

Introduction to Programming Concepts and Tools

Lecture 4:

Classes and Objects

Book

Ira Pohl, Charlie McDowell:
Java by Dissection Chp.6

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 sorts 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 that we want to model.

Objects

- An object might represent a string of characters, a planet, a type of food, a car, 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, update salary).

Classes

Stated differently

A class describes an arbitrary object in a set.

- It specifies the object's attributes.
- It also specifies how the attributes can be touched and manipulated. (instance methods)

A class is a user defined type.

When we have defined a class *Car*, then we can declare variables of type *Car*.

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 Clock{
    int hour, min;
        // instance variables.
        // Not declared in the body of a method

void reset(){ hour = 0; min = 0; }
    // mutator method. Implicit parameter: this object

int getHour(){ return hour;}
    // accessor method. Implicit parameter: this object

int getMin(){ return min;}
    // accessor method. Implicit parameter: this object

String getTime(){ return(hour + ":" + min)}

}

```

A class is a user-defined type. The class Clock is not in itself a program (no main method). It is a type that can be used in programs.

```

class Clock{
    int hour, min;

    void reset(){ hour = 0; min = 0; }

    int getHour(){ return hour;}

    int getMin(){ return min;}

    String getTime(){ return(hour + ":" + min);}

    void tick(){
        boolean turn = ((min+1)%60==0);
        if (turn)
            {min = 0; hour = (hour+1)%24;}
        else
            min = min + 1;
    }
}

```

```

// ClockTest.java - demonstration of class Clock
class ClockTest {
    public static void main(String[] args) {
        int inc = 130;
        Clock c1 = new Clock(); //create a Clock
        Clock c2 = new Clock(); //create a Clock

        c1.tick(); // increment Clock c1
        for (int i = 1; i <= inc; i++){c2.tick();}
        // increment Clock c2 by 130

        System.out.println("Clock1 time is " +
                            c1.getTime());
        System.out.println("Clock2 time is " +
                            c2.getTime());

        c2.reset();
        System.out.println("Reset Clock2, then time is " +
                            c2.getTime());
    }
}

```

Reference variables

```
//create a Clock  
Clock c1 = new Clock();
```

c1 is a variable of type Clock,

It holds a *reference* to an object

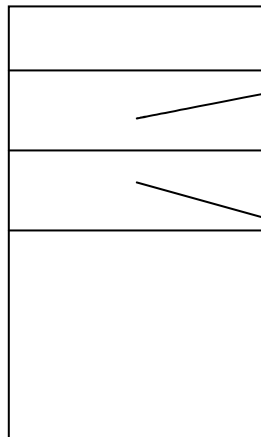
(a pointer to the location where the object is stored)

Objects in memory

variables in main()

c1

c2



hour

0

min

1

hour

2

min

10

class Clock

reset()

getHour()

getMin()

getTime()

tick()

Instance variables and methods

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

```
c1.tick()
```

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

```
c1.hour
```

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 `Clock` we wish to assert that the value of `hour` is always between 0 and 23.

Accessing instance variables from outside the class breaks data hiding.

```
class ClockTest2 {  
    public static void main(String[] args) {  
        Clock c1 = new Clock(); //create a Clock  
        c1.hour = 45; // breaks assumption about Clock  
  
        System.out.println("Clock hour is " +  
                            c1.getHour());  
    }  
}
```

A better Clock.

```
public class Clock {
    private int hour, min;

    public void reset(){ hour = 0; min = 0; } //mutator mth
    public int getHour(){ return hour;}
    public int getMin(){ return min;}

    public String getTime(){ return(hour + ":" + min);}

    public void tick(){ //mutator mth
        boolean turn = ((min+1)%60==0);
        if (turn)
            {min = 0; hour = (hour+1)%24;}
        else
            min = min + 1;
    }
}
```

public/private/default

- **private** methods/fields cannot be accessed from outside of the class
- **public** methods/fields can be accessed from anywhere the class is known
- 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 Clock

```
public class Clock {  
  
    public Clock() {}  
  
    public Clock(int h, int m) {  
        hour = Math.abs(h) % 24;  
        min = Math.abs(m) % 60;  
    }  
    //here instance fields and  
    //instance methods as before  
    // .....  
}
```

Constructors

a constructor has no return type

a constructor has the class name

a constructor can have zero or more arguments (parameters)

Using constructors.

