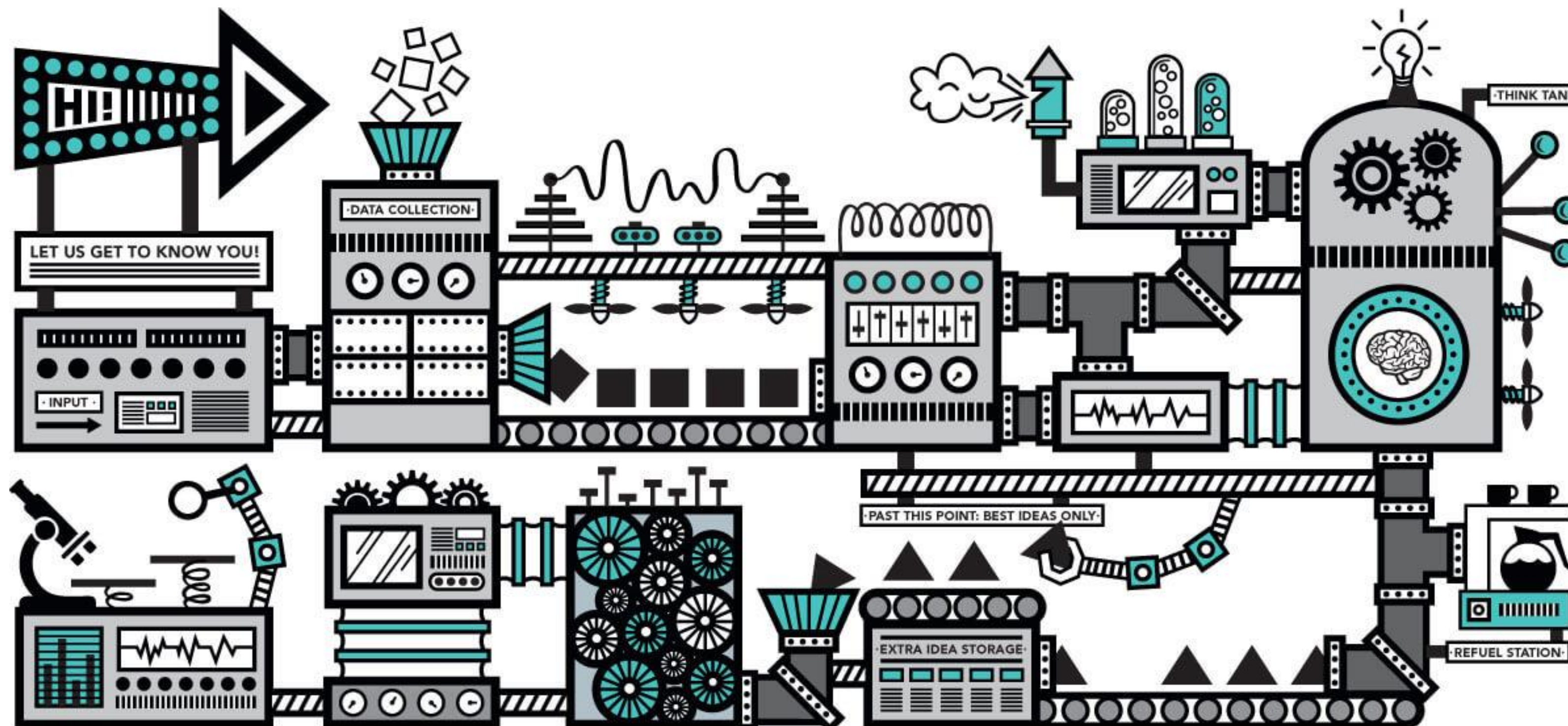
Methods

Sect. 3.3, 8.2

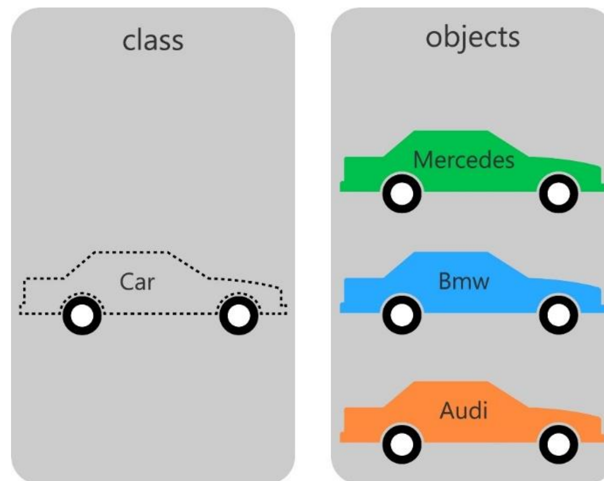
There's a method in my madness.



Example Class: Car

How Cars are Described

- Make
- Model
- Year
- Color
- Owner
- Location
- Mileage



Actions that can be applied to cars

- Create a new car
- Transfer ownership
- Move to a new location
- Repaint
- Delete a car

What's in a class?

- Fields – The data used to describe an object in the class

- Methods – The functions used to manipulate an object in the class

Terminology

- Method: A function in a class definition which implements an action that can be performed on an object
- Receiver: The object from which the method was invoked… most typically, the object which the method acts on. Methods have an implicit parameter of “this” which refers to the receiver object.
- Argument: data passed to a method. Arguments are from the invocation point of view, as compared to parameters.
- Parameter: data passed to a method. Parameters are from the method’s point of view, as compared to arguments

Method Syntax

modifiers returnType name (*parameters*) *throws* {
    body
}

- modifiers : public private static
  - also: protected abstract final synchronize native strictfp
- *returnType* : Any primitive or reference type or void
- *name* : Any valid identifier

Parameters

- Comma separated list of things that look like variable declarations
- Each parameter is a positional place-holders to hold the values that are to be passed a method to allow it to complete its action
- The last parameter in the list may specify a variable length list of values. If so, it must be specified as

  type … *name*

  where *name* is interpreted as an array within the method.

Example Method Headers

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

