More constructors

Constructors

- recall that a public constructor is what a client uses to create an object
- the purpose of a constructor is to initialize the state of an object
  - it should set the values of the non-static fields to appropriate values
    - we should set the fields named **`real`** and **`imag`**
- our complex number class has a single constructor so far

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }
```

Constructors

- our class is missing two constructors commonly found in a value type class
- no-argument constructor
  - a constructor defined as having no parameters
- copy constructor
  - a constructor with a single parameter whose type is the same as the type of the class

No-argument constructor

- a no-argument constructor requires no information from the client
  - i.e., the client does not specify anything regarding the state of the constructed object
- the purpose of a no-argument constructor is to create an object with a well specified standard state
  - for example, we might provide a no-argument constructor that constructs the complex number $(0 + 0i)$

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public Complex() {
    this.real = 0;
    this.imag = 0;
  }
```

Copy constructor

- a copy constructor copies the state of another object of the same type as the class
  - it has a single parameter that is the same type as the class
- a copy constructor for our complex number class would copy the real and imaginary parts of another complex number

```
public Complex(double real, double imag) {
  this.real = real;
  this.imag = imag;
}

public Complex() {
  this.real = 0;
  this.imag = 0;
}

public Complex(Complex other) {
  this.real = other.getReal();
  this.imag = other.getImag();
}
```

Avoiding Code Duplication

- notice that the constructor bodies are almost identical to each other
  - all three constructors have 2 lines of code
  - all three constructors set the real and imaginary parts

- whenever you see duplicated code you should consider moving the duplicated code into a method
- in this case, one of the constructors already does everything we need to implement the other constructors…