```
class ClockTest4 {  
    public static void main(String[] args) {  
        Clock c1 = new Clock(); //a Clock starting at 0:0  
        Clock c2 = new Clock(16,23); //one starting at 16:23  
        Clock c3 = new Clock(-2,-125); //one starting at 2:05  
  
        c1.tick();  
        for (int i = 1; i <= 80; i++){c2.tick();}  
  
        System.out.println("Clock1 time is " +  
                            c1.getTime());  
        System.out.println("Clock2 time is " +  
                            c2.getTime());  
        System.out.println("Clock3 time is " +  
                            c3.getTime());  
    }  
}
```

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

If we had added only the constructor

```
public Clock(int h, int m) {  
    hour = Math.abs(h) % 24;  
    min = Math.abs(m) % 60;}  
}
```

then creating a Counter with

```
Clock c = new Clock();
```

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

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 Clock.

```
public class Clock {
    private static int howMany;

    public static int getHowMany() {
        return howMany;    }

    public Clock() {howMany++; }

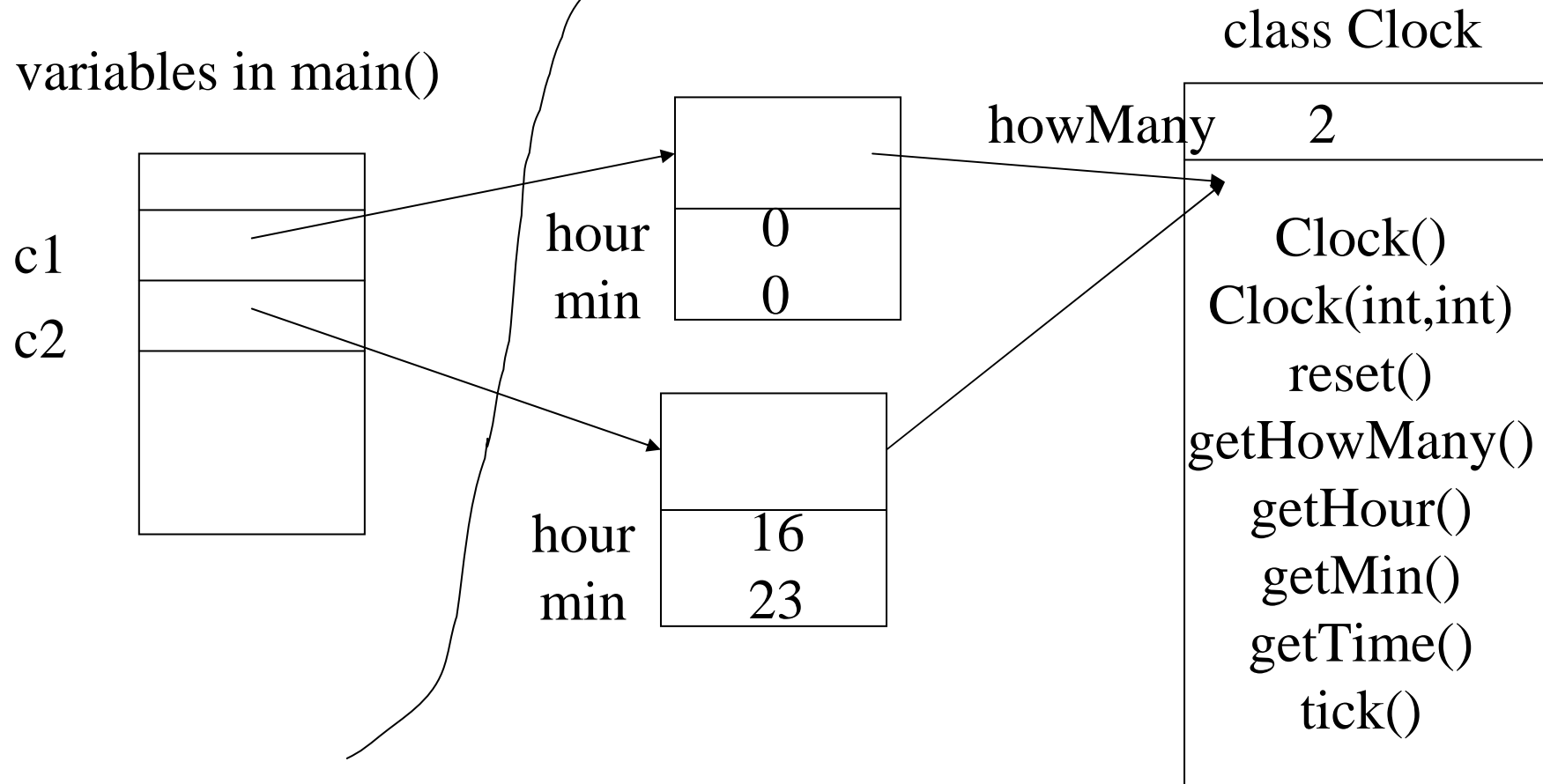
    public Clock(int h, int m) {
        hour = Math.abs(h) % 24;
        min = Math.abs(m) % 60;
        howMany++;
    }