int factorial(int n) { …

public void println(String x) { …

public GridLayout(int rows, int cols,  int hgap, int vgap) {…

public static int sum(int… numbers)
```

Modifiers

Return Type

name

Parameters

Arguments

- The positional list of values or references passed in to a method (to be kept a a parameter)
- An argument may be an expression that evaluates to a value or a reference

```
System.out.println("Hello");
… factorial(10);
… panel.setLayout(new GridLayout(2, 2, 5, 5));
```

Arguments

static methods behave like C

```
class XmpStat {
    static int add3(int x) { return x+3; }
}
```

```
class Tester {
    public static void main() {
System.out.println("20+3=" + XmpStat.add3(20));
    }
}
```

Instance vs. Class Methods

- If a method is declared as “static”, it is a “class method”
  - Invoked like a function… not as an action on an object
  - Class methods do not have a receiver object
  - Class methods do not have a “this” implicit parameter
  - “main” is a class method, not an instance method
  - Constructors are *invoked* like a class method
- Most methods are “instance” methods – methods which are invoked as actions on specific objects
  - Note: Constructors are more like instance methods, even though they are not invoked as instance methods

Method Invocation

- Most often, a method is invoked by specifying an action to occur to an object

```
DollarsAndCents myMoney = new DollarsAndCents(12,50);
myMoney.upOrDown(1.05);
```

Reference to an object in the DollarsAndCents class

Action: Method from the DollarsAndCents class

Argument to be passed to the upOrDown method

Rectangle Class Example

- Rectangles expressed as (left x, top y, width, height)

```
Rectangle rect1 = new Rectangle(70,90,100,150);
```

- "translate" means move a shape without changing it

