Classes, Objects, and Interfaces

CS 5010 Program Design Paradigms

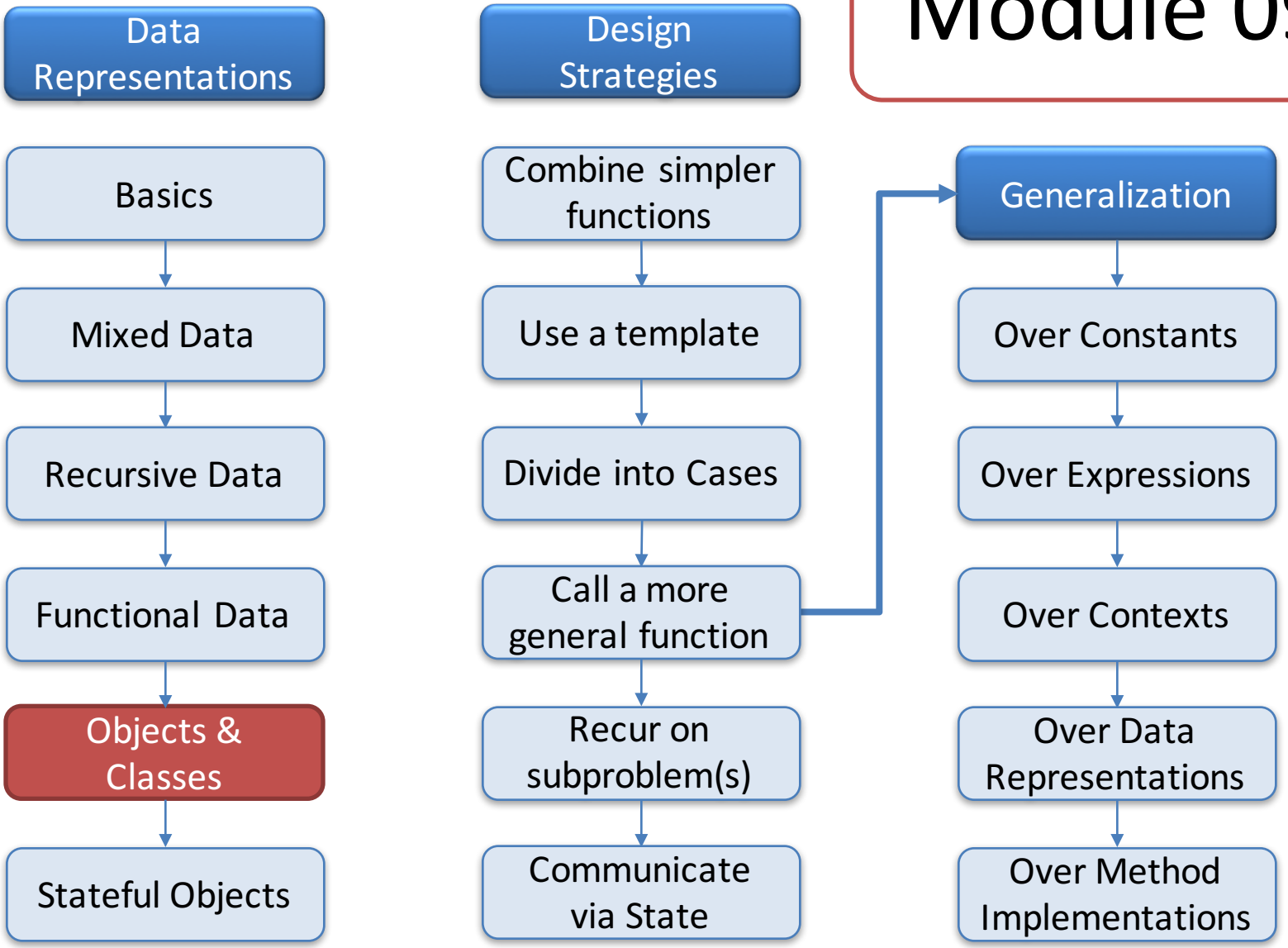
"Bootcamp"

Lesson 9.1

Module Introduction

- In this module, we will see how classes, objects, and interfaces fit into our account of information analysis and data design
- We'll see how the functional and the object-oriented models are related
- We'll learn how to apply the design recipe in an object-oriented setting.

