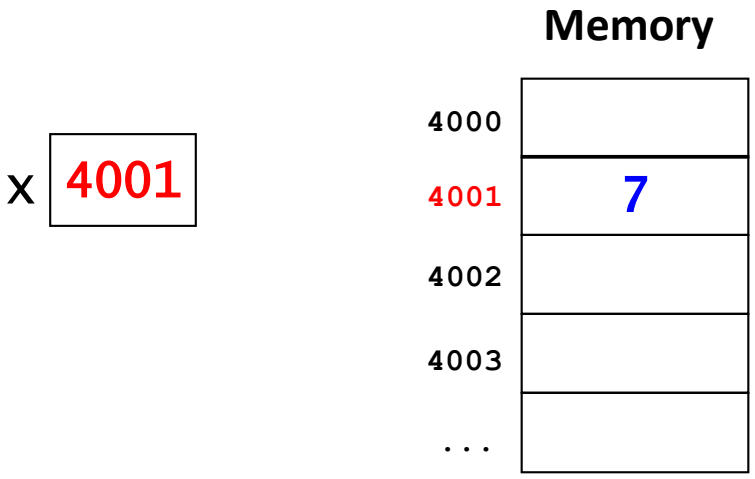
Reference types

Recall: Variables and Values in Python

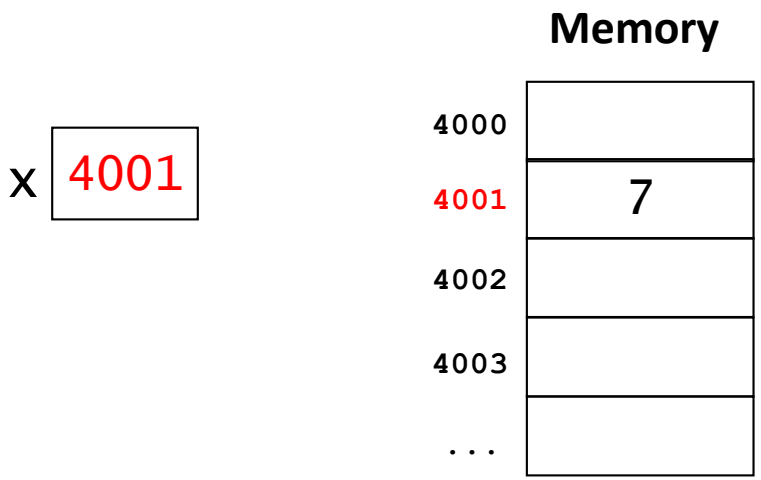
- In Python, when we assign a value to a variable, we're not actually storing the value *in* the variable.

- Rather:
  - the value is somewhere else in memory
  - the variable stores the *memory address* of the value.
  - establishing a ***reference***

- Example: x = 7



Recall: References



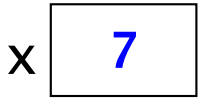
- We say that a variable stores a *reference* to its value.
  - also known as a *pointer*
- Because we don't care about the actual memory address, we use an arrow to represent a reference:

Recall: Simplifying Our Mental Model

- In Python, when a variable represents *immutable* values:
  - integers
  - floats
  - strings
  - other immutable (unchangeable) values

  it's okay to picture the value as being *inside* the variable.

x = 7

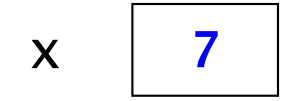


- a simplified picture, but good enough!

Primitive Types

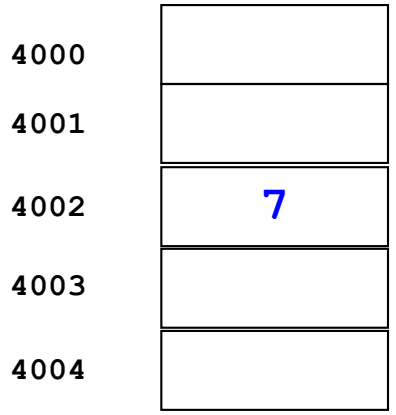
- In Java, values of *primitive* types of data **<u>are</u>** stored directly in the memory cell represented by the variable:

```
int x = 7;
```



  - **there is no reference!**

- Java *primitive types are:*
  - `int`
  - `long`
  - `double`
  - `boolean`
  - `char`
  - a few others: {short, float, byte}



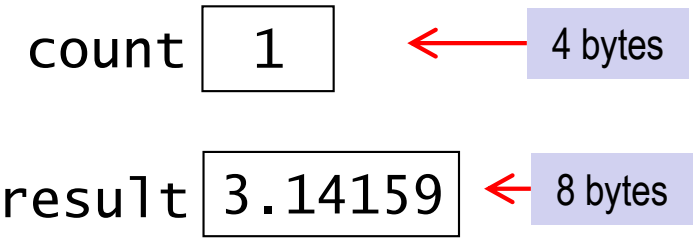
Variable Declarations and Data Types

- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	**size**
`int`	4 bytes
`double`	8 bytes
`long`	8 bytes
`boolean`	1 byte

- Declaring a variable tells the compiler how much memory *(i.e. how many bytes*) to allocate **and** the *type* of the data!

```
int count = 1;
double result = 3.14159;
```



A note about `double` and `float`

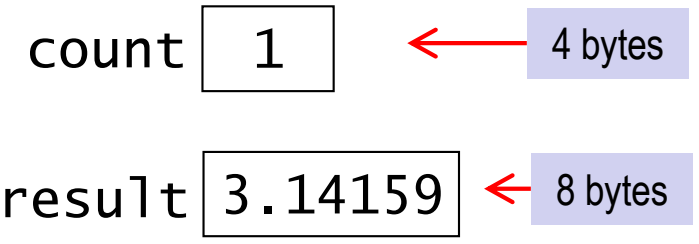
- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	size
`int`	4 bytes
`double`	8 bytes
`long`	8 bytes
`boolean`	1 byte

`float` 4 bytes

- Declaring a variable tells the compiler how much memory *(i.e. how many bytes)* to allocate **and** the *type* of the data!

```
int count = 1;
double result = 3.14159;
```



A note about `double` and `float`

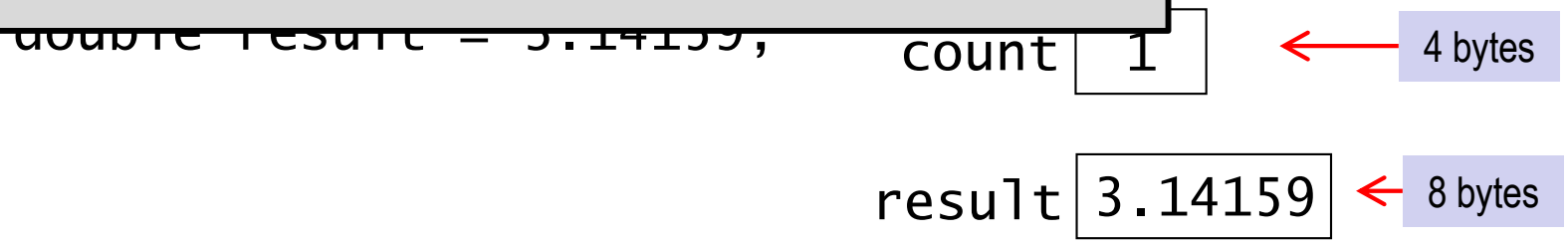
- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	size		
`int`	4 bytes		
`double`	`8 bytes`	`float`	`4 bytes`

The distinction is in the precision. Floats usually allow up to 7 decimal digits of precision, whereas doubles allow up to 15 decimal digits of precision.

```
double dval = 99.5d;
float fval = 99.7f;
```

memory
he data!