```
rect1.translate(100, 80);
```

moves rect1 to (170, 170, 100, 150)

Rectangle example continued

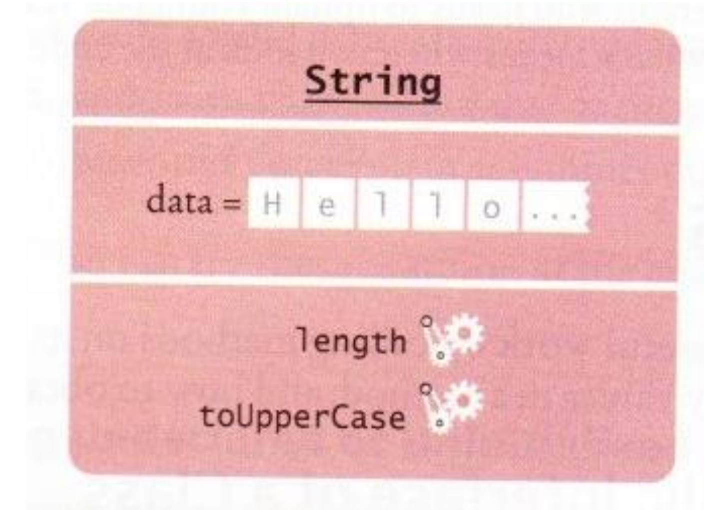
```
public static void main(String[ ] args) {
	Rectangle rect1 = new Rectangle(70,90,100,150);
	Rectangle rect2 = new Rectangle(70,180,200,50);
	rect1.translate(100, 80);
}
```

- There is only <u>one copy</u> of the code for the method *translate*, not one copy for each object
- How is that *translate* moves rect1 and not rect2 ?
- <u>The code of *translate* has to receive a reference to the object it will read information from or modify</u>

See class web page example FirstGUI

No Code in Objects

- In section 2.3 of the text, there are diagrams like:
- Despite these pictures, there is no actual CODE stored in the object!
- The object ONLY contains the values of the fields for that object
- The code is stored in memory only once for the entire class



The Implicit Receiver Parameter

- The object that "receives" the action (the object which the action is performed on) is called the "receiver" object.
- The receiver object is *implicitly* placed in the parameter list of the method, as if (but not actually) you had specified:

```
class Rectangle {
        void translate(int dx, int dy) { … }
```

as if…

```
        void translate(Rectangle this, int dx, int dy) { }
}
```

Constructor "Methods"

- Must be named the same as the class name!
- Implicit "this" parameter is the object being constructed
  - A new object of this class with all non-final fields initialized to zero/null
- Implicit return parameter of "this"
  - **No return type specification in constructor methods**

Implicit Parameter in the Stack

- Parameters are passed using an "activation record" on the program stack
- Local variable values are also kept in the "activation record"
- Using our rectangle example:

```
public static void main(String[ ] args) {
        Rectangle rect1 = new Rectangle(70,90,100,150);
        Rectangle rect2 = new Rectangle(70,180,200,50);
        rect1.translate(100, 80);
}
```

Why class methods?

- Enable “main” before any objects are available
- Enable functions – methods which don’t act on objects
  - Alternative… meaningless empty object which requires creation
- Enable “factory methods”
  - methods which create new objects, but are not constructors

Rectangle Class

```
public class Rectangle {
	int leftx; int topy;
	int w; int h;
	public Rectangle(int lx,  int ty, int wid, int hgt) {
		this.leftx=lx; this.topy=ty;
		this.w=wid; this.h=hgt;
	}
	void translate(int dx,int dy) {
		this.leftx+=dx; this.topy+=dy;
	}
}
```

Rectangle Class

