**Previous Lecture:**
- Instance methods with primitive parameters
- Constructor
- `this`

**Today's Lecture:**
- Methods with non-primitive parameters
- Class (static) variables and methods
- Method `toString`
- Method `overloading`

**Reading:**
- Sec 1.5, 1.7, Sec 3.1.4, 3.2.4, 3.2.5, 3.3.1, 3.3.2

---

```java
public void expand(double f) {
    width *= f;
}
```

---

```java
public class Client {
    public static void main(String[] args)
    {
        Interval i1 = new Interval(0.2, 0.7);
        double x = 2;
        i1.expand(x);
        System.out.println(i1.getEnd());
    }
}
```

---

**Primitive input parameter**
- Write an instance method `expand(double f)` that expands the `Interval` by a factor of `f`.
- Parameter of primitive type: pass-by-value
  I.e., value is copied

```java
/** Expand this Interval by a factor of f */
public void expand(double f) {
    setWidth(width * f);
}
```

---

Use available methods when possible!
Non-primitive input parameter

- Write an instance method
  ```java
  isIn(Interval i)
  ```
  that returns the boolean value true if the instance is in Interval i. Return false otherwise.

- Parameter of non-primitive type: pass-by-reference i.e., Reference is copied; object itself is not copied

```java
/** ="this Interval is in i" */
pubic boolean isIn(Interval i) {
  return ( getBase()>=i.getBase() &&
     getEnd()<=i.getEnd() );
}

public boolean isIn(Interval i) {
  boolean in = getBase()>=i.getBase() &&
     getEnd()<=i.getEnd();
  return in;
}
```

**Not concise!!**

```java
public class Client {
  public static void main(String[] args){
    Interval i1= new Interval(0.2,0.7);
    Interval i2= new Interval(Math.random(),0.2);
    if (i2.isIn(i1))
      System.out.println("Interval i2 “
        + "is in Interval i1.");
    else
      System.out.println("Interval i2 “
        + "is not in Interval i1.");
  }
}
```

- Method toString()
  
  - Every object has default method toString
  
    ```java
    Interval a = new Interval(1,2);
    System.out.println(a);
    ```

  - Automatically invoked by print, println

- 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` would be the mathematical notation for an Interval, e.g., `[3, 7.5]` for an `Interval` object with `base` 3 and `width` 4.5.

Static Variables & Methods

- *Shared* by all instances of a class
- Only one copy no matter how many objects have been instantiated
- Keyword: `static`
- Examples:
  - A constant used by the whole class
  - A variable to keep track of how many `Interval` objects have been created
  - A method that doesn’t need to reference fields

Class (`static`) method

- Write a class method `overlap(Interval a, Interval b)` that returns a new `Interval` if `Interval`s `a` and `b` overlap. Return `null` otherwise.
- What is the method header?

```
blueHigh < redHigh
redHigh < blueHigh
```

The overlap’s left is the rightmost of the two original lefts

```
The overlap’s left is the rightmost of the two original lefts
```

The overlap’s right is the leftmost of the two original rights
The overlap's left is the rightmost of the two original lefts.

The overlap's right is the leftmost of the two original rights.

No overlap if OLeft > ORight.

```
public class Client {
    public static void main(String[] args){
        Interval i1= new Interval(0.2,0.7);
        Interval i2= new Interval(
            Math.random(),0.2);
        Interval o= Interval.overlap(i1,i2);
    }
}
```

An instance overlap method

- Write an instance method
  `overlap(...)`
  that returns a new `Interval` if two `Interval`s overlap. Return `null` otherwise.

- What is the method header? What should be the parameters, if any?

- Are the static and instance versions very different?

```
public class Interval {

    Interval() {
    }

    public static Interval overlap(Interval i1, Interval i2) {
        // Implementation
    }
}
```

Method overloading

- Different methods can have the same name.
- A method has a signature: method name and the parameter types (including the order).
- In a class, all methods must have different signatures.
- E.g., the `abs` method in the `Math` class.