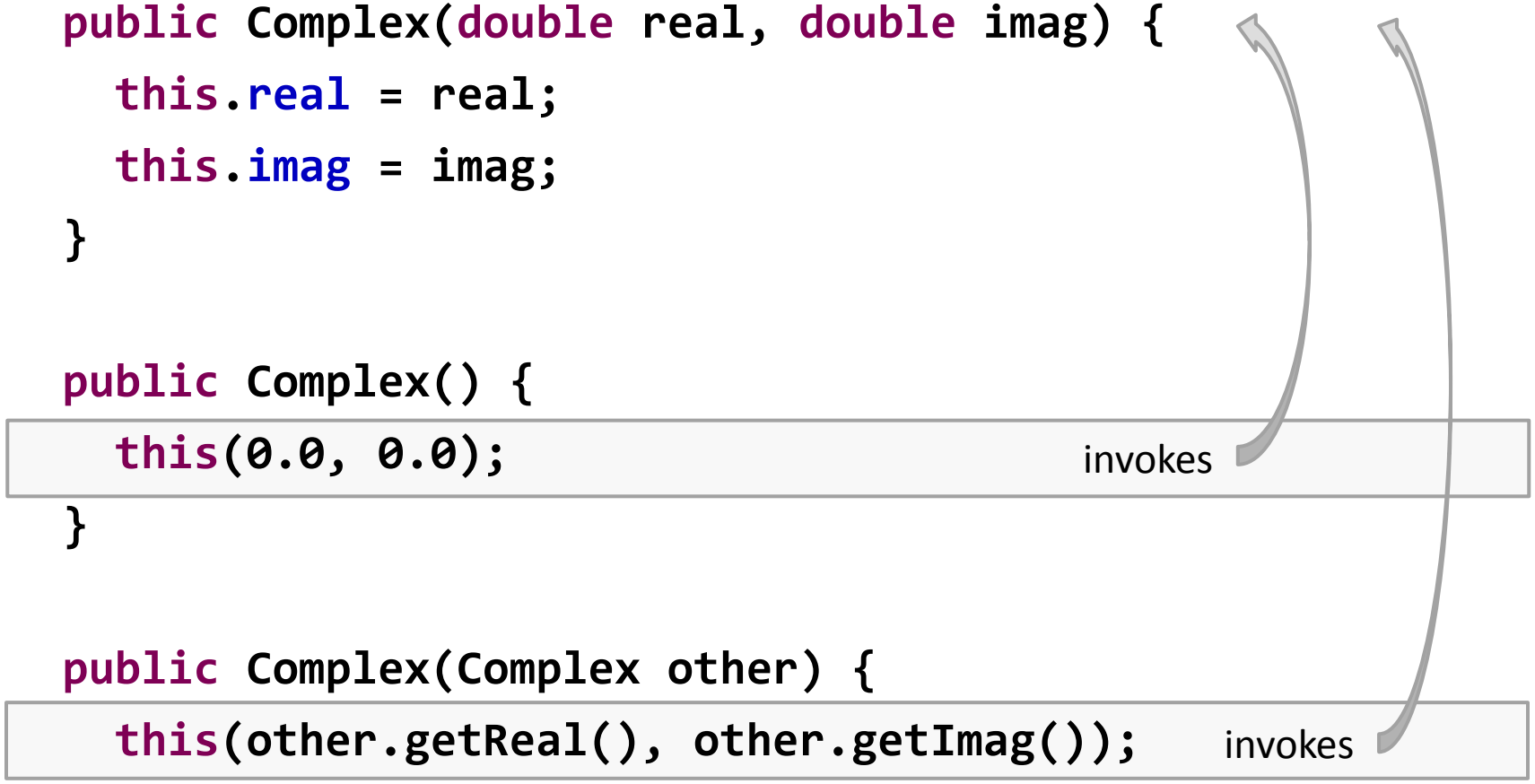
Constructor chaining

- a constructor is allowed to invoke another constructor
- when a constructor invokes another constructor it is called *constructor chaining*
- to invoke a constructor in the same class you use the **`this`** keyword
  - if you do this then it *must occur* on the first line of the constructor body
    - but you *cannot* use **`this`** in a method to invoke a constructor
- we can re-write two of our constructors to use constructor chaining…

```
public Complex(double real, double imag) {
  this.real = real;
  this.imag = imag;
}

public Complex() {
  this(0.0, 0.0);
}

public Complex(Complex other) {
  this(other.getReal(), other.getImag());
}
```



compareTo

Comparable Objects

- many value types have a natural ordering
  - that is, for two objects **`x`** and **`y`**, **`x`** is less than **`y`** is meaningful
    - **`Short`**, **`Integer`**, **`Float`**, **`Double`**, etc
    - **`String`**s can be compared in dictionary order
    - **`Date`**s can be compared in chronological order
    - you might compare **`Complex`** numbers by their absolute value

- if your class has a natural ordering, consider implementing the **`Comparable`** interface
  - doing so allows clients to sort arrays or **`Collection`**s of your object

Interfaces

- an interface is (usually) a group of related methods with empty bodies
  - the **`Comparable`** interface has just one method

```
public interface Comparable<T>
{
  int compareTo(T t);
}
```

- a class that implements an interfaces promises to provide an implementation for every method in the interface

`compareTo()`

- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Throws a **`ClassCastException`** if the specified object type cannot be compared to this object.

Complex compareTo

```
public class Complex implements Comparable<Complex> {
  // fields, constructors, methods...
```

```
  @Override
  public int compareTo(Complex other) {
    double thisAbs = this.abs();
    double otherAbs = other.abs();
    if (thisAbs > otherAbs) {
      return 1;
    }
    else if (thisAbs < otherAbs) {
      return -1;
    }
    return 0;
  }
```

Complex compareTo

- don't forget what you learned in EECS1020
  - you should delegate work to well-tested components where possible
- for complex numbers, we need to compare two **`double`** values
  - **`java.lang.Double`** has methods that do exactly this

Complex compareTo

```
public class Complex implements Comparable<Complex> {
  // fields, constructors, methods...

  @Override
  public int compareTo(Complex other) {
    return Double.compare(this.abs(), other.abs());
  }
```

Comparable Contract

1. the sign of the returned **`int`** must flip if the order of the two compared objects flip
   - if `x.compareTo(y) > 0` then `y.compareTo(x) < 0`
   - if `x.compareTo(y) < 0` then `y.compareTo(x) > 0`
   - if `x.compareTo(y) == 0` then `y.compareTo(x) == 0`

Comparable Contract

2. **`compareTo()`** must be transitive
   - if `x.compareTo(y) > 0 && y.compareTo(z) > 0` then `x.compareTo(z) > 0`
   - if `x.compareTo(y) < 0 && y.compareTo(z) < 0` then `x.compareTo(z) < 0`
   - if `x.compareTo(y) == 0 && y.compareTo(z) == 0` then `x.compareTo(z) == 0`

Comparable Contract

3. if `x.compareTo(y) == 0` then the signs of `x.compareTo(z)` and `y.compareTo(z)` must be the same

Consistency with equals

- an implementation of **`compareTo()`** is said to be consistent with **`equals()`** when

  if **`x.compareTo(y) == 0`** then
  `x.equals(y) == true`

  - and

    if **`x.equals(y) == true`** then
    `x.compareTo(y) == 0`

Not in the Comparable Contract

- it is *not* required that **`compareTo()`** be consistent with **`equals()`**
  - that is

    if **`x.compareTo(y) == 0`** then

    `x.equals(y) == false` is acceptable
  - similarly

    if **`x.equals(y) == true`** then

    `x.compareTo(y) != 0` is acceptable

- try to come up with examples for both cases above
- is **`Complex compareTo`** consistent with equals?

Implementing `compareTo`

- if you are comparing fields of type `float` or `double` you should use `Float.compare` or `Double.compare` instead of `<`, `>`, or `==`

- if your `compareTo` implementation is broken, then any classes or methods that rely on `compareTo` will behave erratically
  - `TreeSet`, `TreeMap`
  - many methods in the utility classes `Collections` and `Arrays`