```
public class Rectangle {
	int leftx; int topy;
	int w; int h;
	public Rectangle(int lx, int ty, int wid, int hgt) {
		this.leftx=lx; this.topy=ty;
		this.w=wid; this.h=hgt;
	}
	void translate(int dx,int dy) {
		this.leftx+=dx; this.topy+=dy;
	}
}
```

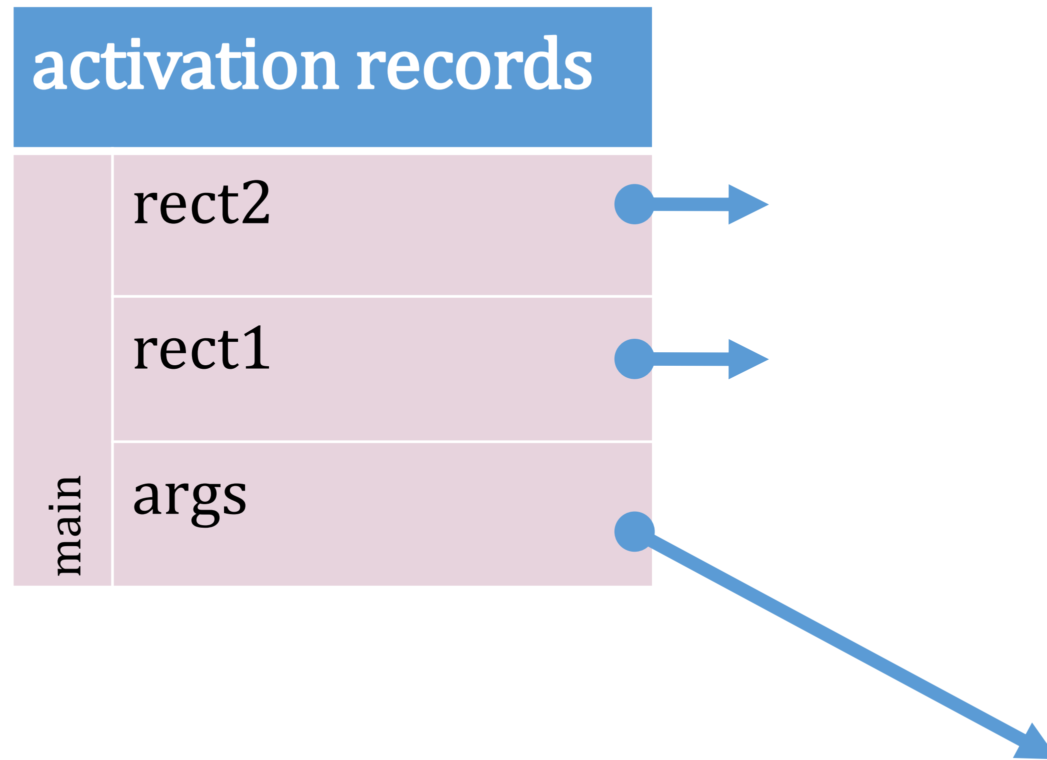
If field names are different than variable names, Java assumes "this."

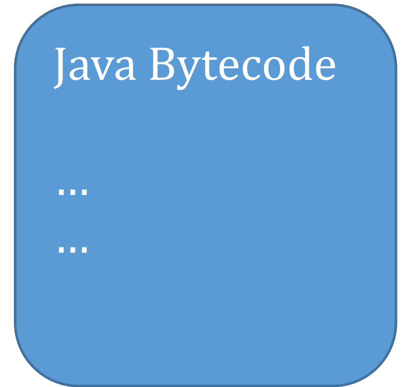
Rectangle Class