    private int hour, min;

    // here instance methods as before
    // .....
}
```

```
// ClockTest5.java - demonstration of a static field
class ClockTest5 {
    public static void main(String[] args) {
        System.out.println("howMany: " + Clock.getHowMany());
        Clock c1 = new Clock();
        Clock c2 = new Clock(23,59);
        c1.tick();
        c2.tick();
        c2.tick();
        System.out.println("Clock1 time is " +
            c1.getTime());
        System.out.println("Clock2 time is " +
            c2.getTime());
        System.out.println("howMany: " + Clock.getHowMany());
    }
}
```

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 `tick()` in `Clock` with

```
void tick(){
    boolean turn = ((getMin()+1)%60==0);
    if (turn)
        {min = 0; hour = (hour+1)%24;}
    else
        min = min + 1;
```

- This call to `getMin()` is operating on the same `Clock` object as the one used to invoke `tick()`.

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 instance variables with the keyword **this**, and the class variables with the classname
- Example: a hidden instance variable, `this.hour`
a hidden static variable, `Classname.x`

Notice that in the following, if you read from top to bottom, we use `hour` and `min` before encountering the definitions.

```
class Clock {  
    void reset() { hour = 0; min = 0; }  
    int getHour() { return hour; }  
    int getMin() { return min; }  
    .....  
    int    hour, min;  
}
```

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

```
public class Clock {  
  
    public Clock() { }  
  
    public Clock(int hour, int min) {  
        this.hour = Math.abs(hour) % 24;  
        this.min = Math.abs(min) % 60;  
    }  
  
    private int hour,min;           // instance variables  
  
    public void reset() { hour = 0; min = 0; }  
  
    public int getHour() { return hour;}  
  
    .....  
}
```

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 `getHowMany()` below, what Clock object is being manipulated?

```
// ClockTest2.java - demonstration of a static field
class ClockTest2 {
    public static void main(String[] args) {
        ...
        System.out.println(Clock.getHowMany());
    }
}
```

Answer: There isn't one. So trying to call `getMin()` from within `getHowMany()` like we did from within `tick()` 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()`.³⁷

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".

immutable string objects versus string variables

A string object when created is immutable
(cannot be changed)

A string variable can be modified to refer to a
different object

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);
    }
}
```