A note about `double` and `float`

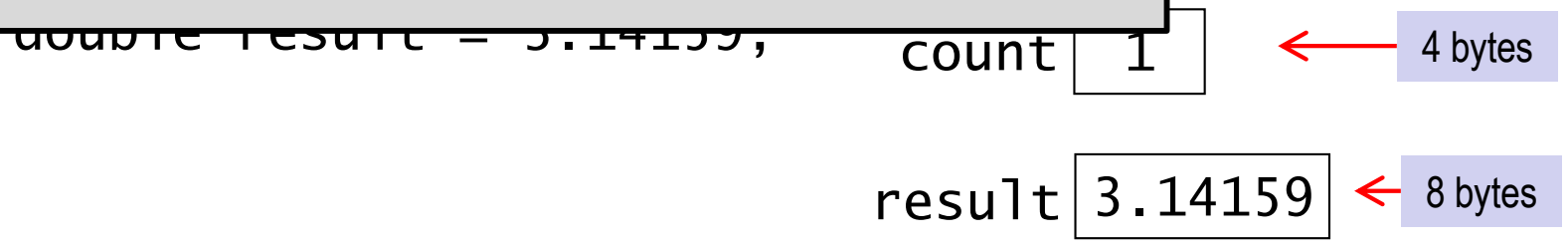
- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	size		
`int`	4 bytes		
`double`	`8 bytes`	`float`	`4 bytes`

The distinction is in the precision. Floats usually allow up to 7 decimal digits of precision, whereas doubles allow up to 15 decimal digits of precision.

```
double dval = 99.5;  // d is the default
float fval = 99.7f;  // f explicitly used
```

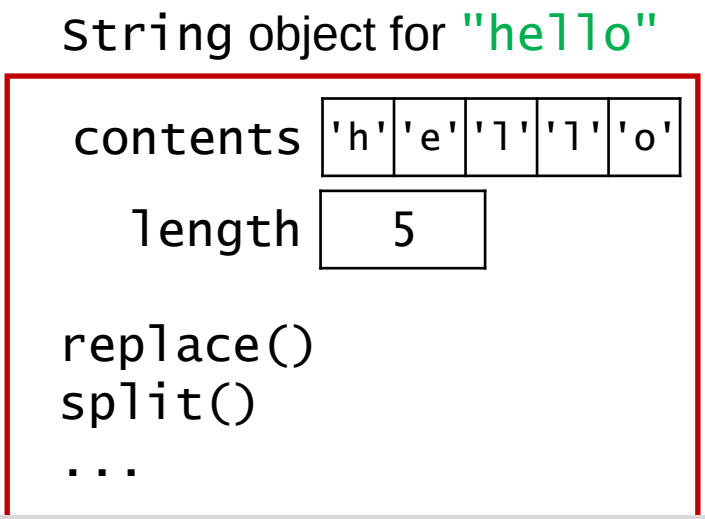
memory
he data!



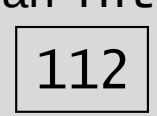
Object vs. Primitive:
summary

- An object is a *physical* construct that groups together:
  - one or more data values
    (the object's *attributes* or *fields*)
  - one or more functions
    (known as the object's *methods*)

- Every object is an *instance*
  of a class.

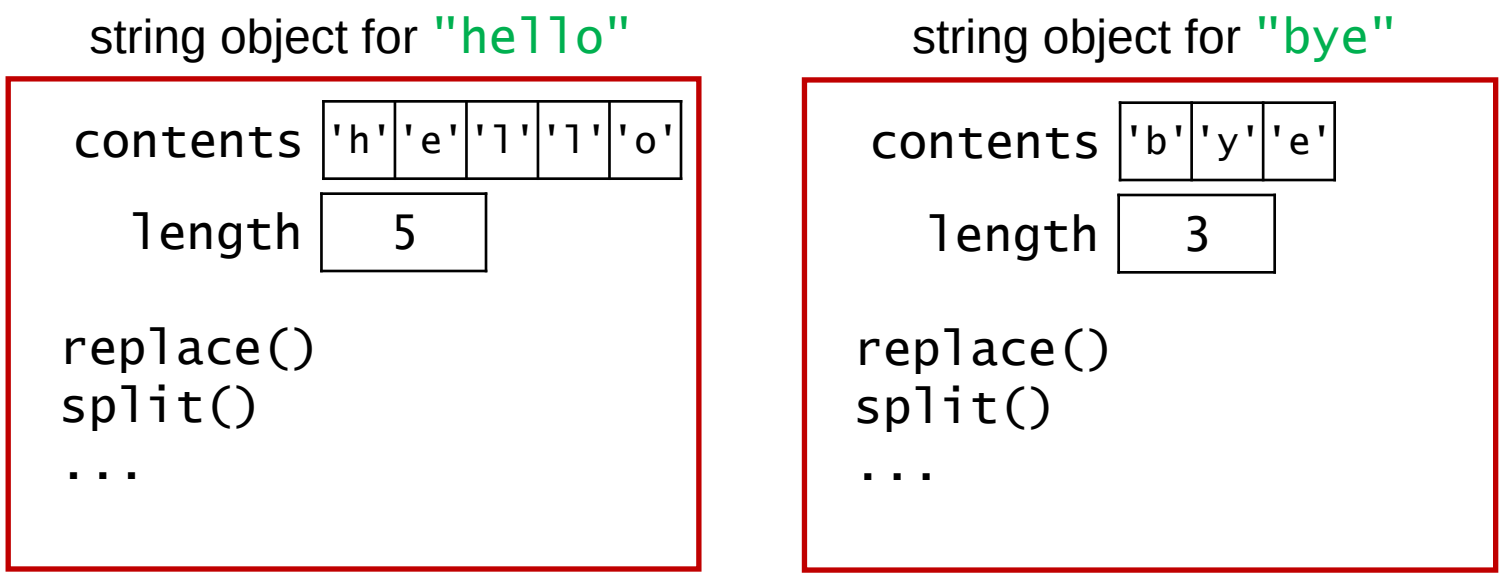


- Primitive values are *not* objects.
  - they are just "single" values
  - there is nothing else grouped with the value
  - they are not instances of a class
  - they require a fixed number of bytes based on their type
    - their value is stored in the allocated memory cell!



Strings Are Objects

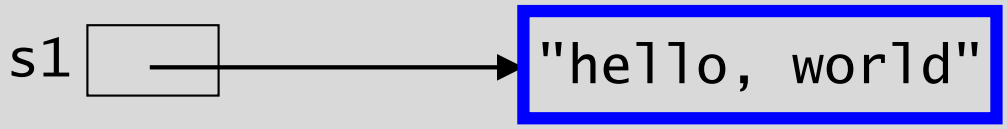
- A string is an object, an instance of class String.
  - ***attributes/fields:***
    - the characters in the string
    - the length of the string
  - ***methods:*** functions inside the string that we can use to operate on the string



Reference Types

- Java stores ***objects*** the same way that Python does:

```
String s1 = "hello, world";
```



  - the object is located elsewhere in memory
  - the variable stores a reference to the object

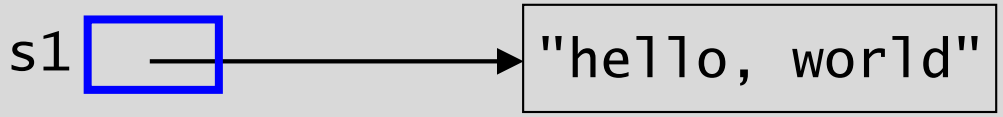
- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- We've worked with two *reference types* thus far:
  - `String`
  - `Scanner`

Reference Types

- Java stores ***objects*** the same way that Python does:

```
String s1 = "hello, world";
```



  - the object is located elsewhere in memory
  - the variable stores a reference to the object

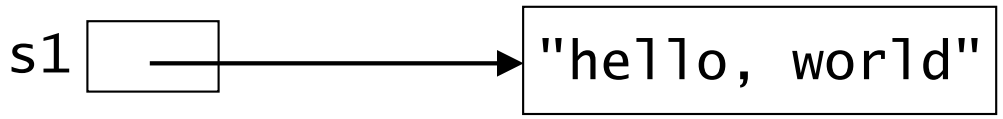
- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- We've worked with two *reference types* thus far:
  - `String`
  - `Scanner`

Reference Types

- Java stores ***objects*** the same way that Python does:

```
String s1 = "hello, world";
```



  - the object is located elsewhere in memory
  - the variable stores a reference to the object

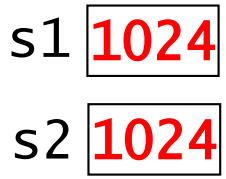
- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

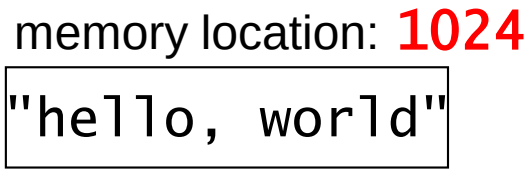
- We've worked with two *reference types* thus far:
  - `String`
  - `Scanner`

Copying References

- When we assign the value of one reference variable to another, we copy the reference to the object.
- We do *not* copy the object itself.
- Example involving strings:

```
String s1 = "hello, world";
String s2 = s1;
```

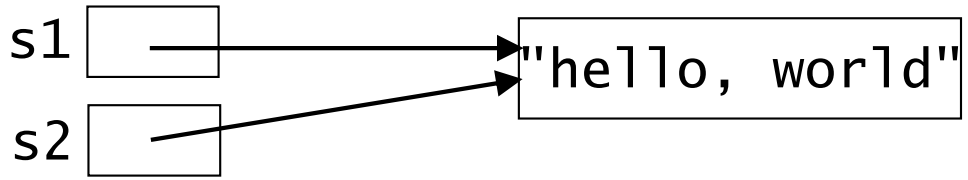




Copying References

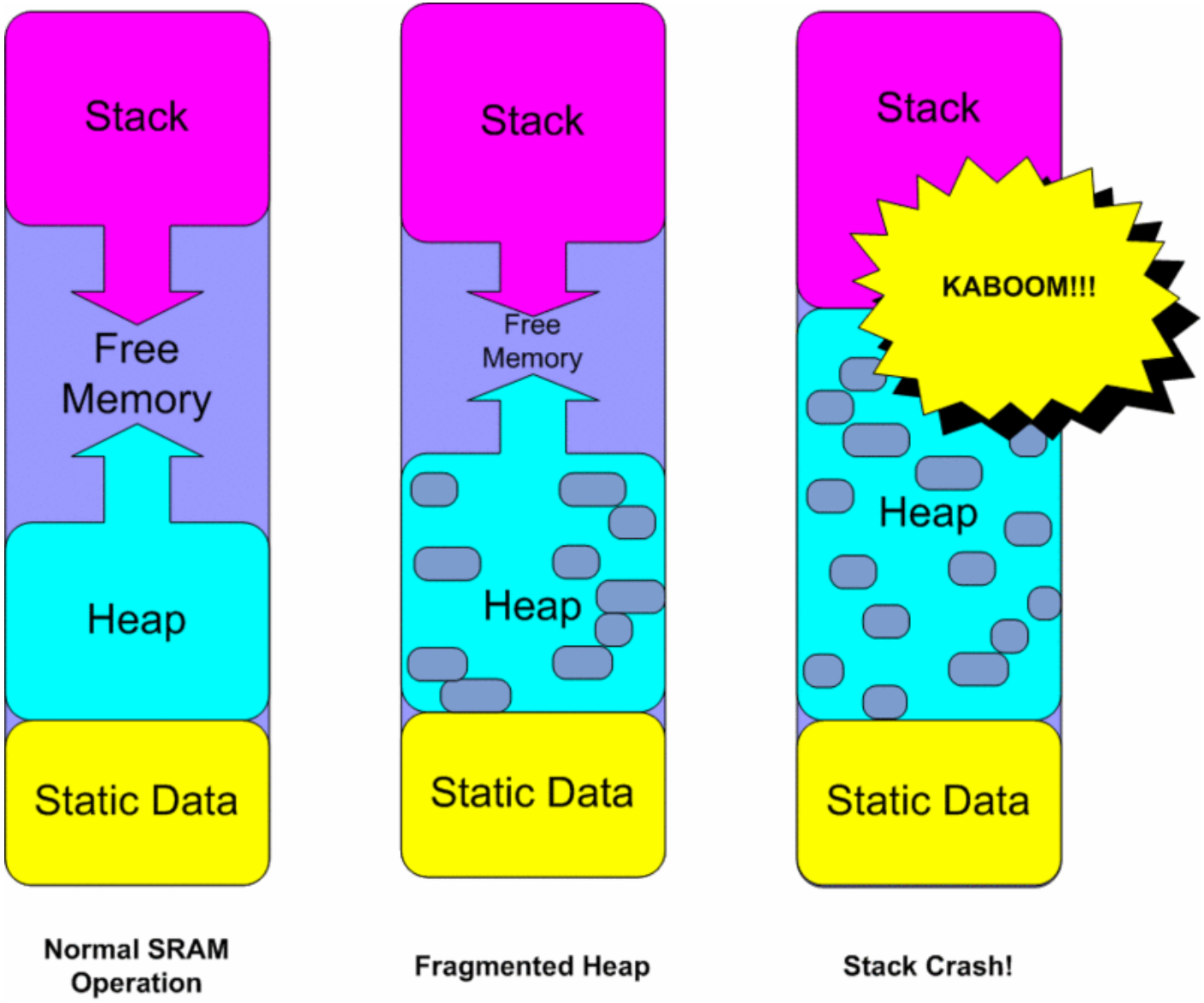
- When we assign the value of one reference variable to another, we copy the reference to the object.
- We do *not* copy the object itself.
- Example involving strings:

```
String s1 = "hello, world";
String s2 = s1;
```



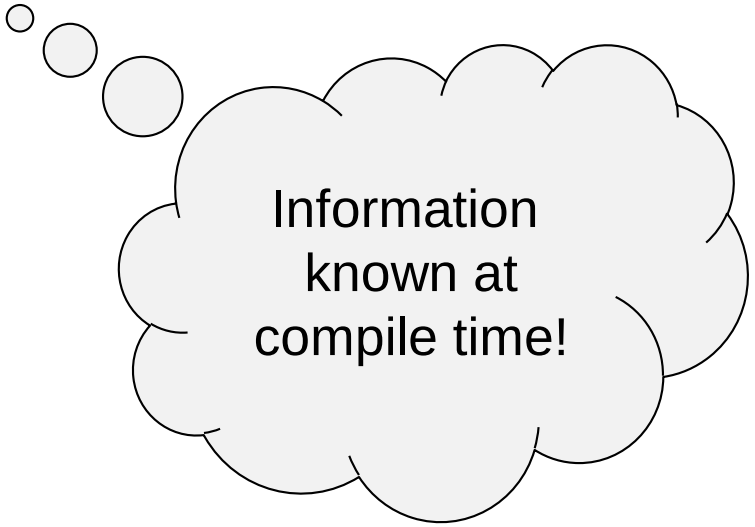
Where in memory do ***variables*** and ***objects*** reside?

Java Memory Model



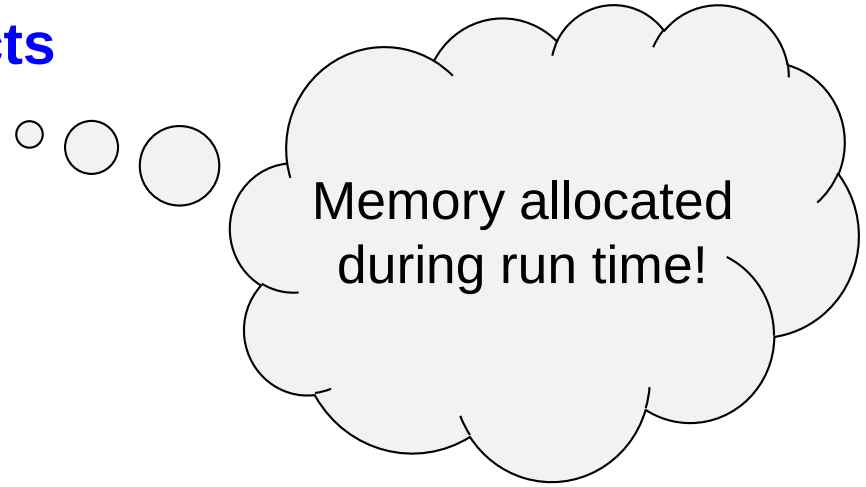
Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static **class variables**
  - Stack **local variables, parameters**
  - Heap objects



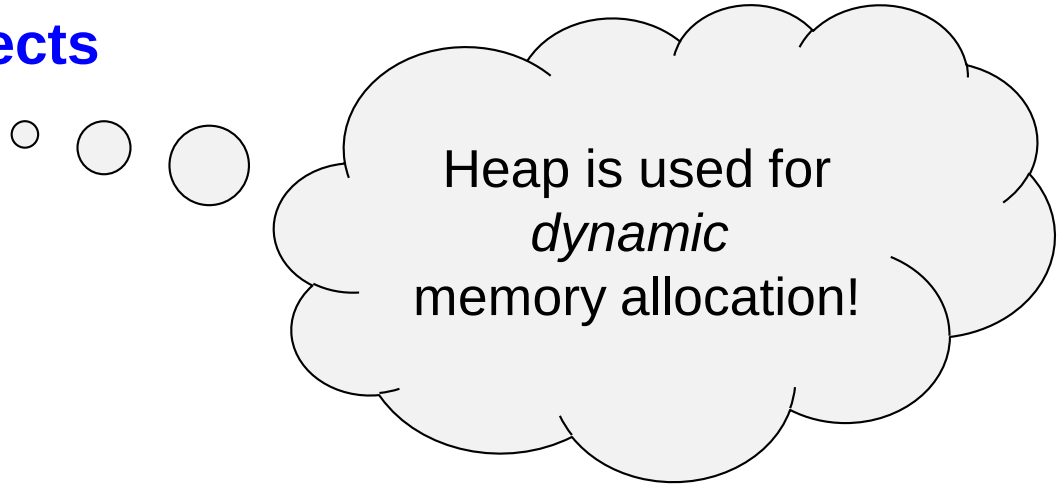
Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects**



Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects**

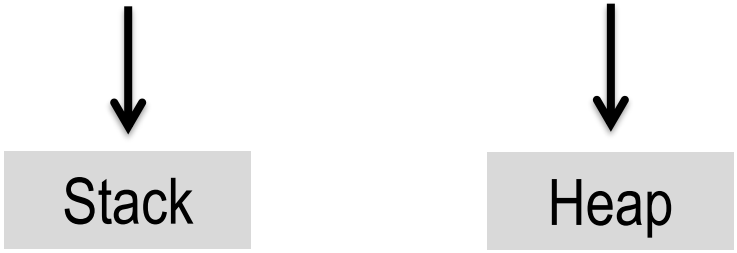


Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects …**

Example: creating a Scanner object

```
Scanner scan = new Scanner(System.in);
```



Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects …**

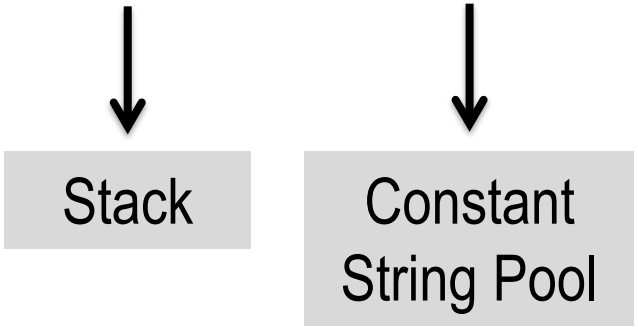
Example: creating a String object

```
String str = new String("A String");
```

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects …** … and The String Constant Pool for Java **literal** strings.

```
String str = "A String";
```



Testing for Equivalent *Strings:*
another look

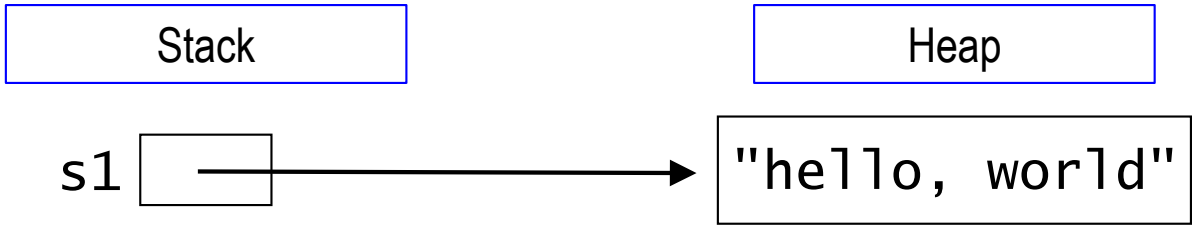
- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
```

Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

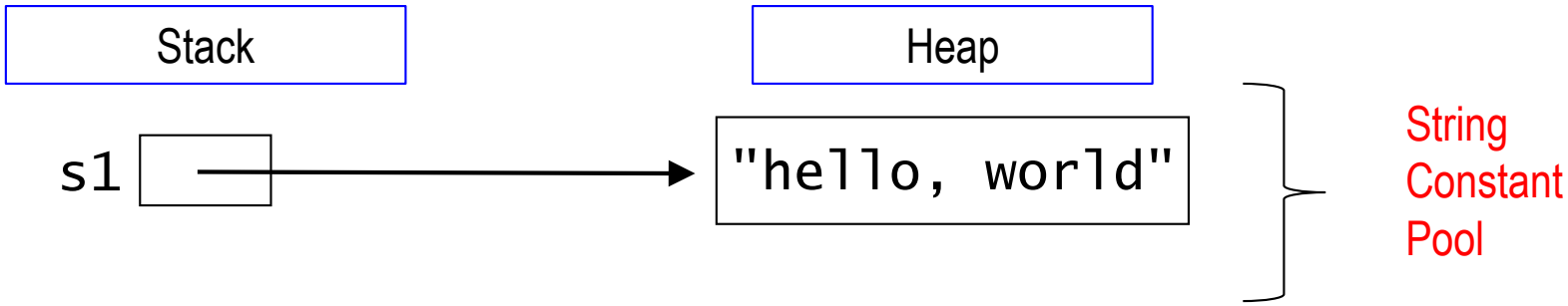
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

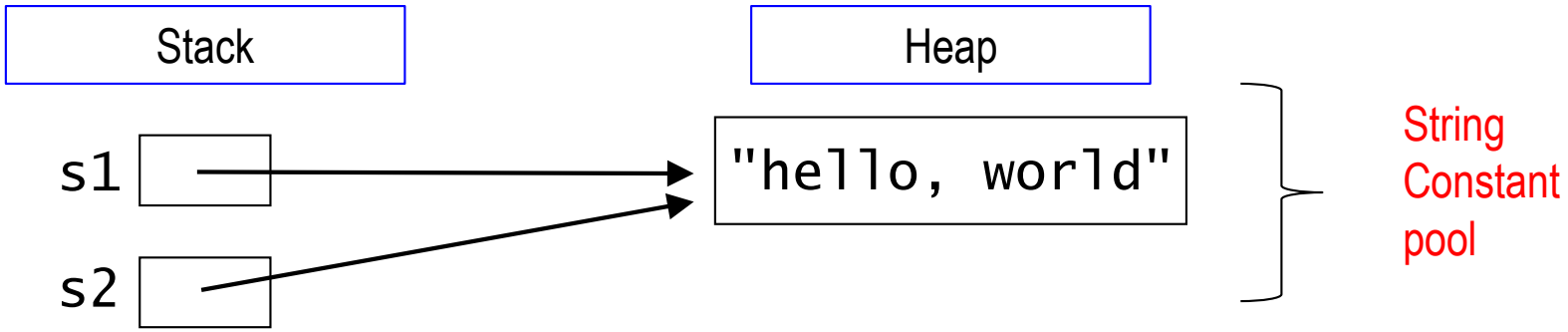
```
String s1 = "hello, world";   // String Constant Pool
String s2 = "hello, world";
String s3 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

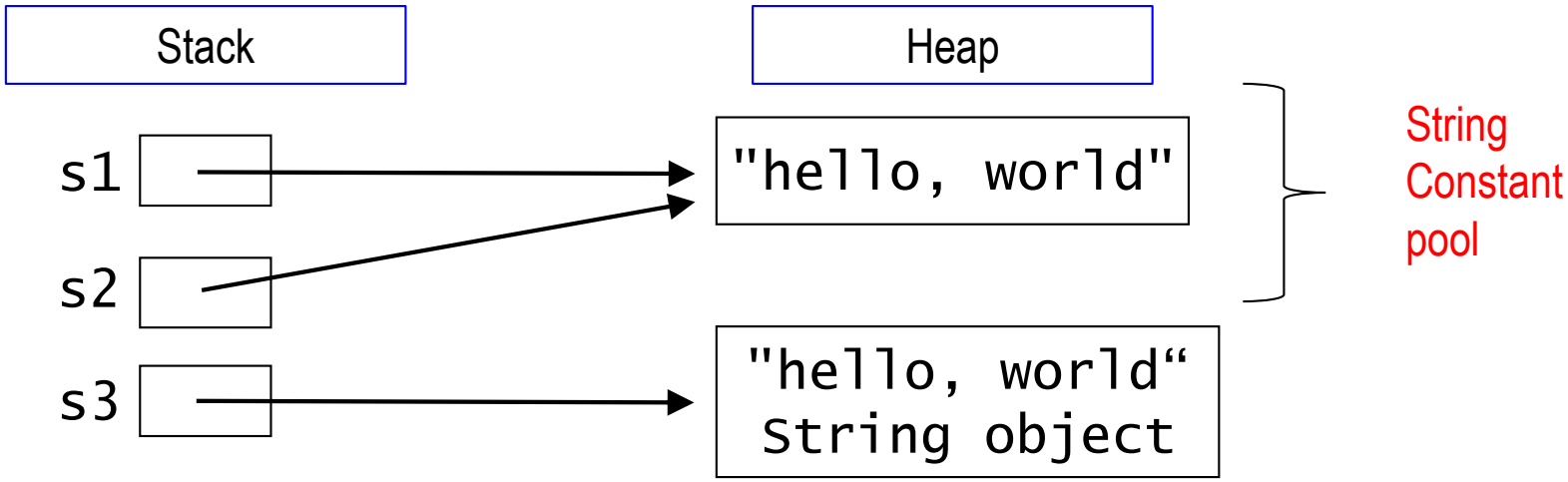
```
String s1 = "hello, world";
String s2 = "hello, world";   // String Constant Pool
String s3 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

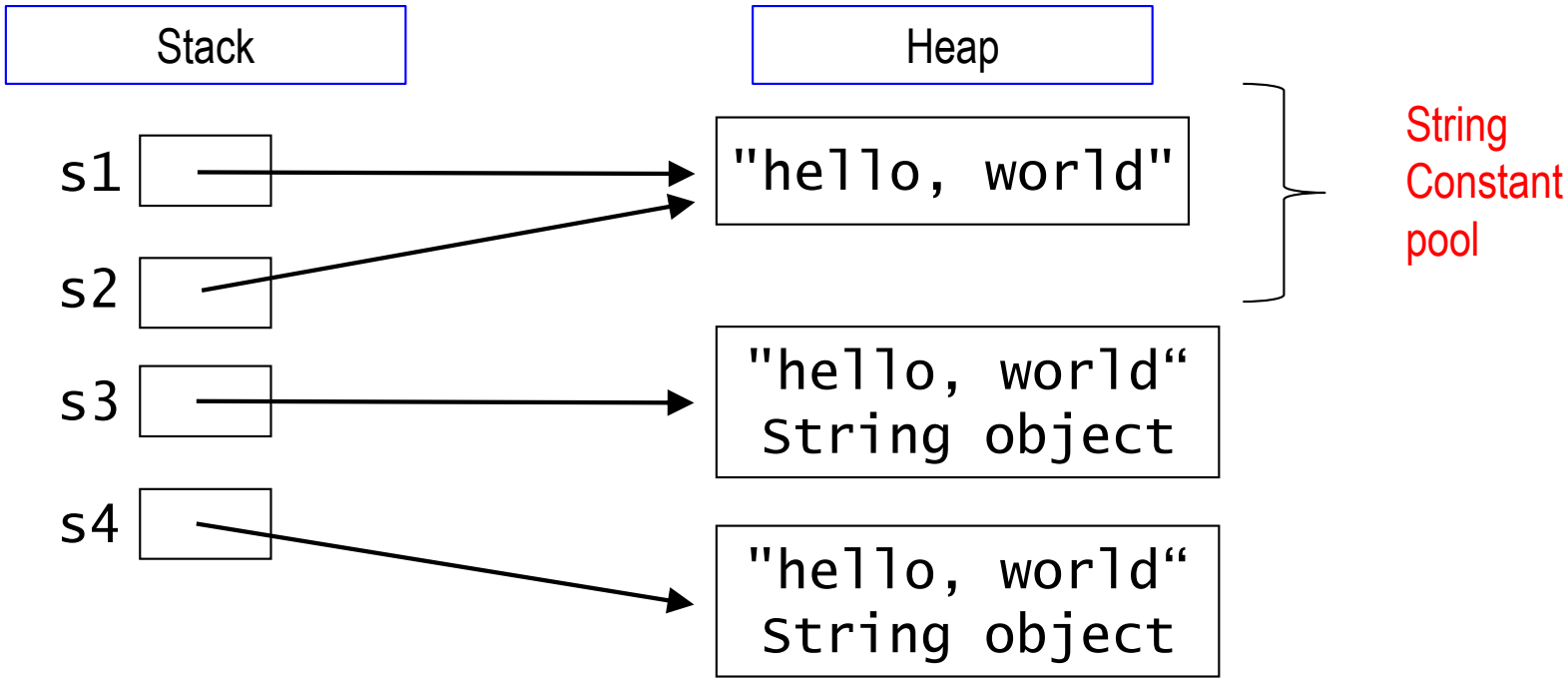
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");   // heap
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

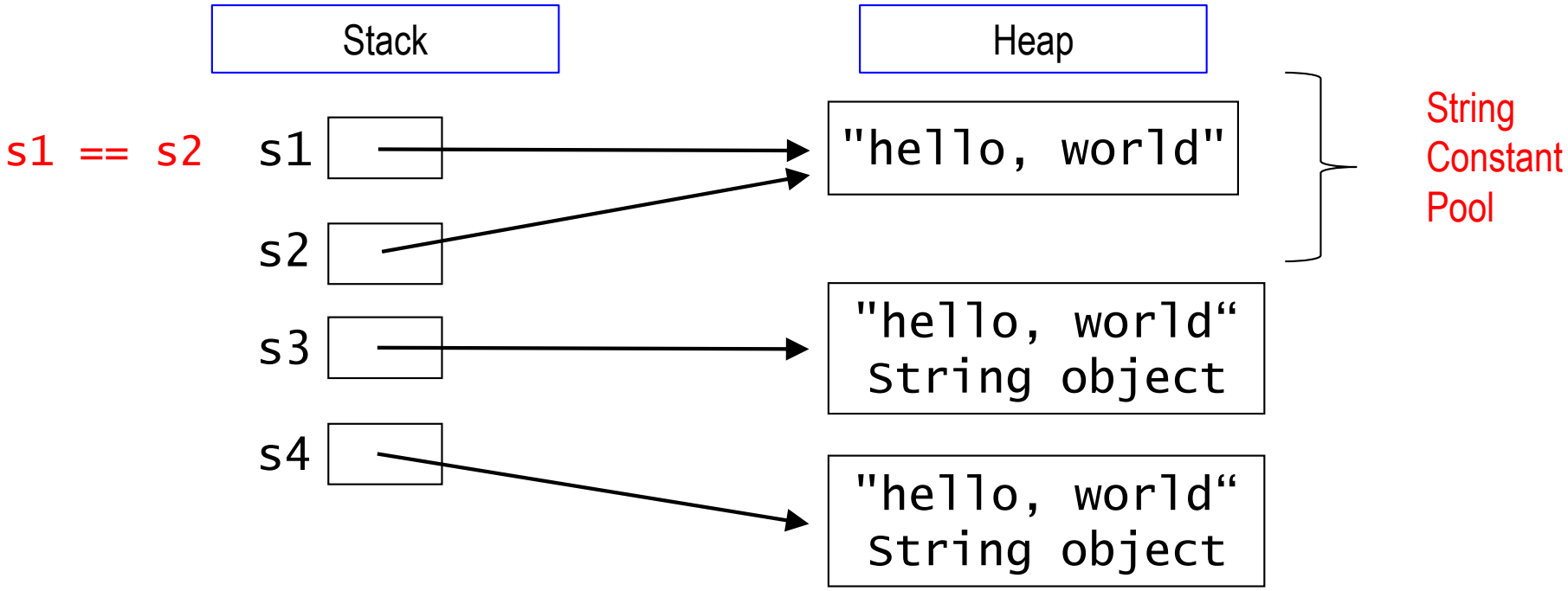
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");    // heap
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

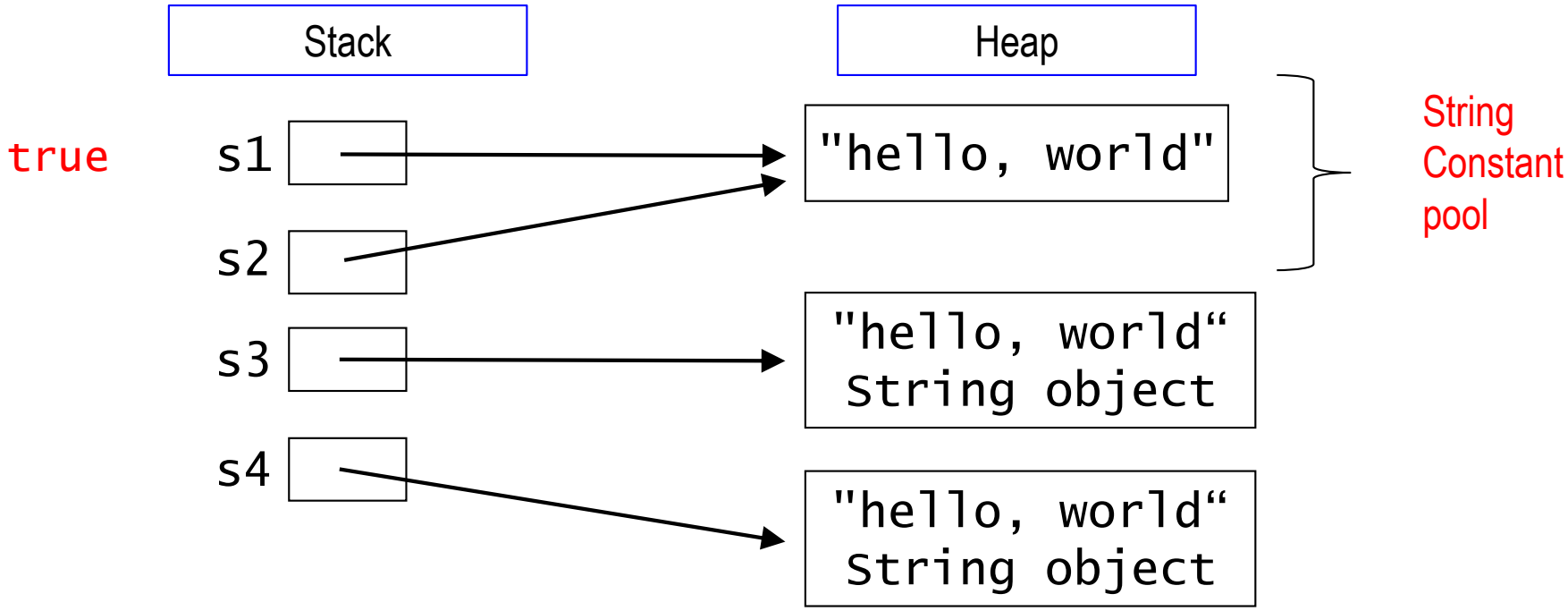
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

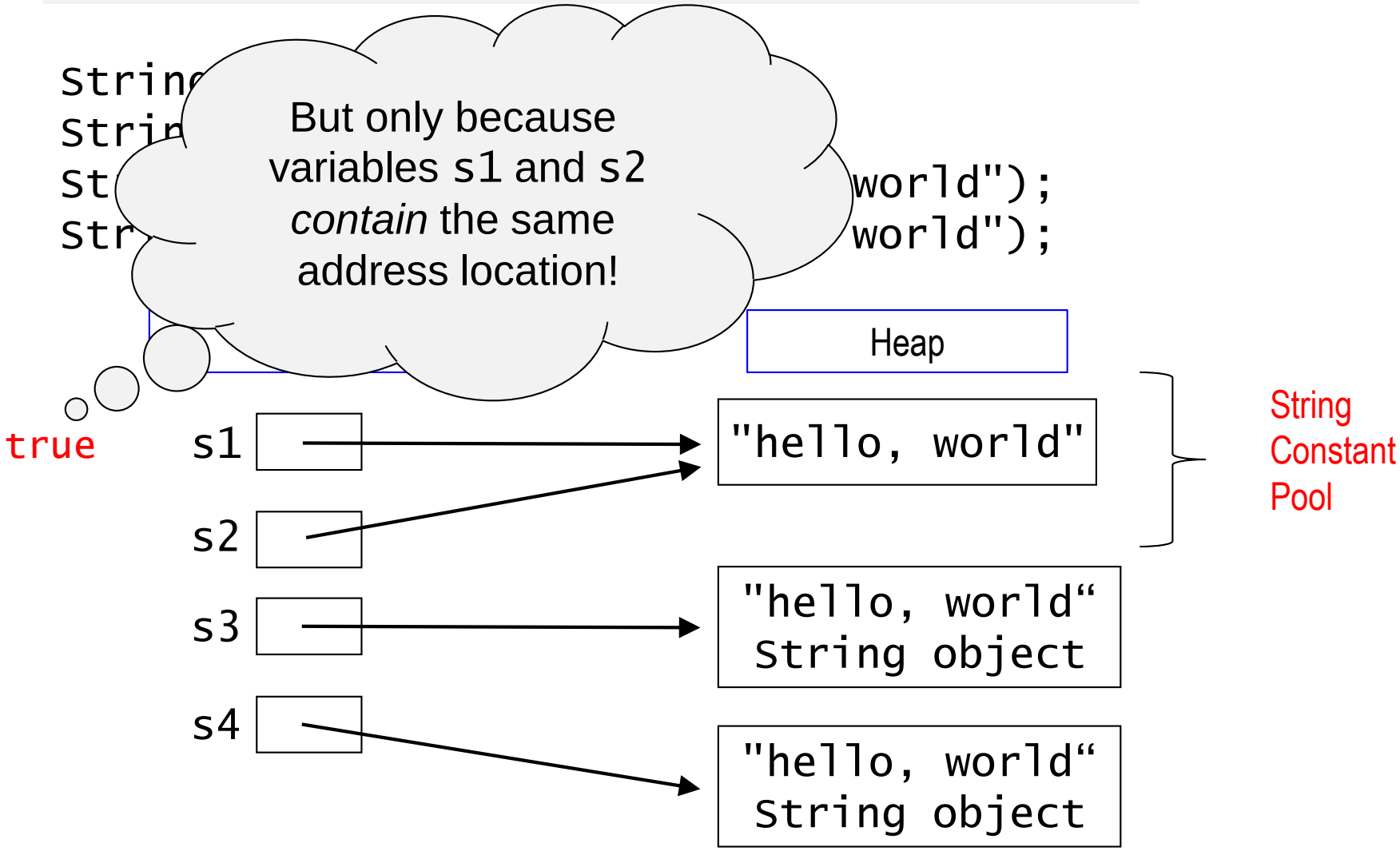
- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

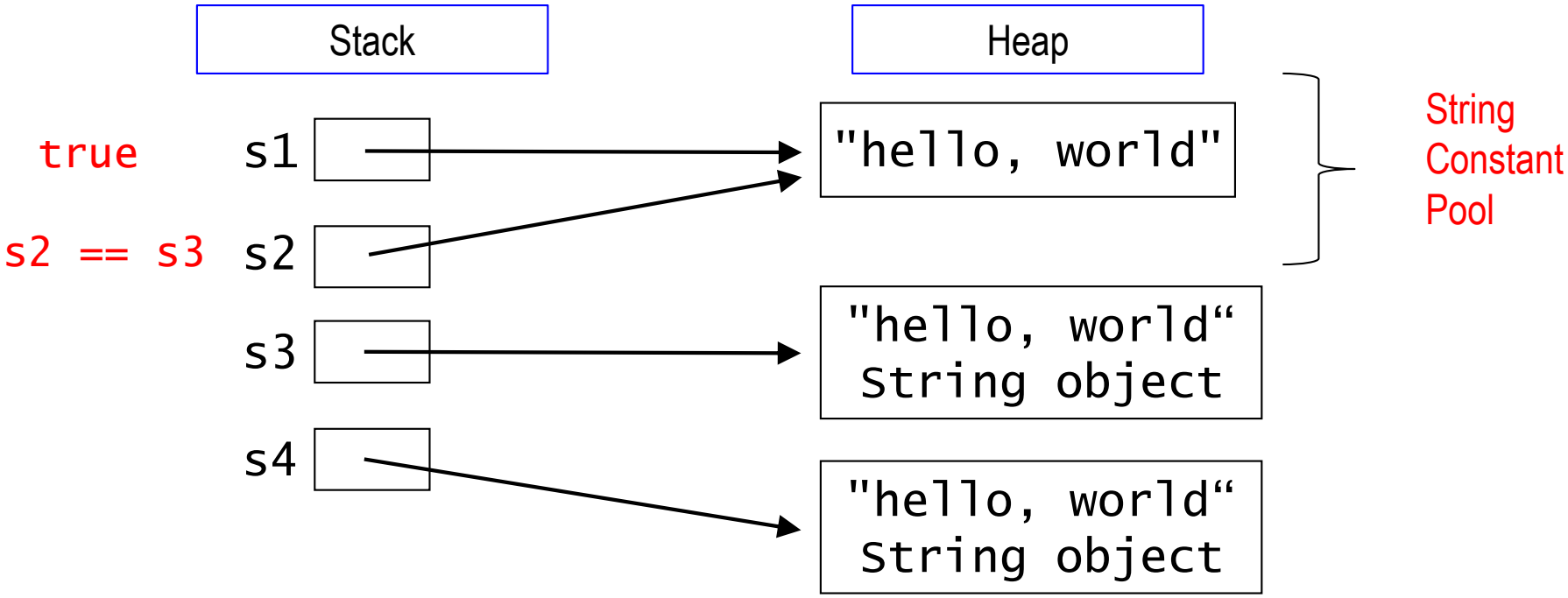
- The `==` and `!=` operators do *not* typically work when comparing *objects*



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

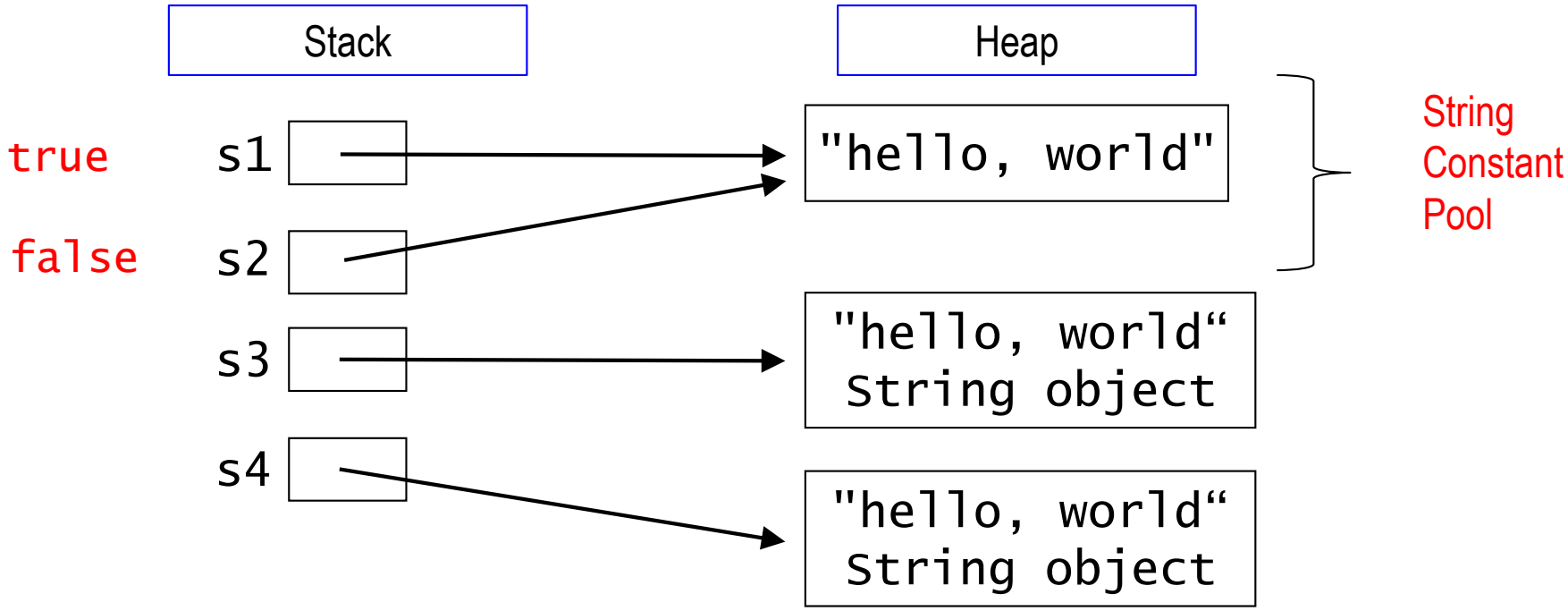
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

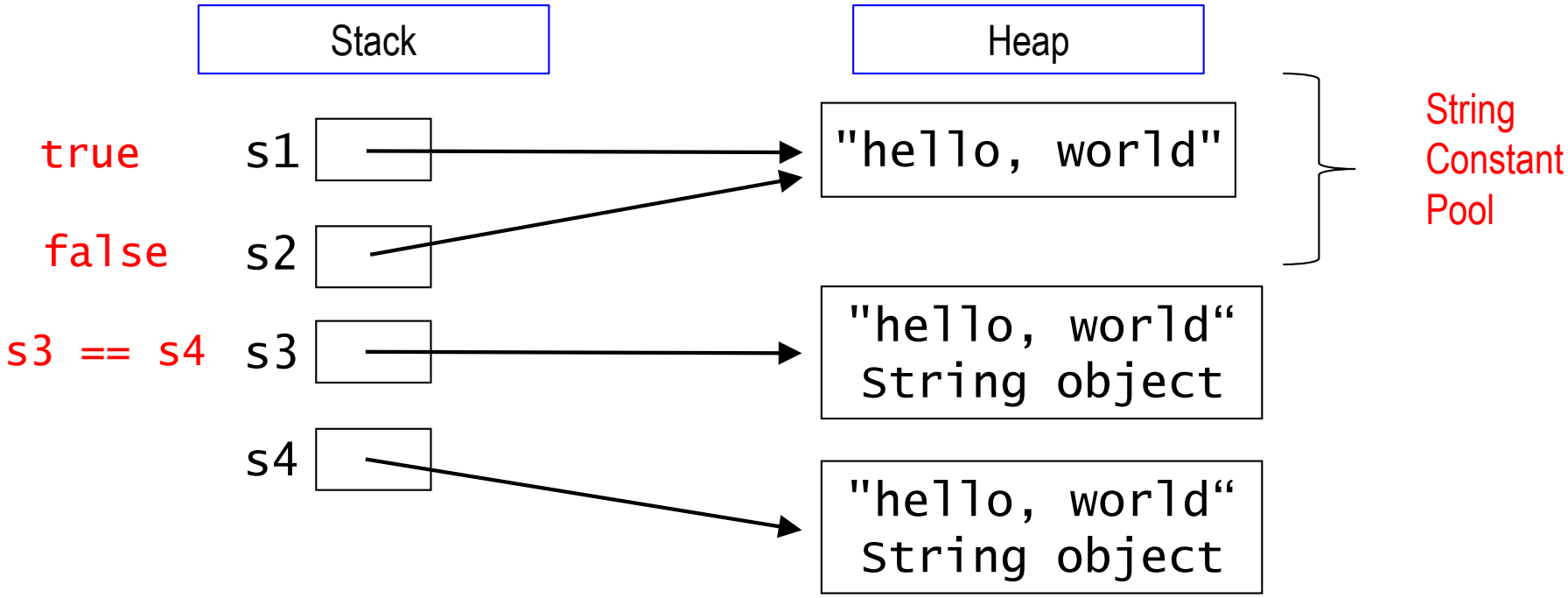
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

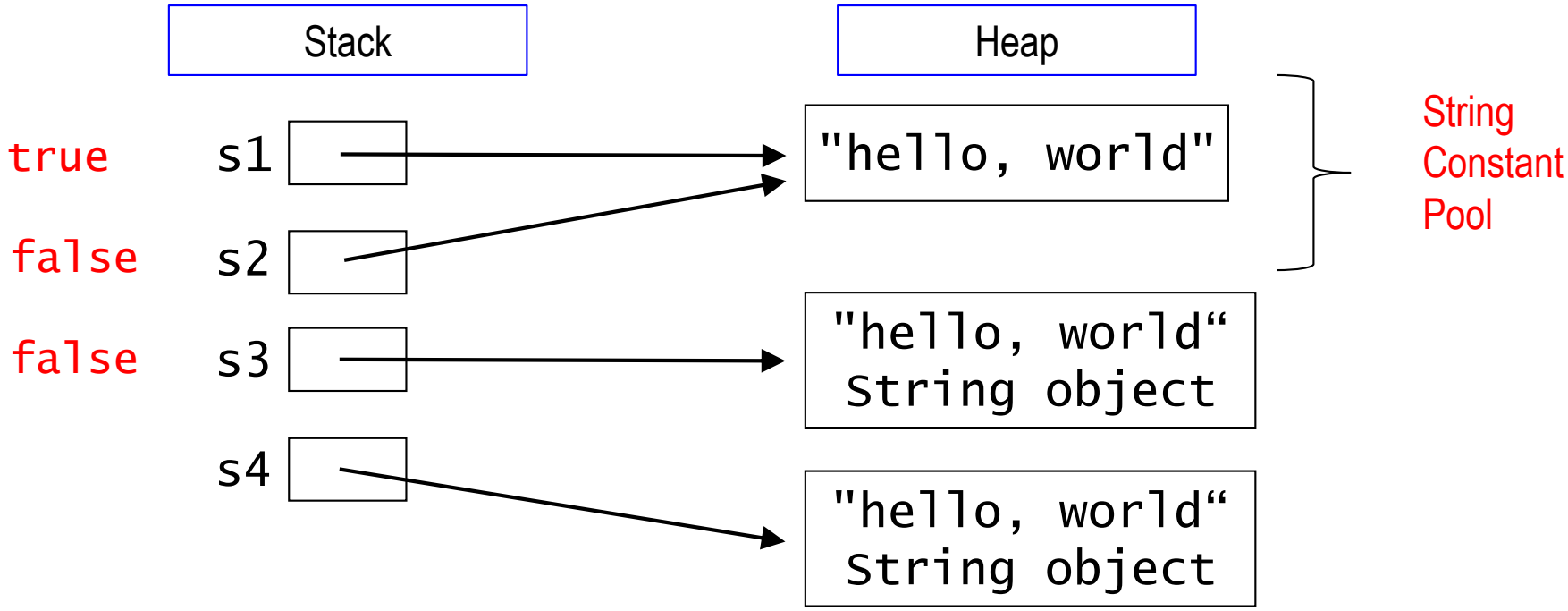
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The == and != operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
Str                        , world");
Str                        , world");
```

Heap

```
llo, world"
```

String Constant Pool

```
true

false

false
```

```
llo, world“
ring object
```

```
llo, world“
ring object
```