Non-static classes

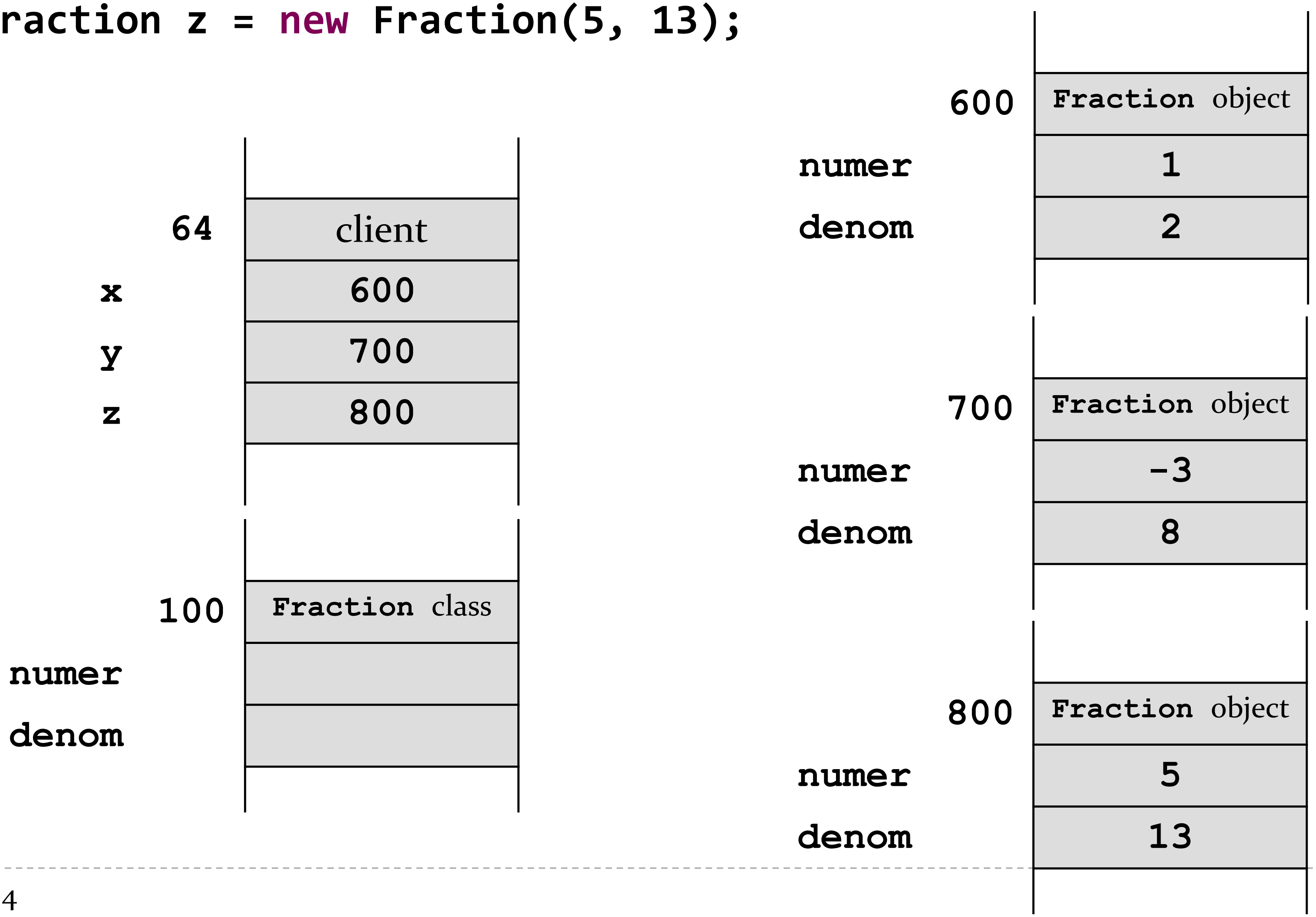
Non-static classes

- a utility class has features (fields and methods) that are all static
  - all features belong to the class
    - therefore, you do not need objects to use those features
      - □ a well implemented utility class should have a single, empty private constructor to prevent the creation of objects
- most Java classes are *not* utility classes
  - they are intended to be used to create to objects
  - each object has its own copy of all non-static fields
  - it is useful to imagine that each object has its own copy of all non-static methods

Why objects?