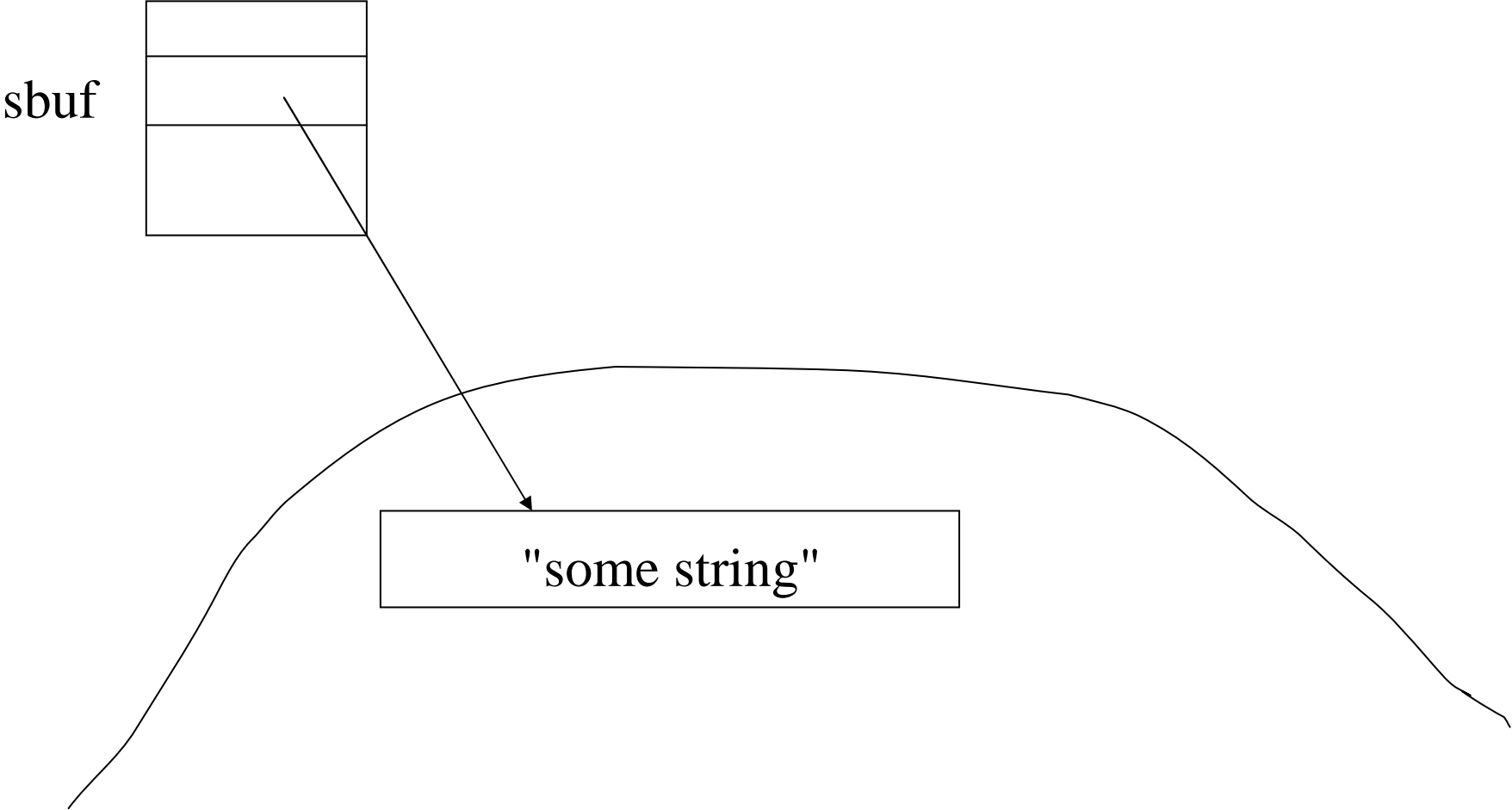
Prints:

```
some mutable string
```

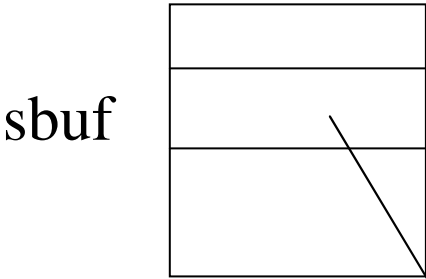
Passing Objects to methods

```
class StringBufferInsert2 {  
  
    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()

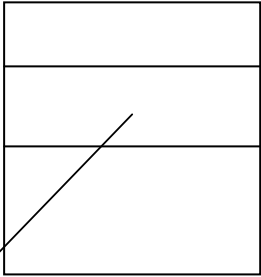


variables in main()



sbuf

variables in insertInMiddle()

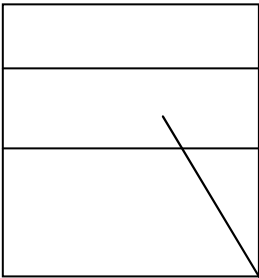


buffer

"some string"

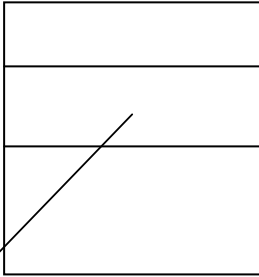
variables in main()

sbuf



variables in insertInMiddle()

buffer



"some mutable string"

Call by Value

- When a reference to an object is passed to a method, then it is possible that the method can modify the fields of the object.