Module 09


Data Representations
Basics
Mixed Data
Recursive Data
Functional Data
Objects & Classes
Stateful Objects
Design Strategies
Combine simpler functions
Use a template
Divide into Cases
Call a more general function
Recur on subproblem(s)
Communicate via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

Goals of this lesson

- Learn the basics about classes, objects, fields, and methods.
- Learn how these ideas are expressed in the Racket object system
- We assume that you already know a little about object-oriented programming.

Slogans for this lesson

- Classes are like define-structs, but with methods (functions) as well as fields.
- Every object knows its class.
- Invoke a method of an object by sending it a message.
- The interface of an object is the set of messages to which it responds.
- Interfaces are data types

Classes and Objects

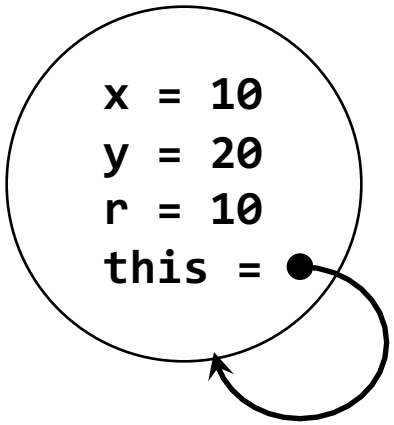
- A class is like a **define-struct**.
- It specifies the names of the fields of its objects.
- It also contains some *methods*. Each method has a name and a definition.
- To create an object of class **C**, we say

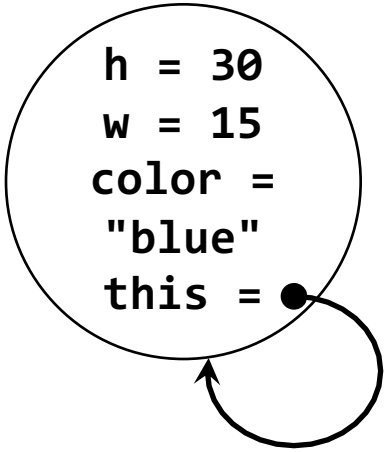
`(new C)`

You say more than this, but this is good enough right now.

What is an object?

- An object is another way of representing compound data, like a struct.
- Like a struct, it has *fields*.
- It has one built-in field, called **this**, which always refers to this object
- Here are pictures of two simple objects:

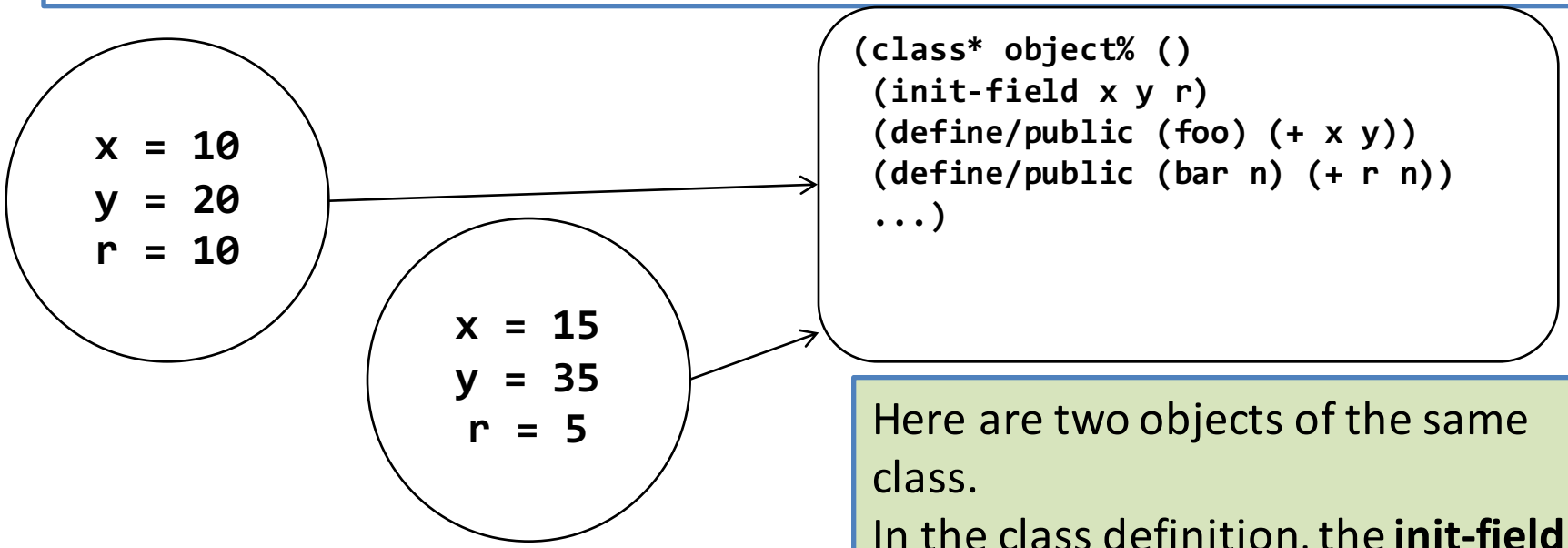




We assume that you've seen some kind of object-oriented programming before, so we're just reviewing vocabulary here.

If you've really never used OOP before, go do some outside reading before continuing.

Every object knows its class (1)



These objects also have a **this** field, but we don't show it unless we need to.

Here are two objects of the same class.
In the class definition, the **init-field** declaration specifies that each object of this class has 3 fields, named **x**, **y**, and **r**.
The class definition also defines two methods, named **foo** and **bar**, that are applicable to any object of this class.

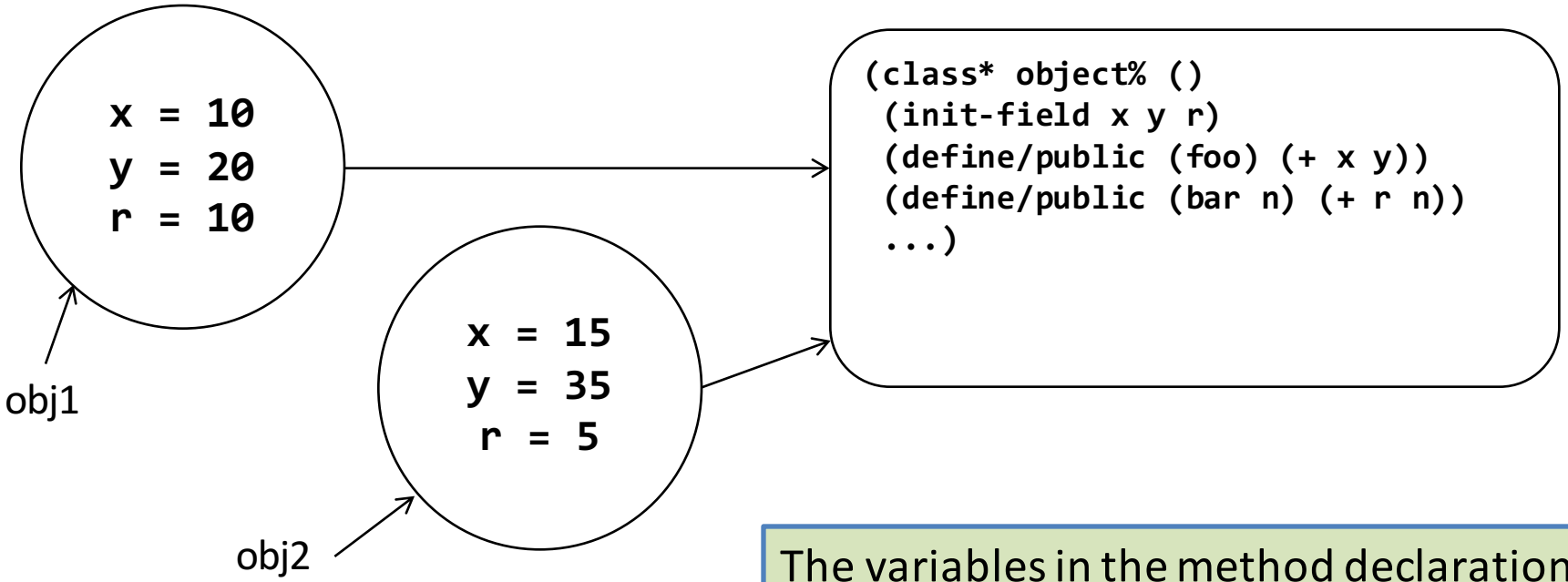
How do you compute with an object?