- each object has its own copy of all non-static fields
  - this allows objects to have their own *state*
    - in Java the state of an object is the set of current values of all of its non-static fields
    - e.g., we can create multiple **`Fraction`** objects that all represent different fraction values

```
Fraction x = new Fraction(1, 2);
Fraction y = new Fraction(-3, 8);
Fraction z = new Fraction(5, 13);
```



64 client

x 600

y 700

z 800

100 Fraction class

numer

denom

600 Fraction object

numer 1

denom 2

700 Fraction object

numer -3

denom 8

800 Fraction object

numer 5

denom 13

Value Type Classes

- a *value type* is a class that represents a value
  - examples of values: name, date, colour, mathematical vector
  - Java examples: **`String`**, **`Date`**, **`Integer`**

- the objects created from a value type class can be:
  - mutable: the state of the object can change
    - **`Date`**
  - immutable: the state of the object is constant once it is created
    - **`String`**, **`Integer`** (and all of the other primitive wrapper classes)

Imaginary numbers

- imaginary numbers occur when you try to take the square root of a negative value
  - for example, $\sqrt{-1}$ has no value in the set of real numbers
- mathematicians have found that it is very useful to say that there exists some number (not real) that when squared is equal to -1
  - this value is usually given the symbol i or j and is called the *imaginary unit*

$$i^2 = -1$$

Imaginary numbers

- an imaginary number is any real valued number multiplied by i

$3i$	$(3i)^2 = -9$
$-3i$	$(-3i)^2 = -9$
$2.5i$	$(2.5i)^2 = -6.25$
$0.01i$	$(0.01i)^2 = -0.0001$

Complex numbers

- a complex number occurs when you add a real number and an imaginary number
  - e.g., $(7 + 2i)$ is a complex number

- the *imaginary part* of a complex number is the imaginary number
  - e.g, the imaginary part of $(7 + 2i)$ is $2i$

- the *real part* of a complex number is the real number (that was added to the imaginary part)
  - e.g, the imaginary part of $(7 + 2i)$ is 7

Complex numbers

- more generally, we say that a complex number is a number that can be written as

$$a + bi$$

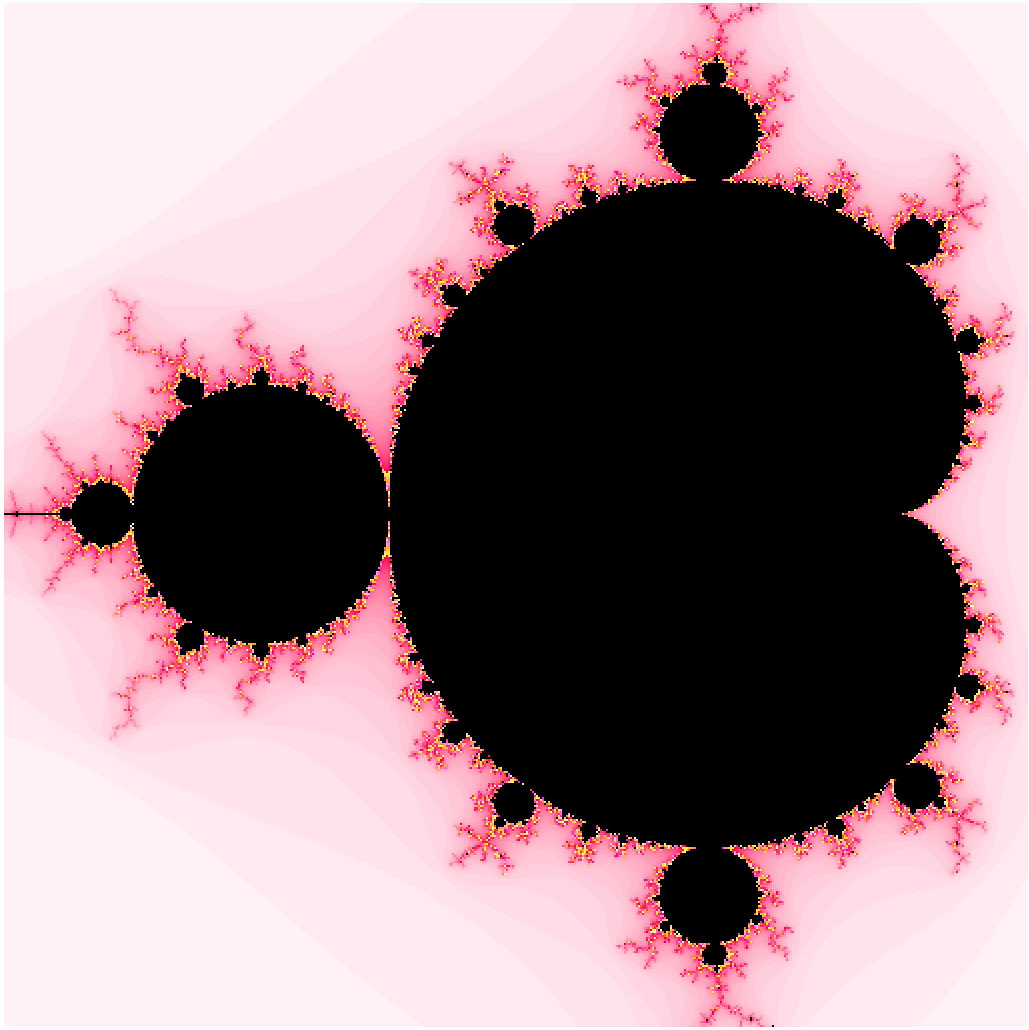
where a and b are real numbers and i is the imaginary unit

