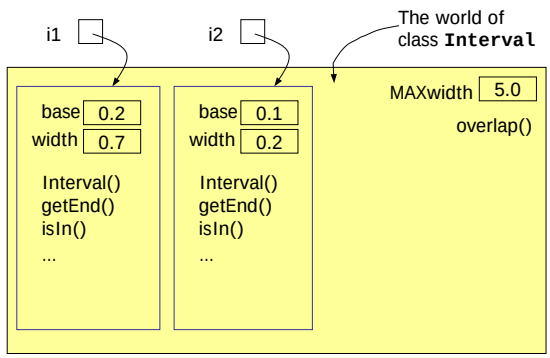
- Previous Lecture:
  - Defining a class:
    - Methods with non-primitive parameters
    - Methods that return non-primitive values
    - Static variables and methods
- Today's Lecture:
  - Method overloading
  - Review with Person class
  - 1-d array
- Reading:
  - Sec 8.1-8.5
  - "Enhanced for-loop" in Sec 8.2 is optional


```
/* =the overlapped Interval between
   Intervals a and b */
public static Interval overlap(Interval a,
                               Interval b) {
  Interval olap;      // overlapped interval
  double left, right; // olap's left & right

  left = Math.max(a.getBase(),b.getBase());
  right = Math.min(a.getEnd(),b.getEnd());
  if ( (right-left) <= 0 )
      olap= null;
  else
      olap= new Interval(left, right-left);
  return olap;
}
```



```
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?


```
/* =the overlapped Interval between
   Intervals a and b */
public static Interval overlap(
                  Interval a, Interval b)


/* =the overlapped Interval between this
   Interval and Interval b */
public Interval overlap(Interval b)
```

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
- The return type is not part of the signature
- E.g., the **abs** method in the **Math** class


```
class Interval {
  private double base;  // low end
  private double width; // interval width
  public static final double maxWidth=5;

  public Interval(double b, double w) {
    setBase(b);
    setWidth(w);
  }

  public Interval() {}

  /* An Interval with base b and maxWdith */
  public Interval(double b) {
    setBase(b);
    setWidth(maxWidth);
  }
  // other methods below
```

this(b,maxWidth)

Chain invocation of methods

- Suppose there are 3 intervals: **i1**, **i2**, **i3**
- You know that **i1** and **i2** overlap
- Write code to find if the overlapped interval of **i1** and **i2** *is in* interval **i3**

```
Interval i1 = new Interval(…);
Interval i2 = new Interval(…);
Interval i3 = new Interval(…);
// Assume i1 and i2 overlap
if (                                  )
   System.out.println("in i3");
else
   System.out.println("not in i3");
```



```
Interval i1 = new Interval(…);
Interval i2 = new Interval(…);
Interval i3 = new Interval(…);
/* Without assuming that i1 and i2
   overlap */
```


A different example

- Create a **Person** class to organize data about a **Person**:
  - **Name**
  - **Age**
  - …


```
public class Person {
  private String name;
  private int age;

  public static final int LEGALage=18;

  /** Constructor */
  public Person(String name, int age)
  { this.name= name; this.age= age; }

  /** =This Person is an adult */
  public boolean isAdult()
  { return age >= LEGALage; }

  /** =String description of this Person */
  public String toString()
  { return name + " is " + age; }
} // class Person
```

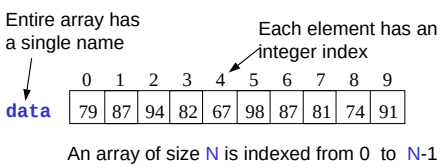
Modify **Person** class

- Modify **Person** class to store data about a **Person**'s best friend: add another instance variable **friend**
- What should be the type of the field **friend**?
- Add two more methods to the class definition: **makeFrend**, **beFriendOf**


```
/** Make a friend with Person p */
public void makeFriend(Person p) {


}


/** Become a friend of Person p */
public void beFriendOf(Person p) {


}
```


Arrays

- An array is an object
- An array is an ordered list of values (or objects)
- Each element is of the same type



An array of size N is indexed from 0 to N-1

Array declaration

***type*[] *identifier*;**

Examples:

```
int[] counts;
double[] price;
boolean[] flip;
char[] vowel;
String[] names;
Interval[] series;
```


Array construction (instantiation)

new *type*[*size*]

must be an integer

Example:

```
new int[4]
```

Declaration & creation:

```
int limit= 4;
double[] price;
price= new double[limit];
```


Array declaration & construction

***type*[] *identifier* = new *type*[*size*];**

Example:

```
int[] counts= new int[4];
```

Then values can be assigned into the cells, e.g.:

```
counts[0]= 6;  counts[2]= 9;
```

Array length and default values

Once created, an array has a fixed length, held in the array's constant called length:

```
int[] counts= new int[4];
System.out.println(counts.length);
// will print 4

System.out.println(counts[2]);
// Array components have default
// values.  Above statement will
// print 0
```


Array creation with initializer list

Create an array using an initializer list:

```
int[] x= new int[]{6,3,4,8};
```

Length of array is determined by length of the initializer list. Shortcut:

```
int[] x= {6,3,4,8};
```

Only when declaring & creating in same statement!

Index operator [ ]

***identifier*[*integer_expression*]**

Accesses an element of the array, e.g.:

```
int[] count= new int[101];
   // declaration & instantiation
count[70+9]= 98;
   // set count[79] to 98
int face= (int) (Math.random()*6);
count[face]= count[face] + 1;
count[face]++;
```


Elements in an array

If **count** is of type **int[]**, i.e., an array of **int**s, then the type of

count[*i*]

is **int** and **count[*i*]** can be used anywhere an **int** variable can be used

Type of count: int[]

Type of count[i]: int

Pattern for processing an array

```
// assume an array has been
// created and is referred to by
// variable A

for (int i=0; i<A.length; i++) {

  // perform some process
  // (on A[i])

}
```


Example

```
// Create an array of length 6
// with random numbers in the range
// of 5 to 9.  Calculate the sum.
```