

# Introduction to Programming Concepts and Tools

Lecture 4:

## Classes and Objects

Ira Pohl, Charlie McDowell:

Java by Dissection

## Today's Lecture:

- How to create your own classes and objects:
  - Instance variables and instance methods
  - Constructor methods, object creation
  - Calling Methods
  - Datahiding, access control
  - Static fields and static methods
  - Passing Objects, reference Types
- The predefined class String
- The predefined class StringBuffer

# Data Abstraction

- In Java there are three types of data values:
  - primitive data values (int, double, boolean, etc.)
  - arrays (actually a special type of object)
  - objects
- Objects in a program are used to represent "real" (and sometimes not so real) objects from the world around us.

# Objects

- An object might represent a string of characters, a planet, a type of food, a student, an employee, ... anything that can't be (easily) represented by a primitive value.
- Just as 3 is a primitive value of type int, every object must also have a type. These types are called classes.

# Classes

A class describes a set of objects.

- It specifies what information will be used to represent an object from the set (e.g. name and salary for an employee).
- It also specifies what operations can be performed on such an object (get the name of the student, update salary).

## Parts of a class:

- Name
- Instance variables
  - tell what kind of data is stored for an object of the class
- Constructor methods
  - tell what happens when objects of the class are created
- Instance methods
  - tell what kind of operations can be performed on an object of the class (an instance of the class)
- Static variables (Class variables)
- Static methods (Class methods)

## Defining Classes:

- Define a class in a separate file called `Classname.java`
- The file should be in the same folder as the programs using objects of the class
- Compile the file containing the class separately to check for syntax errors

# Elements of a Simple Class

- A class describes the data values used to represent an object and any operations that can be performed on that object.
- The data values are stored in *instance variables*, also known as *fields*, or *data members*.
- The operations are described by *instance methods*, sometimes called *procedure members*.

```
class Counter {  
    int value;           // instance variable  
  
    void reset() { value = 0; } // mutator method  
  
    int get()    { return value;} // accessor method  
  
    void click() { value = (value + 1) % 100;}  
}
```

```
// CounterTest.java - demonstration of class Counter
class CounterTest {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //create a Counter
        Counter c2 = new Counter(); //create another

        c1.click(); // increment Counter c1
        c2.click(); // increment Counter c2
        c2.click(); // increment Counter c2 again
        System.out.println("Counter1 value is " +
                           c1.get());
        System.out.println("Counter2 value is " +
                           c2.get());

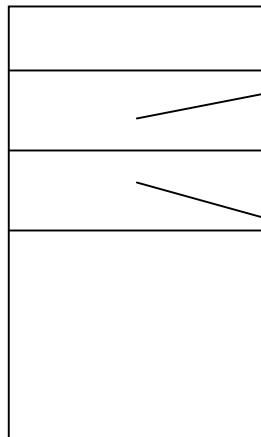
        c1.reset();
        System.out.println("Counter1 value is " +
                           c1.get());
    }
}
```

# Objects in memory

variables in main()

c1

c2



value

1

value

2

class Counter

reset()

get()

click()

The Heap

# Instance variables and methods

- As seen in the `CounterTest` example, you invoke an instance method by expressions such as

```
c1.click()
```

- You can use the same notation to access an instance variable. So in `main()` of `CounterTest` we could use

```
c1.value
```

# Data Hiding

- It is desirable to hide the inner details of a class (abstract data type) from the users of the class.
- We want to be able to determine the correctness of a class without examining the entire program of which it is a part.
- With our current class Counter we wish to assert that the value is always between 0 and 99.

# Accessing instance variables from outside the class breaks data hiding.

```
class CounterTest2 {  
    public static void main(String[] args) {  
        Counter c1 = new Counter(); //create a Counter  
        c1.value = 100; // breaks assumption about Counter  
  
        System.out.println("Counter1 value is " +  
                            c1.get());  
    }  
}
```

# A better Counter.

```
public class Counter {  
    private int value;           // instance variable  
  
    public void reset() { value = 0; } // mutator method  
  
    public int get()      { return value; } // accessor method  
  
    public void click() { value = (value + 1) % 100; }  
}
```

# public/private/default

- **private** methods/fields cannot be accessed from outside of the class
- **public** methods/fields can be accessed from anywhere
- default (no modifier) methods/fields have package access. They can be accessed from other classes in the same package.
  - If you don't specify a package (see section 12.11), all classes in the same directory are part of the same, default - unnamed package.

# Constructing objects

- Objects are created with  

```
new ClassName( )
```
- This allocates space for the object in the heap (memory), and initializes the object by invoking the *constructor* for the class if there is one.
- If there is no constructor, by default all fields are initialized (boolean fields are false, all other primitives are 0, and everything else is initialized to null).