But it still cannot modify a variable that is a reference type variable (a class or array), the variable still points to the same location.
- 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");

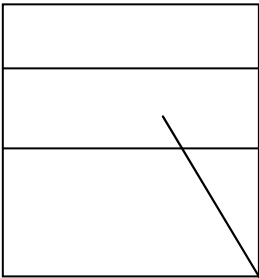
        System.out.println("sbuf is now " + sbuf);
        modify(sbuf);
        System.out.println("sbuf is now " + sbuf);
    }
    static void modify(StringBuffer sb) {
        sb = new StringBuffer("doesn't work");
    }
}
```

Prints:

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

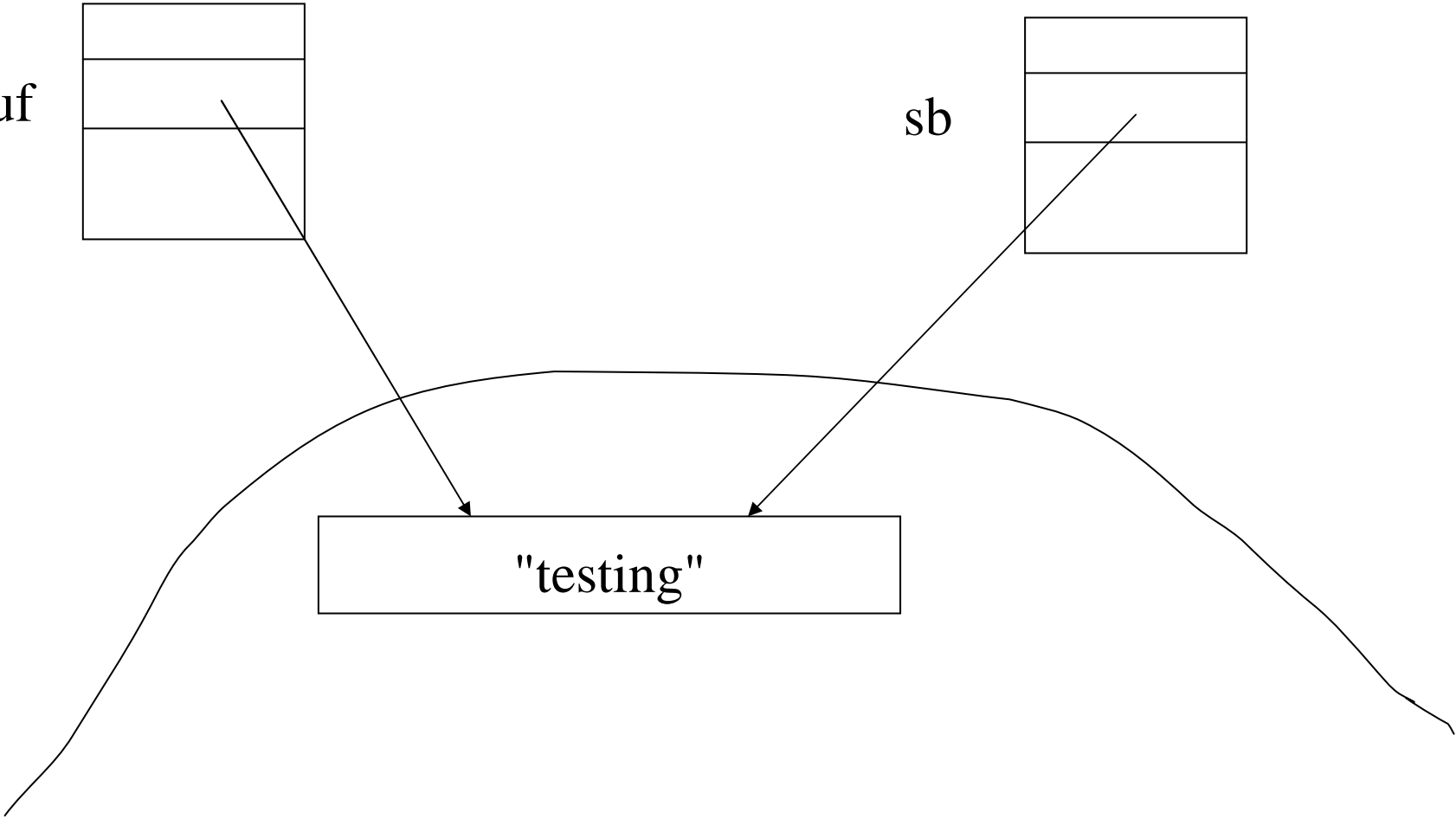
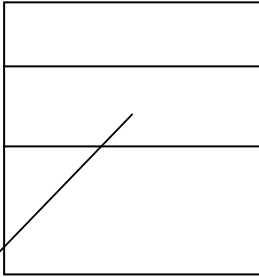
variables in main()