- To invoke a method of an object, we *send the object a message.*
- For example, to invoke the **area** method of an object **obj1**, we write

```
(send obj1 area)
```

- If **obj1** is an object of class **C**, this invokes the area method in class **C**.

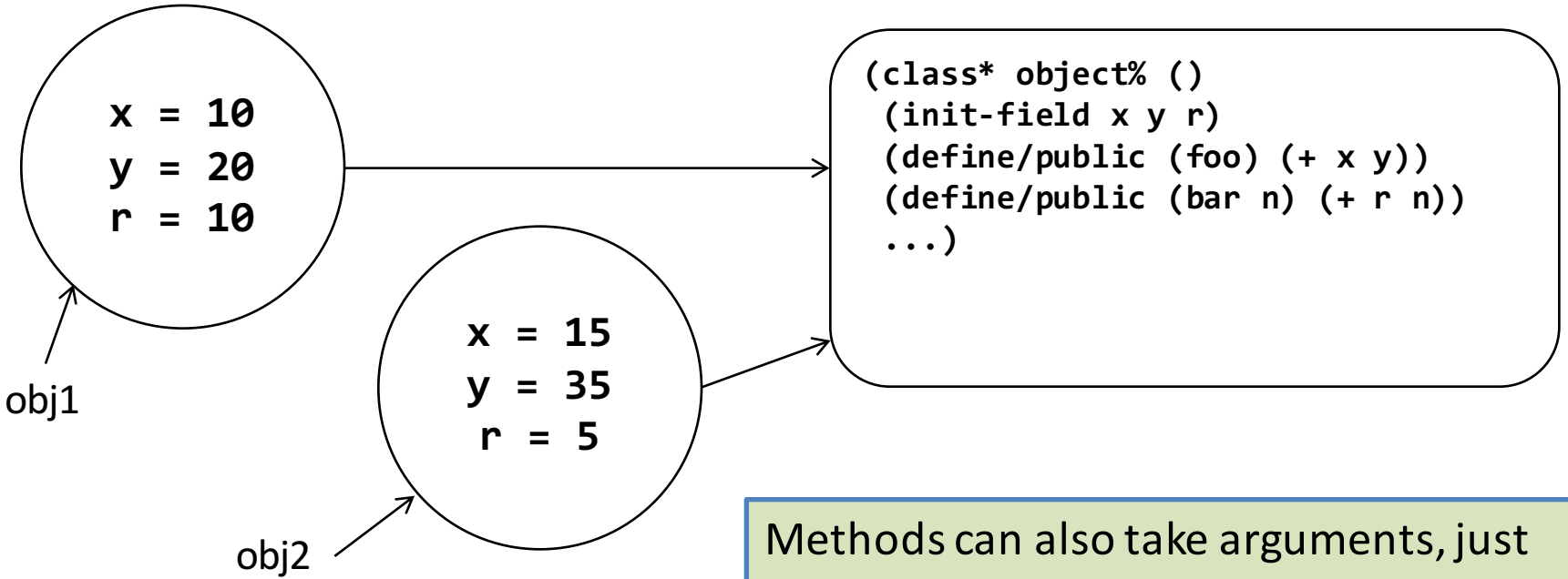
Every object knows its class (2)



The variables in the method declarations refer to the fields in the object. So:

(send obj1 foo) returns 30
(send obj2 foo) returns 50

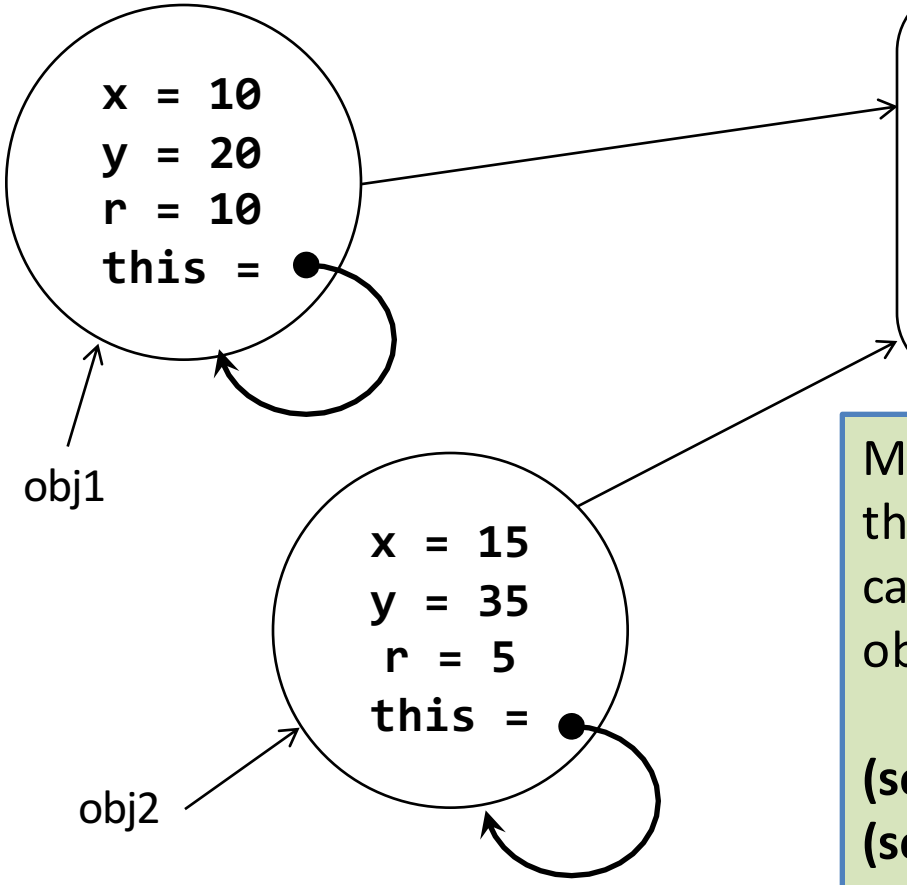
Every object knows its class (3)



Methods can also take arguments, just like functions. So

(send obj1 bar 8) returns 18
(send obj2 bar 8) returns 13

Every object knows its class (4)

