Object Oriented Design in Java

Object Oriented Programming:
an evolution of Structured Programming

*Couples the **logic** being performed with the data it is being performed on to create a physical entity that models a real life object.*

Object Oriented programming allows this coupling to be defined implicitly within the language!

In the world of OO the buildings blocks are no longer the code modules, but the physical entities or objects that are created!

Coding vs. Programming
Understanding Design Principles

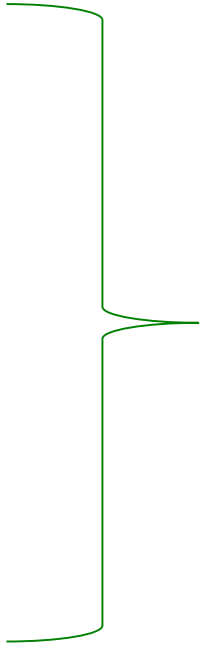
Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion/Injection



"Agile Software Development: Principles, Patterns, and Practices“ by **Robert Martin**

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - *A class should have only one reason to change.*

- Open Closed Principle
  - *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

- Liskov Substitution Principle
  - *Derived types must be completely substitutable for their base types.*

- Interface Segregation Principle
  - *Clients should not be forced to depend upon interfaces that they don't use.*

- Dependency Inversion/Injection
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

Understanding Design Principles

Three common characteristics of a BAD design:

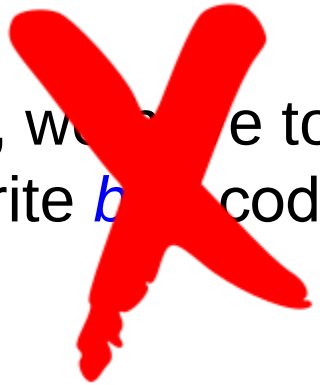
Rigidity - It is hard to change because every change affects too many other parts of the system.

Fragility - When you make a change, unexpected parts of the system break.

Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

Robert Martin

Abstraction is the Key



In order to go fast, we have to accept that we will write *bad* code…

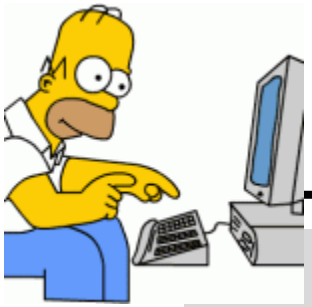
Abstraction does not mean you have to solve every specific problem!

Abstraction allows you to build an architecture that will not force you to start from scratch every time you need to *pivot* or discover a new requirement.

More pragmatic and economical to build flexible architecture.

Java

Compiled vs. Interpreted



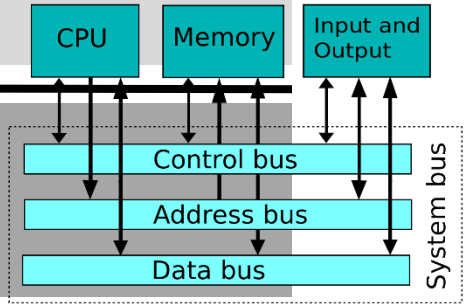
Programmer

High Level Languages

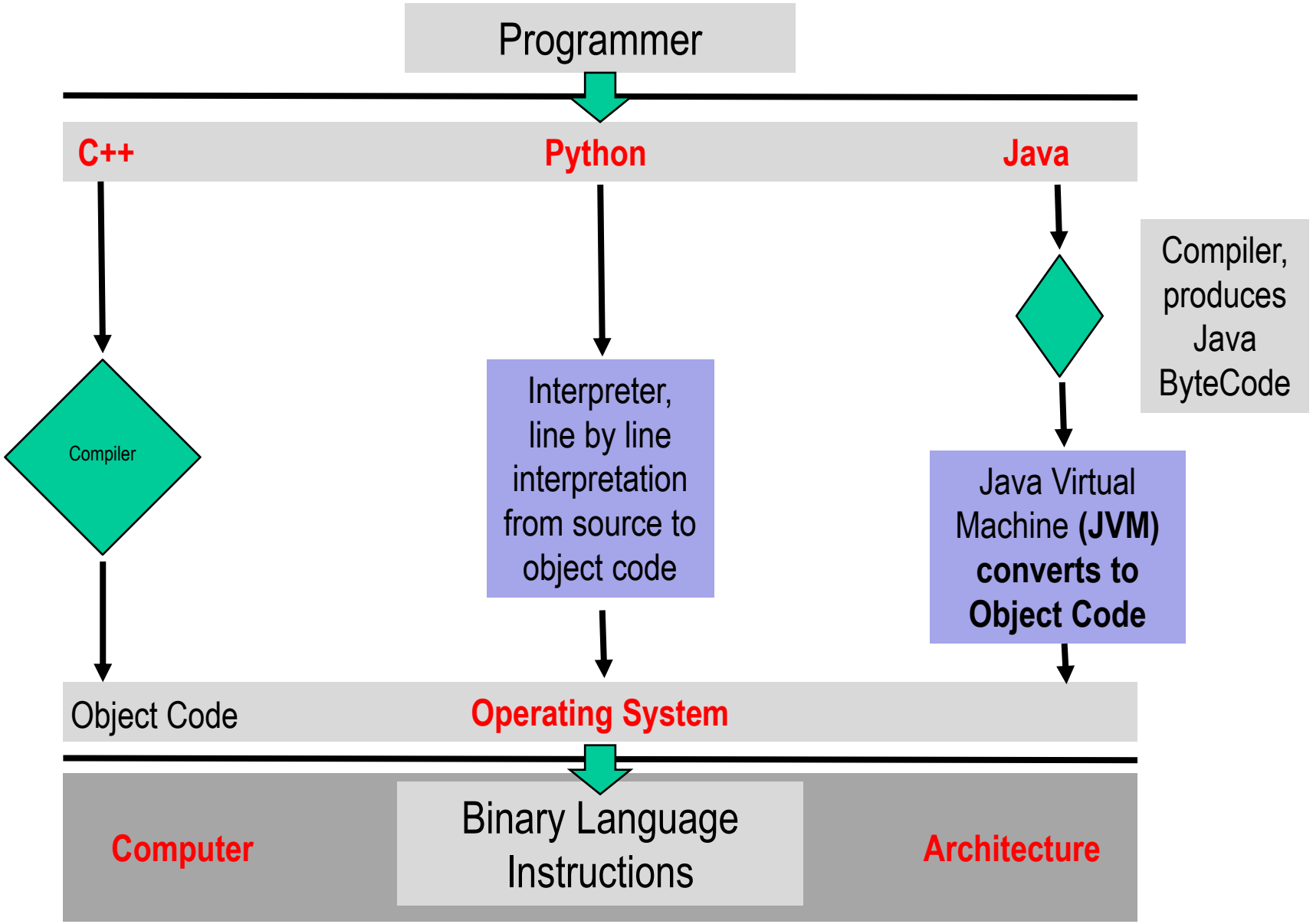
Operating System

Computer Architecture

Binary Language

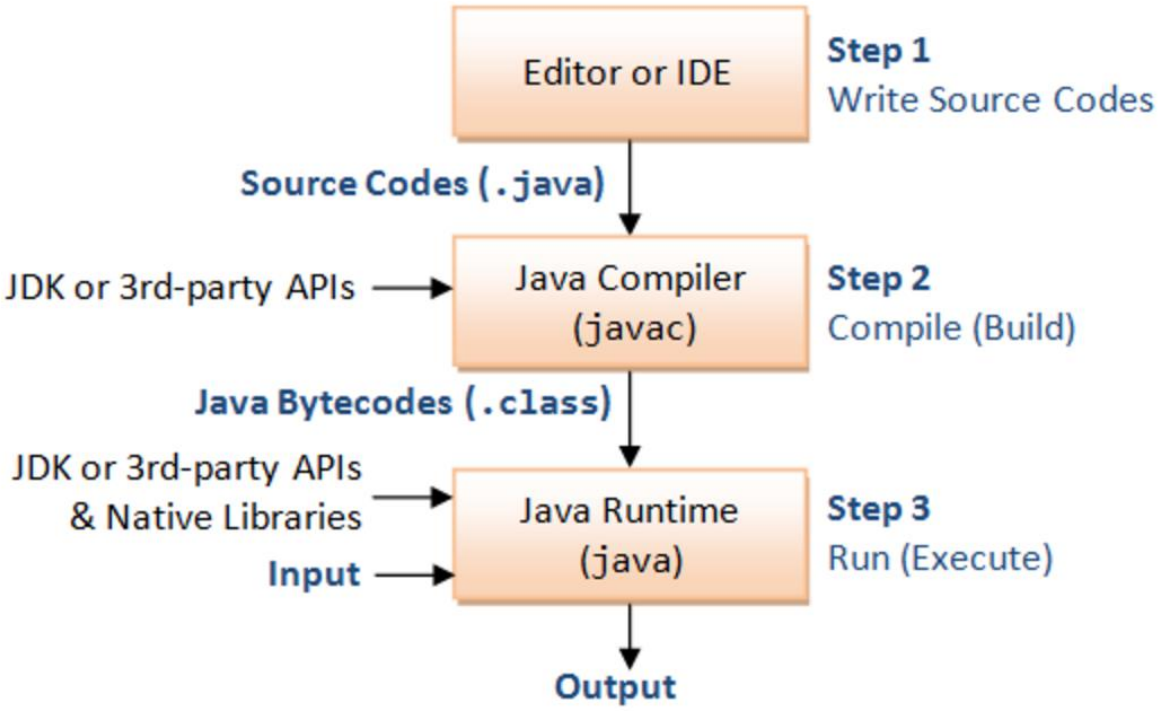


Compiled vs. Interpreted



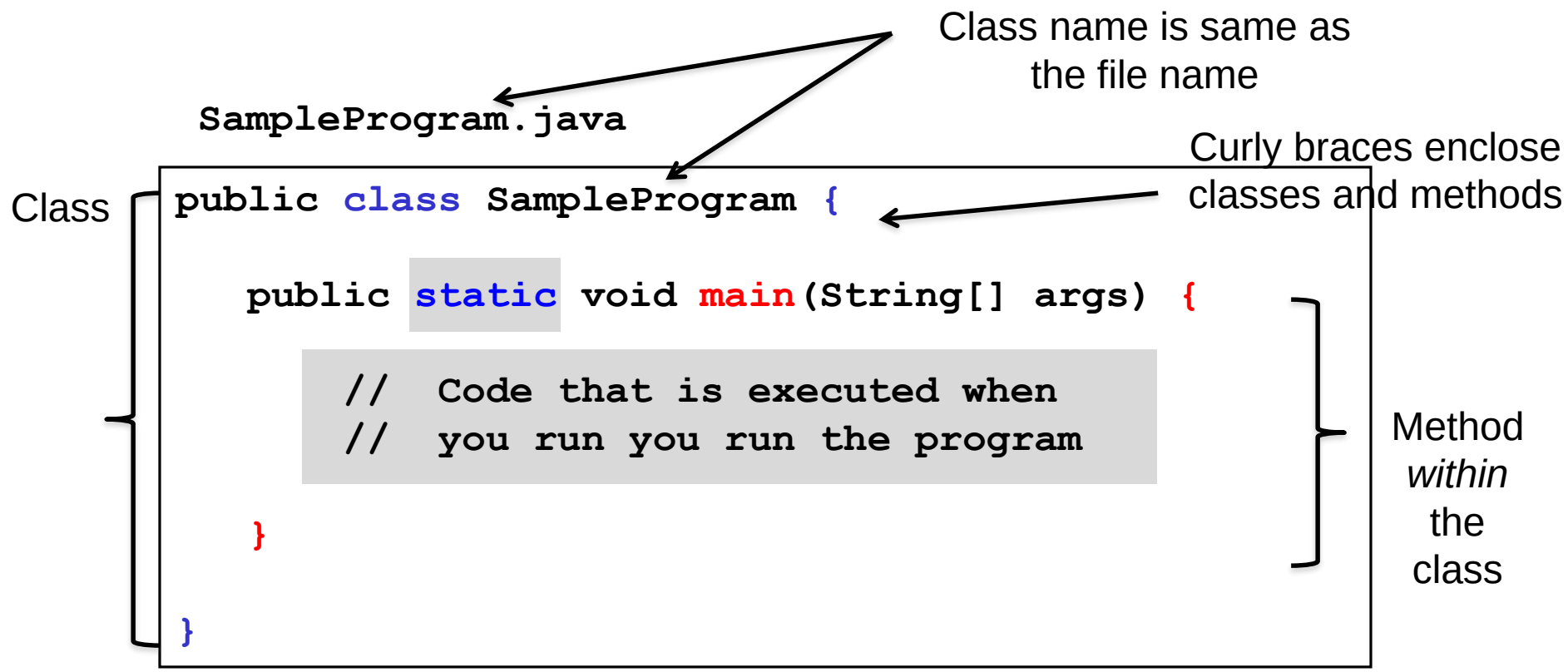
Compilation vs Interpretation

- Java is an example of a language which is compiled; before executing any code, your program must be transformed into a lower-level code called byte-code. The byte-code is then passed to the Java Virtual Machine (JVM), which runs the program and produces output:



Java Basic Program Structure

- Java programs are organized as classes stored in files with the ".java" extension, and with code written inside methods delimited by curly braces; each program must have a method called main, which contains the code that will be executed when you run your program:



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);
              .
              .
              .
              .
   }
}
```

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);
                .
                .
                .
                .
   }
}
```

Simple Java Program

```
import java.util.*;

public class Play {

    public static void main( S

        Scanner scan = new

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

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

In Java operators are *overloaded* such that the operation performed depends on the *datatype* of the operands.

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);
              .
              .
              .
              .
   }
}
```

Java Data Types

- `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!";
```

Java Data Types

- `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: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

-

-

-

```
boolean isPrime = false;
```

Why is this important?

- `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;
```

-

-

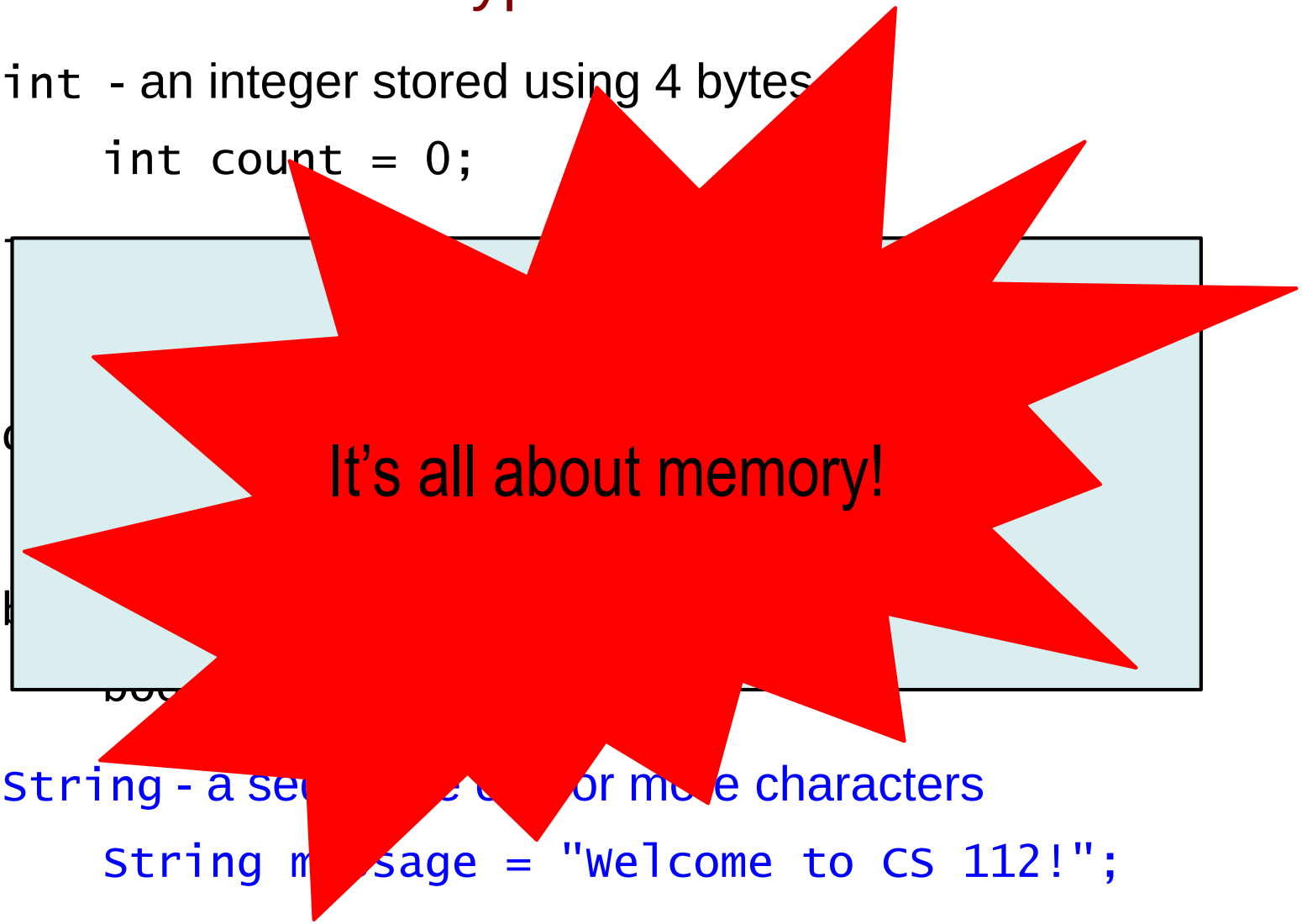
-

- `String` - a se[illegible] or more characters

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

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

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

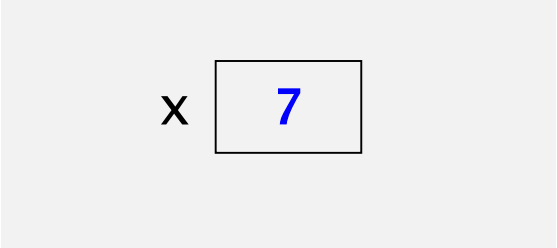


Primitive Types

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

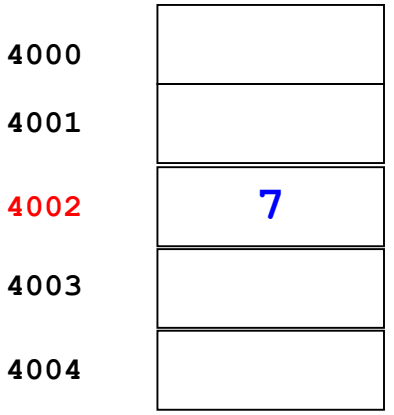
```
int x = 7;
```

A visual simplification



  - **there is no reference!**

- Java *primitive types are:*
  - `int`
  - `long`
  - `double`
  - `boolean`
  - a few others

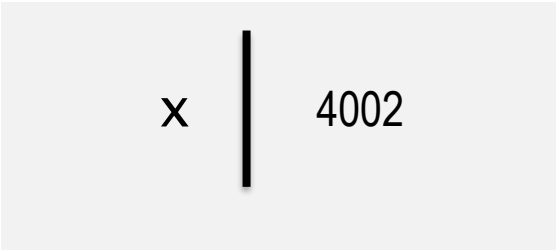


Primitive Types

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

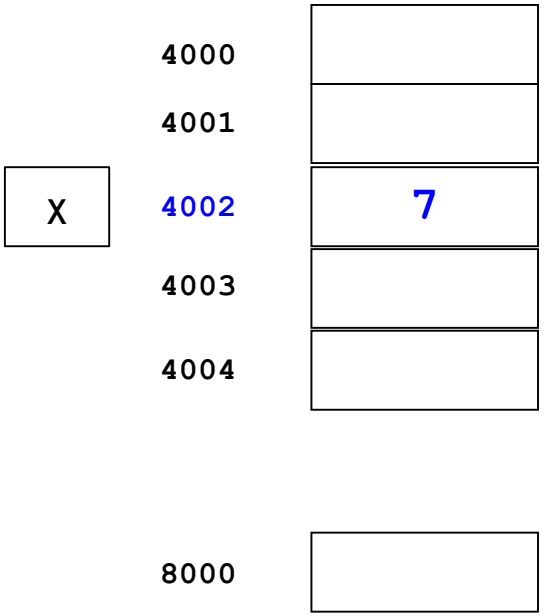
```
int x = 7;
```

memory map



  - **there is no reference!**

- Java *primitive types are:*
  - `int`
  - `long`
  - `double`
  - `boolean`
  - a few others



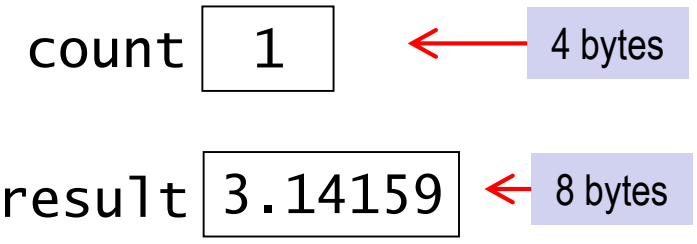
Variable Declarations and Data Types

- 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

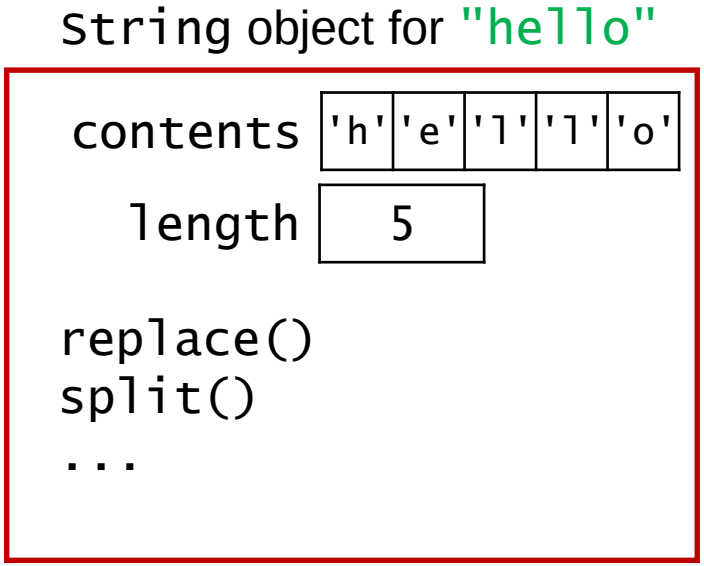
- Declaring a variable tells the compiler how much memory *(i.e. how many bytes*) to allocate **and** the *type* of the data!