Why study complex numbers?

- applications
  - any scientific or engineering application that involves vibrations, waves, or signals probably
  - complex analysis in mathematics
  - quantum mechanics in physics and chemistry
  - differential equations
  - many others
- from an EECS1030 perspective
  - easily implemented value type

- also, you can make pretty pictures

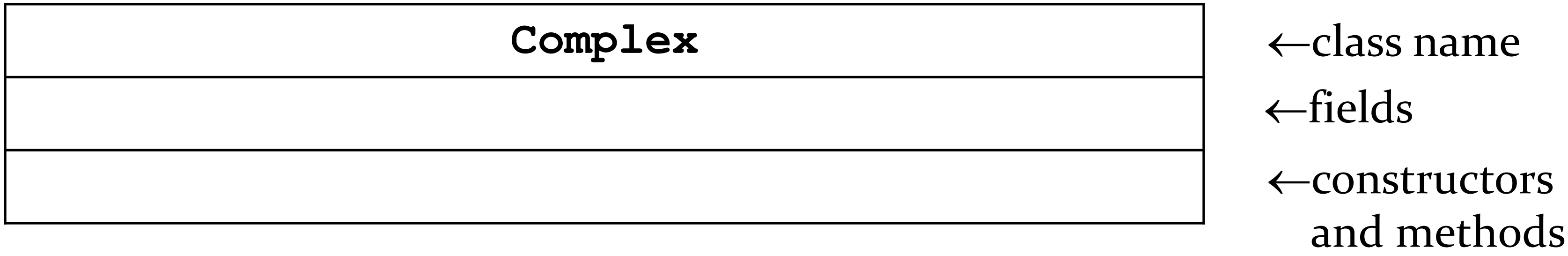
Mandelbrot set



Class **`Complex`**

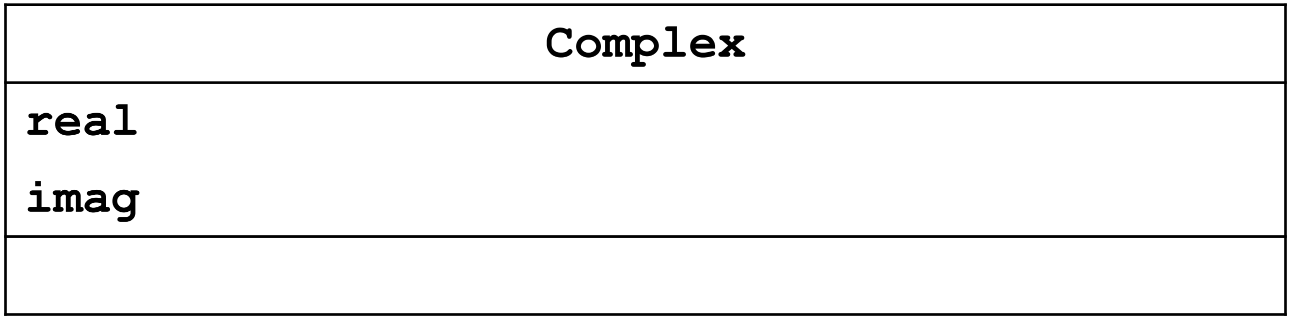
- when creating a class you should first analyze the requirements of the class
  - what fields does each object need?
  - how do you construct an object?
  - what methods should each object provide?

- this information can be summarized in a UML class diagram



Class **`Complex`**

- what fields does each **`Complex`** object need?
  - a field to represent the real part
  - a field to represent the complex part



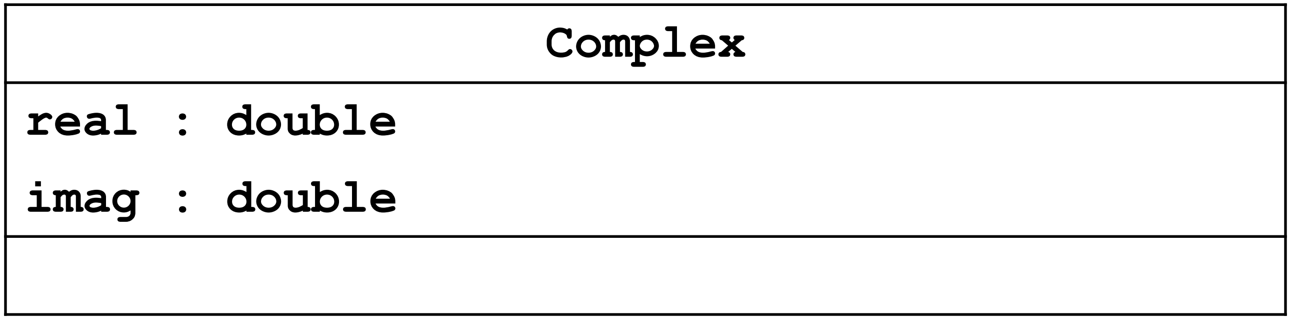
`Complex`

`real`

`imag`

Class **`Complex`**

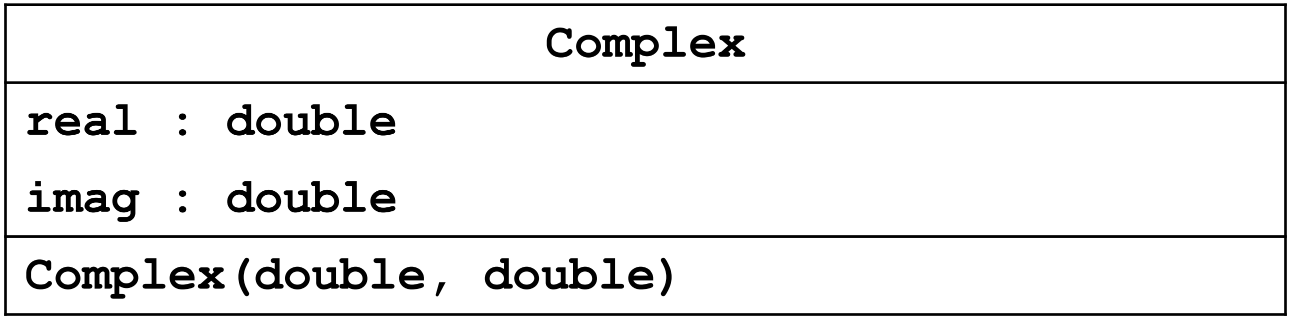
- what are appropriate types for the fields?
  - the real part
    - **`double`**
  - the complex part
    - **`double`**



```
Complex
real : double
imag : double
```

Class **`Complex`**

- how do you create a **`Complex`** object?
  - by specifying the values of the real and imaginary parts



```
Complex
real : double
imag : double
Complex(double, double)
```

What operations?

- there are many possible operations involving complex numbers
  - implementing them all is impractical for our current purposes
- we will consider the following
  - complex conjugate
  - absolute value
  - addition
  - multiplication

Complex conjugate

- to compute the complex conjugate of a complex number, simply change the sign of the imaginary part
  - the complex conjugate of

$$a + bi$$

is

$$a + (-b)i$$

- note that the result is a complex number

Absolute value

- the absolute value or magnitude of

$$a + bi$$

is

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

- note that the result is a real number

Addition

- addition of two complex number is defined as

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

- that is, you sum the real parts and sum the imaginary parts separately

- note that the result is a complex number

Multiplication

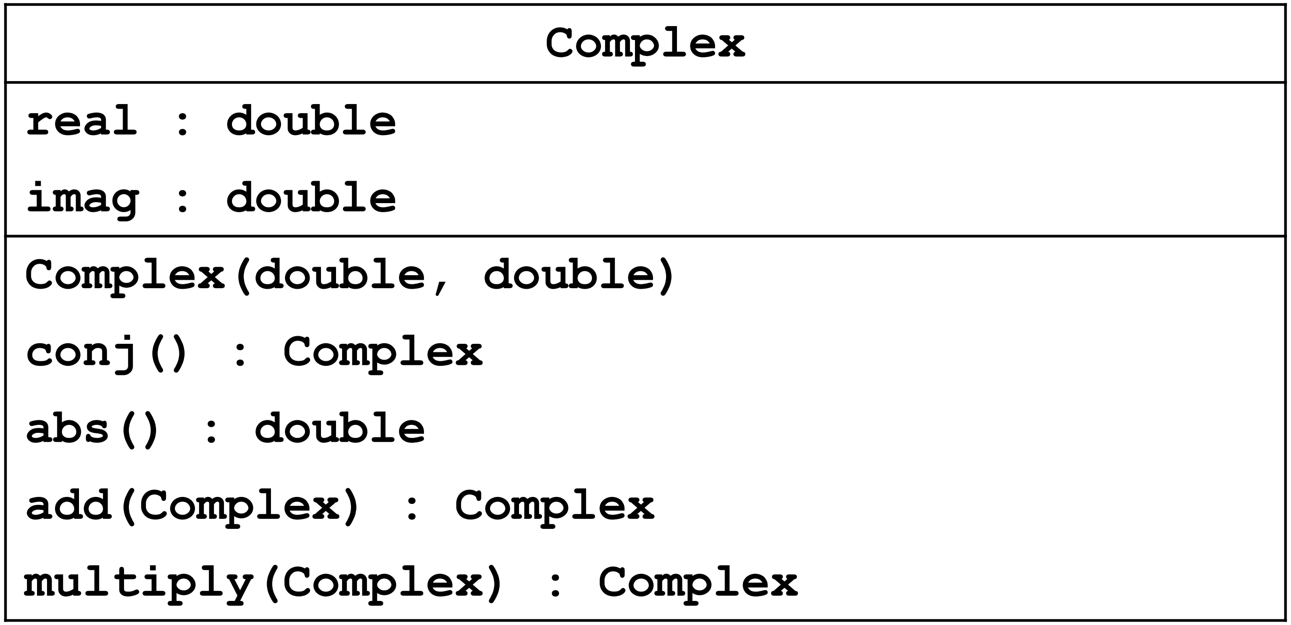
- multiplication of two complex number is defined as

$$(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$$

- you can easily derive this
- note that the result is a complex number

Class **`Complex`**

- what methods should **`Complex`** provide?



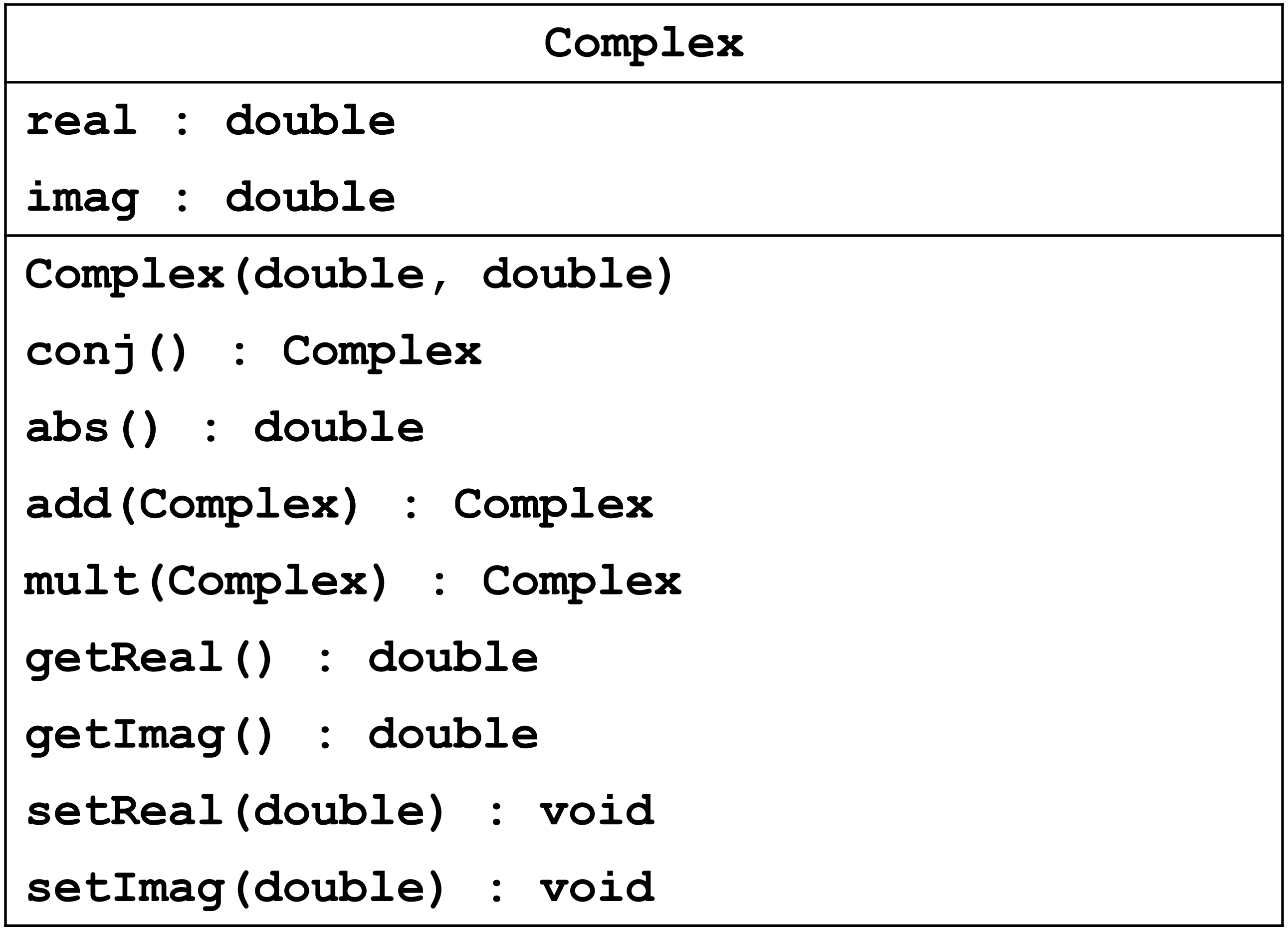
```
Complex
real : double
imag : double
Complex(double, double)
conj() : Complex
abs() : double
add(Complex) : Complex
multiply(Complex) : Complex
```

Class `Complex`

- what other methods might a client find useful?
  - get the value of the real part
  - get the value of the imaginary part
  - set the value of the real part
  - set the value of the imaginary part

- methods that get information about the state of an object are called *accessor methods*
- methods that change the state of an object are called *mutator methods*

Class **Complex**



```
Complex
real : double
imag : double
Complex(double, double)
conj() : Complex
abs() : double
add(Complex) : Complex
mult(Complex) : Complex
getReal() : double
getImag() : double
setReal(double) : void
setImag(double) : void
```

Class **`Complex`**

- there are three more important methods, but we will look at these later

Class and fields

- start by creating the class and adding the fields

- if you decide to organize your classes into packages, then you should first create the appropriate package

```java
public class Complex {

  private double real;
  private double imag;

}
```

Class and fields

- notice that the class is marked **`public`**
  - this means that the class is visible to all clients
- notice that the fields are marked **`private`**
  - this means that the fields are visible only inside of the class

Constructor

- we can now implement the constructor
- a constructor:
  - must have the same name as the class
  - never returns a value (not even void)
    - constructors are not methods
  - can have zero or more parameters
- 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`**

```
public class Complex {

  private double real;
  private double imag;

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

`this`

- every constructor and non-static method has a parameter that does not explicitly appear in the parameter list
- the parameter is called an implicit parameter and its name in Java is always **`this`**
- in a constructor, **`this`** is a reference to the object currently being constructed

this

- in our constructor

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

`this.real` refers to the field named `real`

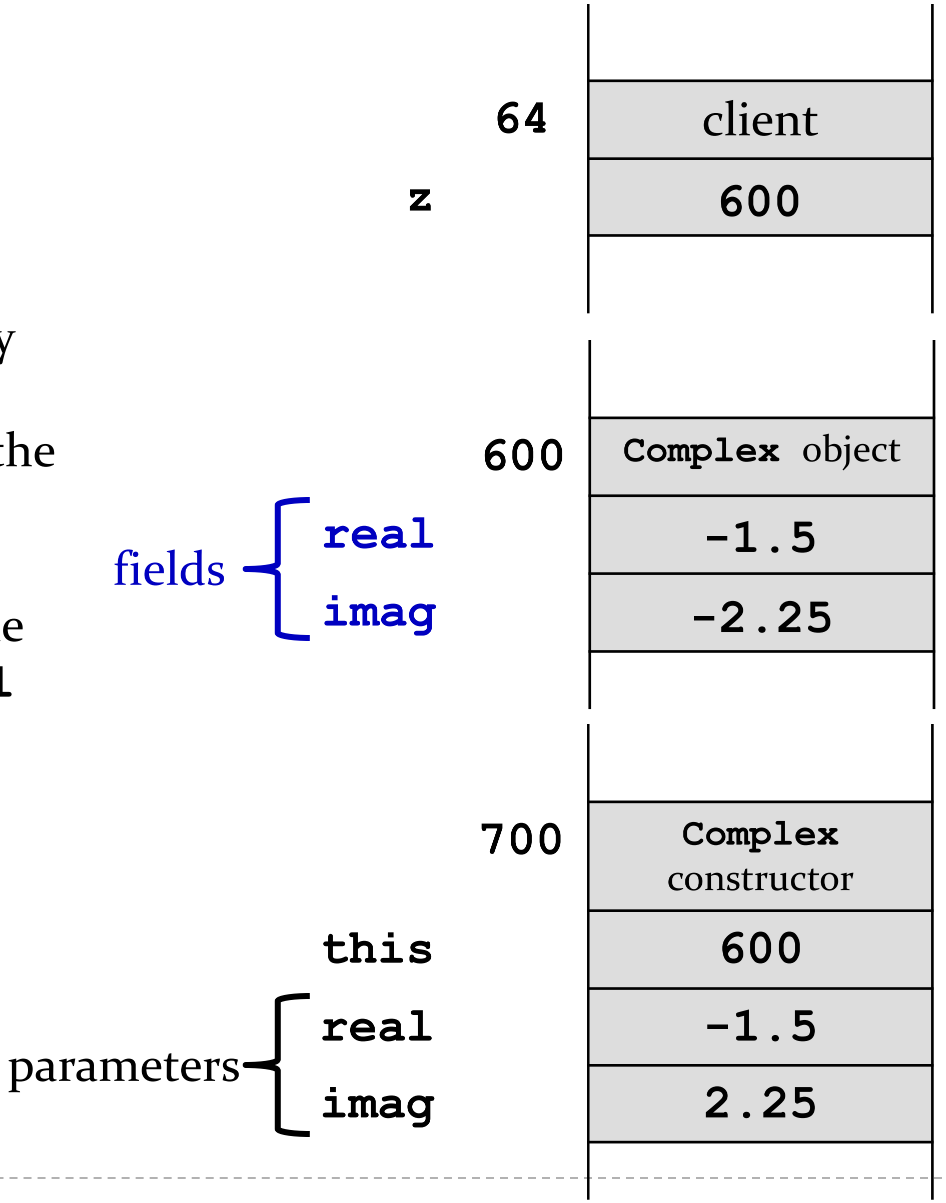
`this.imag` refers to the field named `imag`

`real` refers to the parameter named `real`

`imag` refers to the parameter name `imag`

```
Complex z = new Complex(-1.5, 2.25);
```

1. **`new`** allocates memory for a **`Complex`** object
2. the **`Complex`** constructor is invoked by passing the memory address of the object and the arguments **`-1.5`** and **`2.25`** to the constructor
3. the constructor runs, setting the values of the fields **`this.real`** and **`this.imag`**
4. the value of **`z`** is set to the memory address of the constructed object



this

- in our constructor

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

there are parameters with the same names as fields

- when this occurs, the parameter has precedence over the field
  - we say that the parameter *shadows* the field
  - when shadowing occurs you must use **`this`** to refer to the field