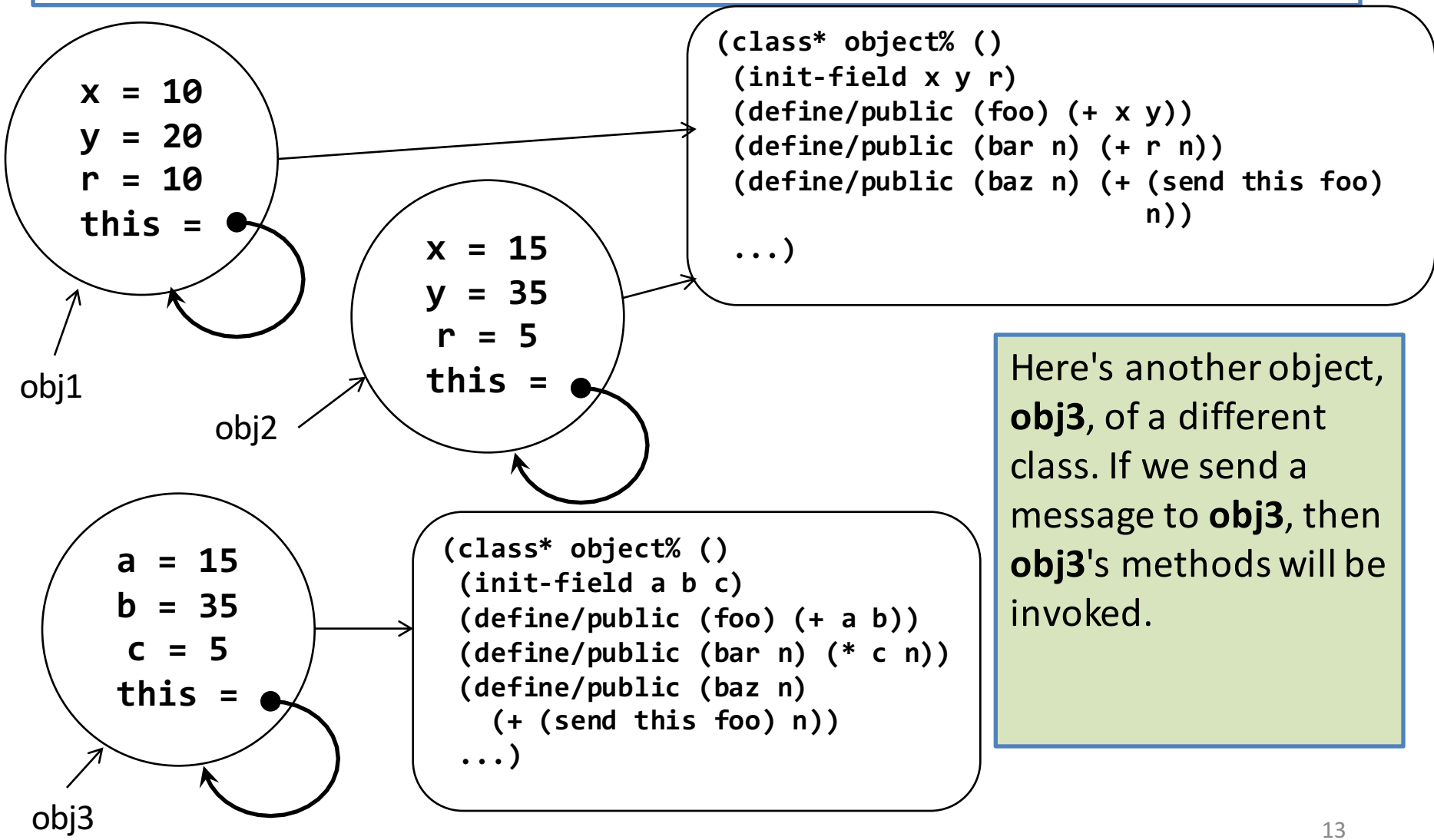



```
(class* object% ()
 (init-field x y r)
 (define/public (foo) (+ x y))
 (define/public (bar n) (+ r n))
 (define/public (baz n) (+ (send this foo)
                           n))
 ...)
```