```
public class Rectangle {
	int leftx; int topy;
	int w; int h;
	public Rectangle(int lx, int ty, int wid, int hgt) {
		leftx=lx; topy=ty;
		w=wid; h=hgt;
	}
	void translate(int dx,int dy) {
		leftx+=dx; topy+=dy;
	}
}
```

If field names are different than variable names, Java assumes "this."

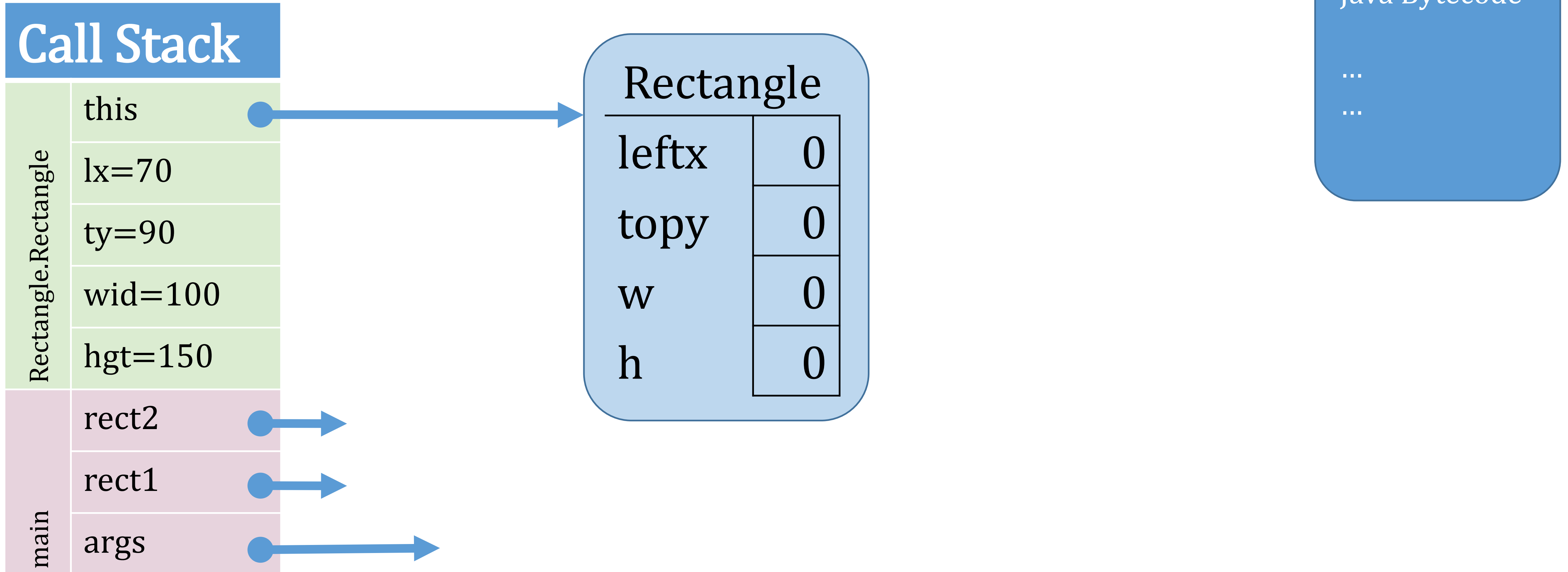
Memory (at start of main)





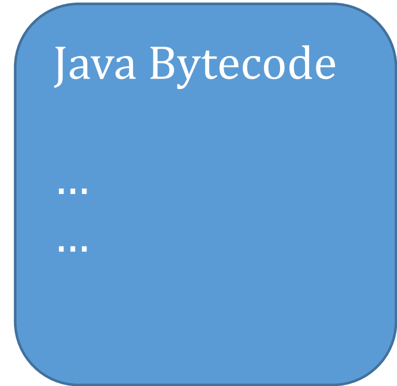
Next, “new” will create a rectangle object, and invoke the Rectangle creator…

Memory (at start of rect1 constructor)



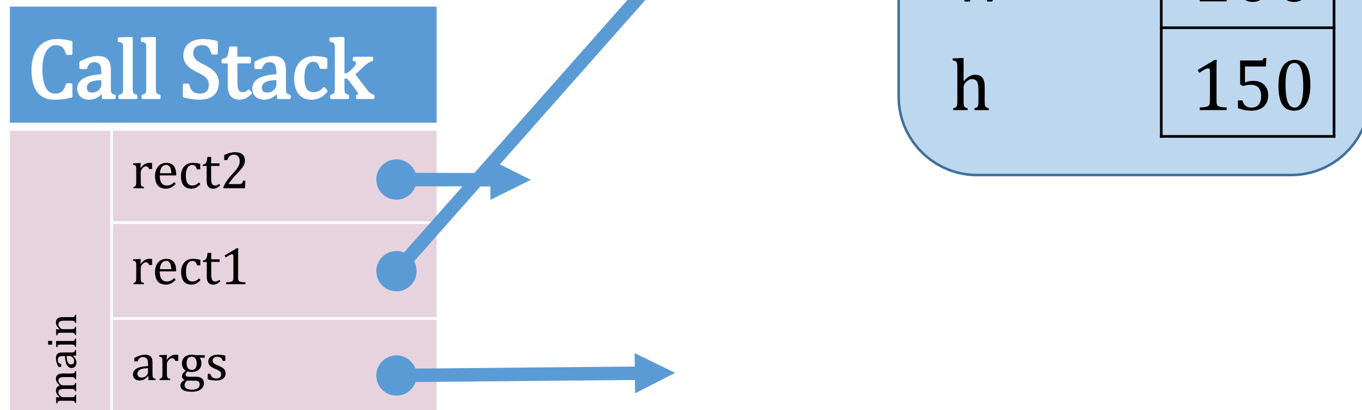
Next, the Rectangle creator returns…

Memory (after rect1 constructor)



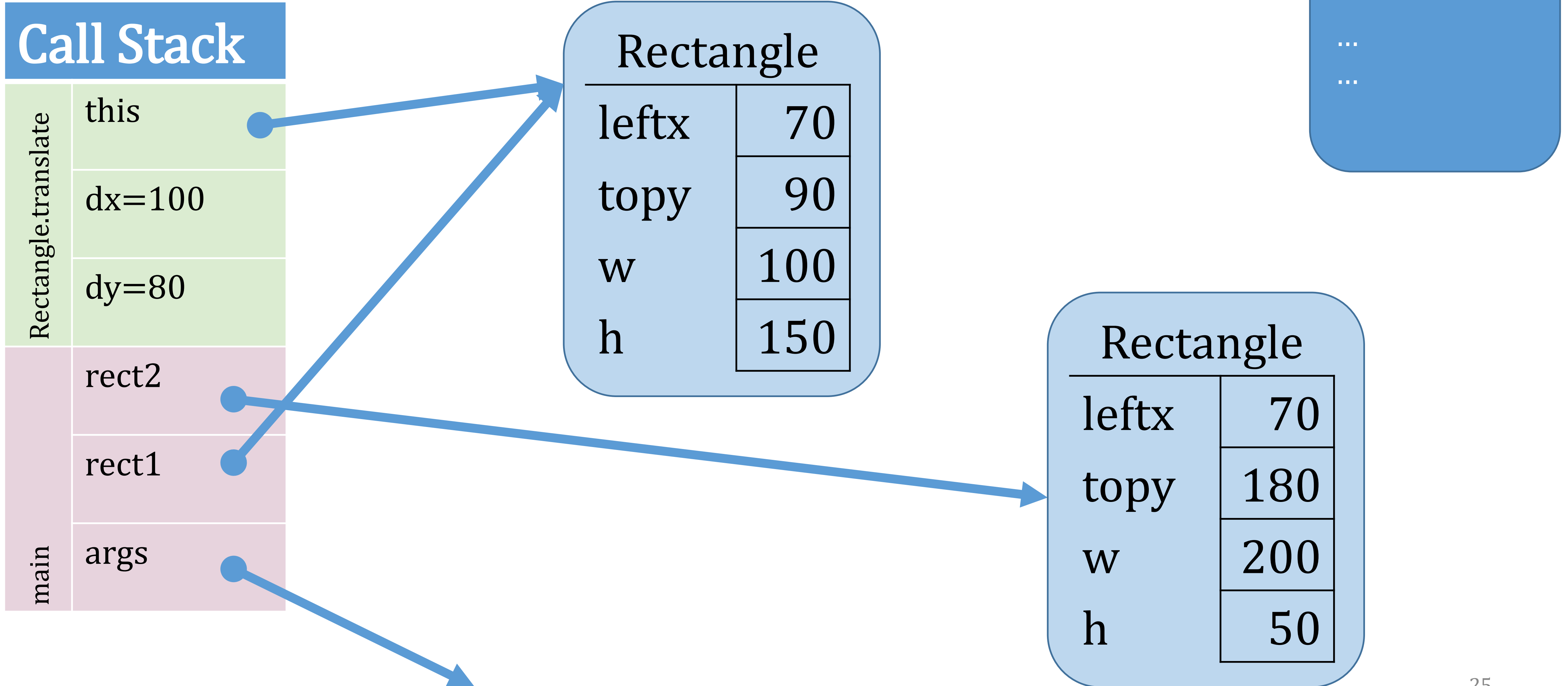
Rectangle

leftx	70
topy	90
w	100
h	150



Next, rect2 is created similarly, and then the translate method is invoked

Memory (at start of translate method)



Accessor vs. Mutator Methods

Sect. 2.5

- If a method changes the receiver object field values, it is called a ***mutator*** method.
  - For example… the "translate" method in Rectangle changes the value of leftx and topy… it is a mutator method
- If a method does not change the receiver object field values, it is called an ***accessor*** method.
  - For example… the "getWidth" method in Rectangle does not change any of the fields in rectangle… it is an accessor method.
- If all methods are accessor methods, the Class is called an ***immutable*** class
  - For example… the String class is immutable.

Example: Immutable String Class

- Strings are arrays of characters, so index starts at 0

