Non-static classes

Part 2

Methods

- like constructors, all non-static methods have an implicit parameter named **`this`**
- for methods, **`this`** refers to the object that was used to call the method

Accessors

- an accessor method enables the client to gain access to an otherwise private field of the class
- the name of an accessor method often begins with **`get`**
- for fields of primitive type or immutable type, the accessor method implementation simply returns the value of the field
  - for fields that are object references the implementer must think more carefully about the implementation
    - this will be discussed later on in the course

```java
public class Complex {

  private double real;
  private double imag;

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

  public double getReal() {
    return this.real;
  }

  public double getImag() {
    return this.imag;
  }
```

Mutators

- a mutator method enables the client to modify (or mutate) an otherwise private field of the class
- the name of an accessor method often begins with **`set`**
- for fields of primitive type or immutable type, the mutator method implementation simply modifies the value of the field
  - for fields that are object references the implementer must think more carefully about the implementation
    - this will be discussed later on in the course

```java
public void setReal(double newReal) {
  this.real = newReal;
}

public void setImag(double newImag) {
  this.imag = newImag;
}
```

`conj`

- to compute the complex conjugate of

$$a + bi$$

we return a new complex number equal to

$$a + (-b)i$$

```java
public Complex conj() {
  return new Complex(this.getReal(), -this.getImag());
}
```

`abs`

- to compute the absolute value of

$$a + bi$$

we return a new real number equal to

$$\sqrt{a^2 + b^2}$$

```java
public double abs() {
  // "obvious" implementation
  double a = this.getReal();
  double b = this.getImag();
  return Math.sqrt(a * a + b * b);
}
```

`abs`

- the problem with the obvious implementation is that it fails in cases where the value of

```
z.abs()
```

can be represented using **`double`** but the value of

```
a * a + b * b
```

cannot be represented using **`double`**

`abs`

- examples of underflow and overflow

a	b	Computed value of a^2+b^2	Computed value of $\sqrt{a^2+b^2}$	Actual value of $\sqrt{a^2+b^2}$
1e-200	0	0	0	1e-200
1e-170	1e-169	0	0	1.00498756211208 9E-169
0	1e200	Infinity*	Infinity*	1e200
1e170	1e169	Infinity*	Infinity*	1.004987562112089E170

* **`Double.POSITIVE_INFINITY`**

- **`java.lang.Math`** provides a way to avoid intermediate under- and overflow for $\sqrt{a^2+b^2}$

```java
public double abs() {
  // avoids intermediate under- and overflow
  return Math.hypot(this.getReal(), this.getImag());
}
```

`abs`

- the field that studies solving mathematical problems using computational techniques is called *numerical analysis*
  - of interest in computer science, mathematics, engineering, and science
- how does `Math.hypot` work?
  - for a pure Java implementation the ideas described in the following link work
    - http://blogs.mathworks.com/cleve/2012/07/30/pythagorean-addition/

`add`

- to add two complex numbers

$$\underbrace{(a + bi)}_{\texttt{this}} + \underbrace{(c + di)}_{\texttt{other}}$$

we return a new complex number equal to

$$(a + c) + (b + d)i$$

```java
public Complex add(Complex other) {
  double a = this.getReal();
  double b = this.getImag();

  double c = other.getReal();
  double d = other.getImag();

  return new Complex(a + c, b + d);
}
```

`multiply`

- to multiply two complex numbers

$$\underbrace{(a + bi)}_{\texttt{this}} \times \underbrace{(c + di)}_{\texttt{other}}$$

we return a new complex number equal to

$$(ac - bd) + (bc + ad)i$$

```java
public Complex multiply(Complex other) {
  double a = this.getReal();
  double b = this.getImag();

  double c = other.getReal();
  double d = other.getImag();

  return new Complex(a * c - b * d,
                     b * c + a * d);
}
```

Obligatory methods

- recall that all classes in Java inherit from **`java.lang.Object`**
  - http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html
- any class you create inherits all of the **`public`** and **`protected`** fields and methods of **`java.lang.Object`**
  - the course notes refers to the methods inherited from **`java.lang.Object`** as *obligatory methods*
  - there are 11 such methods in total, but we are only interested in 3 of them
    - **`toString`**, **`equals`**, **`hashCode`**

toString

- **`toString()`** returns a **`String`** representation of the calling object
  - we can call **`toString()`** with our current **`Complex`** class even though we have not implemented it

```
// client of Complex

Complex z = new Complex(1, 2);
System.out.println(z.toString());
```

  - this prints something like **`Complex@fff003c1`** on my computer

`toString`

- **`toString()`** should return a concise but informative representation that is easy for a person to read
- it is recommended that all subclasses override this method
  - this means that any non-utility class you write should redefine the **`toString`** method
  - for our complex number class we might decide that **`toString`** should return strings that look like complex numbers
    - e.g., $\mathbf{2.2 + 3.7}\boldsymbol{i}$ or $\mathbf{-1.00001 - 92851.35}\boldsymbol{i}$

```java
@Override
public String toString() {
  StringBuilder b = new StringBuilder();
  b.append(this.getReal());
  double imag = this.getImag();
  if (imag < 0) {
    b.append(" - ");
  }
  else {
    b.append(" + ");
  }
  b.append(Math.abs(imag));
  b.append('i');
  return b.toString();
}
```