# Adding constructors to Counter

```
public class Counter {  
  
    public Counter() { }  
    public Counter(int v) { value = v % 100; }  
  
    private int value;           // instance variable  
  
    public void reset() { value = 0; } // mutator method  
  
    public int get() { return value; } // accessor method  
  
    public void click() { value = (value + 1) % 100; }  
}
```

# Using constructors.

```
class CounterTest3 {  
    public static void main(String[] args) {  
        Counter c1 = new Counter(); //a Counter starting at 0  
        Counter c2 = new Counter(50); //one starting at 50  
  
        c1.click();  
        c2.click();  
        System.out.println("Counter1 value is " +  
                            c1.get());  
        System.out.println("Counter2 value is " +  
                            c2.get());  
    }  
}
```

The default, no-arg constructor is only provided when there are no user specified constructors.

If we had added only the constructor

```
public Counter(int v) { value = v % 100; }
```

then creating a Counter with

```
Counter myCounter = new Counter();
```

would be a syntax error. There no longer is a constructor that takes zero arguments.

# The ADT Stack in Java

- A stack is a data structure that supports three operations: push, pop, and top.
- The following is an implementation of a generic stack in Java.

```
public class Stack {  
    private StackElement top;  
    private int count=0;  
    public void push(Object obj){  
        StackElement element = new StackElement(obj);  
        element.next = top;  
        top = element;  
        count++;  
    }  
    public Object pop() {  
        if(top == null) return null;  
        Object obj = top.value;  
        top = top.next;  
        count--;  
        return obj;  
    }  
}
```

```
class StackElement {  
    Object value;  
    StackElement next;  
    StackElement(Object obj)  
    {  
        value = obj;  
        next = null;  
    }  
}
```

```
public class StackTest {
    public static void main(String[] args)
    {
        Stack stack = new Stack();
        stack.push("Testing");
        stack.push("One");
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

# Static fields and methods

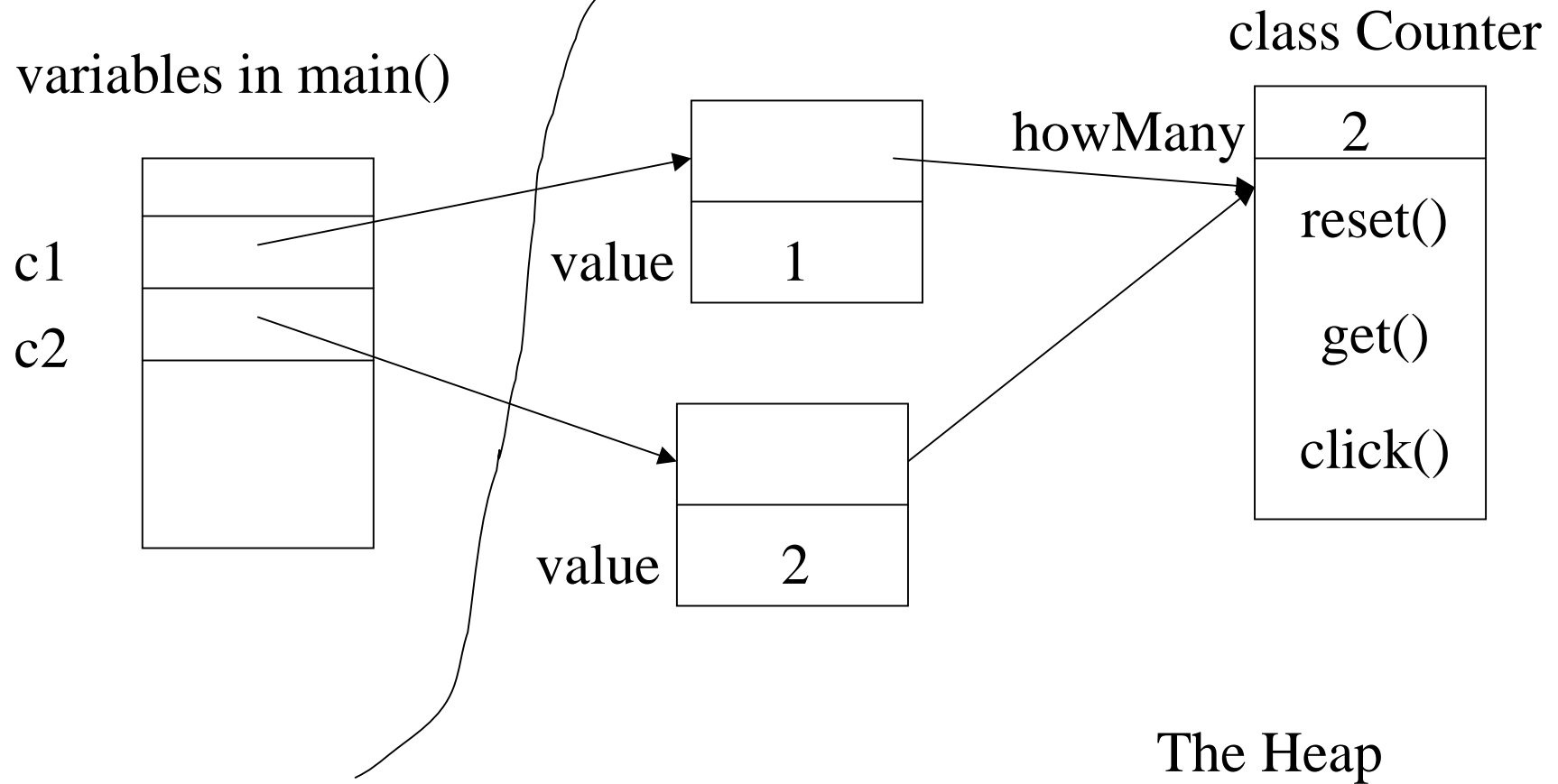
- Static methods don't operate (implicitly) on an instance of the class containing the method.
- Likewise, static fields are not part of an object, they are instead part of the class, hence also called *class variables*.

## Adding a static method and a static field to Counter.