```
int z = "CS 140".length( ); //gives z the value 6
String str = "CS 140";
String s1 = str.substring(0, 2);  // s1 is “CS”
char ch = str.charAt(3); // gives ch the value '1‘
```

Substring

- Two methods can have the same name if they take different parameters

String substring(int beginIndex) : Returns a new string that is a substring of "this". The substring begins with the character at the specified index and extends to the end of "this".

String substring(int beginIndex, int endIndex) : Returns a new string that is a substring of "this". The substring begins at beginIndex and extends to the character at (endIndex – 1). Thus the length of the substring is endIndex-beginIndex.

- Note that since String is immutable, "this" is not modified… a new string is returned

Replace

- String replace(char oldChar, char newChar) : Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
- String replace(CharSequence target, CharSequence replacement) : Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

  Note a CharSequence can be a String
- Note – there is also replaceAll, but it uses regular expressions

Changing Case

- String toLowerCase() : Returns a new String that is “this” with all of the characters in “this” converted to lower case using the rules of the default locale.
- String toUpperCase() : Returns a new String that is “this” with all of the characters in “this” converted to upper case using the rules of the default locale.

Equality

- boolean equals(Object anObject) : Compares "this" to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
- boolean equalsIgnoreCase(String anotherString) : Compares "this" to another String, ignoring case considerations.

equals vs. ==

```
String str1="CS 140";
String str2="CS 140";
String str3="CS 220";
String str4 = str1;
```

str1.equals(str2) ?

str1.equals(str3)?

str1==str2?

str1==str4?

strx==stry &&
    !strx.equals(stry)?

Searching in Strings

- int indexOf(int ch): returns the index within "this" of the first occurrence of the specified character.
- int indexOf(int ch, int fromIndex): returns the index within "this" of the first occurrence of the specified character, starting the search at the specified index.
- int indexOf(String str): returns the index within "this" of the first occurrence of the specified substring.
- int indexOf(String str, int fromIndex): returns the index within "this" of the first occurrence of the specified substring, starting at the specified index.