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



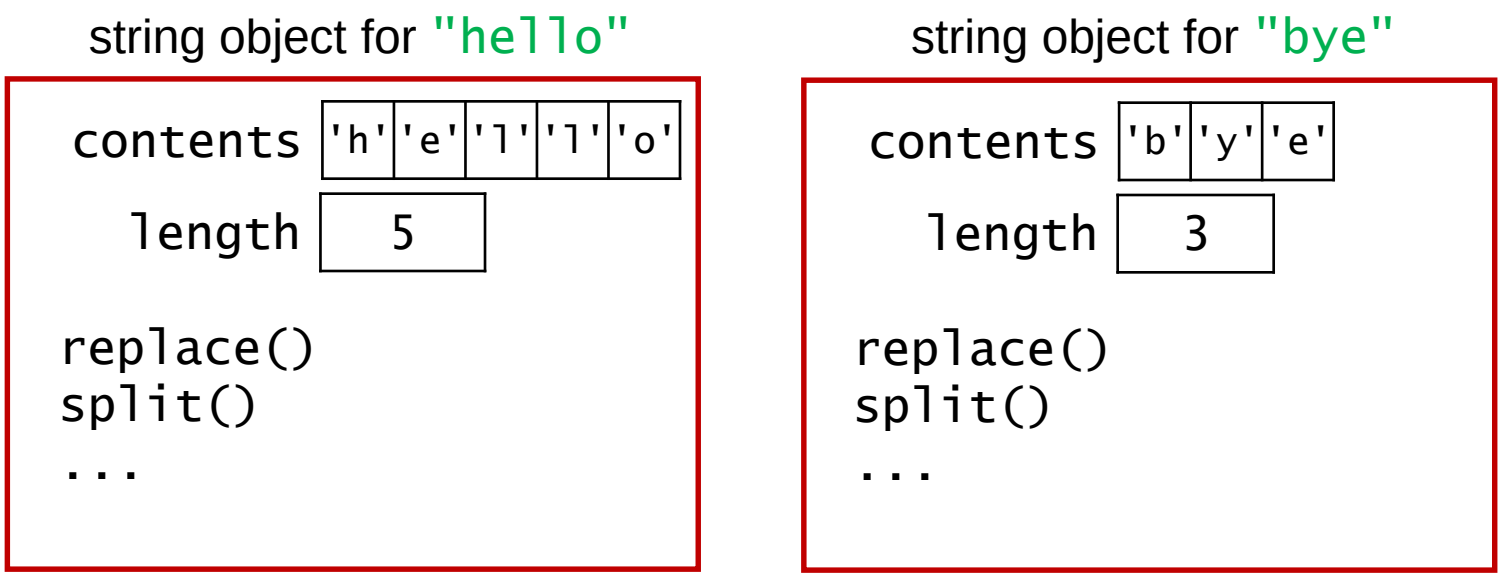
Object vs. Primitive:
summary

- An object is a 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.



Strings Are Objects

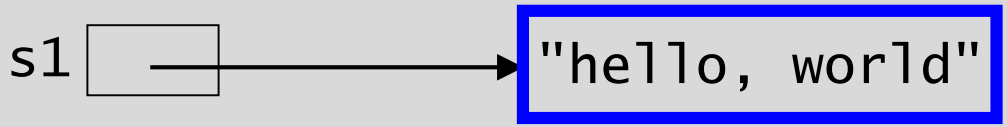
- A string is an object.
  - ***attributes/fields:***
    - the characters in the string
    - the length of the string
  - ***methods:*** functions inside the string that we can use to operate on the string



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

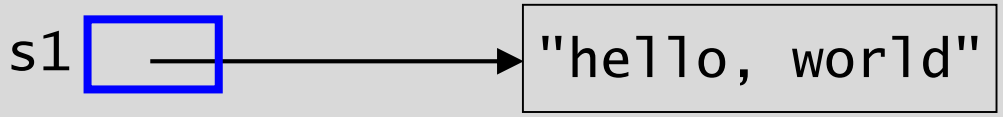
- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- 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

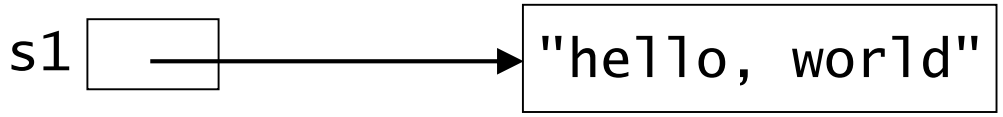
- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- 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;
```

s1 1024

s2 1024

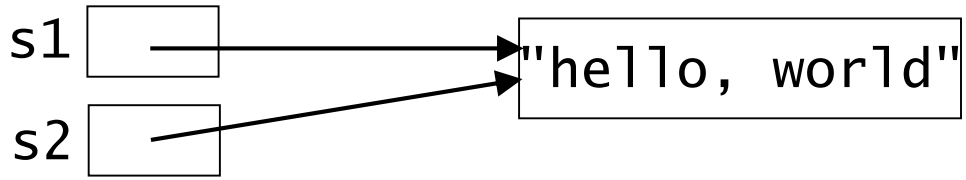
memory location: 1024

"hello, world"

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;
```

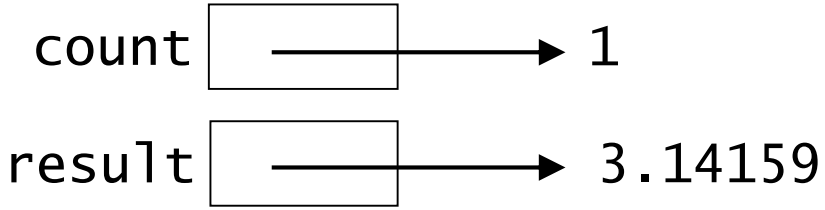


What About Python?

- In Python, *everything* is an object.
  - thus, all variables hold references (memory addresses)

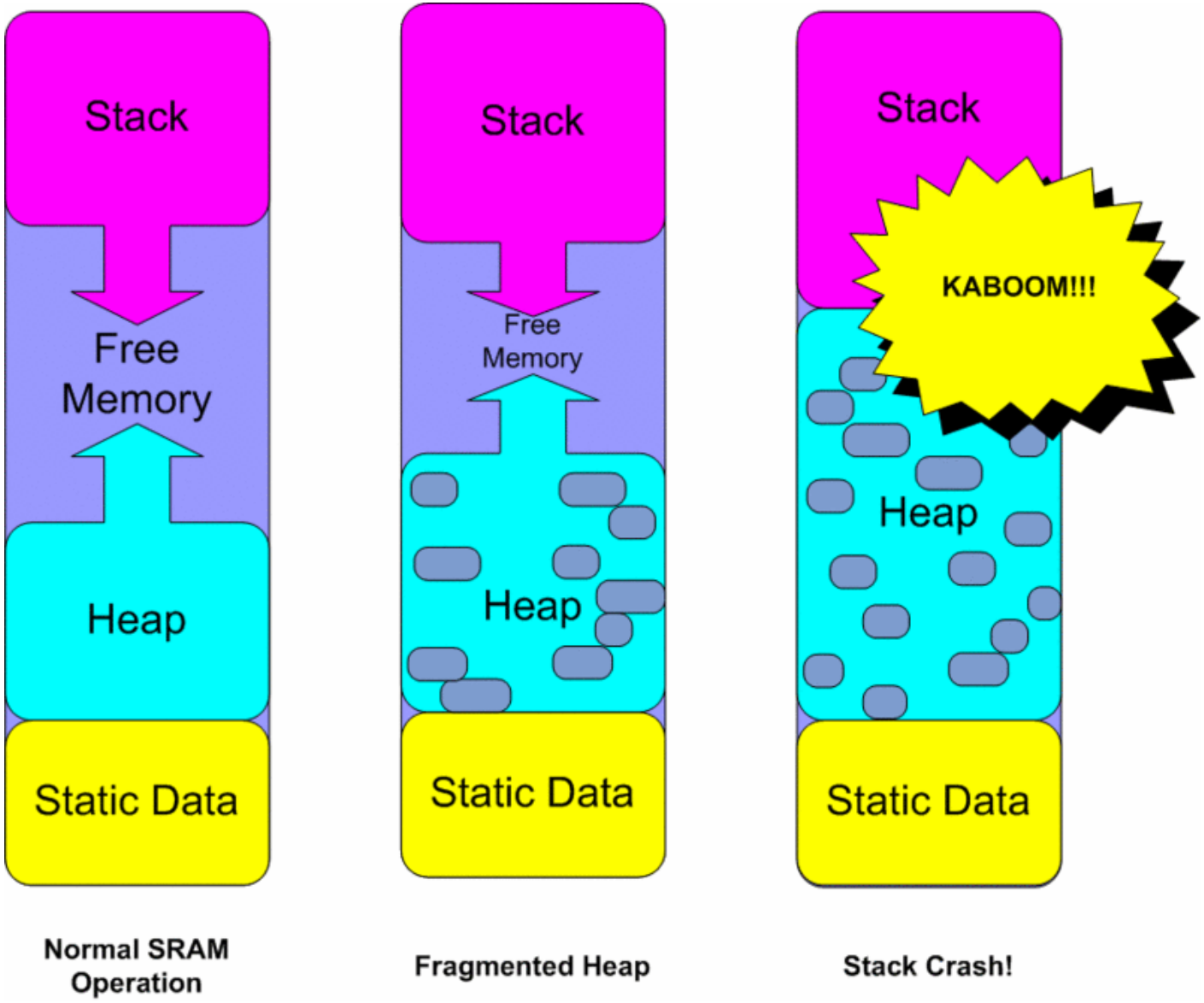
- As a result, Python can make every variable the same size.
  - and thus we don't need to declare the variable!

```
count = 1
result = 3.14159
```



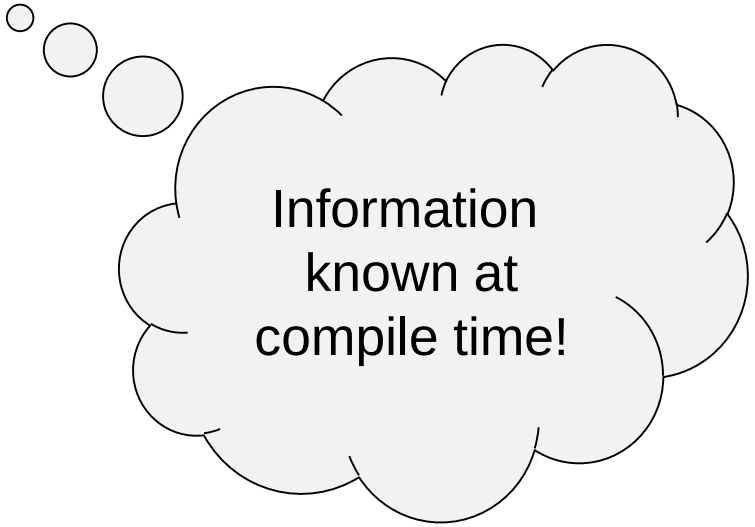
both variables are the same size, because they both store a memory address!