sbuf



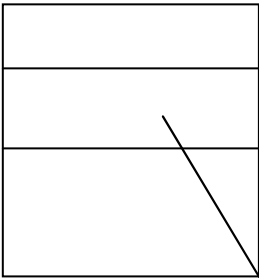
variables in modify()

sb



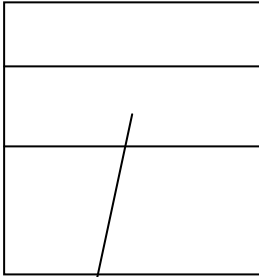
variables in main()

sbuf



variables in modify()

sb



"testing"

"doesn't work"

Symbolic Constants

```
public class Clock {  
  
    public Clock() { }  
  
    public Clock(int hour,int min) {  
        hour = Math.abs(h) % HOUR_MODULUS;  
        min  = Math.abs(m) % MIN_MODULUS;  
    }  
    private int hour,min;  
    void tick(){  
        boolean turn = ((min+1)%MIN_MODULUS ==0);  
        if (turn)  
            {min = 0; hour = (hour+1)%HOUR_MODULUS;}  
        else  
            min = min + 1;  
    }  
  
    private static final int HOUR_MODULUS = 24;  
    private static final int MIN_MODULUS = 60;  
}
```

Symbolic Constants

- The keyword final means that once initialized the value of `HOUR_MODULUS` cannot be changed
- We can now use the constant `HOUR_MODULUS` everywhere where `24` was used before
- Makes the program easier to modify without introducing errors
- Makes the code easier to understand

Calling Methods

example 1 :

```
methodName ( argument-list )
```

Can be used calling a method in the same class

- call a class-method from a class-method

- call a class-method from an instance-method

- call an instance-method form an instance-method

When calling one instance method from another instance method the same implicit object is used.

```
Example: void tick(){  
    boolean turn = (getMin() +1)%60;.....}
```

Calling Methods

Example 2: `c1.tick()`

`c1` is a variable of type `Clock` that references a `Clock` object

The method name is preceded by an expression that evaluates to a value from the class that contains the method definition (often a variable of the appropriate type)

Can be used calling an instance method in another class

Calling Methods

Example 3: `Math.sqrt(x)`

`Math` is a class name

`sqrt(-)` is a static method in the class `Math`

The method name is preceded by the class name
where the static method is defined

Can be used calling a *static* method in another class.

Calling an instance method can be thought of as
sending a message to an object.

Calling a static method is just a function call.

Programming Style

- The style in which programs are written should make the program easier to understand
 - for others
 - for your self at later times
- Consistent indentation
- In class definitions, consistent order e.g. public members before private

Object Oriented Programming and Design

- Data and behavior packed together
 - Abstract Data Types (ADT)
 - User defined ADT's
- Hide internal details of implementation
- Data hiding, restricted access

Abstract Data Types

- Integrates data-values and methods
- Models real world
- Makes access control to data possible
- Facilitates maintenance

- The user does not need to bother with details of implementation