Overriding methods

- our class is a *subclass* or *child class* of **`java.lang.Object`**
- when a subclass redefines a **`public`** or **`protected`** method inherited from its superclass, we say that the subclass *overrides* the inherited method
- to override a method, you create a method that has the exact same signature and return type of the method that you want to override
  - the return type may also be a subtype of the return type of the overridden method (but this is not important for now)

Overriding methods

- when you override a method you may use the annotation **`@Override`** immediately before the method header
  - if you do so, the compiler will generate an error message if your method does not have the identical signature and return type of a method in a superclass

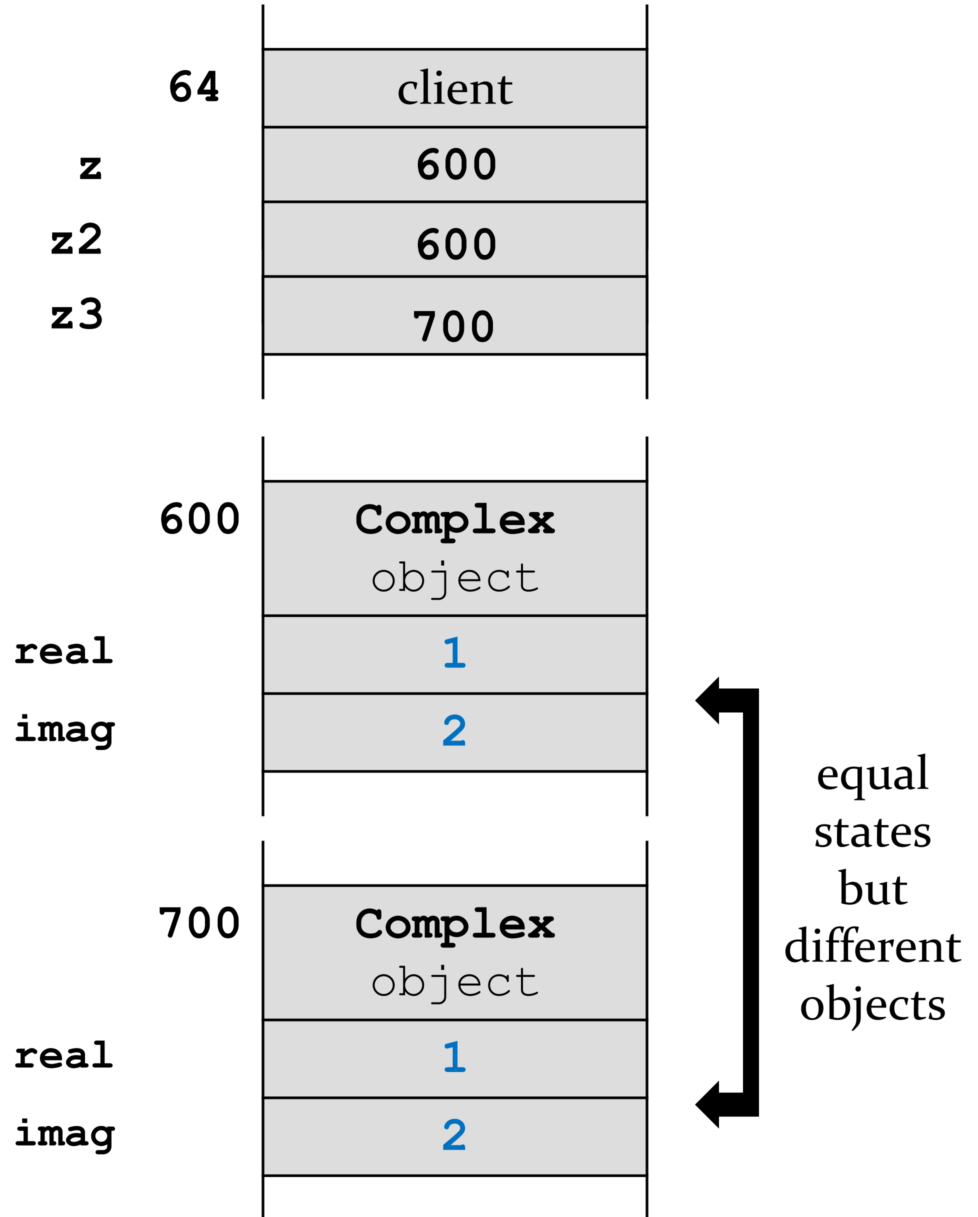
`equals()`

- suppose you write a value class that extends **`Object`** but you do not override **`equals()`**
  - what happens when a client tries to use **`equals()`**?
    - **`Object.equals()`** is called

```
// Complex client

Complex z = new Complex(1, 2);
System.out.println( z.equals(z) );        // true

Complex z2 = z;
System.out.println( z2.equals(z) );       // true

Complex z3 = new Complex(1, 2);
System.out.println( z3.equals(z));        // false!
```


[notes 3.2.4]

```
Complex z = new Complex(1, 2);
Complex z2 = z;
Complex z3 = new Complex(1, 2);
```



z and **z2** refer to the object at address 600

z3 refers to the object at address 700

`Object.equals`

- **`Object.equals`** checks if two references refer to the same object
  - **`x.equals(y)`** is true if and only if **`x`** and **`y`** are references to the same object

Complex.equals

- most value classes should support logical equality
  - an instance is equal to another instance if their states are equal
    - e.g. two complex numbers are equal if their real and imaginary parts both have the same values

- implementing **`equals()`** is surprisingly hard
  - "One would expect that overriding **`equals()`**, since it is a fairly common task, should be a piece of cake. The reality is far from that. There is an amazing amount of disagreement in the Java community regarding correct implementation of **`equals()`**. Look into the best Java source code or open an arbitrary Java textbook and take a look at what you find. Chances are good that you will find several different approaches and a variety of recommendations."
    - Angelika Langer, Secrets of equals() – Part 1
  - `http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html`

- what we are about to do does not always produce the result you might be looking for
  - but it is always satisfies the `equals()` contract
  - and it's what the notes and textbook do

EECS1030 Requirements for **`equals`**

1. an instance is equal to itself
2. an instance is never equal to **`null`**
3. only instances of the exact same type can be equal
4. instances with the same state are equal

1. An Instance is Equal to Itself

- **`x.equals(x)`** should always be **`true`**
- also, **`x.equals(y)`** should always be true if **`x`** and **`y`** are references to the same object
- you can check if two references are equal using **`==`**

```java
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }

}
```

2. An Instance is Never Equal to `null`

- Java requires that `x.equals(null)` returns `false`
- and you must not throw an exception if the argument is `null`
  - so it looks like we have to check for a `null` argument…

```java
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }


}
```

3. Instances of the Same Type can be Equal

- the implementation of **`equals()`** used in the notes and the textbook is based on the rule that an instance can only be equal to another instance of the same type
- you can find the class of an object using **`Object.getClass()`**

```
public final Class<? extends Object> getClass()
```

- Returns the runtime class of an object.

```java
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }


}
```

Instances with Same State are Equal

- recall that the value of the attributes of an object define the state of the object
  - two instances are equal if all of their attributes are equal

- unfortunately, we cannot yet retrieve the attributes of the parameter **`obj`** because it is declared to be an **`Object`** in the method signature
  - we need a cast

```java
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  Complex other = (Complex) obj;


}
```

Instances with Same State are Equal

- there is a recipe for checking equality of fields

1. if the field is a primitive type other than **`float`** or **`double`** use **`==`**
2. if the attribute type is **`float`** use **`Float.compare()`**
3. if the attribute type is **`double`** use **`Double.compare()`**
4. if the attribute is an array consider **`Arrays.equals()`**
5. if the attribute is a reference type use **`equals()`**, but beware of attributes that might be null

```java
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  Complex other = (Complex) obj;
  if (Double.compare(this.getReal(), other.getReal()) != 0) {
    return false;
  }
  if (Double.compare(this.getImag(), other.getImag()) != 0) {
    return false;
  }
  return true;
}
```

The `equals()` Contract

- for reference values `equals()` is
  1. reflexive
  2. symmetric
  3. transitive
  4. consistent
  5. must not throw an exception when passed `null`

The `equals()` contract: Reflexivity

1. reflexive :
   - an object is equal to itself
   - **`x.equals(x)`** is **`true`**

The `equals()` contract: Symmetry

2. symmetric :
   - two objects must agree on whether they are equal
   - **`x.equals(y)`** is **`true`** if and only if **`y.equals(x)`** is **`true`**

The `equals()` contract: Transitivity

3. transitive :
   - if a first object is equal to a second, and the second object is equal to a third, then the first object must be equal to the third
   - if
     `x.equals(y)` is **`true`**
     and
     `y.equals(z)` is **`true`**
     then
     `x.equals(z)` must be **`true`**

The `equals()` contract: Consistency

4. consistent :
   - repeatedly comparing two objects yields the same result (assuming the state of the objects does not change)

The **`equals()`** contract: Non-nullity

5. **`x.equals(null)`** is always **`false`** and never does not throw an exception

The **`equals()`** contract and **`getClass()`**

- using **`getClass()`** makes it relatively easy to ensure that the **`equals()`** contract is obeyed
  - e.g., symmetry and transitivity are easy to ensure
- however, using **`getClass()`** means that your **`equals()`** method won't work as expected in inheritance hierarchies
  - more on this when we talk about inheritance

One more thing regarding **`equals()`**

- if you override **`equals()`** you must override **`hashCode()`**
  - otherwise, the hashed containers won't work properly

- we will see how to implement **`hashCode()`** in the next lecture or so