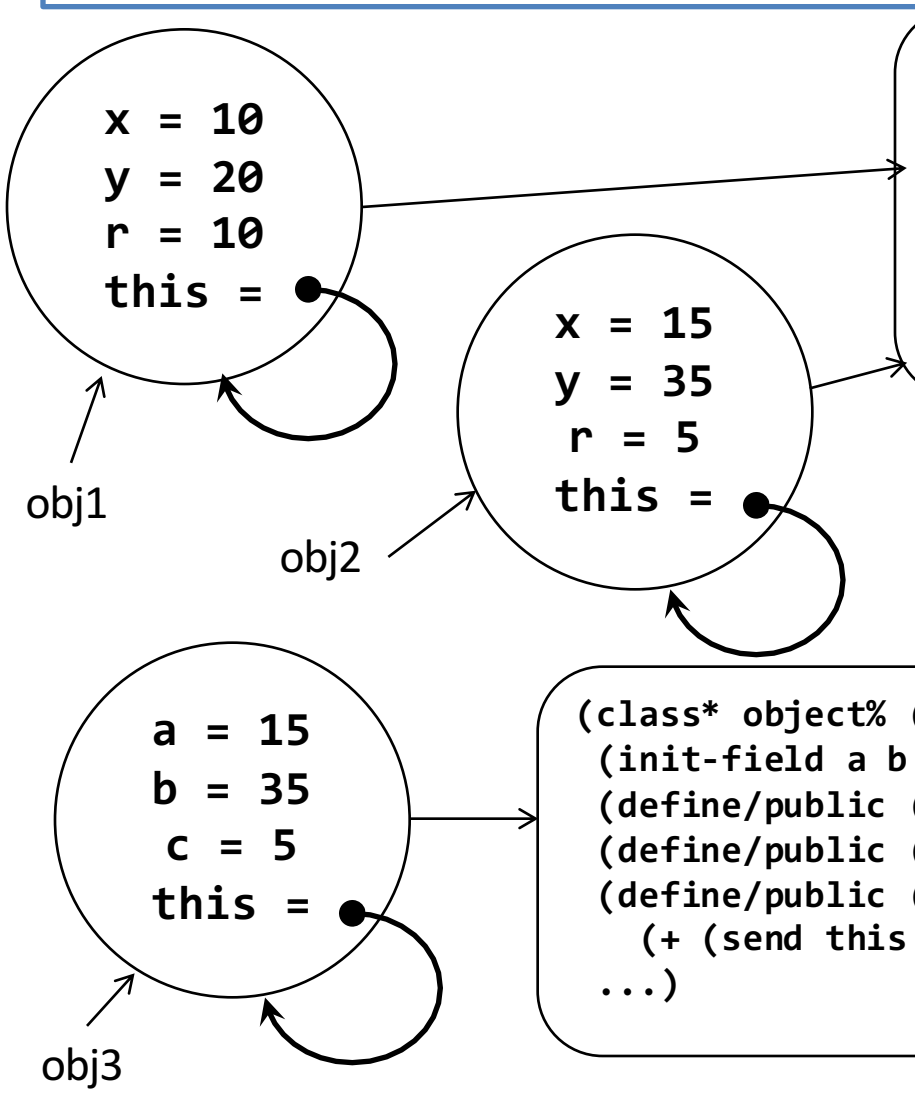
Methods are just Racket functions, so they can do anything a Racket function can do, including send messages to objects.

(send obj1 baz 20) returns (+ 30 20) = 50
(send obj2 baz 20) returns (+ 50 20) = 70

Every object knows its class (5)



Every object knows its class (6)



So
(send obj2 bar 8)
= **(+ 5 8)**
= **13**
(send obj3 bar 8)
= **(* 5 8)**
= **40**

The important thing about an object is what methods it responds to

- So if I wrote

```
(define (foo1 x) (send x bar 8))
```

- I could call **foo1** on **obj1**, **obj2**, or **obj3**, because all of them respond to the **bar** message with an integer argument.
- The contract for **foo1** should specify that its argument will accept a **bar** message with an integer argument.

Interfaces are data types

- The set of messages to which an object responds (along with their contracts) is called its *interface*.
- So the contract for **foo1** (or any other function that takes an object as an argument) should be expressed in terms of *interfaces*.
- So interfaces play the role of data types in the OOP setting.

Using The Racket Class System

- We will use full Racket (yay!)
- Write

```
#lang racket
```

  at the beginning of each file
- And set the Language level to "Determine Language from Source"

Interface definition

```
#lang racket

(require "extras.rkt")
(require rackunit)

;; examples from Lesson 9.1

(define Interface1<%>
  (interface ()
    foo  ; -> Int
    bar  ; Int -> Int
    baz  ; Int -> Int
    ))
```

In Racket, names of interfaces end with **<%>** (by convention)

Ignore that **()** for now

We write down each method name, with its contract as a comment. We can write them in any order

foo is a function of no arguments (legal in #lang racket)

A Class Definition (1)

```
(define Class1%
  (class* object% (Interface1<%>)

    (init-field x y r)  ;;  x,y,r : Int

    (super-new)   ;; required magic

    ;; foo : -> Int
    (define/public (foo) (+ x y))

    ;; bar : Int -> Int
    (define/public (bar n) (+ r n))

    ;; baz : Int -> Int
    (define/public (baz n)
      (+ (send this foo) n))

    ))
```

