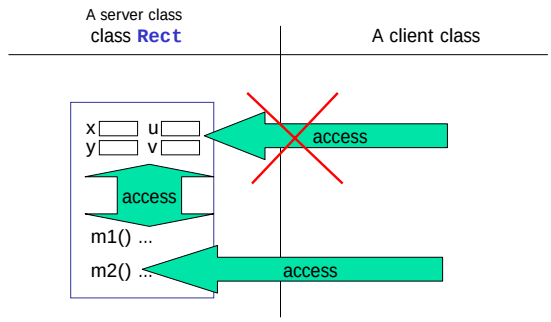
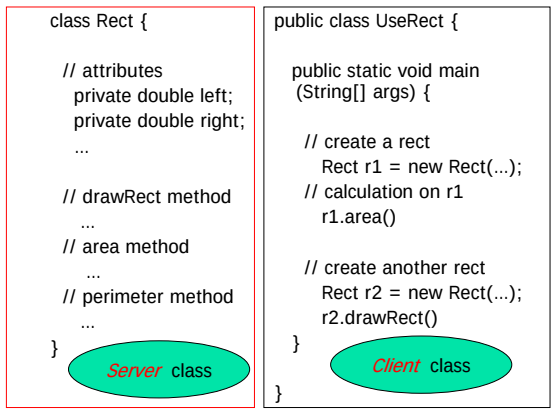
- Previous Lecture:
  - Creating objects and calling their methods
  - OO thinking
- Today's Lecture:
  - Defining a class:
    - Instance variables and methods
    - Constructors
    - Methods with parameters
    - Method toString *(in lab this week)*
- Reading: Sec 6.2-6.5
- Announcement:
  - Section in the computer lab this week

OOP ideas

- Aggregate variables/methods into an abstraction (a class) that makes their relationship to one another explicit
- Objects (instances of a class) are self-governing (protect and manage themselves)
- Hide details from client, and restrict client's use of the services
- Allow clients to create/get as many objects as they want



Data within objects should be protected: **`private`**
Provide only a set of methods for **`public`** access.



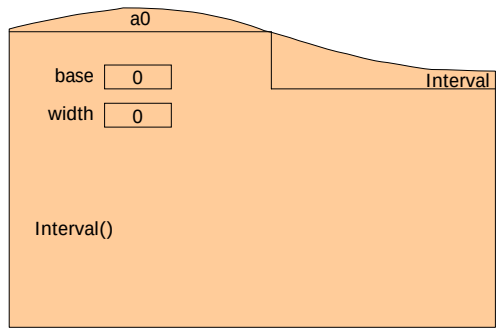
Class Definition

```
public class class-name {

    declaration (and initialization)

    constructor

    methods

}
```


Class definition: declarations

```
class Interval {
  private double base;  // low end
  private double width; // interval width
}
```

- *Declarations* in a class define ***fields*** (***instance variables***) of the class
- Each class is a *type*. Classes are *not* primitive types.
- Declarations are made *outside* of a method (but in a class, of course)



Declarations Revisited

- Syntax: *type name*;
- Examples:
```
int count;
double width;
Interval in1;
Interval in2;
```
- Instance variables have default initial values
  - `int` variables: `0`
  - Non-primitive (reference) variables: `null`
    Value `null` signifies that no object is referenced
  - Different from local variables, which have no default values

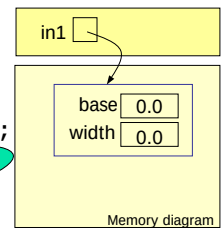
Object instantiation

- An expression of the form
  `new` *class-name*`( )`
  computes a reference to a newly created object of the given class
- Examples:
```
Interval in1;           //declaration
in1 = new Interval();   //instantiation

//Combined declaration & instantiation
Interval in2 = new Interval();
```


Do not access fields directly

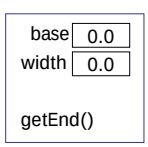
```
public class Client {
  public static void main(String[] args){

    Interval in1;
    in1= new Interval();
    System.out.println(
     in1.base+in1.width);
  }
}
```

base, width are private

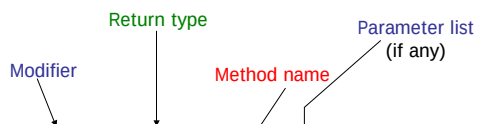


Class definition

```
class Interval {
  private double base;  // low end
  private double width; // interval width

  /* =Get right end of interval */
  public double getEnd() {
    return base + width;
  }

}
```



Instance method




```
public double getEnd() {
  return base + width;
}
```

The absence of the keyword `static` → an instance method
(There isn't a keyword "instance")

Methods

A method is a named, parameterized group of statements

```
modifier return-type method-name ( parameter-list ) {
    statement-list
}
```

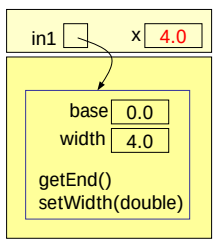
return-type `void` means nothing is returned from the method

There must be a `return` statement, unless return-type is `void`

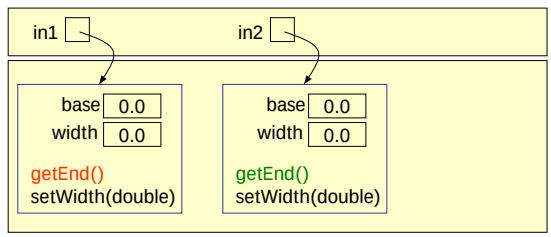

```
/* An Interval is [base, base+width] */
class Interval {
   private double base;  // low end
   private double width; // interval width

   /* =Get right end of interval */
   public double getEnd() {
     return base + width;
   }
   /* set width to w */
   public void setWidth(double w) {
     width= w;
   }
}
```


Calling an instance method

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

    Interval in1;
    in1= new Interval();
    double x;
    in1.setWidth(4);
    x= in1.getEnd();
  }
}
```




```
Interval in1= new Interval();
Interval in2= new Interval();
if ( in1.getEnd() > in2.getEnd() )
  System.out.println("blah...");
```



Constructor

- A *constructor* is used to create objects
- Each class has a default constructor
- You can define your own constructor:

```
modifier class-name ( parameter-list ) {
    statements-list
}
```

- Use `public` as the modifier for now
- an instance method that has *no* return type


```
class Interval {
  private double base;  // low end
  private double width; // interval width

  /* An Interval with base b, width w */
  public Interval(double b, double w) {
    base= b;
    width= w;
  }

  public double getEnd() {
    return base + width;
  }
}
```

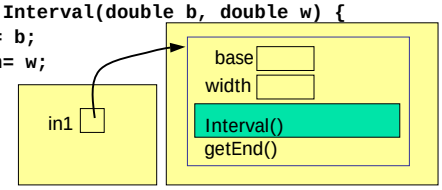
```
class Interval {
  private double base;  // low end
  private double width; // interval width

  /* Default constructor */
  public Interval() {}




  public double getEnd() {
    return base + width;
  }
}
```



```
public class Client {
  public static void main(String() args) {
    Interval in1= new Interval(3,0.1);
  }
}
```

Create an object

```
class Interval {
  private double base, width;

  public Interval(double b, double w) {
    base= b;
    width= w;
  }
  ...
}
```



Method `toString()`

- Every object has default method `toString`
- *Automatically* invoked by `print`, `println`

```
Interval a = new Interval(1,2);
System.out.println(a);
```

- Some default text will be printed unless you define a `toString` method

Method `toString()`

- Usually defined to give a *useful* description of an instance of a class
- E.g., useful description of an instance of `Interval` could be the mathematical notation for an Interval, e.g.,

  `[3,7.5]`

  for an `Interval` object with `base` 3 and `width` 4.5.


```
class Interval {
  private double base;  // low end
  private double width; // 

  public Interval(
    base= b;
    width= w;
  }

  /** =String description of Interval */
  public String toString() {
    return "[" + getBase() + "," + getEnd() +
           "]";
  }
}
```

Note that `toString` doesn't print—it returns a String. Another method (e.g., `println`) does the actual printing.

More instance methods with input parameters

- Write an instance method

  `expand(double f)`

  that expands the `Interval` by a factor of `f`.

- What should be the method header?

- Parameter of primitive type