Java Memory Model



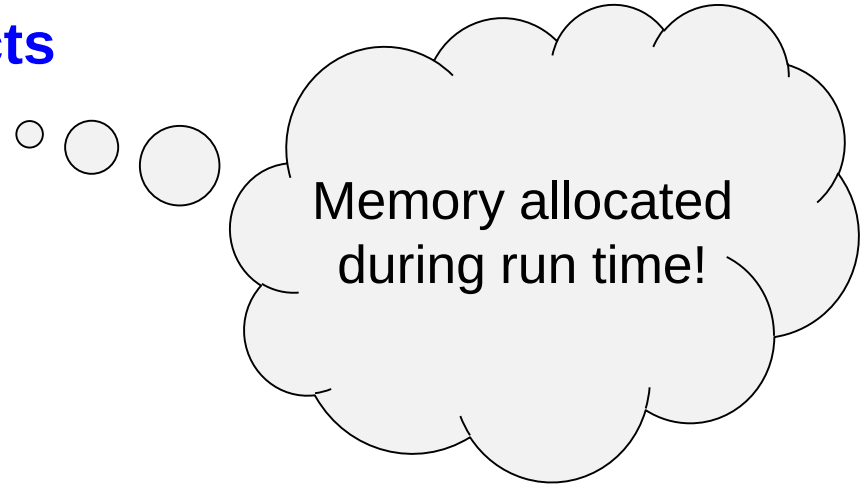
Memory Management: Looking Under the Hood

- 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



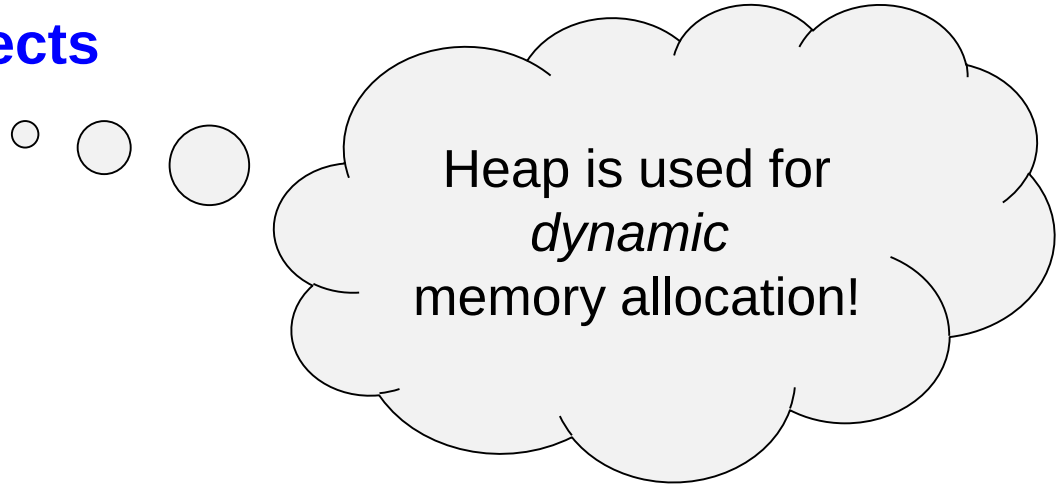
Memory Management: Looking Under the Hood

- 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 …**

Constant string pool for Java literal strings

```
String s = "some string";

vs.

String s =
    new String("some string“);
```

Memory Management, Type I: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

```
public static final PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.

- The Java runtime system allocates memory for *class variables* when the class is first encountered.
  - this memory stays fixed for the duration of the program

Memory Management, Type II: Stack Storage

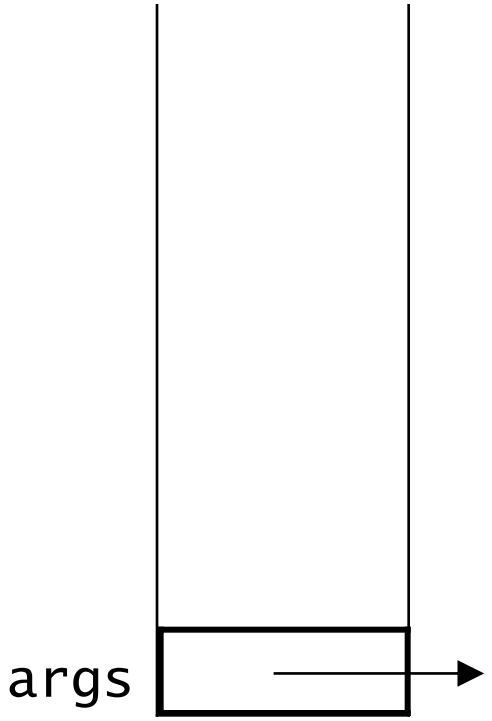
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

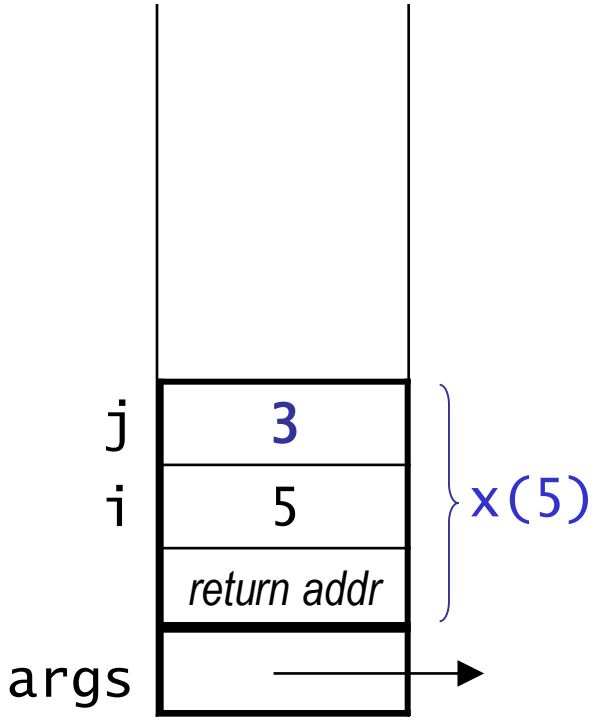
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

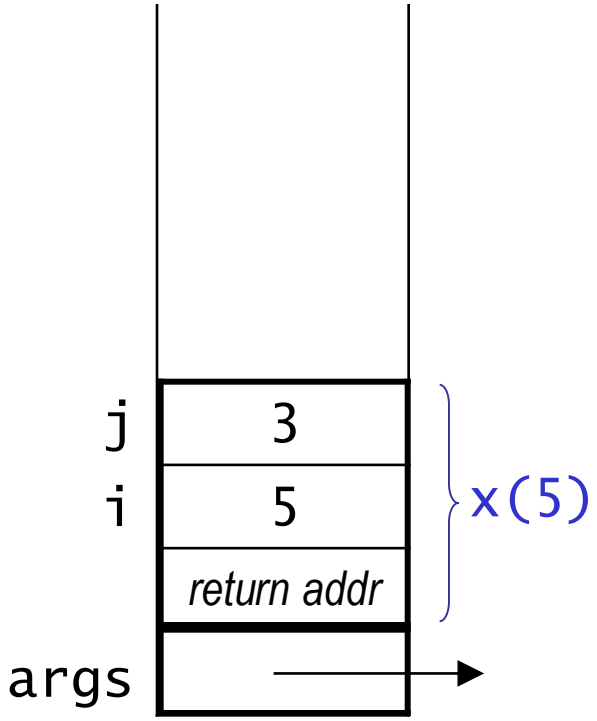
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

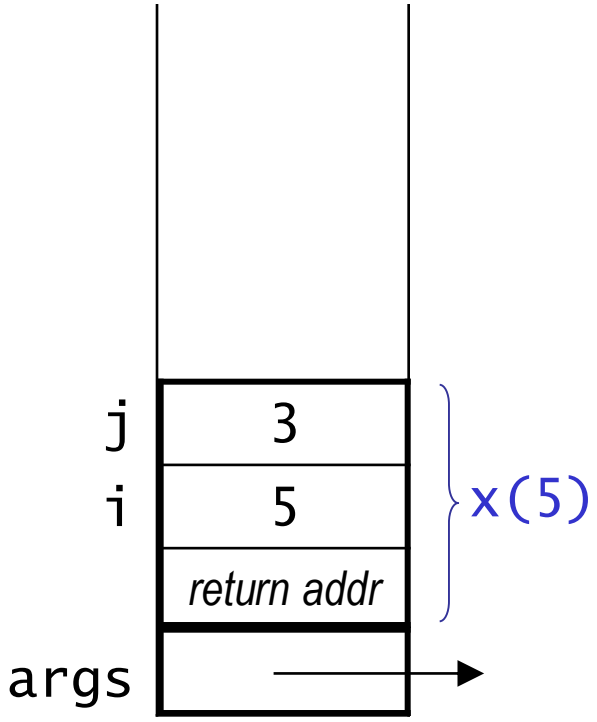
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

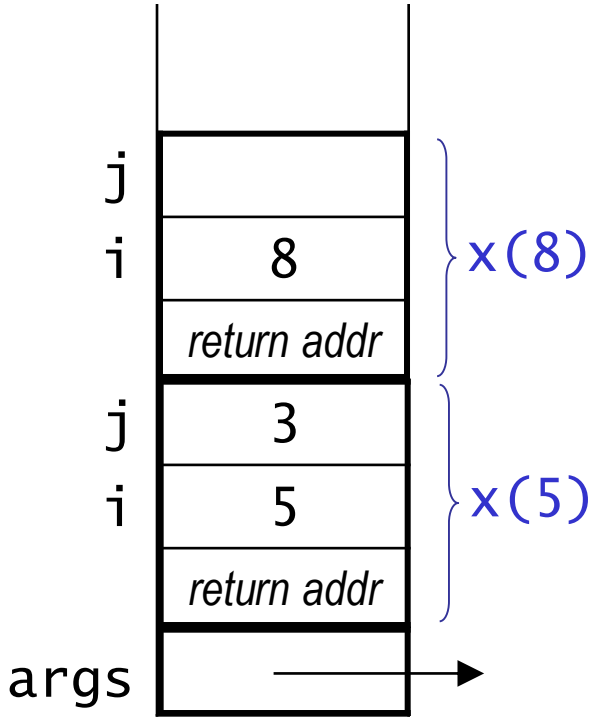
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

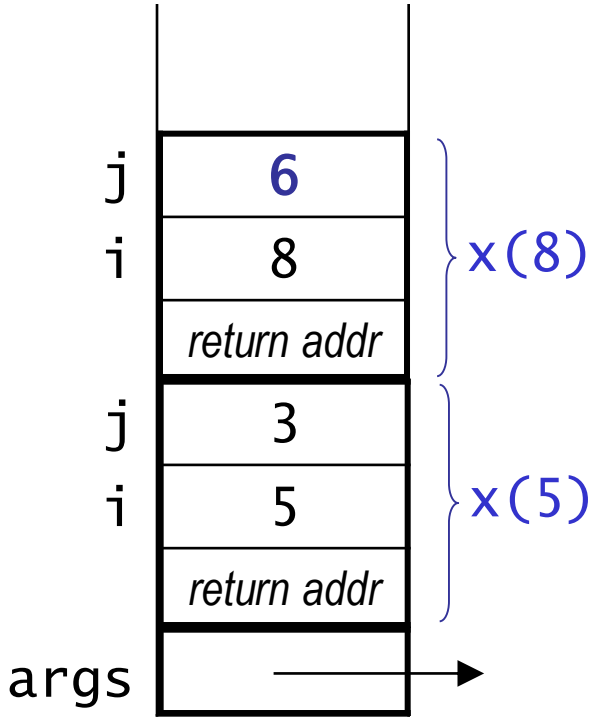
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

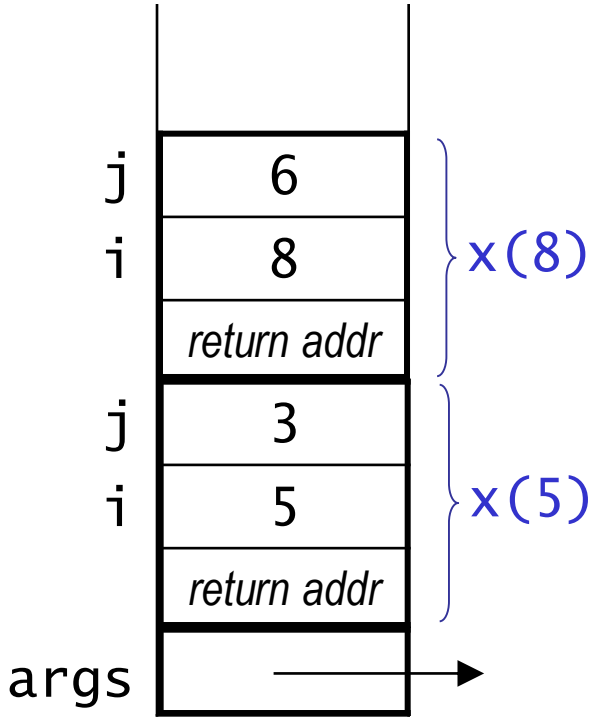
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

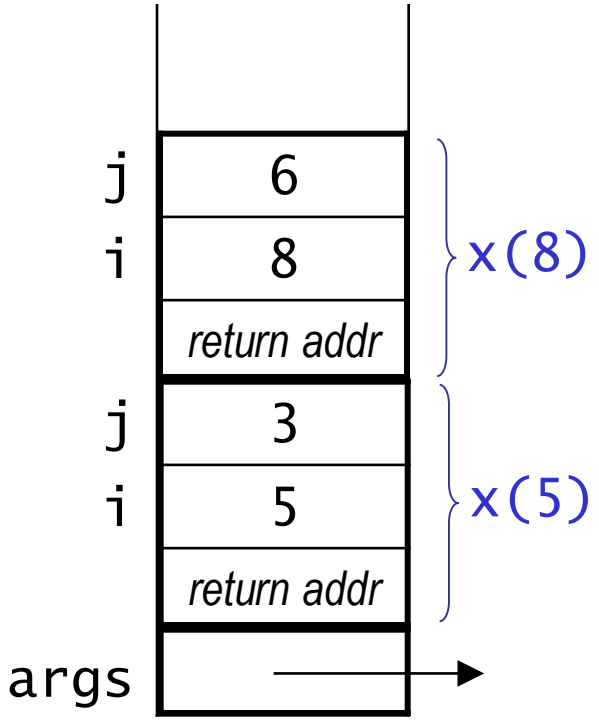
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

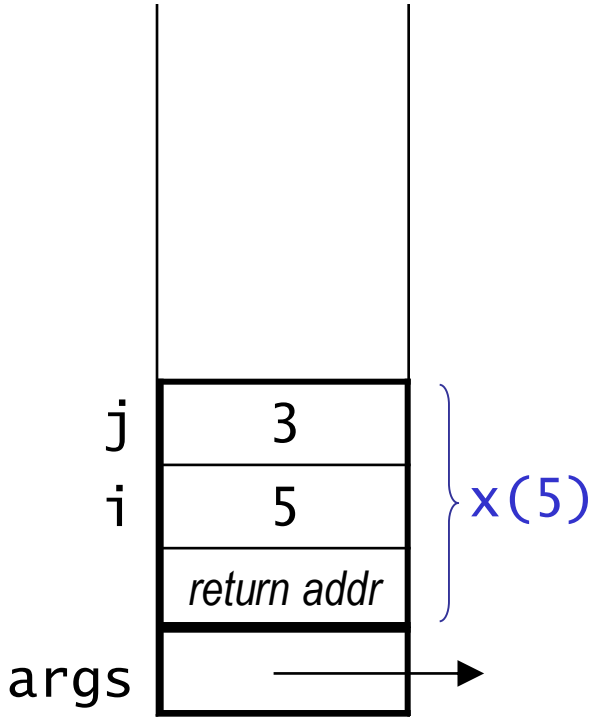
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

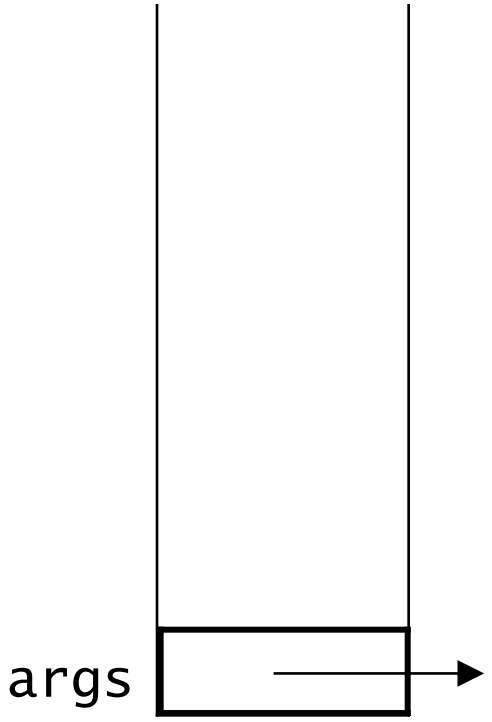
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

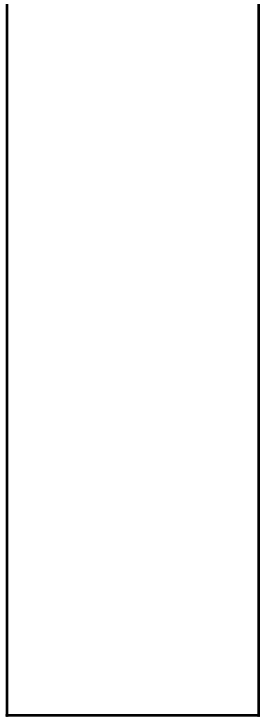
- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type III: Heap Storage

- Objects are stored in a memory region known as *the heap*.
- Memory on the heap is allocated using the `new` operator:

```
Scanner inp = new Scanner();
```

- `new` returns the memory address of the start of the object on the heap.
  - a reference!
- An object persists until there are no remaining references to it.
- Unused objects are automatically reclaimed by a process known as *garbage collection*.
  - makes their memory available for other objects

Memory Management, Type III: Heap Storage

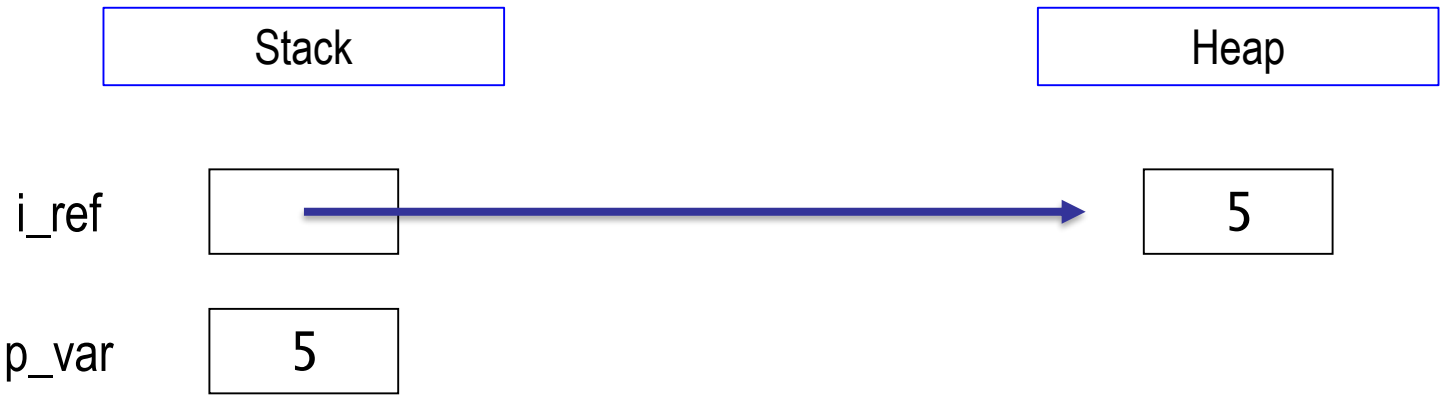
- Objects are stored in a memory region known as *the heap*.
- Memory on the heap is allocated using the `new` operator:

```
Integer val = new Integer(5); // an integer object
Scanner inp = new Scanner();
```

- `new` returns the memory address of the start of the object on the heap.
  - a reference!
- An object persists until there are no remaining references to it.
- Unused objects are automatically reclaimed by a process known as *garbage collection*.
  - makes their memory available for other objects

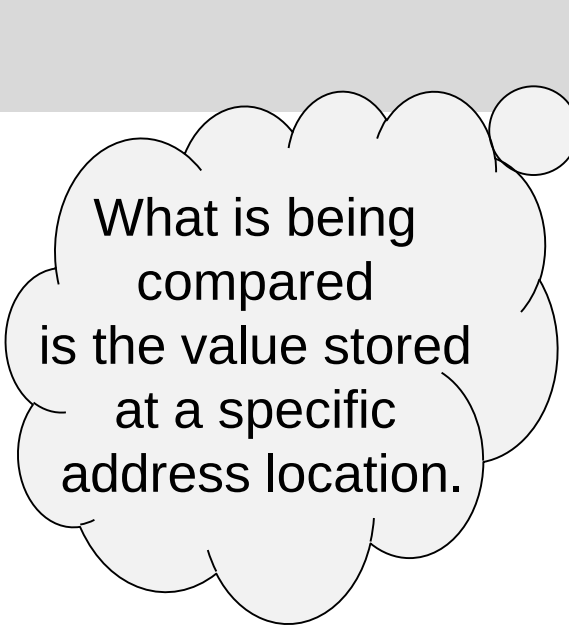
int Primitive vs. Integer Object

```
Integer i_ref = new Integer(5);      // an integer object
int p_var = 5;
```



Testing for Equivalent *Primitive* Values

- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.



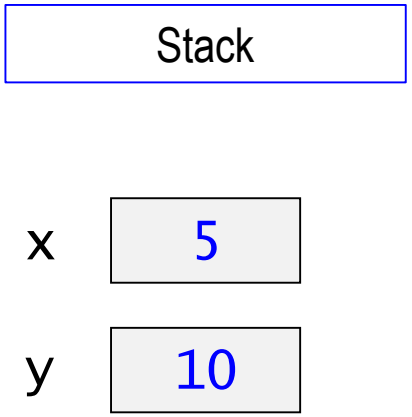
Testing for Equivalent *Primitive* Values

- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.

```
int x = 5;
int y = 10;
if ( x == y ) {


}
```

Comparing
the actual values
5 and 10.

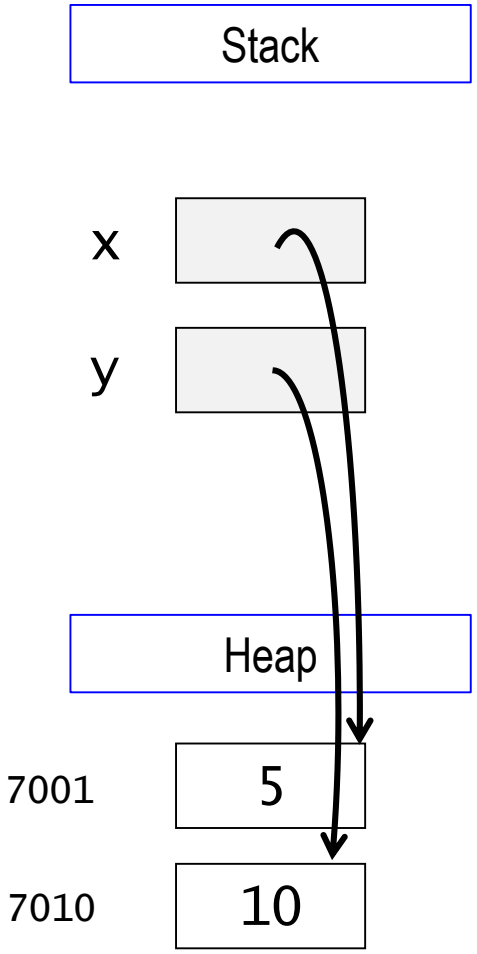


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

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

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```

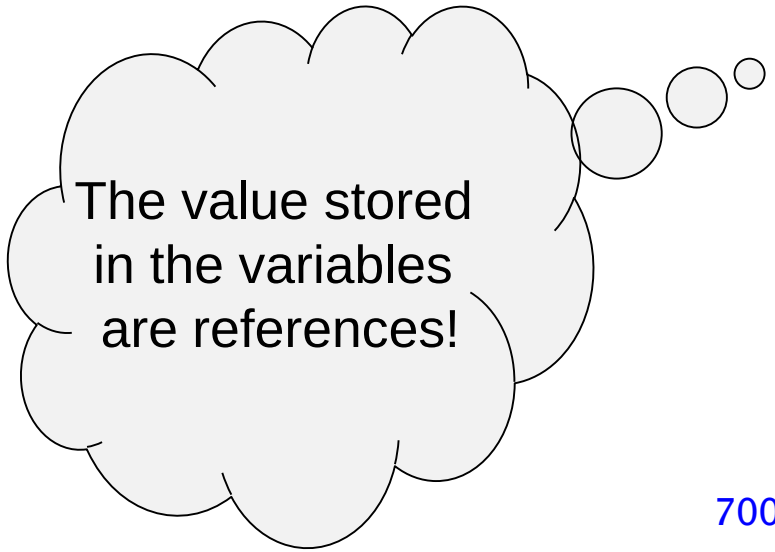


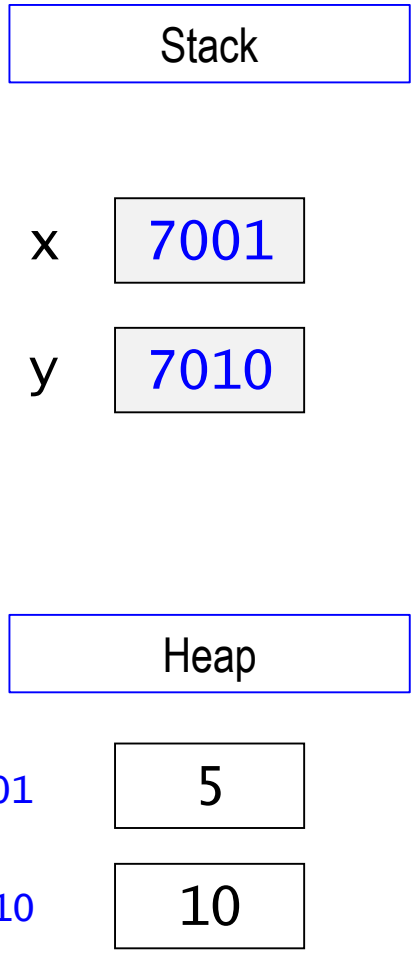
Testing for Equivalent *Objects:*
Numeric Wrapper Classes

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

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```



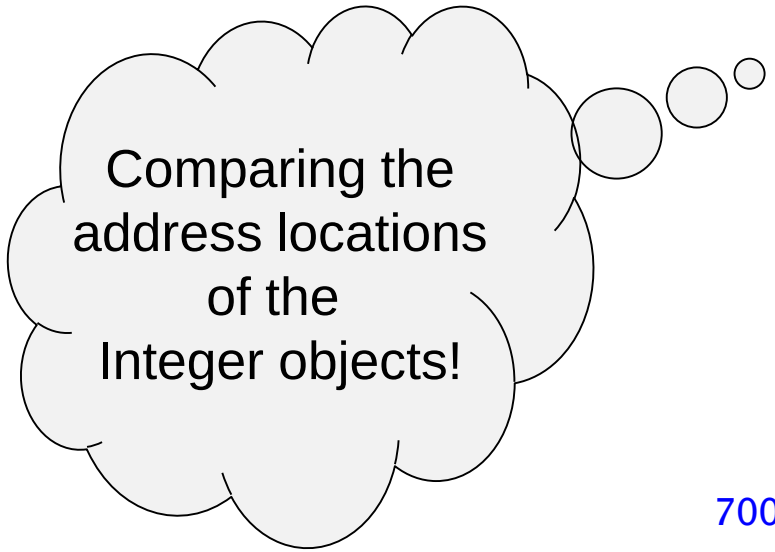


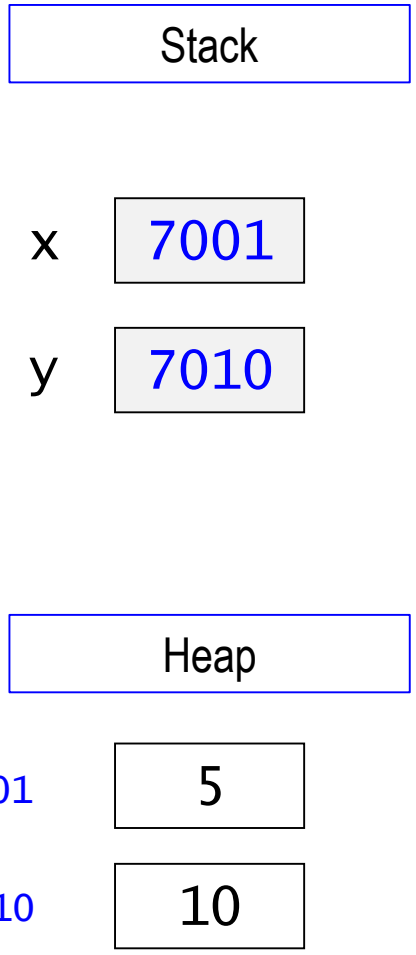
Testing for Equivalent *Objects:*
Numeric Wrapper Classes

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

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```

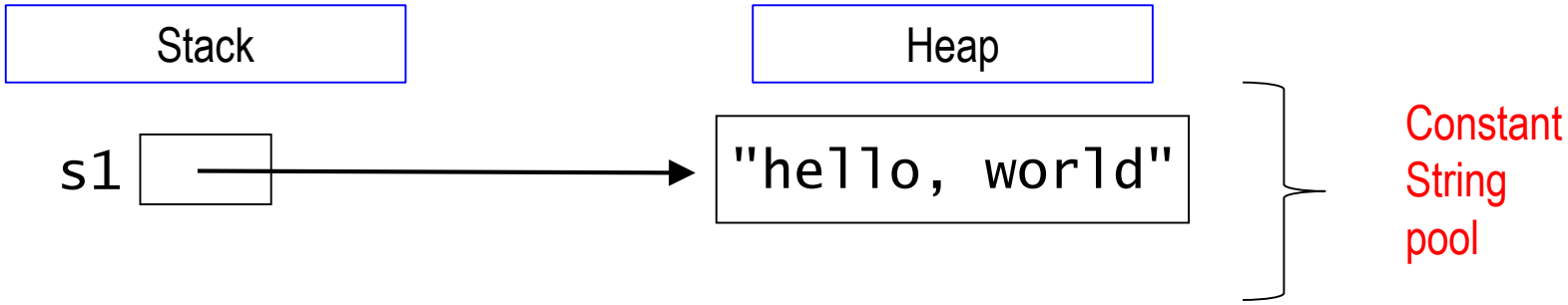




Testing for Equivalent *Strings:*
another look

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

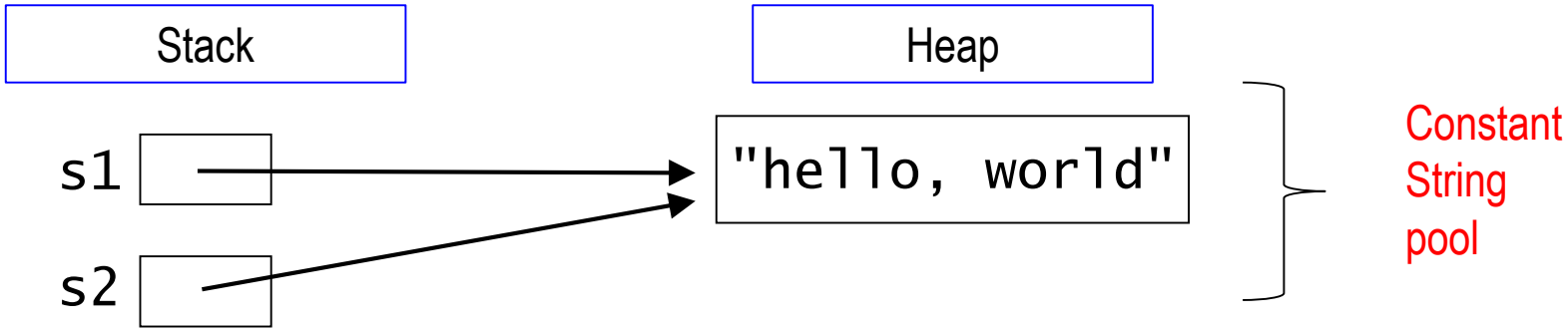
```
String s1 = "hello, world";   // constant string pool
String s2 = "hello, world";
String s3 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

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

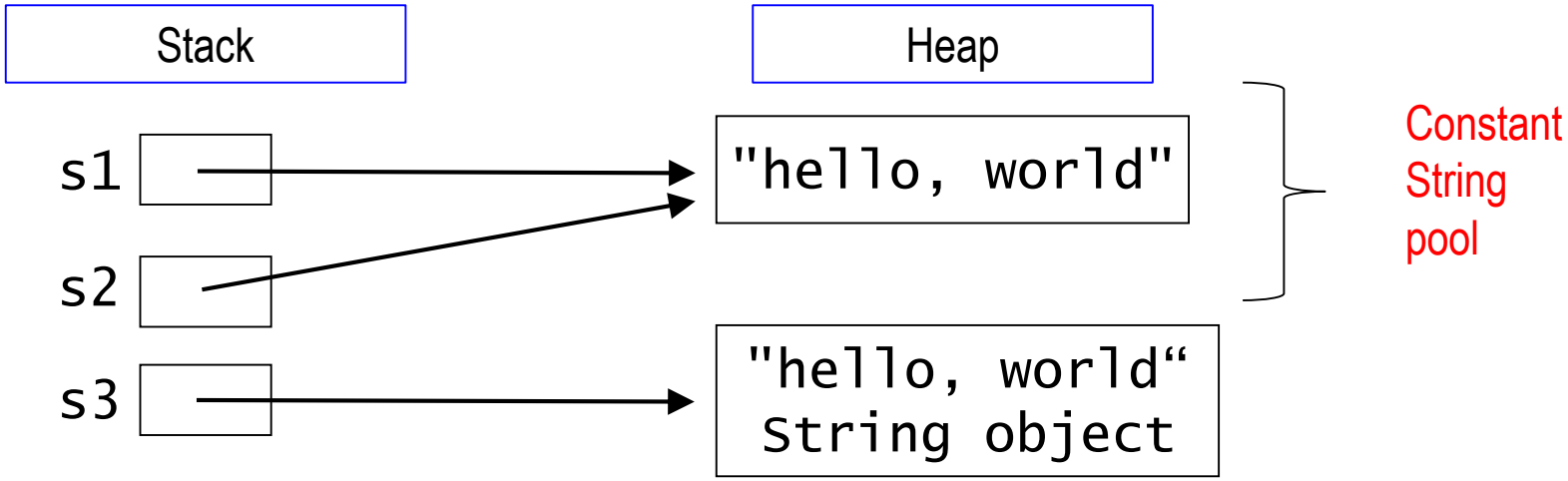
```
String s1 = "hello, world";
String s2 = "hello, world";   // constant string pool
String s3 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

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

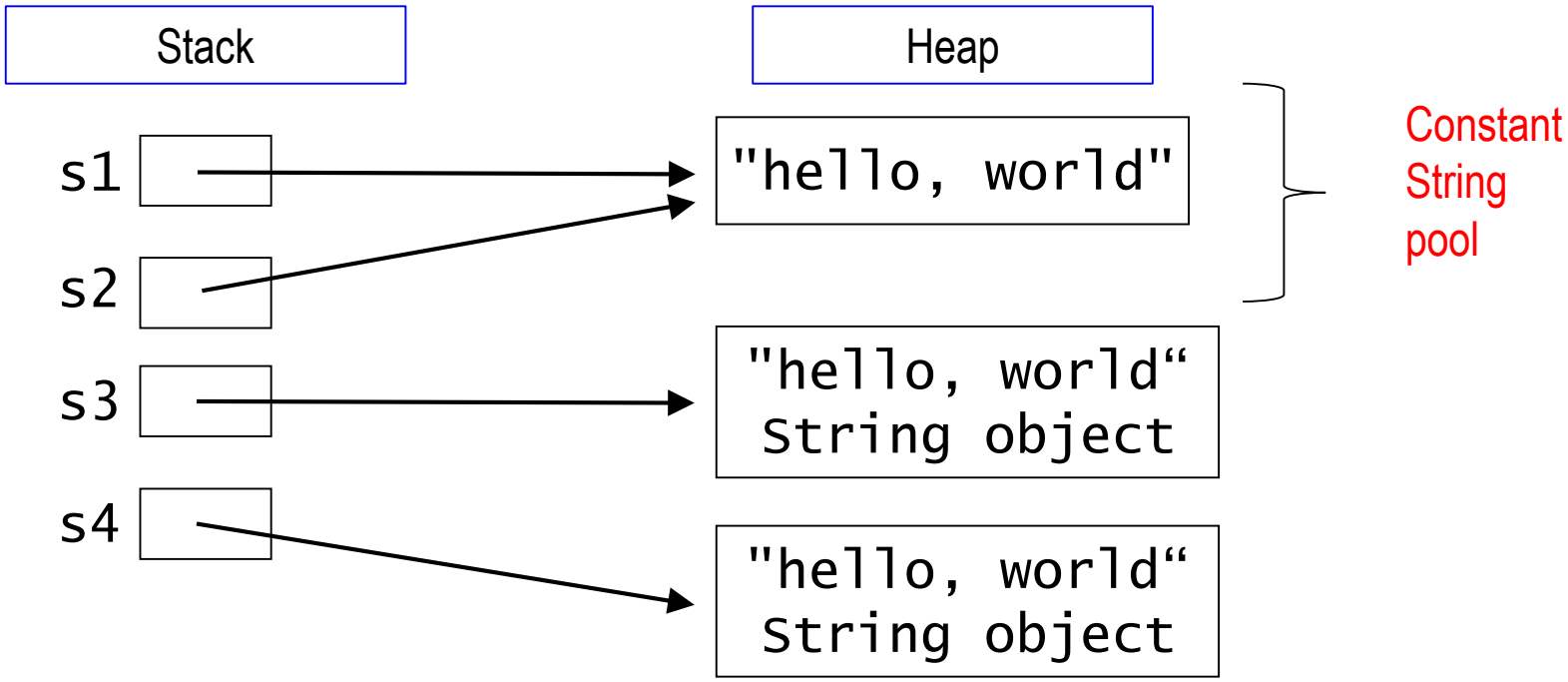
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");   // heap
```



Testing for Equivalent *Strings:*
another look

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

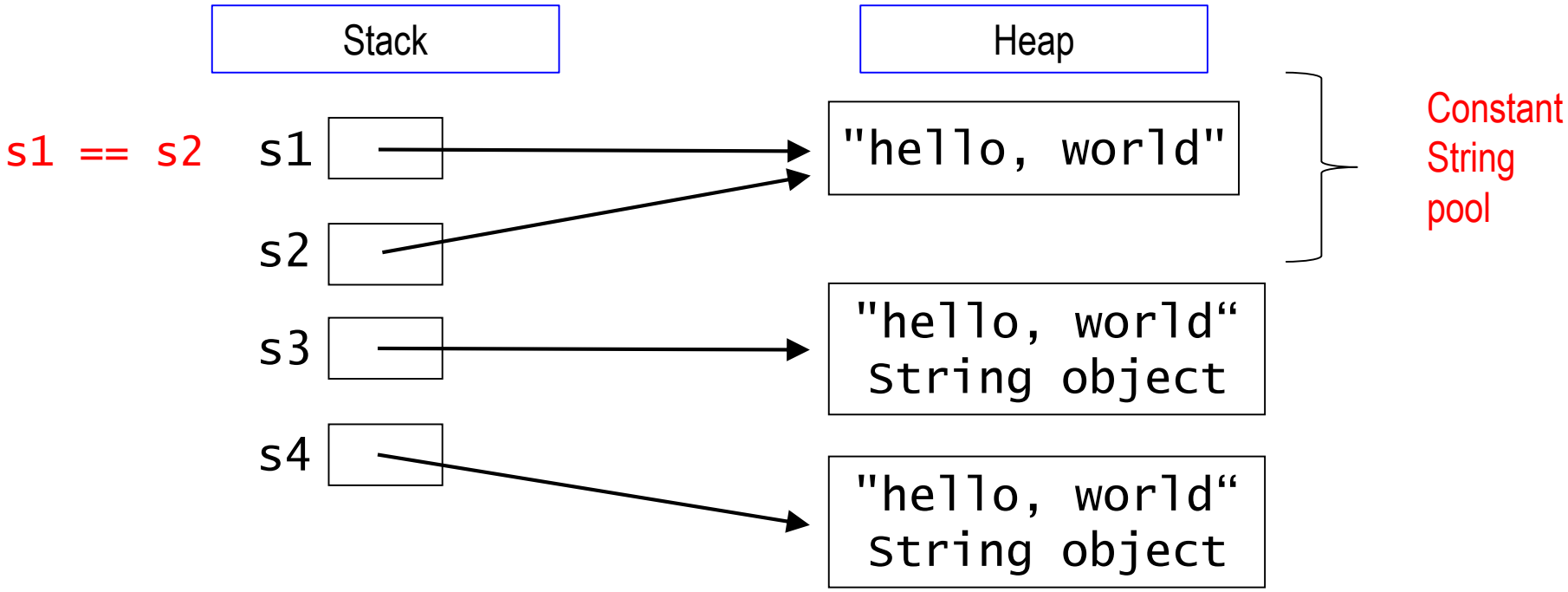
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");    // heap
```



Testing for Equivalent *Strings:*
another look

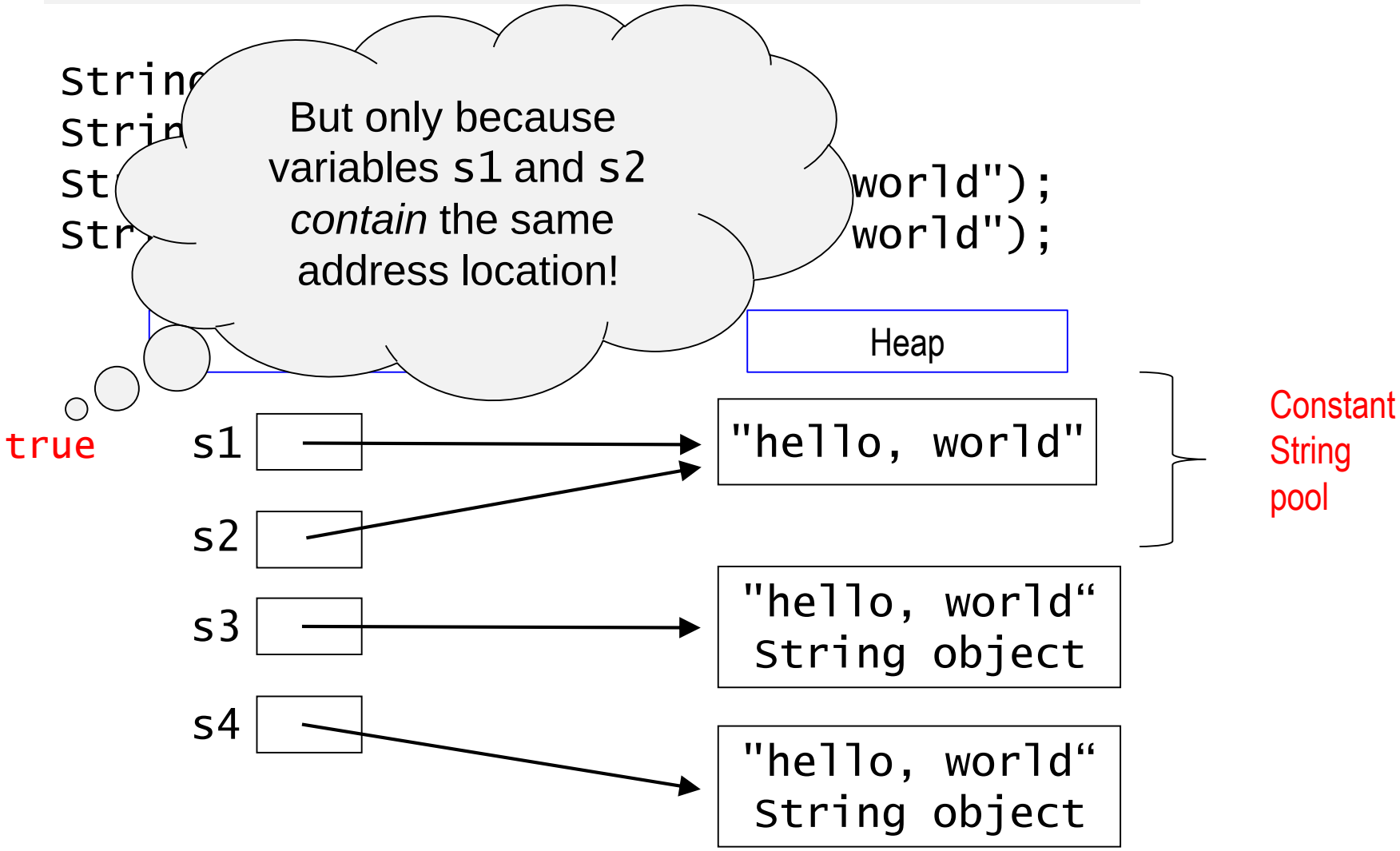
- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
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

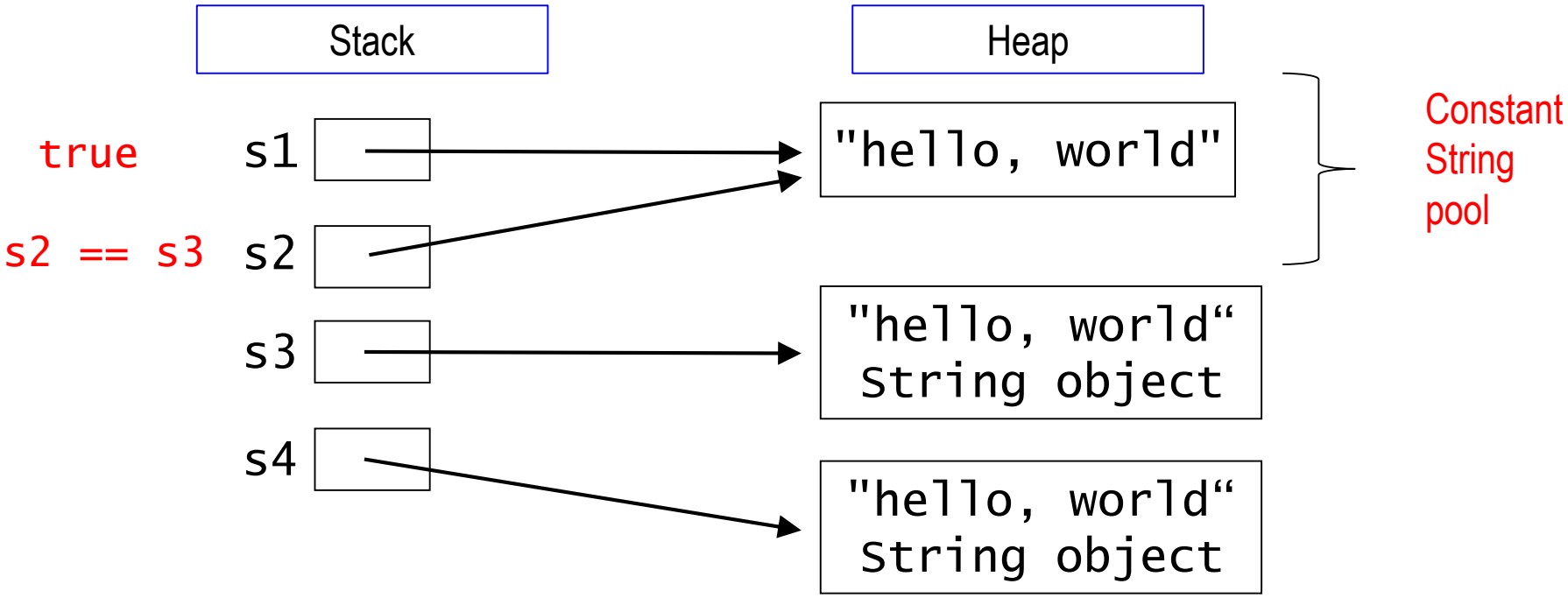
- The `==` and `!=` operators do *not* typically work when comparing *objects*



Testing for Equivalent *Strings:*
another look

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

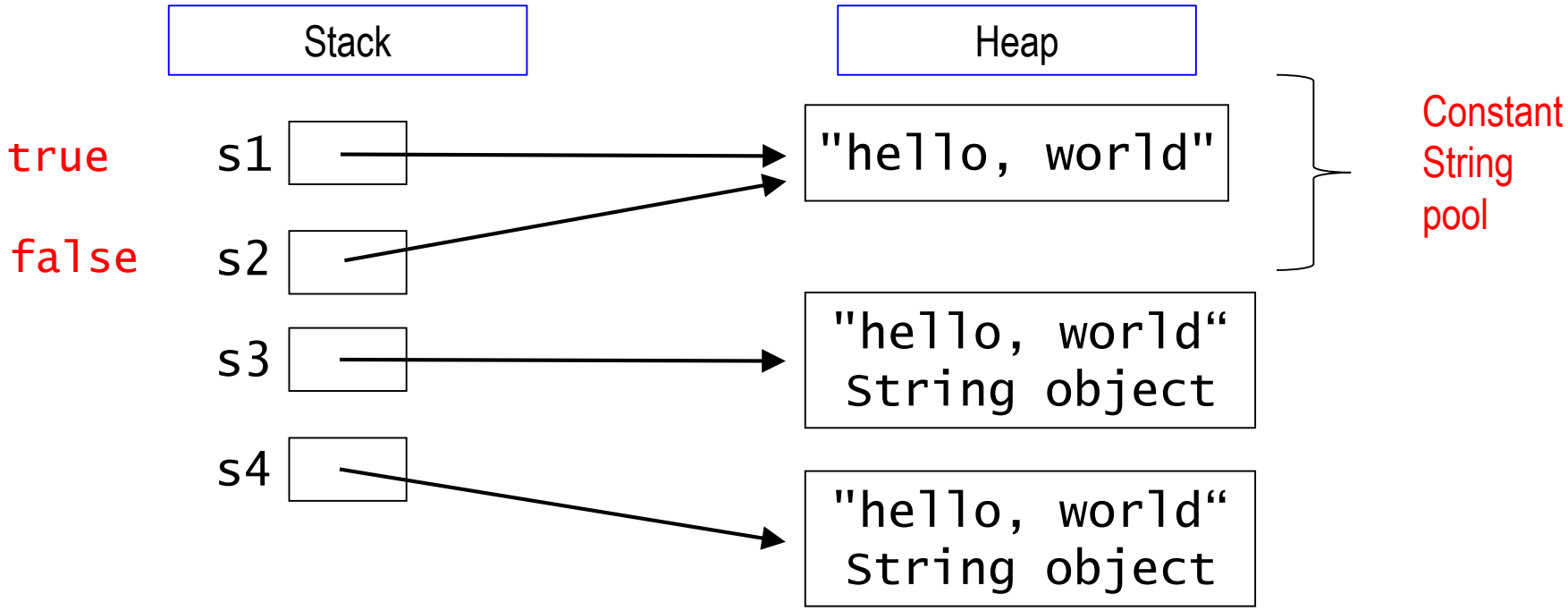
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

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

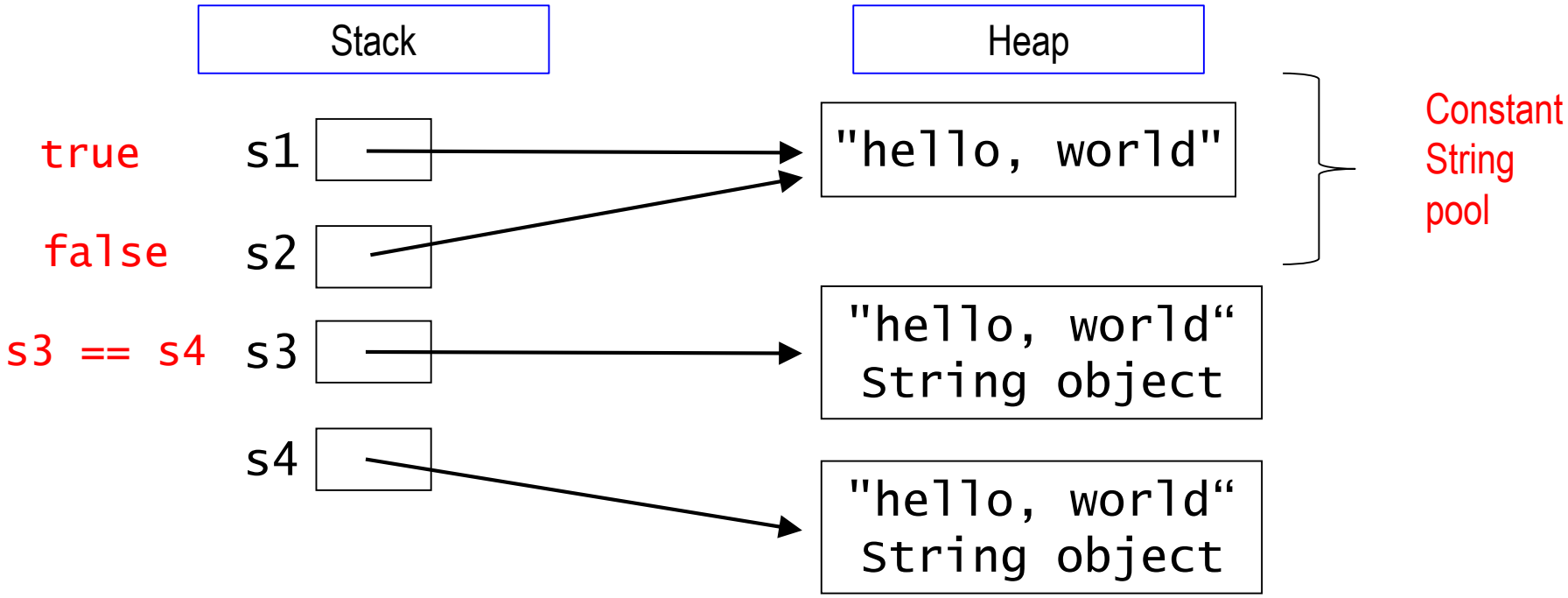
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

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

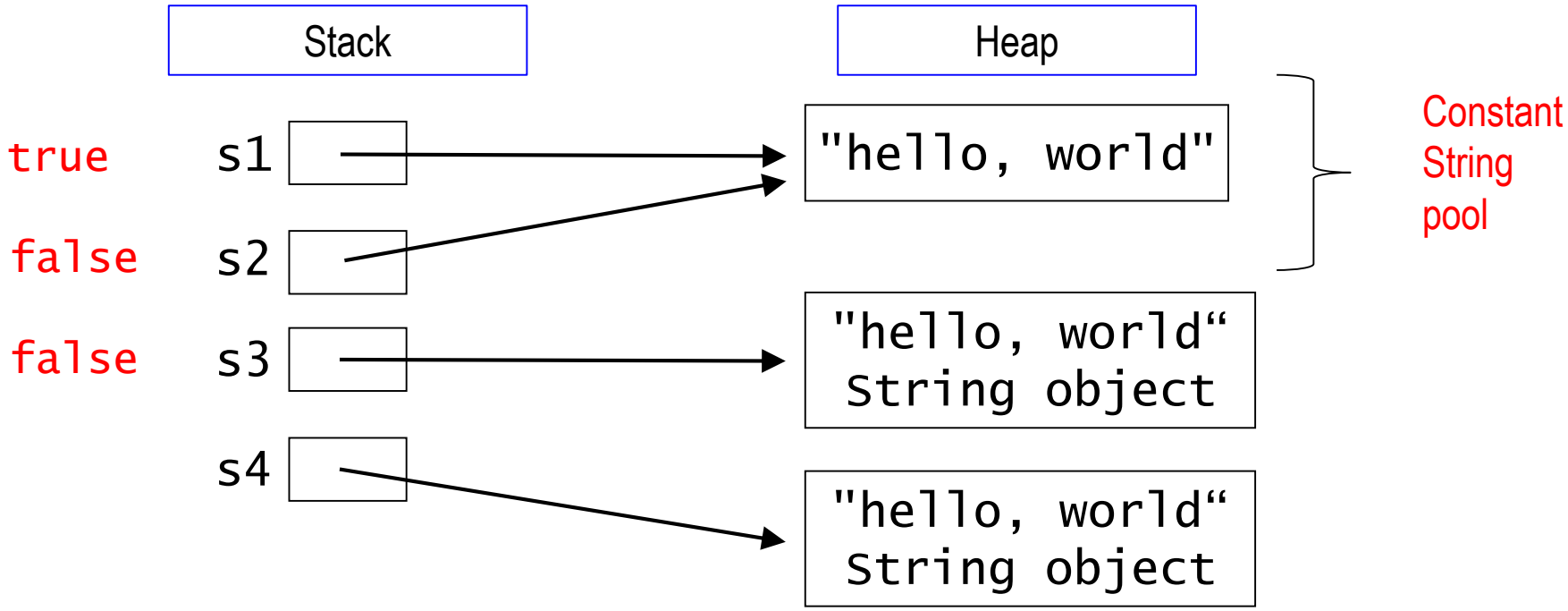
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



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");
```


```
true

false

false
```

Heap

`llo, world"`

Constant String pool

`llo, world“`
`ring object`

`llo, world“`
`ring object`

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"

Constant String pool

llo, world“
ring object

llo, world“
ring object

```
true

false

false
```

Back to Our 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.println("Show which number ? ");
      int number = scan.nextInt();

                        .
                        .
                        .
                        .
                        .


   }
}
```

Our 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.println("Show which number ? ");
      int number = scan.nextInt();

      System.out.printl("Number entered is: ");
      if (number == 1 )
          System.out.println(num1);
      else if (number == 2 )
         System.out.println(num2);
      else
         System.out.println(num3);
   }
}
```

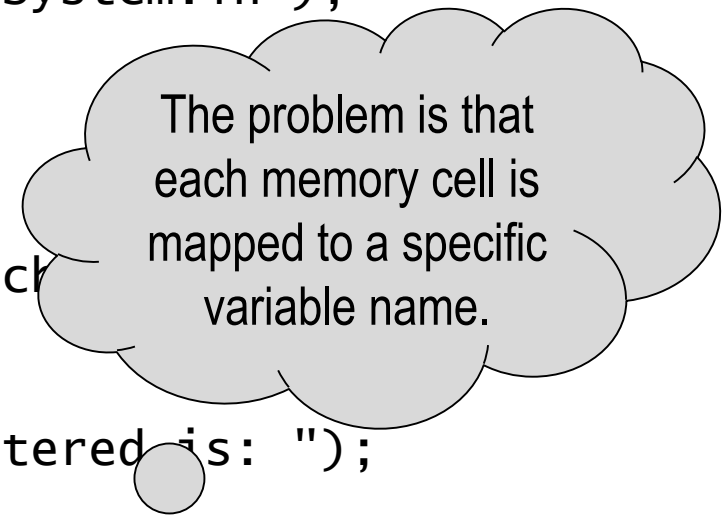
Our Simple Java Program

```
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.println("Show which
      int number = scan.nextInt();

      System.out.printl("Number entered is: ");
      if (number == 1 )
          System.out.println(num1);
      else if (number == 2 )
         System.out.println(num2);
      else
         System.out.println(num3);
   }
}
```



Our Simple Java Program with an array

```
import java.util.*;

public class Play {
   public static void main( String[] args ) {
      Scanner scan = new Scanner( System.in );
      int [] nums = int[3];

      nums[0] = scan.nextInt();
      nums[1] = scan.nextInt();
      nums[3] = scan.nextInt();

      System.out.println("Show which number ? ");
      int number = scan.nextInt();

      System.out.printl("Number entered is ");
      nums[number-1];


   }
}
```

Can access any one of the three integers entered through the *array* variable **nums** by using an offset.