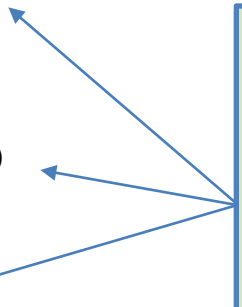
This means that this class is supposed to implement **Interface1<%>**. If we leave off one of the methods, we'll get an error message.

x, **y**, and **r** are the field names. We've put in their contracts as a comment. In a real example, you'd put an interpretation for each field, just as you do the fields of a **struct**.

object% and **(super-new)** are required magic. We'll learn about them in a later module

A Class Definition (2)

```
(define Class1%
  (class* object% (Interface1<%>)

    (init-field x y r)  ;;  x,y,r : Int

    (super-new)   ;; required magic

    ;; foo : -> Int
    (define/public (foo) (+ x y))

    ;; bar : Int -> Int
    (define/public (bar n) (+ r n))

    ;; baz : Int -> Int
    (define/public (baz n)
      (+ (send this foo) n))

    ))
```



We use **define/public** to define methods. Here we've written the contract for each method; later we'll see what the Design Recipe deliverables for methods are.

Another class definition

```
(define Class2%
  (class* object% (Interface1<%>)

    (init-field a b c)  ; a, b, c : Int

    (super-new)

    ;; foo : -> Int
    (define/public (foo) (+ a b))

    ;; bar : Int -> Int
    (define/public (bar n) (* c n))

    ;; baz : Int -> Int
    (define/public (baz n)
      (+ (send this foo) n))

    ))
```

Here's the definition of Class2% . Observe that it has different field names, but the same method names. The method definitions refer to the new field names.

Yet another class definition

```
(define Class2%
  (class* object% (Interface1<%>)

    (init-field a b c)  ; a, b, c : Int

    (super-new)

    ;; foo : -> Int
    (define/public (foo) (+ a b))

    ;; bar : Int -> Int
    (define/public (bar n) (* c n))

    ;; baz : Int -> Int
    (define/public (baz n)
      (+ (send this foo) n))

    ))
```

Objects of Class2% and Class2a% are built the same way and give the same answer for every method call. Any procedure that works with one will work the same way with the other.

```
(define Class2a%
  (class* object% (Interface1<%>)

    (init-field a b c)  ; a, b, c : Int

    ; add a new field, initialized to (- a)
    (field [a1 (- a)])

    (super-new)

    ;; foo : -> Int
    (define/public (foo) (- b a1))

    ;; bar : Int -> Int
    (define/public (bar n) (* c n))

    ;; baz : Int -> Int
    (define/public (baz n)
      (+ (send this foo) n))

    ))
```

This is another reason we write contracts in terms of interfaces, not classes.

Creating objects and testing

```
(define obj1 (new Class1% [x 10][y 20][r 10]))
(define obj2 (new Class1% [y 35][x 15][r 5]))
(define obj3 (new Class2% [a 15][b 35][c 5]))

(begin-for-test

  (check-equal? (send obj1 foo) 30)
  (check-equal? (send obj2 foo) 50)

  (check-equal? (send obj1 bar 8) 18)
  (check-equal? (send obj2 bar 8) 13)

  (check-equal? (send obj1 baz 20) 50)
  (check-equal? (send obj2 baz 20) 70)

  (check-equal? (send obj2 bar 8) 13)
  (check-equal? (send obj3 bar 8) 40)

)
```

Here is the syntax for creating objects. The fields can be listed in any order.

And here we send the objects some messages and check that the results are as we predicted on the slides above.

Lesson Summary

- In this lesson we've learned:
  - Classes are like define-structs, but with methods (functions) as well as fields.
  - Every object knows its class.
  - Invoke a method of an object by sending it a message.
  - The interface of an object is the set of messages to which it responds.
  - Interfaces are data types.
- We've seen how to define classes, objects, and interfaces in the Racket object system.

Next Steps

- Study the file 09-1-basics.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson