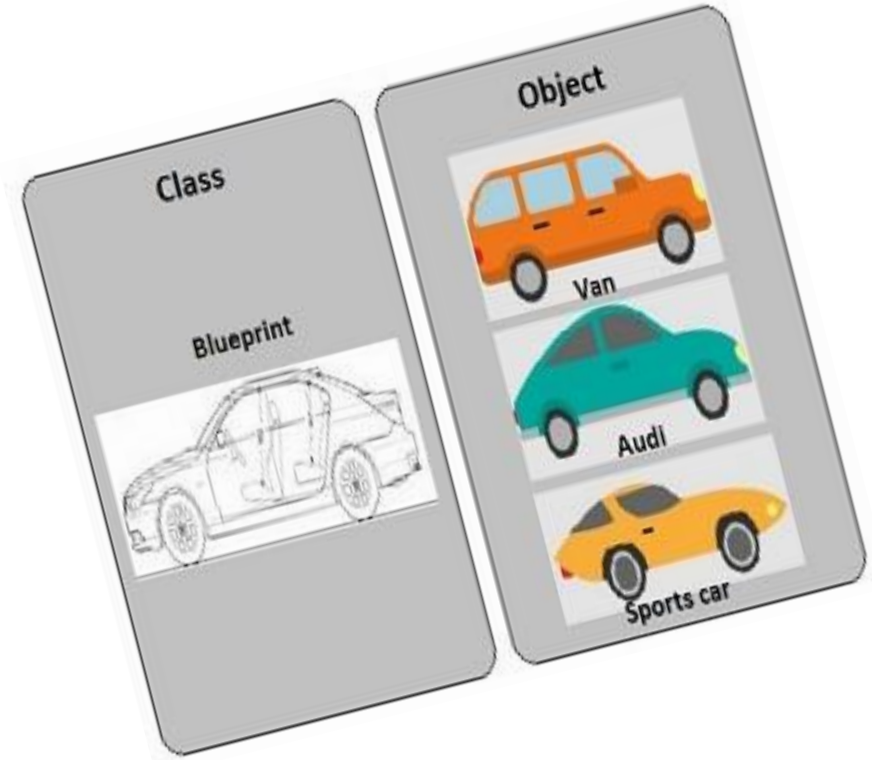


Writing our own classes to build *custom* data types

Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Why the need for classes?

- Let's say I want to keep track of data on my students:
  - **Name {first, last, middle}**
  - **Date of birth {month, day, year}**
  - **Student ID**

- Let's assume all data is stored as strings.

- How can I maintain all this data for each student?

- We wouldn't create a variable for each piece of data multiplied by all students?

- We could use arrays. So let's start there!

Why the need for classes?

```
String [] studentName = new String[N];
String [] dob = new String[N];
String [] sid = new String[N];
```

strudentName

chris	**pam**	**nick**	**john**	**molly**					

dob

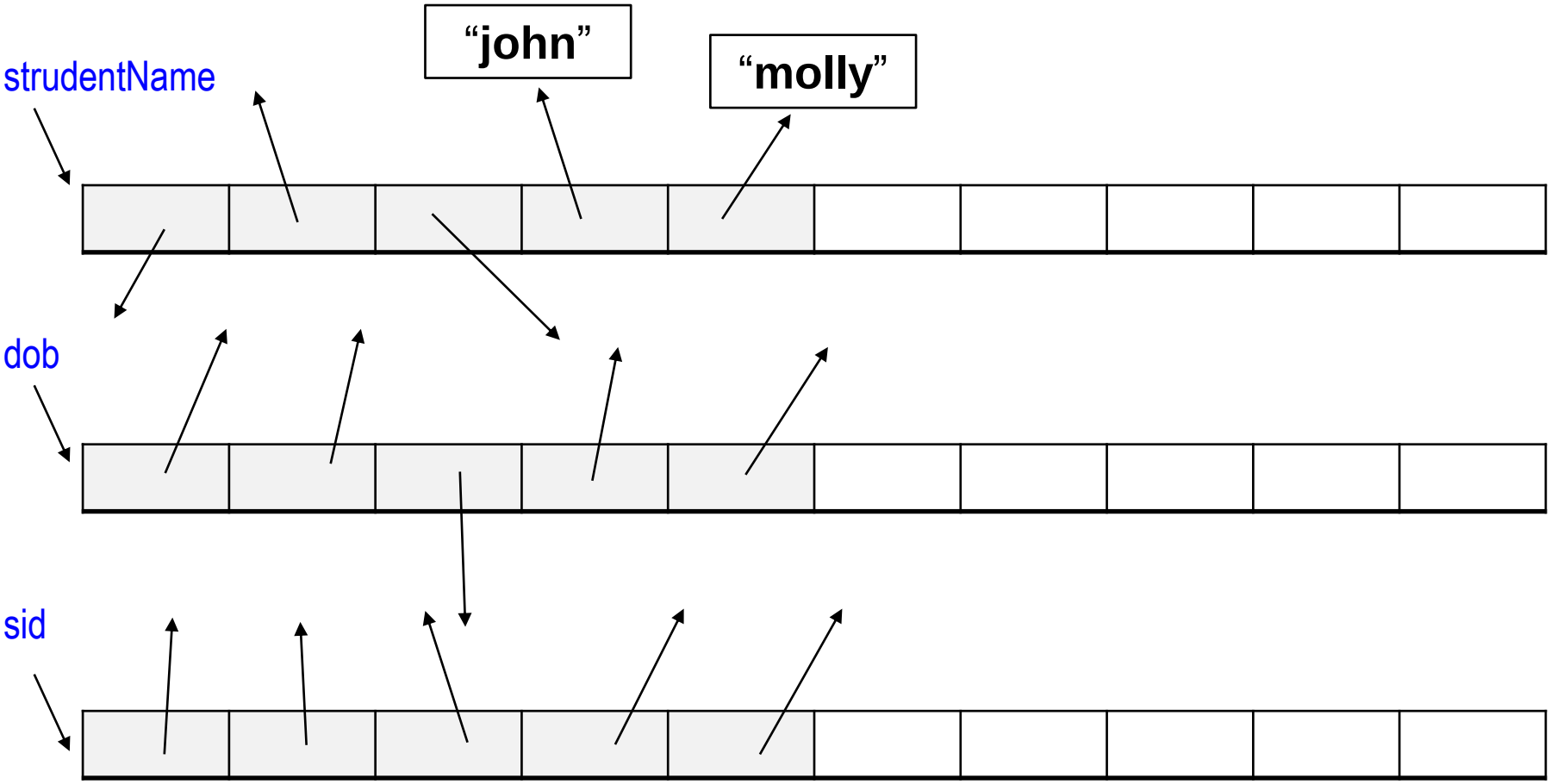
2/21	**3/14**	**12/9**	**5/3**	**8/9**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for classes?

```
String [] studentName = new String[N];
String [] dob = new String[N];
String [] sid = new String[N];
```



Why the need for classes?

```
String [] studentName = new String[N];
int [] month = new int[N];
int [] day = new int[N];
int [] year = new int[N];
```

… three *integer* arrays to properly represent the date of birth!

strudentName

chris	**pam**	**nick**	**john**	**molly**					

month

2	**3**	**12**	**5**	**8**					

day

21	**14**	**9**	**3**	**9**					

year

1989	**2001**	**1996**	**2012**	**1999**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for cla

```
String [] studentName = ne
String [] dob = new String
String [] sid = new String
```

Recall the physical memory layout of an array is contiguous.

strudentName

chris	**pam**	**nick**	**john**	**molly**					

dob

2/21	**3/14**	**12/9**	**5/3**	**8/9**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for classes?

```
String [] studentName = new String
String [] dob = new String[N];
String [] sid = new String[N];
```

We would usually be interested in all the data for a specific student

strudentName

chris	**pam**	**nick**	**john**	**molly**					

dob

2/21	**3/14**	**12/9**	**5/3**	**8/9**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for classes?

```
String [] studentName = new String
String [] dob = new String[N];
String [] sid = new String[N];
```

Therefore we rely on the index:
studentName[**4**]
dob[**4**]
sid[**4**]

strudentName

chris	**pam**	**nick**	**john**	**molly**					

dob

2/21	**3/14**	**12/9**	**5/3**	**8/9**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for cla

```
String [] studentName = ne
String [] dob = new String
String [] sid = new String
```

…shift down to open the spot to add the new student…

strudentName

chris	**pam**	**nick**	**john**		**molly**				

dob

2/21	**3/14**	**12/9**	**5/3**	**8/9**					

sid

a00	**a11**	**a22**	**a33**	**a44**					

Why the need for cla

```
String [] studentName = ne
String [] dob = new String
String [] sid = new String
```

and again…

strudentName

dob

sid

chris	**pam**	**nick**	**john**		**molly**				
2/21	**3/14**	**12/9**	**5/3**		**8/9**				
a00	**a11**	**a22**	**a33**	**a44**					

Why the need for cla

```
String [] studentName = ne
String [] dob = new String
String [] sid = new String
```

and again…

strudentName

dob

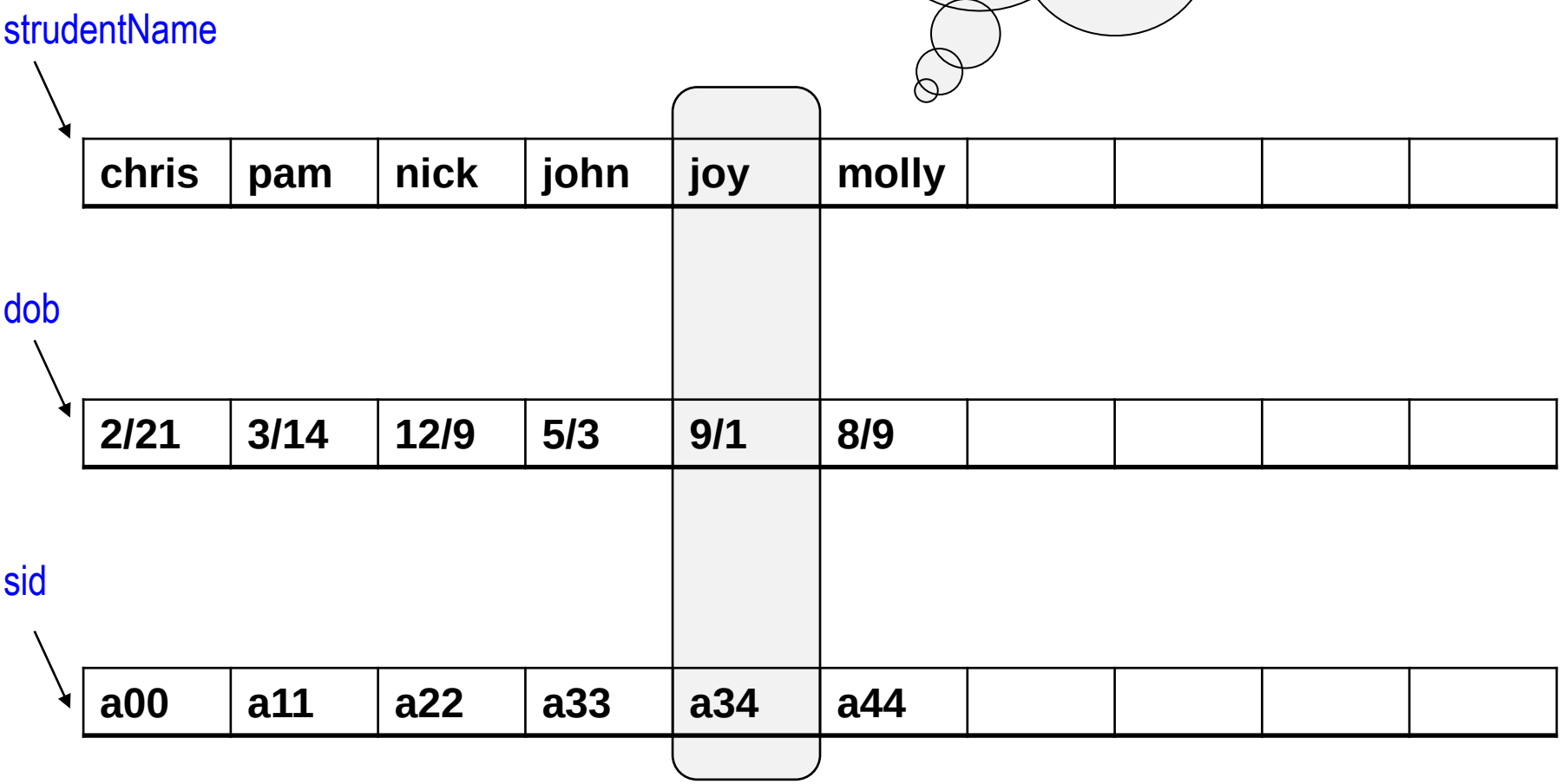
sid

chris	**pam**	**nick**	**john**		**molly**				
2/21	**3/14**	**12/9**	**5/3**		**8/9**				
a00	**a11**	**a22**	**a33**		**a44**				

Why the need for cla

```
String [] studentName = ne
String [] dob = new String
String [] sid = new String
```

Consider the *inefficiency* of the memory use in this scenario?



Why the need for class ?

```
String [] studentName = ne
String [] dob = new String[
String [] sid = new String[
```

The problem is that our ***logical*** view does not match the physical memory *layout* of our data!

strudentName

chris	**pam**	**nick**	**john**	**joy**	**molly**				

dob

2/21	**3/14**	**12/9**	**5/3**	**9/1**	**8/9**				

sid

a00	**a11**	**a22**	**a33**	**a34**	**a44**				

Why the need for class ?

```
String [] studentName = ne
String [] dob = new String[
String [] sid = new String[
```

We need a physical representation that matches our logical view!

strudentName

chris	**pam**	**nick**	**john**	**joy**	**molly**				

dob

2/21	**3/14**	**12/9**	**5/3**	**9/1**	**8/9**				

sid

a00	**a11**	**a22**	**a33**	**a34**	**a44**				

Why the need for class ?

```
String [] studentName = ne
String [] dob = new String[
String [] sid = new String[
```

This is what *class definitions* allow us to do!

strudentName

chris	**pam**	**nick**	**john**	**joy**	**molly**				

dob

2/21	**3/14**	**12/9**	**5/3**	**9/1**	**8/9**				

sid

a00	**a11**	**a22**	**a33**	**a34**	**a44**				

Why the need for classes?

```
class Student {
   String name;
   String dob;
   String sid;
}
```

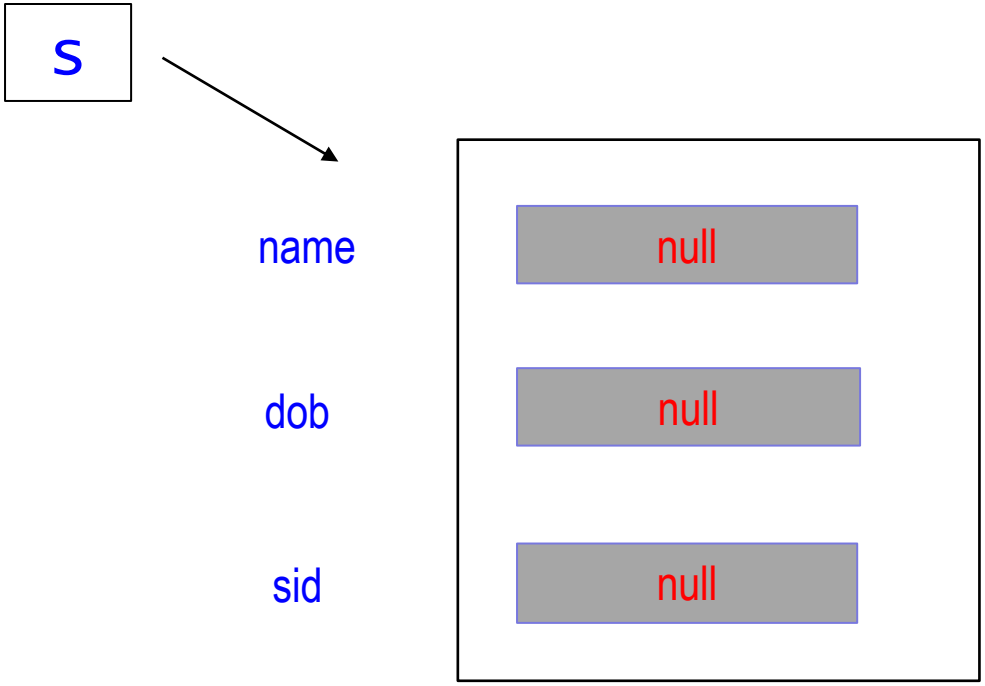
Student is a custom datatype that we created and can use to create an ***instance*** of the Student class: the physical object created from this class definition.

Why the need for classes?

```
class Student {
   String name;
   String dob;
   String sid;
}
```

Create an instance of the Student class:

```
Student s = new Student();
```

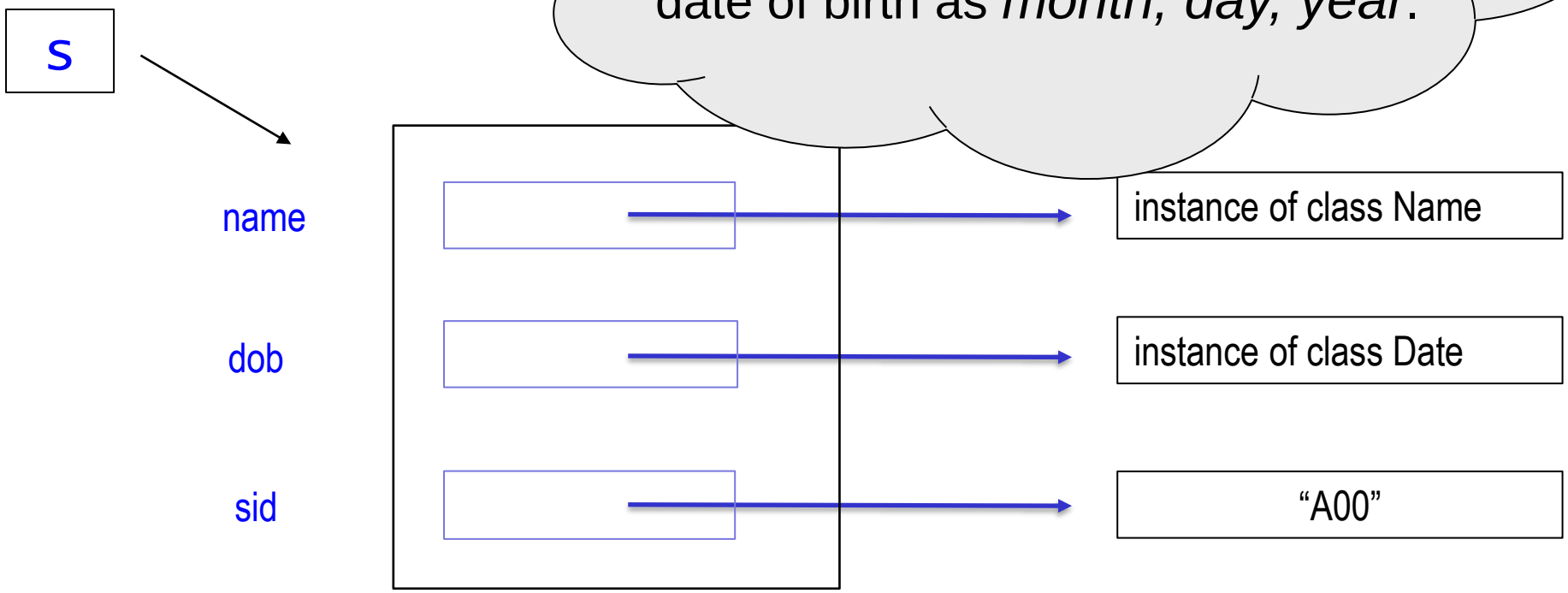


Why the need for classes?

```
class Student {
    Name name;
    Date dob;
    String sid;
}
```

Create an instance of the Student class!

We can even define
new classes,
that logically represent a
person's name as *first, middle, last* and
date of birth as *month, day, year*.

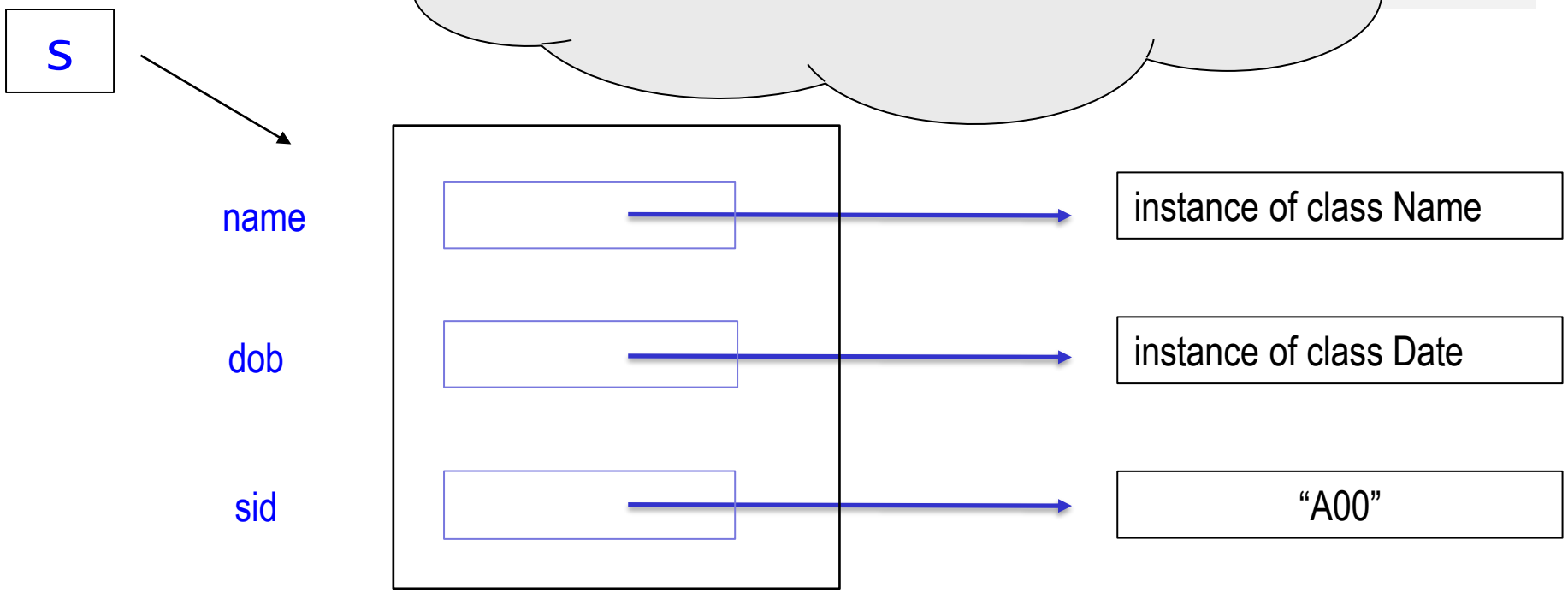


Why the need for classes?

```
class Student {
    Name name;
    Date dob;
    String sid;
}
```

Create an instance of the Student class!

This is referred to as *object composition*. When one class is composed of objects of other classes!

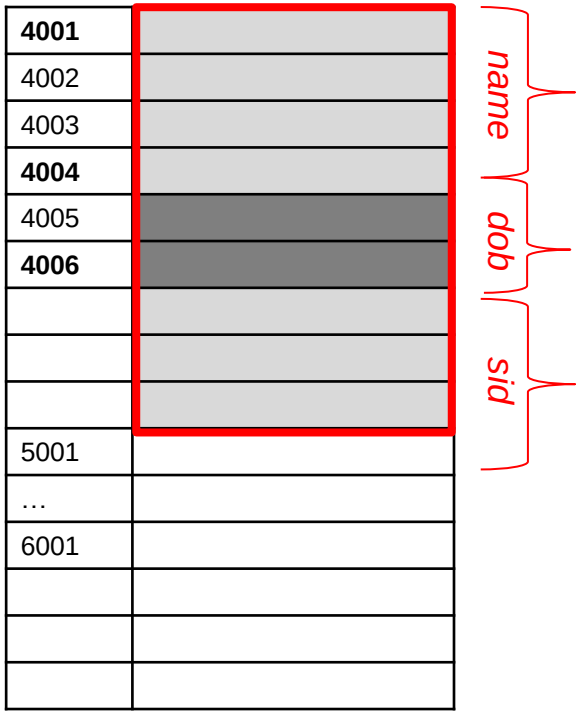


From Arrays to C-Structures…

```
struct Student {
    char name[20];
    char dob[6];
    char sid[10];
};
```

student.h

```
#include "student.h"

int main() {
   Student s;

}
```



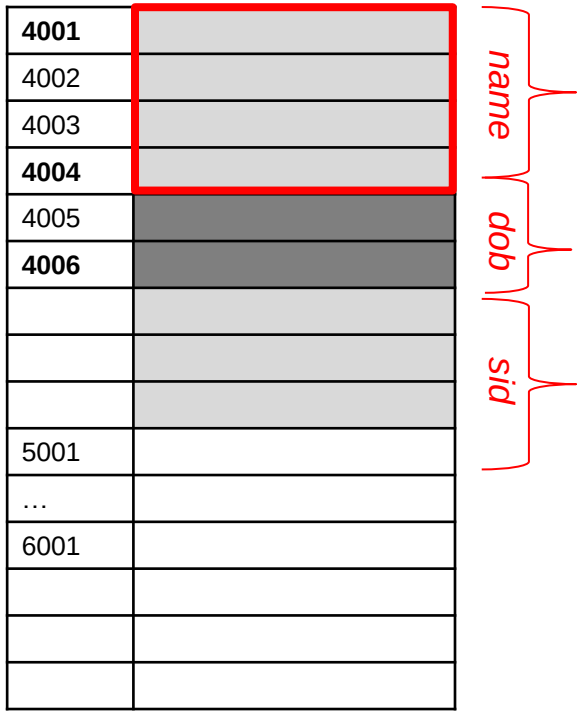
In the world of C/C++ you can declare the physical structure to be part of the Stack frame,

From Arrays to C-Structures…

```
struct Student {
    char name[20];
    char dob[6];
    char sid[10];
};
```

student.h

```
#include "student.h"

int main() {
   Student s;

   printf("Enter a name: " );
   scanf("s", s.name );
}
```



From Arrays to C-Structures…

```
struct Student {
    char name[20];
    char dob[6];
    char sid[10];
};
```

student.h

```
#include "student.h"

int main() {
   Student s;

   initialize(s);
   output(s);
}
```

4001	
4002	
4003	
4004	
4005	
4006	
5001	
…	
6001	

name

dob

sid

From Arrays to C-Structures…

```
struct Student {
    char name[20];
    char dob[6];
    char sid[10];
};
```

student.h

```
#include "student.h"

int main() {
   Student *s;

}
```

4001	
4002	
4003	
4004	
4005	
4006	
5001	
…	
6001	

s

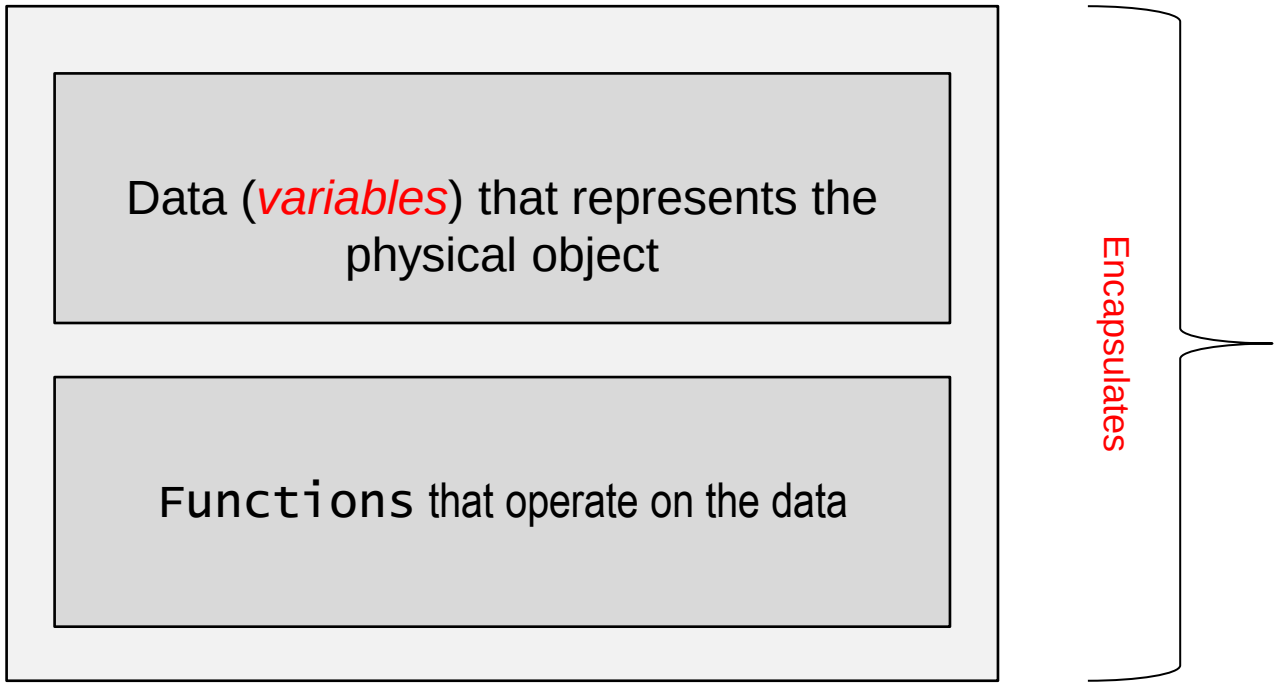
This declares a variable that will *point* to a student structure which will dynamically allocated at run-time.

Functions to work on the structures:

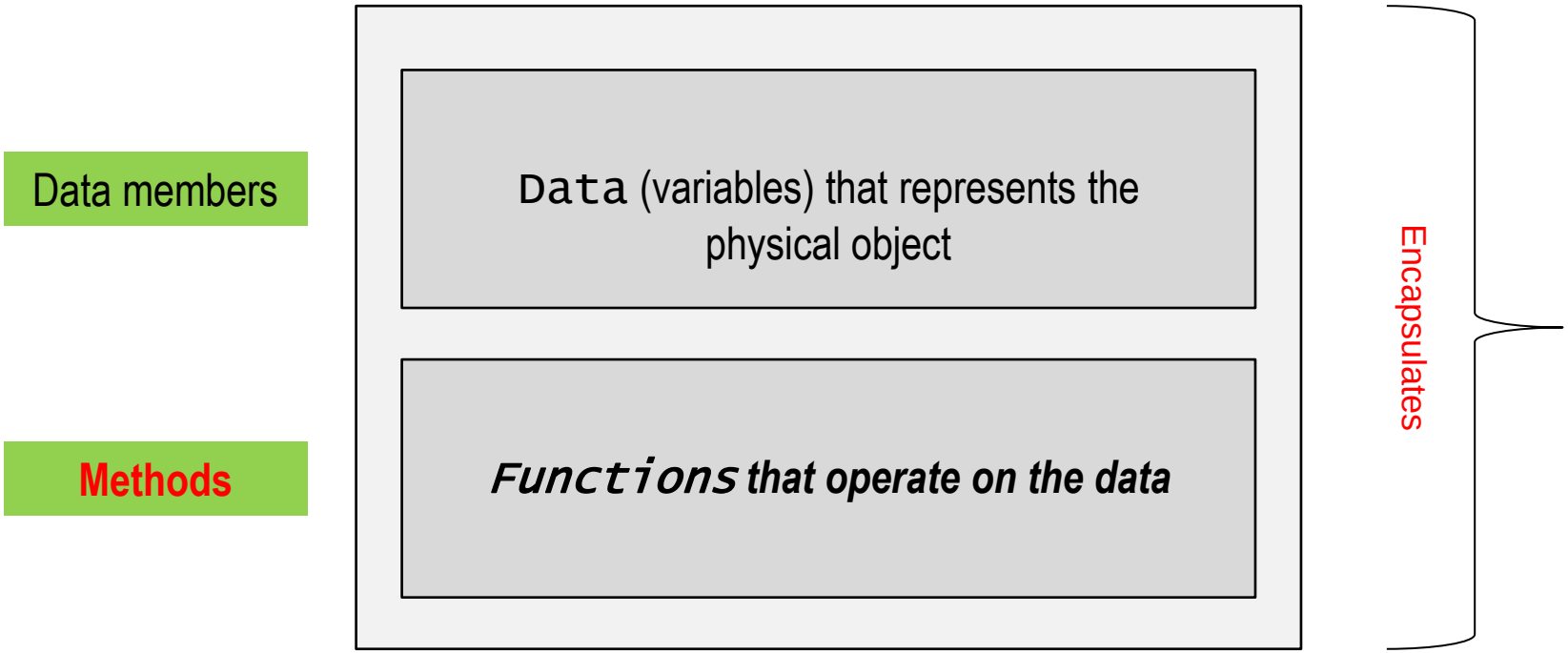
decoupling of data and functions

```
#include "student.h"

void initialize(struct Student s) {
   printf("Enter a name: " ); scanf("%s", s.name );
   printf("Enter a dob: " ); scanf("%s", s.dob );
   printf("Enter a student id: " ); scanf("%s", s.id );
}

void output(struct Student s) {
    printf(("Name: %s, Dob: %s, sid: %s\n"
           s.name, s.dob, s.sid );
}

int main() {
   struct Student s1, s2;

   initialize(s1); // Call the function and pass the object to initialize

   output(s1); // call the function and pass the object to output

}
```

From Structures to Classes

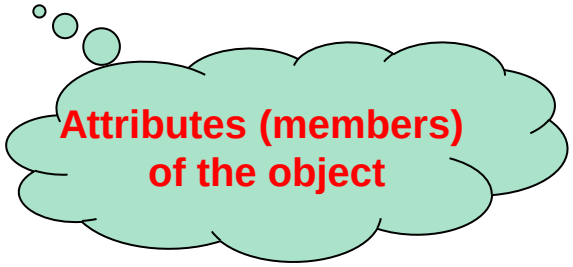


From Structures to Classes

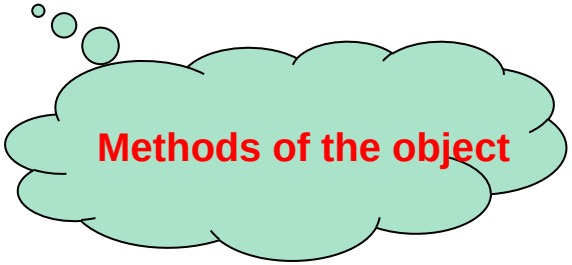


What Is An Object?

- An object is a construct that groups together:
  - one or more data values that describe an object



  - one or more functions that operate on those data values



Java Methods

There are two types of methods
we can define for a class:

Static

non Static

Java Methods

There are two types of methods
we can define for a class:

A **static** method can be called on the class, but **NOT** on an instance of the class.

```
className.method()
```

non Static

Java Methods

There are two types of methods
we can define for a class:

A **static** method can be called on the class, but **NOT** on an instance of the class.

```
Math.pow(8, 3)
```

non Static

Java Methods

There are two types of methods
we can define for a class:

A **static** method can be called on the class, but **NOT** on an instance of the class.

```
className.method(object)
```

If the static method needs access to a specific object, then the object must be explicitly passed to the method as an input parameter.

non Static

Java Methods

There are two types of methods
we can define for a class:

A **static** method can be called on the class, but **NOT** on an instance of the class.

```
int[] arr = {1,2,3};
Arrays.toString(arr)
```

If the static method needs access to a specific object, then the object must be explicitly passed to the method as an input parameter.

non Static

Java Methods

There are two types of methods
we can define for a class:

A static method can be called on the class, but NOT on an instance of the class.

```
className.method(object)
```

If the static method needs access to a specific object, then the object must be explicitly passed to the method as an input parameter.

A ***non* static** method must be called on an *instance* of the class.

```
object.method()
```

As the method is being called on an object, the method has direct access to all the data and methods of the class.

Java Methods

There are two types of methods
we can define for a class:

A static method can be called on the class, but NOT on an instance of the class.

```
className.method(object)
```

If the static method needs access to a specific object, then the object must be explicitly passed to the method as an input parameter.

A ***non* static** method must be called on an *instance* of the class.

```
String str = "Hello!";
int len = str.length()
```

As the method is being called on an object, the method has direct access to all the data and methods of the class.

Designing a Custom Class

- What's in a Date?

- Example:
  - 02/25/1962
  - 03/02/1996
  - 10/04/1999

- **month** // int
- **day** // int
- **year** // int

Attributes of a *date*

Class Definition:

a *user defined* custom datatype

```
public class Date {

    int month;
    int day;
    int year;



}
```

Attributes of the class

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date(); // instance of
        bday.month = 2;
        bday.day = 25;
        bday.year = 1962;
    }
}
```

Class Definition:

a *user defined* custom datatype

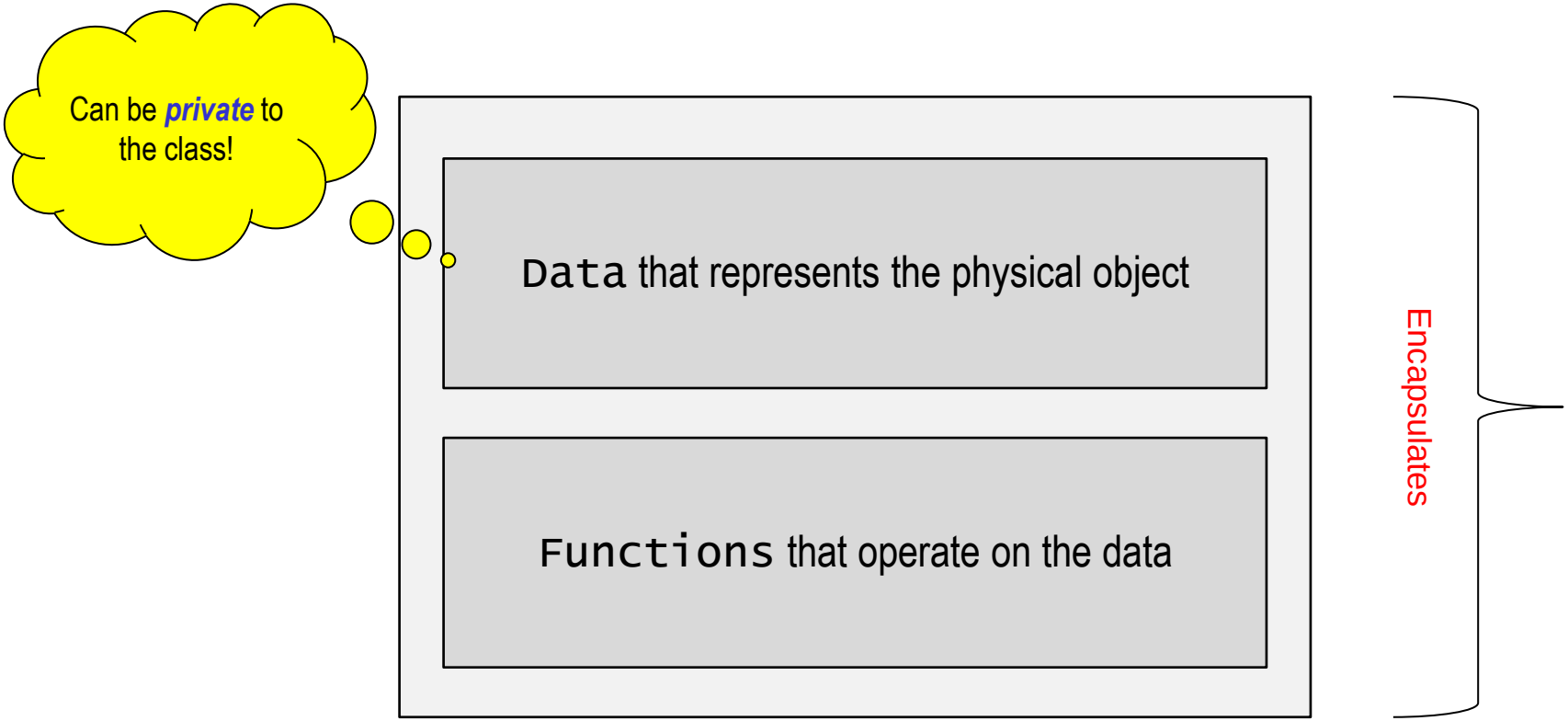
```
public class Date {

    int month;
    int day;
    int year;

}
```

Attributes
class

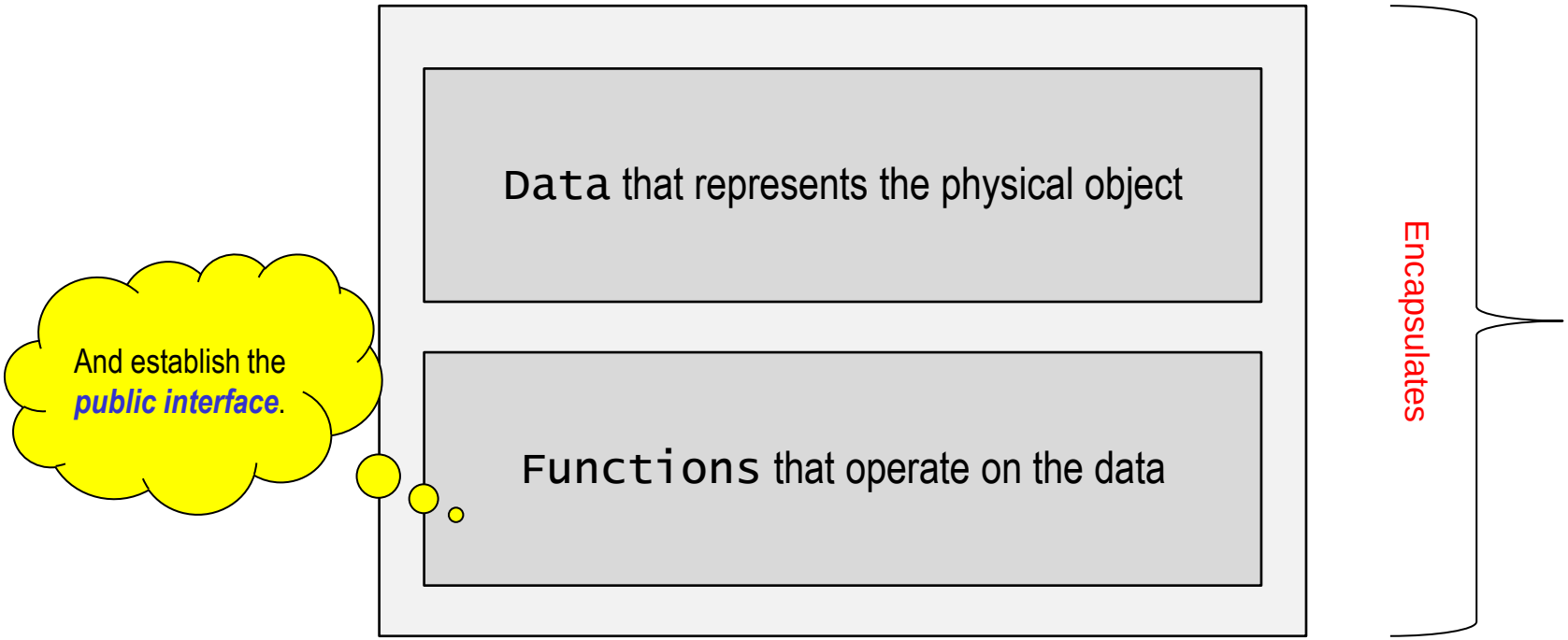
How can we ensure that *applications* or *clients* who create instances of our class use them correctly?

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date();
        bday.month = 13;        // Invalid Data
        bday.day = 25;
        bday.year = 1962;
    }
}
```

Classes and Data and Function Encapsulation



Classes and Data and Function Encapsulation



Classes with a **Domain** Specific Interface
C++ and Java

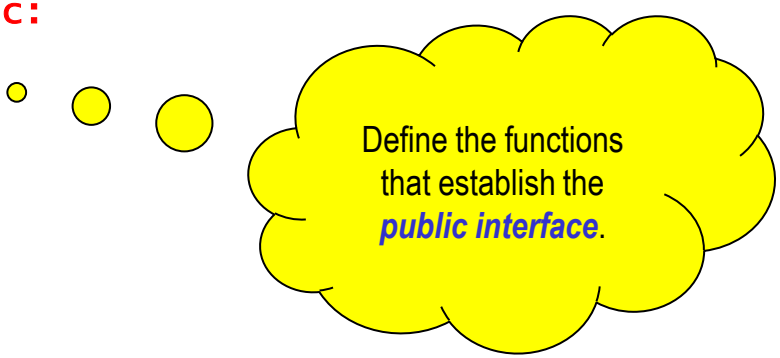
```
class className
   private:
```

Declare the data members that define the **physical object**.

```
   public:
```

Classes with a **Domain** Specific Interface
C++ and Java

```
class className
   private



   public:
```



Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int month;
    private int day;
    private int year;



}
```

Attributes of the class

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date();
        bday.month = 2;
        bday.day = 25;
        bday.year = 1962;
    }
}
```

Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int month;
    private int day;
    private int year;

    public boolean isHoliday() { ... }

    public int calculateAge() { ... }

    public int daysUntil( Date someDate ) { ... }

    public String formatDate() { ... }

    public boolean equals( Date someDate ) { .. }

}
```

The class defines the public interface!

Behaviors of the class

Functions that each object can perform!

Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int month;
    private int day;
    private int year;

    public boolean isHoliday() { ... }

    ...
}
```

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date();
        // can only access public items
        // need to assign the date by calling
        // a public method of the class!
    }
}
```

Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int month;
    private int day;
    private int year;

    public boolean isHoliday() { ... }
    public boolean setDate(int m, int d, int y) {...}
    ...
}
```

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date();
        bday.setDate( 02, 25, 1962 );



    }
}
```

Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int mont
    private int da
    private int y

    public boolear
    public boolean
    ...
}
```

But what if I want to create an instance of Date with specific initial values, should I always have to call the setDate method?

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date();
        if ( !bday.setDate( 02, 25, 1962 ) )
           // throw an exception...

    }
}
```

Class Definition:

a *user defined* custom datatype

```
public class Date

    private int month
    private int day;
    private int year;

    public boolean is
    public boolean setDate(
    ...
}
```

Class can provide special methods called ***constructors*** *which can assign initial values at object creation!*

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date(2, 25, 1962);



    }
}
```

Class Definition:

a *user defined* custom datatype

```
public class Date {

    private int month;
    private int day;
    private int year;

    public boolean isV
    public boolean setDa
    ...
}
```

And constructors can also ensure that our objects are only initialized with valid data!

```
public class testDate {

    public static void main( String[] args ) {
        Date bday = new Date(2, 30, 1962);



    }
}
```

Class Definition

```
public class className {

    public static variables    // class scope

    private instance members   // object scope

            Instance Methods of the class

    public className() { ... };

    public datatype setMethod() { ... };

    public datatype getMethod() { ... };

    // ... methods to print, compare, etc.

            Static Methods of the class

    // methods called at the class level
}
```

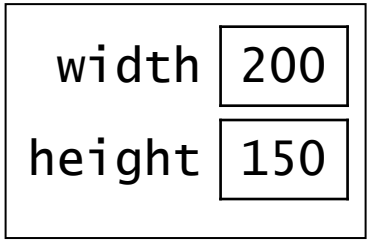
Attributes of the *class*.

Methods of the *class*.

Case Study: A Rectangle Class

- Let's say that we want to create a data type for objects that represent rectangles.

- Every Rectangle object should have two variables inside it (*`width`* and *`height`*) for the rectangle's dimensions.
  - these variables are referred to as *fields*
  - also known as: attributes, instance variables



- We'll also put functions/methods inside the object.

Sample Rectangle Class

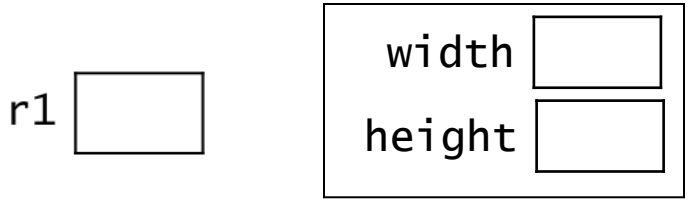
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }


    .
    .
    .
    .



}
```

Constructor

- The constructor has the same name as the class.
  - it is non-static
  - it has no return type
- The purpose of the constructor is to initialize the members.
- Constructors can be overloaded.
- A constructor that defines no parameters is referred to as the a no-arg constructor.
- If a class does not define ***any*** constructors, Java will provide a ***default*** *no-arg* constructor for the class.

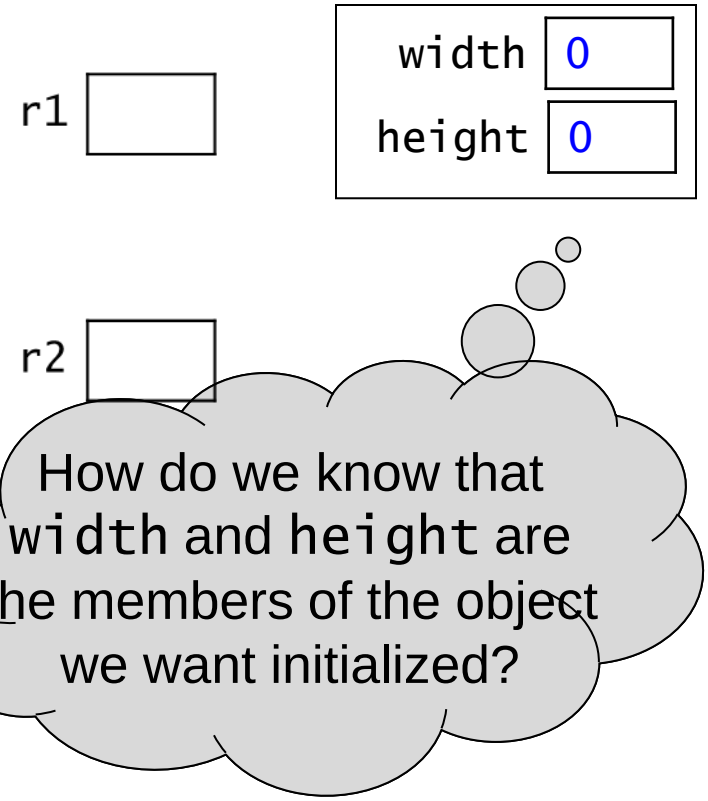
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



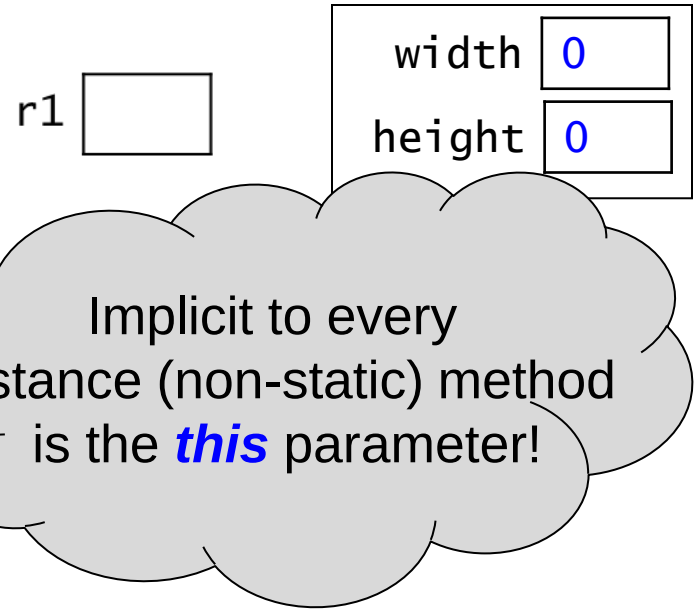
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }


}
```



Sample Rectangle Class

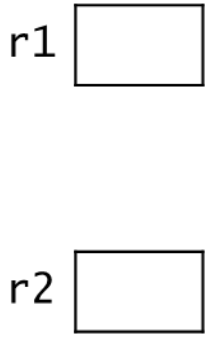
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```

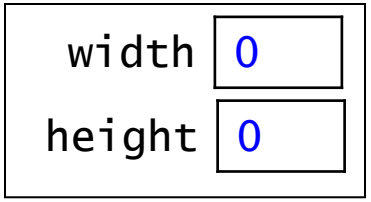
r2

*The **this*** paramater contains the address location of the object the method was called on.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        this.width = this.height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```





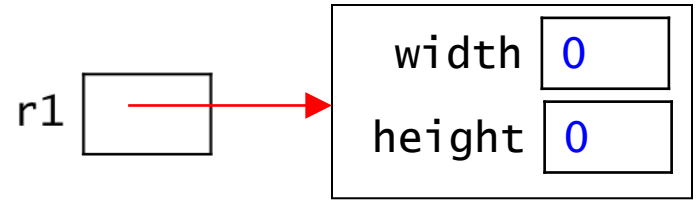
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        this.width = this.height = 0
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```

Note that this call is part of an assignment statement.

Sample Rectangle Class

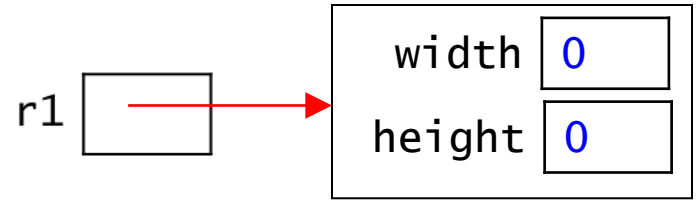
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        this.width = this.height = 0
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



Constructors return the address location of the object constructed via the *this* parameter!

Sample Rectangle Class

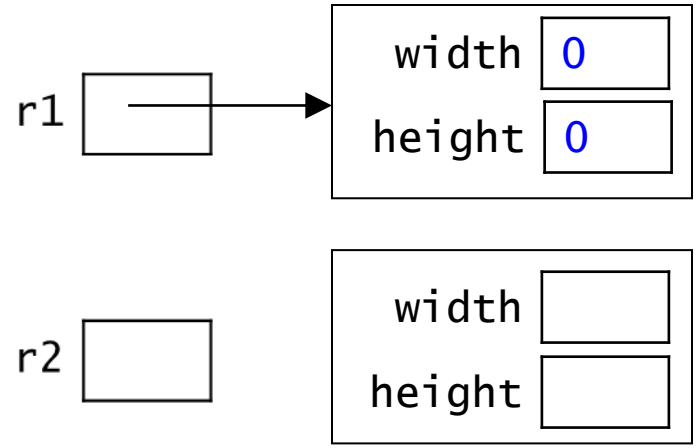
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        this.width = this.height = 0
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



This is why constructors cannot be declared to be void methods!

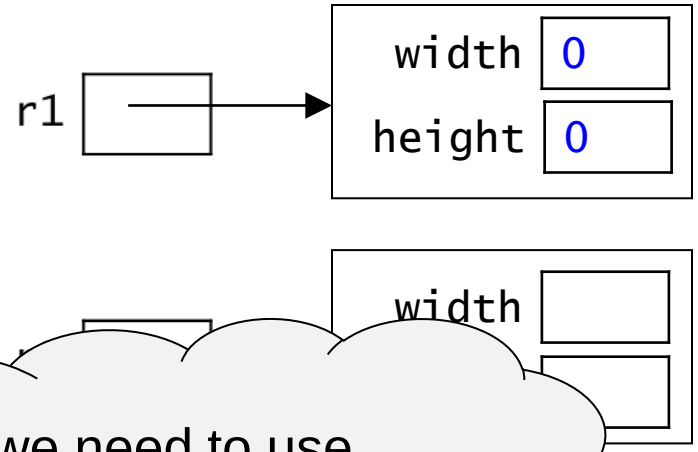
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



Sample Rectangle Class

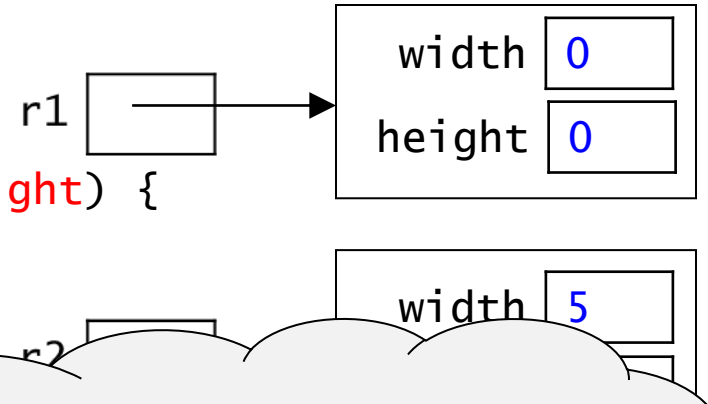
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```

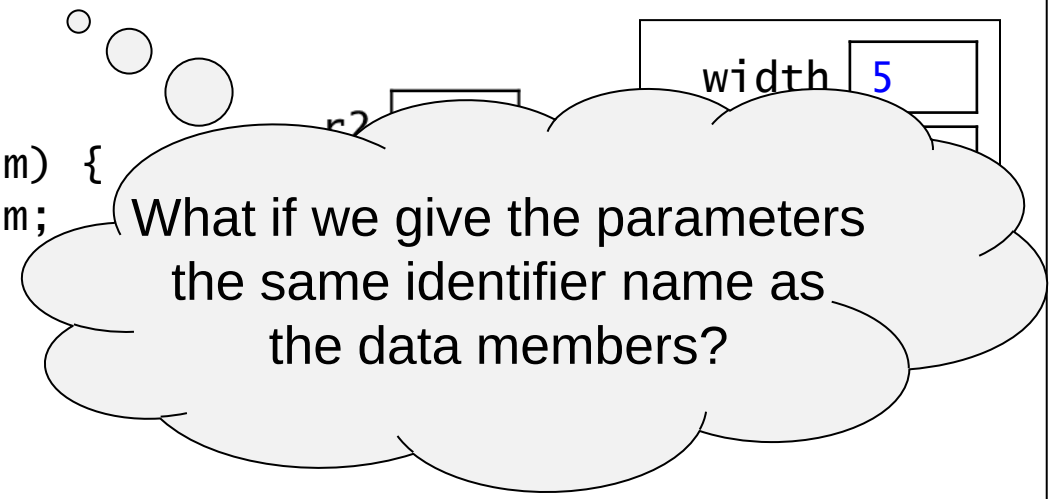


Do we need to use the *this* reference to access the data members?

Sample Rectangle Class

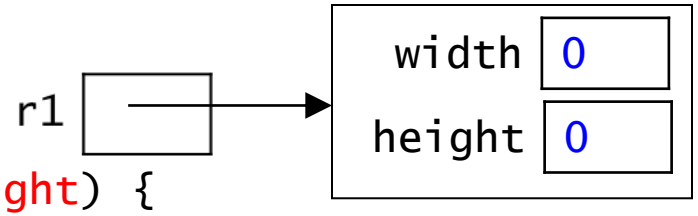
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        width = width;
        height = height;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```

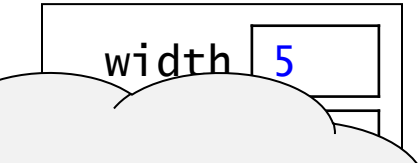




Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        width = width;
        height = height;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```

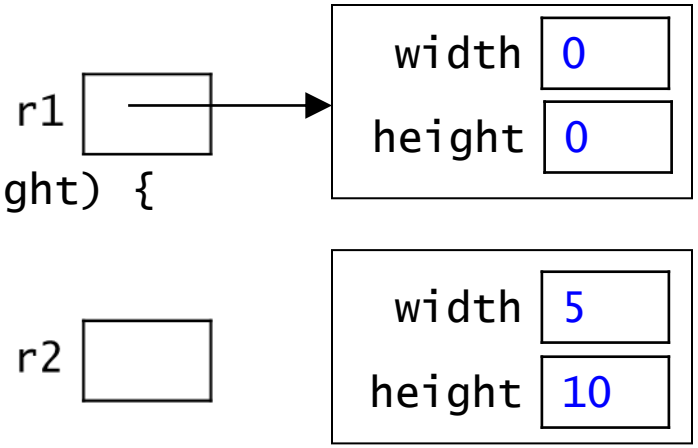




Now we have a scope issue!

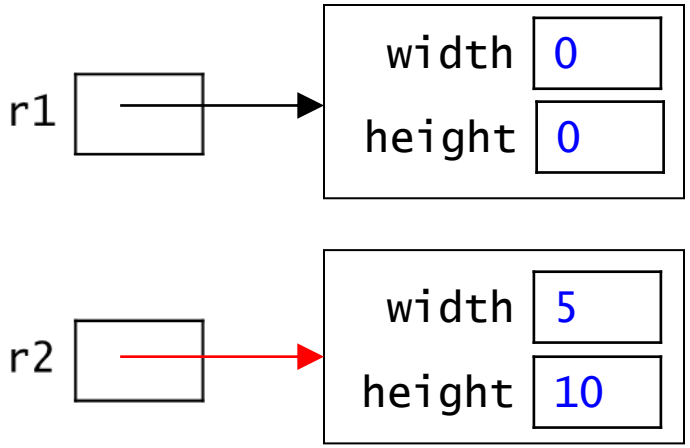
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }

}
```



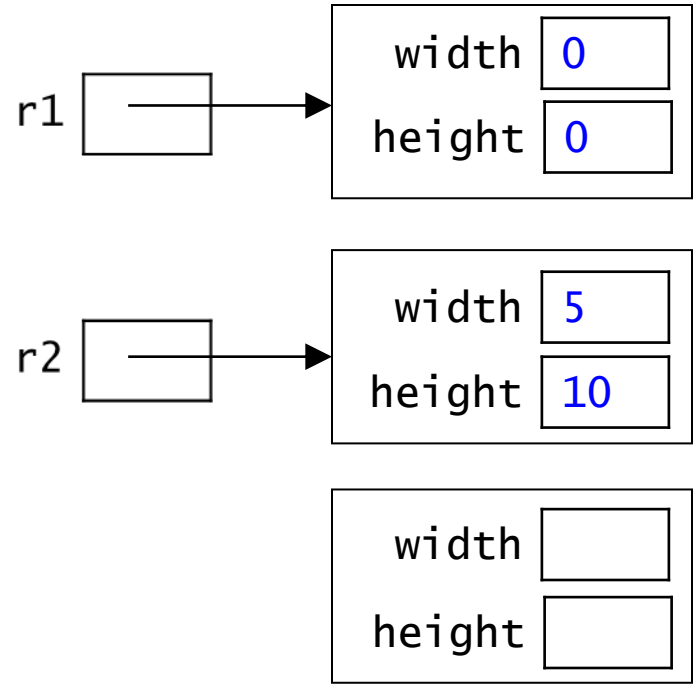
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
    }


}
```



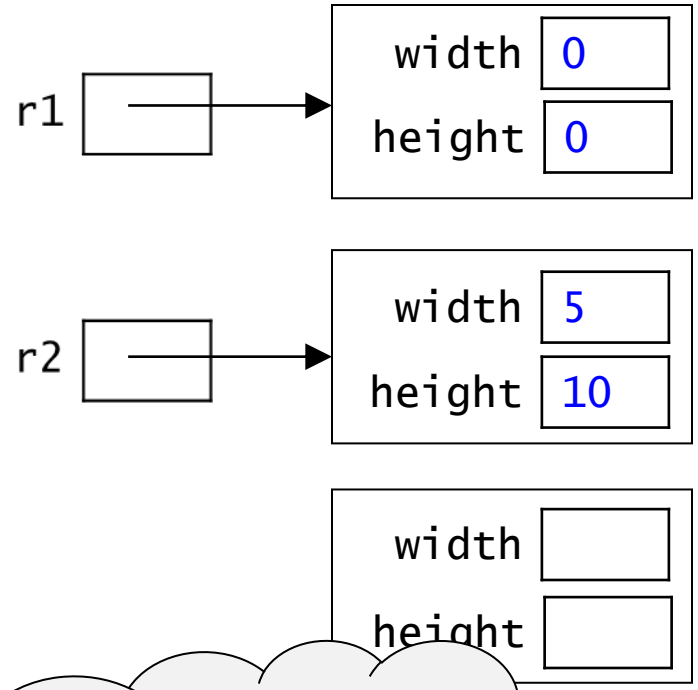
Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
        Rectangle r3 = new Rectangle(7);
    }

}
```



Sample Rectangle Class

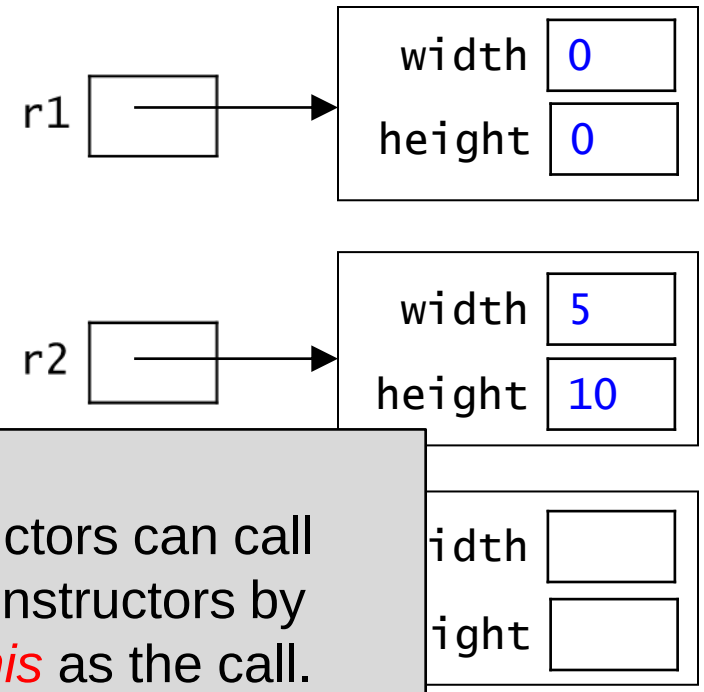
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        width = height = dim;
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String

        Rectangle r1 = new Rectangle(),
        Rectangle r2 = new Rectangle(5, 10),
        Rectangle r3 = new Rectangle(7);
    }

}
```



Note that both constructors are doing the same thing.

Sample Rectangle Class

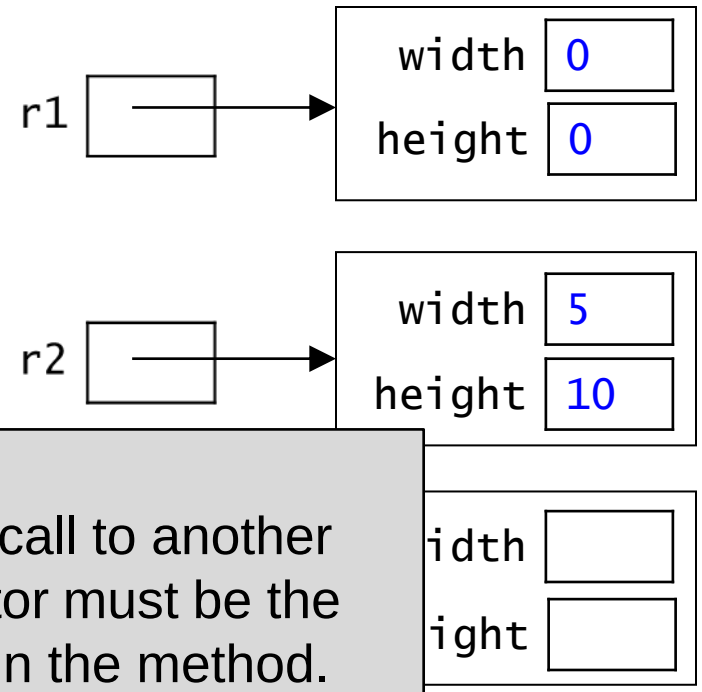
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        this(dim, dim);
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
        Rectangle r3 = new Rectangle(7);
    }

}
```



Constructors can call *other* constructors by using *this* as the call.

Sample Rectangle Class

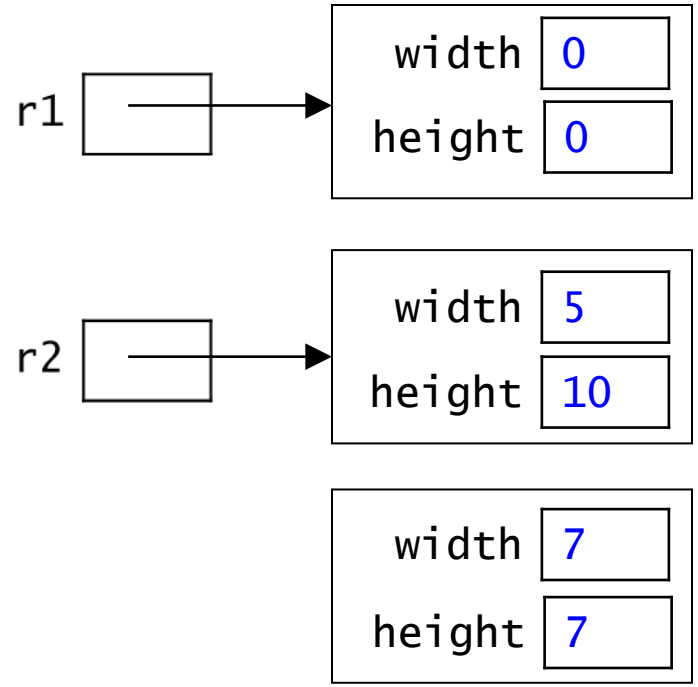
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        this(dim, dim);
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main( String [] args ) {

        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle(5, 10);
        Rectangle r3 = new Rectangle(7);
    }

}
```



The *this* call to another constructor must be the first call in the method.

Sample Rectangle Class

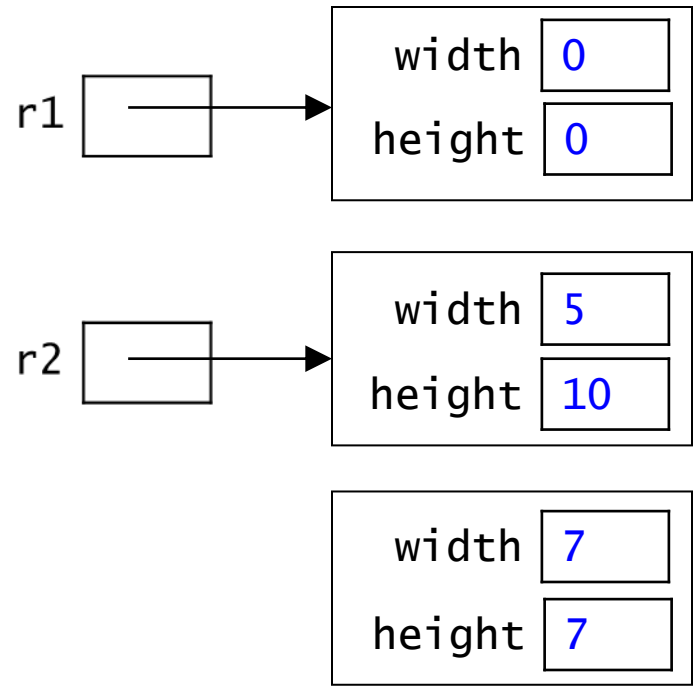
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        this(dim, dim);
    }
    public Rectangle() {
        width = height = 0;
    }
    .
    .
    .
    public static void main

        Rectangle r1 = new
        Rectangle r2 = new Re
        Rectangle r3 = new Rectang
    }

}
```



This constructor is also doing the same as the other two constructors.

Sample Rectangle Class

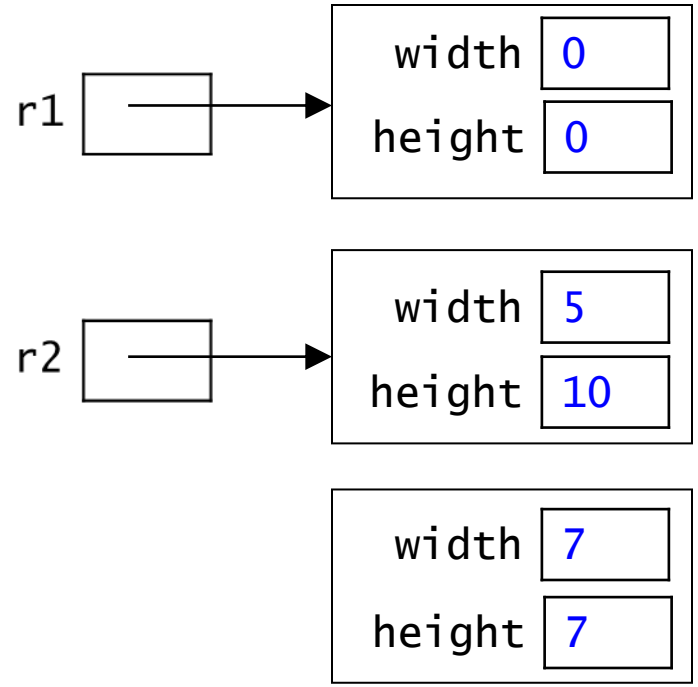
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        this(dim, dim);
    }
    public Rectangle() {
        this(0, 0);
    }
    .
    .
    .
    public static void main

        Rectangle r1 = new
        Rectangle r2 = new Re
        Rectangle r3 = new Rectang
    }

}
```



Can use *this* to call one of the other constructors.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public Rectangle(int dim) {
        this(dim, dim);
    }
    public Rectangle() {
        this(0);
    }
    .
    .
    .
    public static void main

        Rectangle r1 = new
        Rectangle r2 = new Re
        Rectangle r3 = new Rectang
    }

}
```



Can use *this* to call one of the other constructors.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    ...

}
```

Accessor Methods

Mutator Methods

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
    public void grow(int dw, int dh) {
        width += dw;
        height += dh;
    }
    public double area() {
        return( width*height );
    }

    ...

}
```

Accessor Methods

- Allow *applications* or *client methods* to gain access to the data stored in private data members!

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
    public void grow(int dw, int dh) {
        width += dw;
        height += dh;
    }
    public double area() {
        return( width*height );
    }

    ...

}
```

Accessor Methods

- Allow *applications* or *client methods* to gain access to the data stored in private data members!
- Or perform a necessary operation of the class without altering the values of the data members.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
    public void grow(int dw, int dh) {
        width += dw;
        height += dh;
    }
    public double area() {
        return( width*height );
    }

    ...

}
```

Mutator Methods

- Alter the values of the data members.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int
        width = w;
        height = h;
    }
    .
    .
    .
```

```
public void grow(int width, int height) {
      this.width += width;
      this.height += height;
  }
```

```
    .
    .
    .
}
```

What if the variables passed to the method result in inappropriate dimensions, a negative width or height?

Allowing Appropriate Changes

- To allow for appropriate changes to an object, we add whatever mutator methods make sense.

- These (*setter*) methods can prevent inappropriate changes:

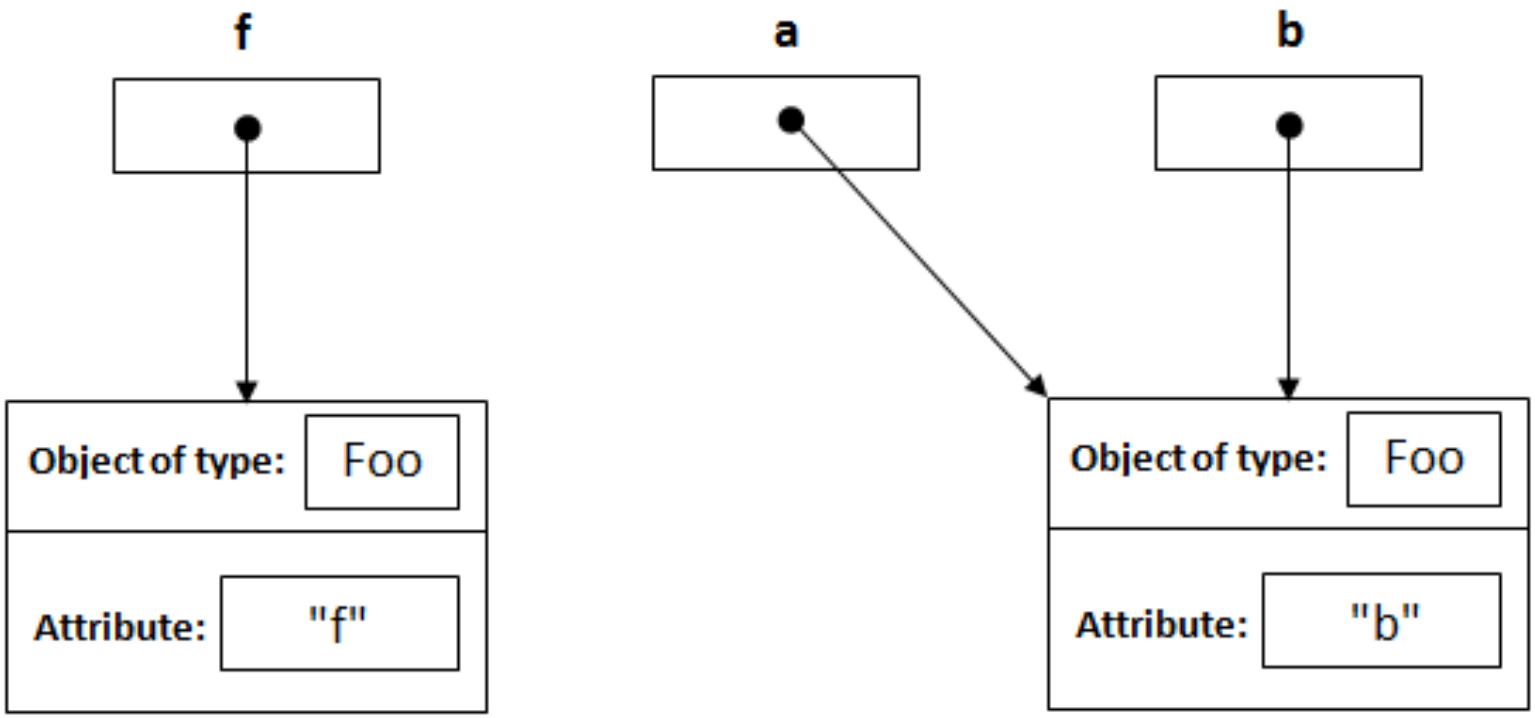
```
public void setWidth(int w) {
    if (w <= 0) {
        throw new IllegalArgumentException();
    }

    this.width = w;
}

public void setHeight(int h) {
    if (h <= 0) {
        throw new IllegalArgumentException();
    }

    this.height = h;
}
```

Throwing an exception ends the method call.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;
    public Rectangle(int w, int h) {
        this.setWidth(w);
        this.setHeight(h);
    }
    .
    .
    .
    public void grow(int dw, int dh) {
        this.setWidth(width+dw);
        this.setHeight(height+dh);
    }
    .
    .
    .
}
```

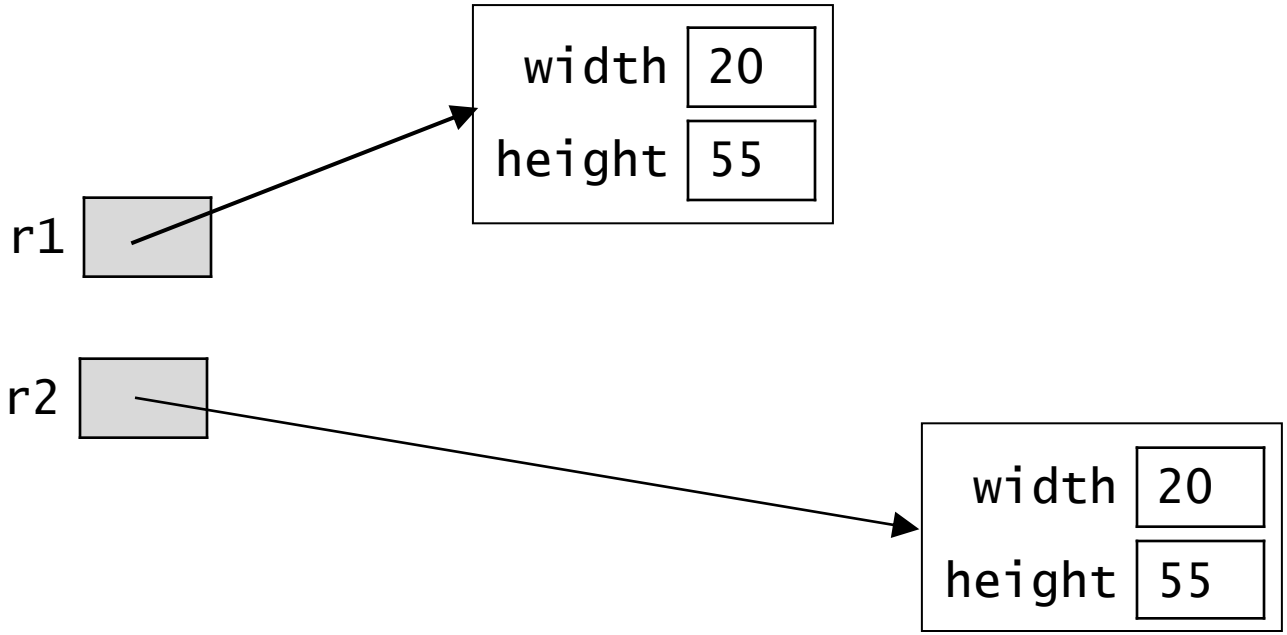
Objects are *Reference* types



Testing for Equivalent Objects

- Let's say that we have two different `Rectangle` objects, both of which represent the same rectangle:

```
Rectangle r1 = new Rectangle(20, 55);
Rectangle r2 = new Rectangle(20, 55);
```



- What is the value of the following condition?

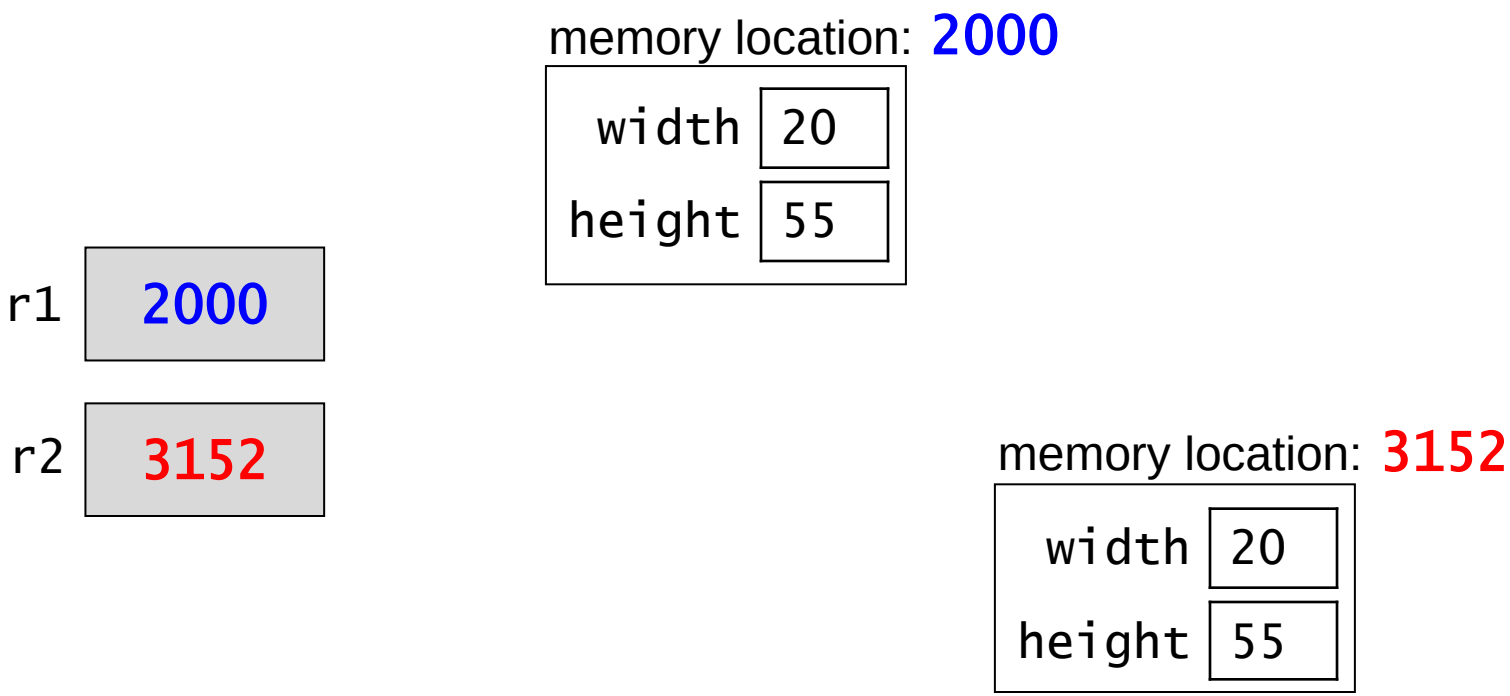
```
r1 == r2
```

Testing for Equivalent Objects (cont.)

- The condition

  `r1 == r2`

  compares the *references* stored in `r1` and `r2`.



- It doesn't compare the objects themselves.

Testing for Equivalent Objects (cont.)

- To test for equivalent objects, we need to use the `equals` method:

```
r1.equals(r2) // commutative
```

Testing for Equivalent Objects (cont.)

- To test for equivalent objects, we need to use the `equals` method:

```
r2.equals(r1) // commutative
```

Testing for Equivalent Objects (cont.)

- To test for equivalent objects, we need to use the `equals` method:

```
r1.equals(r2)
```

- Java's built-in classes have an *equals* methods that:
  - returns `true` if the two objects are equivalent to each other
  - returns `false` otherwise

```
String s1 = "CS112";
String s2 = "CS611";
if ( s1.equals(s2) )
   System.out.println("I am not doing my job!");
```

Default `equals()` Method

- If we don't write an `equals()` method for a class, objects of that class get a default version of this method.

- The default `equals()` just tests if the memory addresses of the two objects are the same.
  - the same as what == does!

- To ensure that we're able to test for equivalent objects, we need to write our own `equals()` method.

`equals()` Method for Our `Rectangle` Class (cont.)

- Here's an alternative version:

```
public boolean equals(Rectangle other) {
    return (other != null
            && this.width == other.width
            && this.height == other.height);
}
```

Converting an Object to a String

```
Rectangle r1 = new Rectangle(10, 20);
System.out.println(r1.toString());
```

Java makes an *implicit* call to the `toString` method

Converting an Object to a String

- The `toString()` method allows objects to be displayed in a human-readable format.
  - it returns a string representation of the object

- This method is called *implicitly* when you attempt to print an object or when you perform string concatenation:

```
Rectangle r1 = new Rectangle(10, 20);
System.out.println(r1);
```

equivalent to:

```
System.out.println(r1.toString());
```

Converting an Object to a String

- The `toString()` method allows objects to be displayed in a human-readable format.
  - it returns a string representation of the object

- This method is called implicitly when you attempt to print an object or when you perform string concatenation:

```
Rectangle r1 = new Rectangle(10, 20);
System.out.println(r1);

// the second line above is equivalent to:
System.out.println(r1.toString());
```

- If we don't write a `toString()` method for a class, objects of that class get a default version of this method.
  - here again, it usually makes sense to write our own version

`toString()` Method for Our `Rectangle` Class

```
public String toString() {
    return width + " x " + height;
}
```

- Note: the method does not do any printing. It returns a `String` that can then be printed.

Sample Rectangle Class

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHheight(h);
    }
    .
    .
    .
    public void grow(int dw, int dh) {
        setWidth(width+dw);
        setHeight(height+dh);
    }
    public double area() {
        return(width*height);
    }
    public boolean equals(Rectangle other) {
        return (other != null && this.width == other.width
                                && this.height == other.height );
    }
    public String toString() {
       return (width + " x " + height);
    }
```