```
public class Counter {  
  
    private int value;  
    private static int howMany = 0;  
  
    public Counter(){ howMany++; }  
    public void reset() { value = 0; }  
    public int get()    { return value; }  
    public void click() { value = (value + 1) % 100; }  
    public static int howMany() { return howMany; }  
}
```

```
// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        System.out.println(Counter.howMany());
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.click();
        c2.click();
        c2.click();
        System.out.println("Counter1 value is " +
            c1.get()); //prints Counter1 value is 1
        System.out.println("Counter2 value is " +
            c2.get()); //prints Counter2 value is 2
        System.out.println(Counter.howMany()); // prints 2
    }
}
```

# Objects in memory



# Calling Methods

- methods in the same class
  - just use the name
  - works for
    - instance method to instance method
    - instance method to static method
    - but NOT static method to instance method
- instance methods
  - `objectReference.methodName()`
- class methods
  - `ClassName.methodName()`

# instance to instance

- We could implement `click()` in `Counter` with

```
void click() { value = (get() + 1) % 100; }
```

- This call to `get()` is operating on the same `Counter` object as the one used to invoke `click()`.

# Scope

- The scope of class and instance variables is the entire class, regardless of where the declaration appears.
- Local variables can hide class/instance variables (have the same name). The local variable takes precedence. You can still access the class/instance variables with the keyword **this**.

Notice that in the following, if you read from top to bottom, we use value before encountering the definition.

```
class Counter {  
    void reset() { value = 0; }  
    int get()    { return value; }    //current value  
    void click() { value = (value + 1) % 100; }  
    int  value;           //0 to 99  
}
```

## Accessing hidden instance variables using the keyword **this**

```
public class Counter {  
  
    public Counter() { }  
    public Counter(int value) {  
        this.value = value % 100; }  
  
    private int value;           // instance variable  
  
    public void reset() { value = 0; } // mutator method  
  
    public int get() { return value; } // accessor method  
  
    public void click() { value = (value + 1) % 100; }  
}
```

The scope of the class variable x, overlaps the scope of the local variable x.

```
//Scope2.java: class versus local scope
class Scope2 {
    public static void main(String[] args) {
        int x = 2;

        System.out.println("local x = " + x);
        System.out.println("class variable x = "
            + Scope2.x);
    }
    static int x = 1;
}
```

# Why not static to instance?

- When calling one instance method in the same class from another in the same class, they both operate on the same, implicit object.
- When executing a static method there is NO implicit object being operated on, hence calling an instance method in the same class using only the method names, doesn't specify what object to operate on.

When executing the call to `howMany()` below, what Counter object is being manipulated?

```
// CounterTest2.java - demonstration of a static field
class CounterTest2 {
    public static void main(String[] args) {
        ...
        System.out.println(Counter.howMany()); // prints 2
    }
}
```

Answer: There isn't one. So trying to call `get()` from within `howMany()` like we did from within `click()` won't work.

# The class `String`

- `String` is a standard Java class. Values from the class `String` are called objects, so "hello" is an object from the class `String`.
- We can also say "hello" is an *instance of* the class `String`.
- The class `String` has instance methods that operate on an instance of class `String`. For example: `length()` and `charAt()`.<sup>37</sup>

# An example using String

Write a program that determines if a String is a palindrome.

## Pseudocode

```
compare the first character with the last character
if they are not equal then return false
compare the second character with the next to last
if they are not equal then return false
continue until middle two characters are compared
```

# Refined Palindrome Pseudocode

```
set left to index the leftmost or first character
set right to index the rightmost or last character
while left is less than right
    compare the left character with the right character
    if they are not equal then return false
    increment left
    decrement right
end of the while loop
return true
```

```
public class Palindrome {
    public static void main(String[] args) {
        String str1 = "eye", str2 = "bye";
        System.out.println("Palindrome detection");
        System.out.println(str1 + " "
                            + isPalindrome(str1));
        System.out.println(str2 + " "
                            + isPalindrome(str2));
    }
}
```

```
static boolean isPalindrome(String s) {  
    int left = 0;  
    int right = s.length() - 1;  
    while (left < right) {  
        if (s.charAt(left) != s.charAt(right))  
            return false;  
        left++;  
        right--;  
    }  
    return true;  
}  
}
```

# String is a bit special

Because strings are so common, Java has two pieces of special syntax for the class `String`.

- There is syntactic support for string concatenation.
- There is syntactic support for creating string literals.

# String concatenation

The operator `+` is overloaded to implement concatenation of strings.

```
"hello, " + "world"
```

is equivalent to

```
"hello, ".concat("world")
```

# String literals

String literals are supported.

```
String s = "hello"
```

is equivalent to

```
char[] temp = { 'h', 'e', 'l', 'l', 'o' };  
String s = new String(temp);
```

# Strings are immutable

- Instances of the class `String` are *immutable*. This means once created, a `String` object cannot be changed.
- One implication of this is that in the following code fragment:

```
String s = "some string";  
...someFunction(s)...
```

we know for certain that when the function returns, `s` will still be "some string".

# A mutable class - StringBuffer

StringBuffer is another standard Java class for representing strings. Unlike String, instances of the class StringBuffer are mutable. The class StringBuffer has mutator methods - operations (instance methods) that actually change the object.

```
class StringBufferInsert {
    public static void main(String[] args) {
        StringBuffer sbuf = new StringBuffer("some string");
        sbuf.insert(sbuf.length() / 2, "mutable ");
        System.out.println(sbuf);
    }
}
```

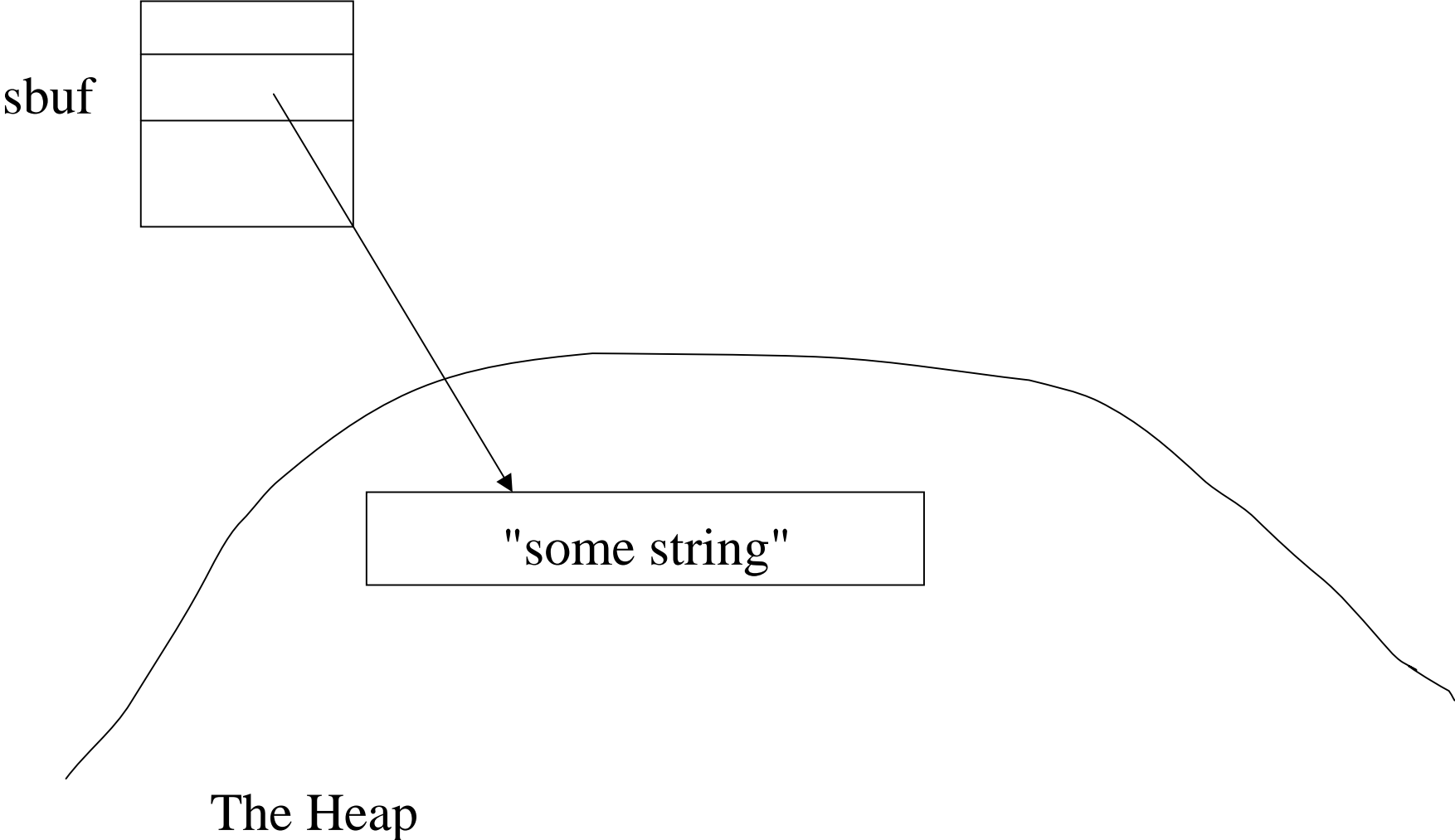
## **Prints:**

some mutable string

# Passing Objects to methods

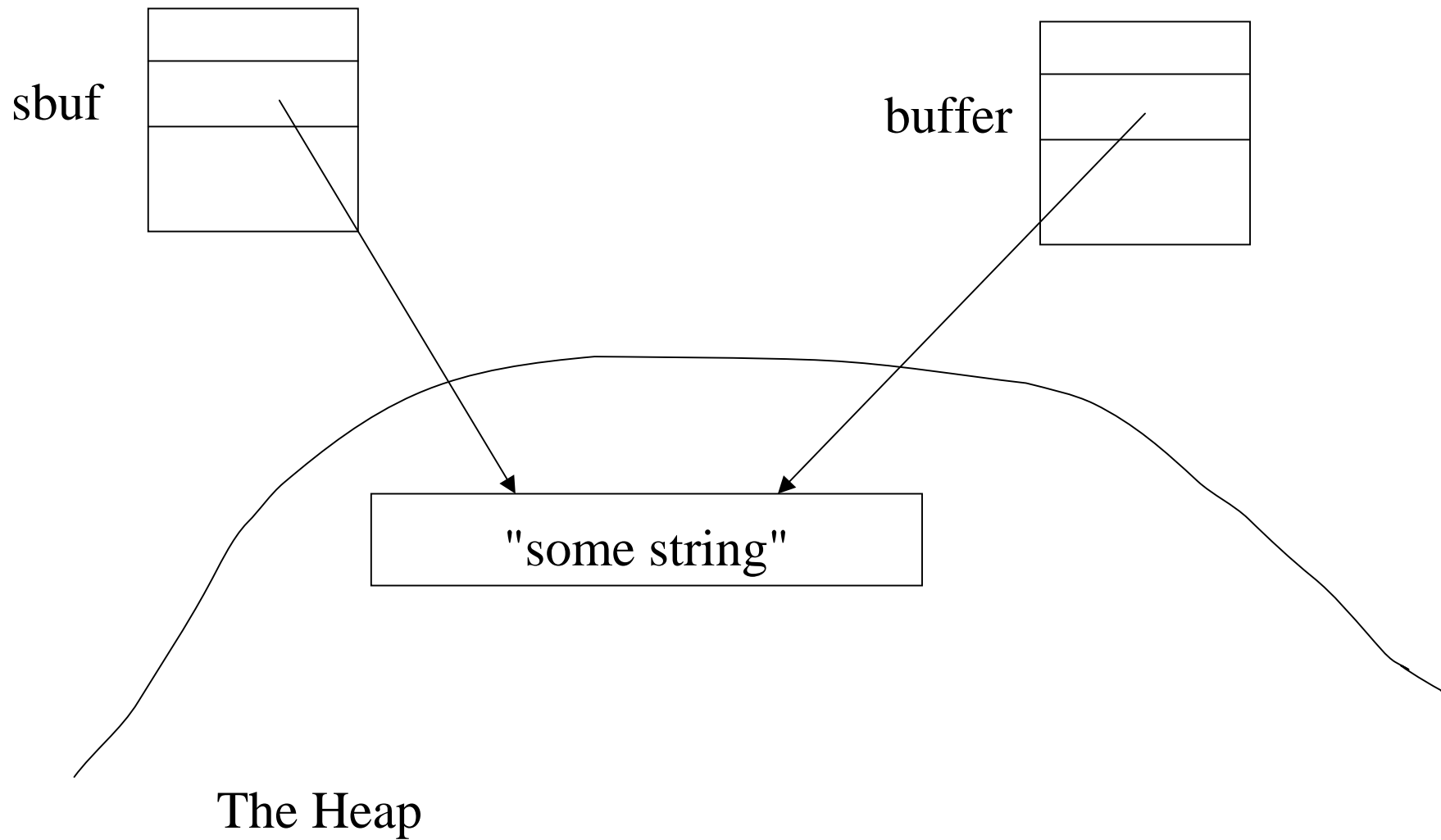
```
class StringBufferInsert {
    public static void main(String[] args) {
        StringBuffer sbuf = new StringBuffer("some string");
        insertInTheMiddle(sbuf, "mutable ");
        System.out.println(sbuf);
    }
    static void insertInTheMiddle(StringBuffer buffer,
                                   String insertThis)
    {
        buffer.insert(buffer.length() / 2, insertThis);
    }
}
```

variables in main()



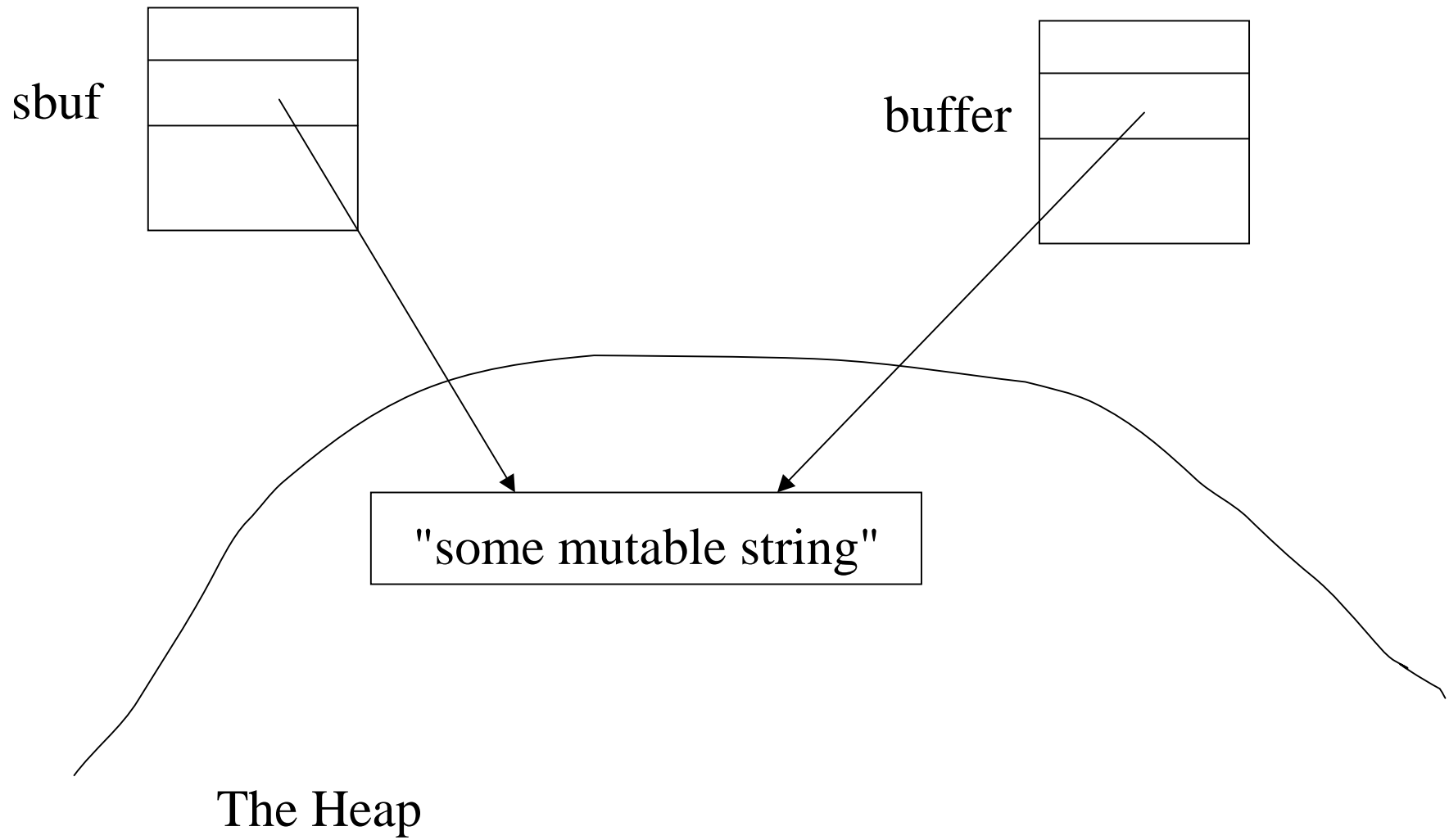
variables in main()

variables in insertInMiddle()



variables in main()

variables in insertInMiddle()



# Call by Value

- Although you can modify an object that is passed to a method, you still cannot modify a variable that is a reference type variable (a class or array).
- There is a difference between modifying an object, and modifying a variable, when the variable is a reference (anything other than a primitive type in Java).

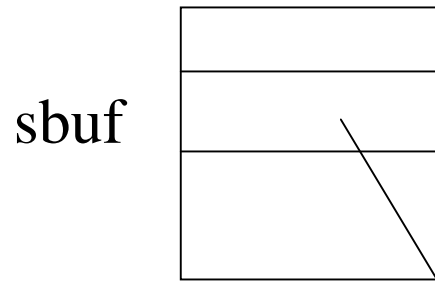
```
// ModifyParameters.java - you can't modify the actual
//      arg even when it is a reference
class ModifyParameters {
    public static void main(String[] args) {
        StringBuffer sbuf = new StringBuffer("testing");

        System.out.println("sbuf is now " + sbuf);
        modify(sbuf);
        System.out.println("sbuf is now " + sbuf);
    }
    static void modify(StringBuffer sb) {
        sb = new StringBuffer("doesn't work");
    }
}
```

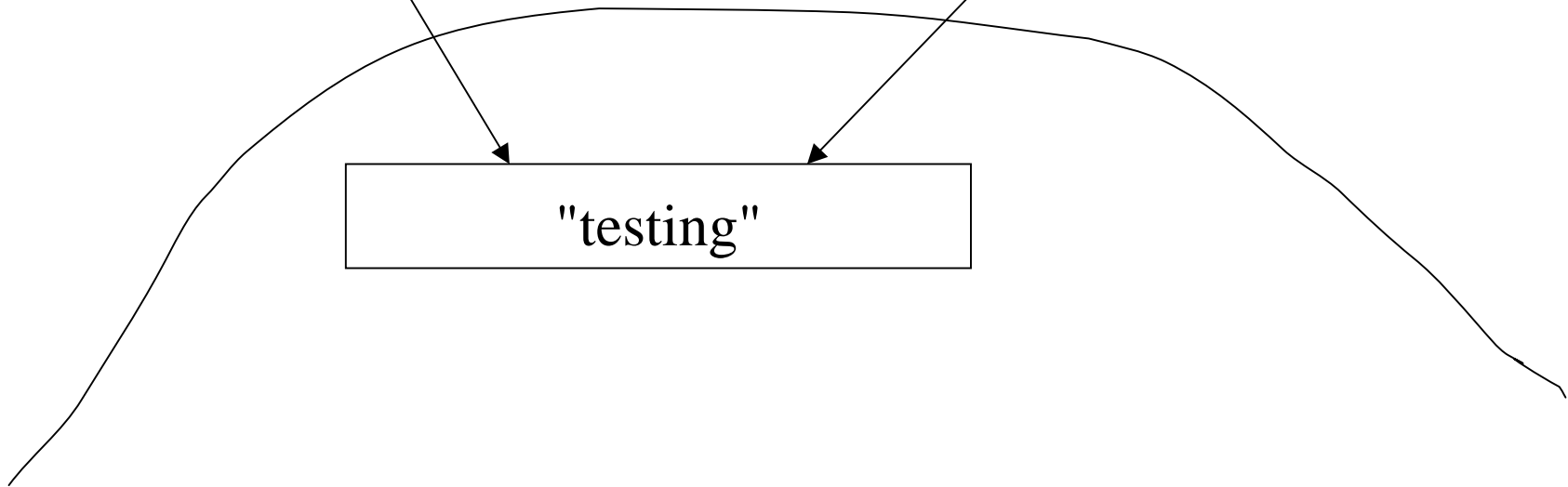
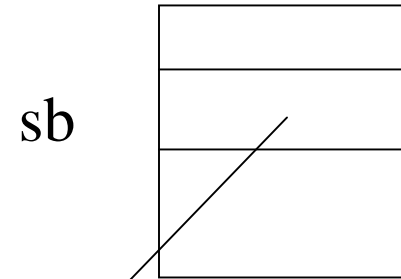
## Prints:

```
sbuf is now testing
sbuf is now testing
```

variables in main()

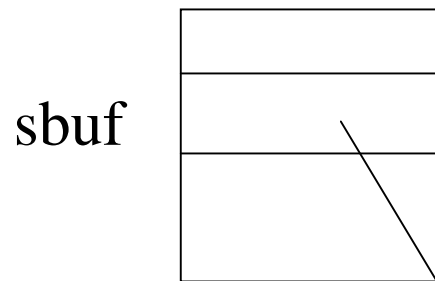


variables in modify()

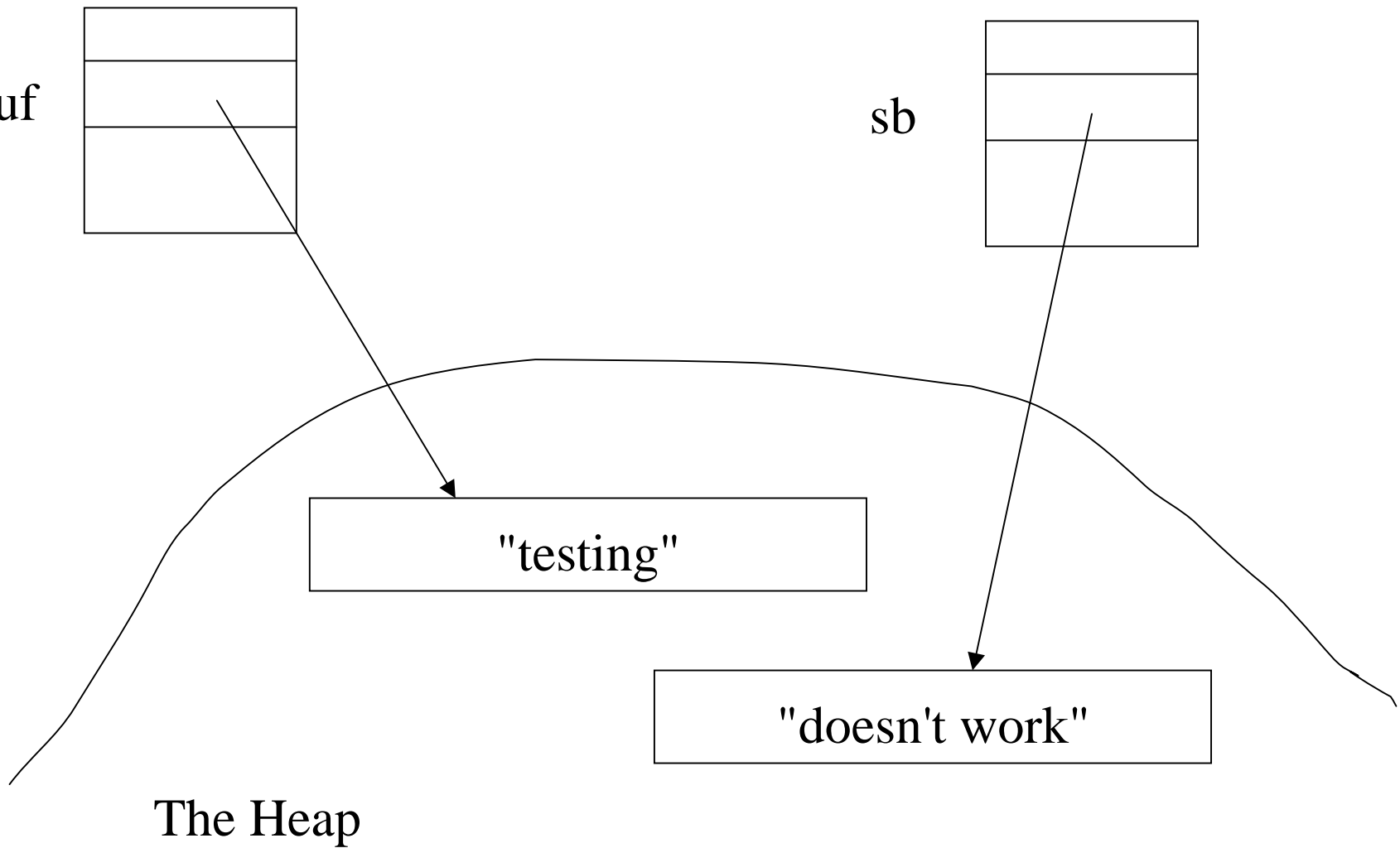
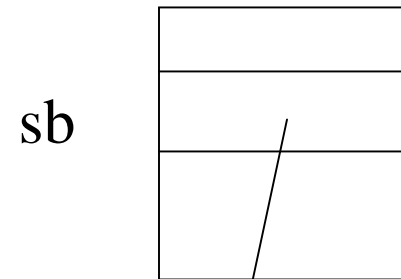


The Heap

variables in main()



variables in modify()



# Symbolic Constants

```
public class Counter {  
  
    public Counter() { }  
    public Counter(int v) { value = v % MODULUS; }  
  
    private int value;           // instance variable  
  
    public void reset() { value = 0; } // mutator method  
  
    public int get() { return value; } // accessor method  
  
    public void click() { value = (value + 1) % MODULUS; }  
  
    private static final int MODULUS = 100;  
}
```