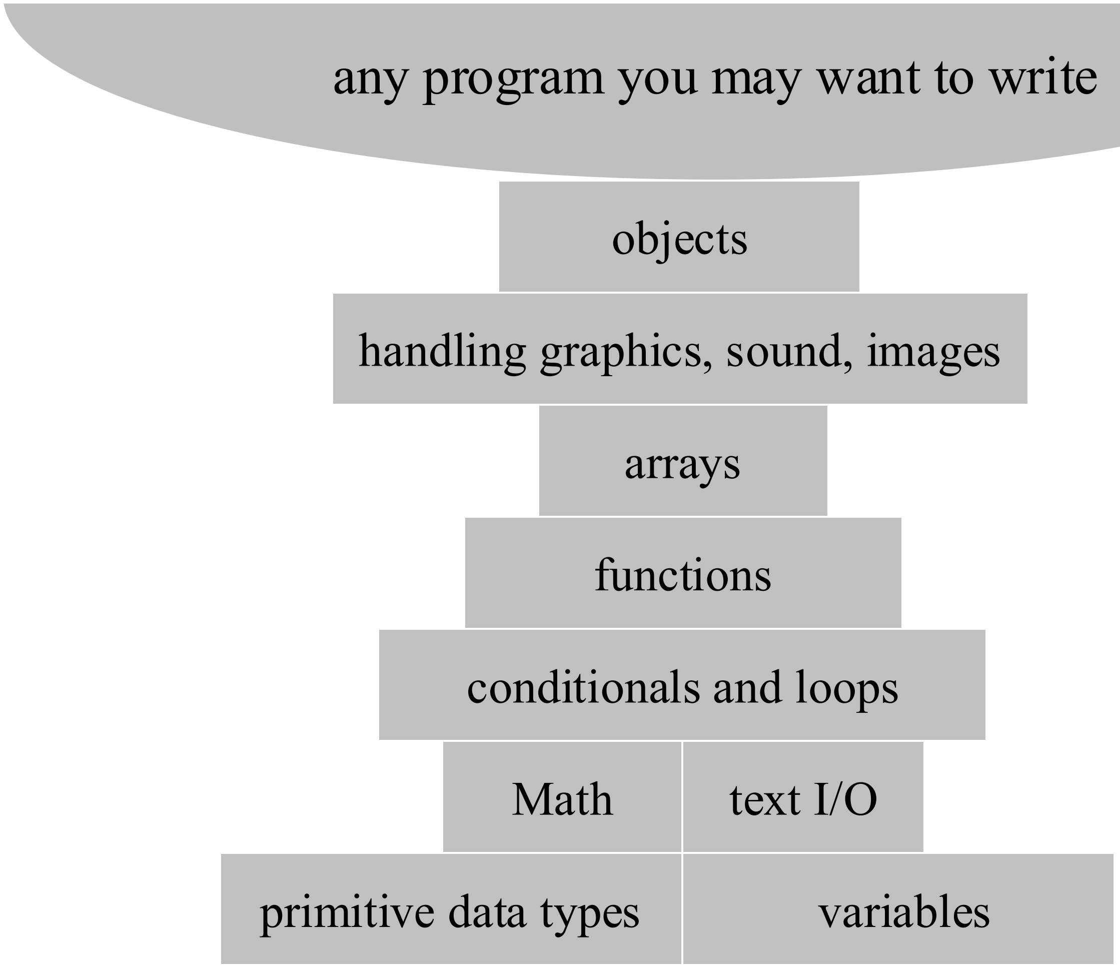
Introduction to Computer Science

Reichman University

Lecture 8-2

Object-Oriented Programming I

The big picture



any program you may want to write

objects

handling graphics, sound, images

arrays

functions

conditionals and loops

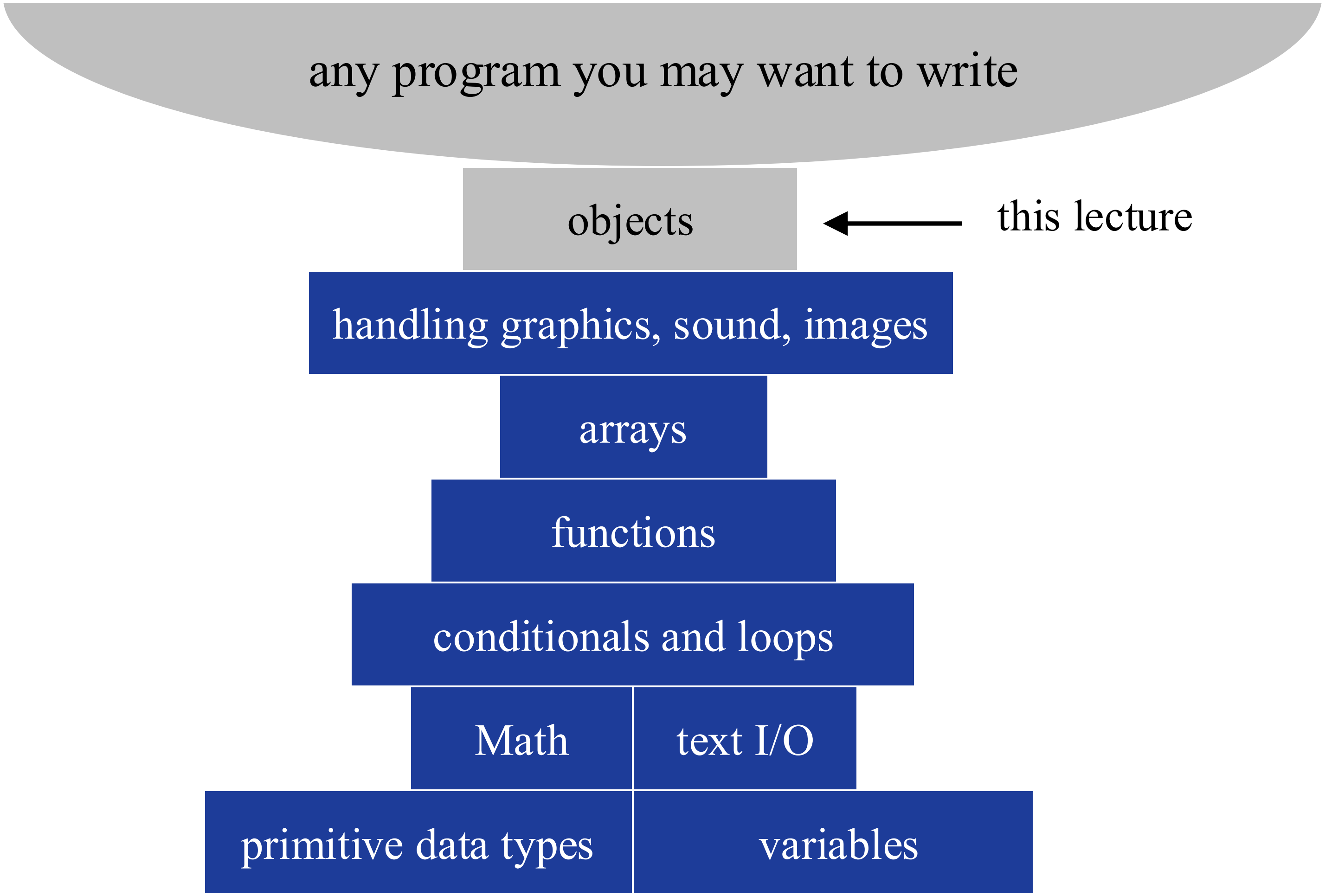
Math

text I/O

primitive data types

variables

The big picture



any program you may want to write

objects

this lecture

handling graphics, sound, images

arrays

functions

conditionals and loops

Math

text I/O

primitive data types

variables

Objects as types

The basic type system of Java

`int`, `char`, `double`, `boolean`, ... (8 primitive types)

Real life is much richer:

- `Fraction:` numbers like 1/2, 1/3, 5/6 , ...
- `Date:` values like 12/5/2014, 28/7/1995, ...
- `Point:` pairs of numbers, like (3,5), (-17,5), ...
- `Set:` collections of unique elements without order
- `BigInteger:` big ints, like 762543423543265378483488834 34…
- `Color:` RGB triplets like (212, 17, 15), , ...
- `BankAccount:` Financial data relevant to bank accounts
- . . .

Type = a set of structured values,
and operations on these values

The structured values are called *objects*

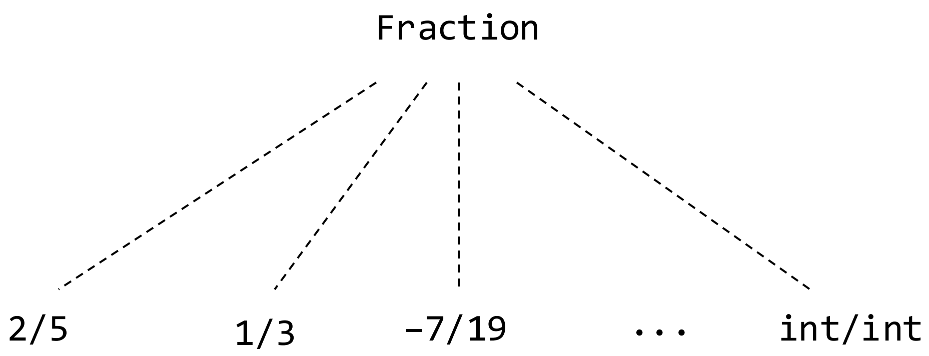
Object-oriented programming (OOP)

A programming technique for *representing* and *using* objects.

Fractions

In OOP, objects are derived from, and handled by, classes

Class (example)



`Fraction`

`2/5` `1/3` `-7/19` `...` `int/int`

Each fraction is an object / instance of type `Fraction`

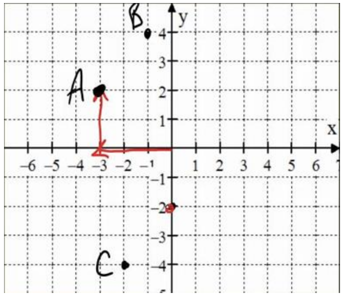
The `Fraction` class is designed to provide Fraction-oriented operations: *add*, *multiply*, *invert, divide*, ...

Class examples

$$\frac{1}{2} + \frac{1}{3}$$

`Fraction`

`BankAccount`



`Point`

Fractions

Bob: I am writing a program that needs to compute expressions
like `(1/2 + 1/3)` and `(5/17 * 19/23)`, without losing any precision

Alice: Lucky you! A while ago I developed a class – I called it `Fraction` –
that enables creating and manipulating such objects

Fractions

Bob: I am writing a program that needs to compute expressions like `(1/2 + 1/3)` and `(5/17 * 19/23)`

Alice: Lucky you! A while ago I developed a class – I called it `Fraction` – that enables creating and manipulating such objects

Bob: Great… How can I use these fractions in my programs?

Alice: Easy – I'll give you my compiled `Fraction.class` file, and then you can simply call its methods, as needed

Bob: And how will I know how to call / use these methods?

Alice: I will also give you the `Fraction` class API.
I think that I documented the class well enough, so that people will be able to use it without bothering me.

Fraction abstraction (API / class skeleton)

`Fraction` class skeleton / API

```
/** Represents a signed fraction, like 2/3 or -1/5. */                          API
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
     *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

Constructors

`Fraction` class skeleton / API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
     *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

Constructor

A *method* that constructs, and returns, a new object;

OOP convention: The name of a constructor is the name of the class

Constructors

`Fraction` class skeleton / API

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)
    // More Fraction methods
}
```

Constructor

A *method* that constructs, and returns, a new object;

OOP convention: The name of a constructor is the name of the class

```
// client code (in any class)
int x = 17;
...
Fraction a = new Fraction(2,5);
Fraction b = new Fraction(4,8);
...
```

Constructor calls

Used to create new objects

Object variables

`Fraction` class skeleton / API

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)
    // More Fraction methods
}
```

<u>Constructor</u>

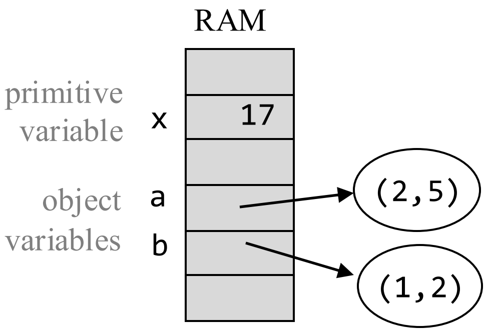
A *method* that constructs, and returns, a new object;

OOP convention: The name of a constructor is the name of the class

```
// client code (in any class)
int x = 17;
...
Fraction a = new Fraction(2,5);
Fraction b = new Fraction(4,8);
...
```

<u>Constructor calls</u>

Used to create new objects



<u>Object variables</u> (like `a` and `b`)

Object variable hold *addresses in memory*

Like array variables, they are also called *pointers, or references.*

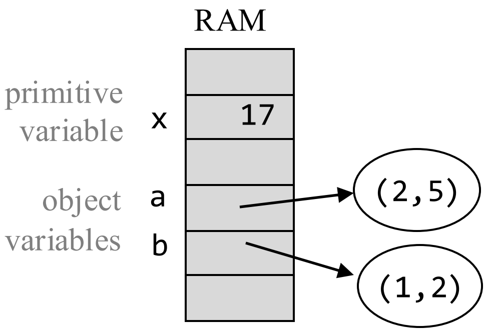
Constructors

`Fraction` class skeleton / API

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

   // More Fraction methods
}
```

```
// client code (in any class)
int x = 17;
...
Fraction a = new Fraction(2,5);
Fraction b = new Fraction(4,8);
...
```



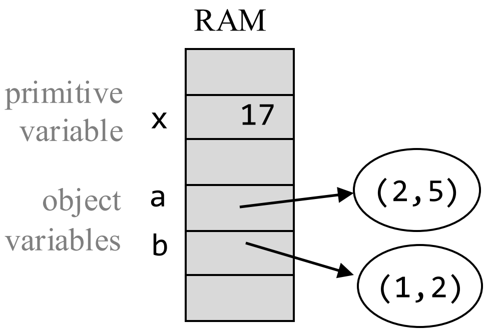
Anatomy of `Fraction a = new Fraction(2,5)`

Constructors

`Fraction` class skeleton / API

```
/** Represents a signed fraction, like 2/3 or -1/5. */                API
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    // More Fraction methods
}
```

```
// client code (in any class)
int x = 17;
...
Fraction a = new Fraction(2,5);
Fraction b = new Fraction(4,8);
...
```



Anatomy of `Fraction a = new Fraction(2,5)`

1. The client code calls the constructor, using the statement **`new`** *ClassName*(*arguments*)

2. The constructor's code (not seen here):

   - Causes the OS to allocate a free memory block for storing the new object's data
   - Typically does some object initialization work
   - Returns the base address of the allocated memory block to the caller

   The API hides all these implementation details from the caller

3. This value is then stored in the object variable `a`.

Fraction abstraction (API / class skeleton)

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

✓  /** Constructs a fraction from the two integers */
   public Fraction(int numerator, int denominator)

   /** Returns the numerator of this fraction */
   public int getNumerator()

   /** Returns the denominator of this fraction */
   public int getDenominator()

   /** Returns a fraction which is the sum of this fraction and the other one. */
   public Fraction add(Fraction other)

   /** Returns a fraction which is the product of this fraction and the other one. */
   public Fraction multiply(Fraction other)

    /** Returns the inverse of this fraction. */
    public Fraction invert()

   /** Returns a textual representation of this fraction,
    *  in the form "numerator/denominator". */
   public String toString()

   // More Fraction methods
}
```

Fraction abstraction

The toString() method

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
     *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

Fraction abstraction

The `toString()` method

`Fraction` API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)
    ...

    /** Returns a textual representation of this fraction,
     *  in the form "numerator/denominator". */
    public String toString()
    ...
}
```

`toString`

- A method that returns a textual representation of the current object
- Basic debugging service
- It's the responsibility of the class designer to write a `toString` method
- Why? Because programmers who write client code that uses the class expect to be able to use a `toString` method.

Client code (example)

```
public class FractionDemo {
   public static void main(String args[]) {
      Fraction a = new Fraction(2,5);
      Fraction b = new Fraction(3,6);
      System.out.println("a = " + a.toString());
      System.out.println("b = " + b);
   }
}
```

```
% java FractionDemo
a = 2/5
b = 1/2
```

Convention: When an object variable is cast as a string, Java automatically invokes the `toString()` method on this object.

Fraction abstraction (API / class skeleton)

`Fraction` API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

✓   /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
✓    *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

`Fraction` abstraction

Accessor / getter methods

`Fraction` API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
      *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

`Fraction` abstraction

Accessors / Getters

Provide access to the objects' data;
Their names typically start with "`get`".

Accessor / getter methods

Fraction API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */                API
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    ...
}
```

Fraction abstraction

Accessors / Getters

Provide access to the objects' data;
Their names typically start with "get".

```
// client code (in any class)
...
Fraction a = new Fraction(2,5);
System.out.println("The numerator of " + a + " is " + a.getNumerator());
System.out.println("The denominator of " + a + " is " + a.getDenominator());
...
```

```
% java FractionDemo
The numerator of 2/5 is 2
The denominator of 2/5 is 5
```

Observations (from this example):

- Methods are *functions that operate on objects*
- For example, the method call `a.getNumerator` applies the `getNumerator` method to the object `a`.

Fraction abstraction (API / class skeleton)

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

✓   /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

✓   /** Returns the numerator of this fraction */
    public int getNumerator()

✓   /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
✓    *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

Fraction abstraction

Fraction methods

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    /** Returns the numerator of this fraction */
    public int getNumerator()

    /** Returns the denominator of this fraction */
    public int getDenominator()

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    /** Returns a textual representation of this fraction,
     *  in the form "numerator/denominator". */
    public String toString()

    // More Fraction methods
}
```

Fraction abstraction

Fraction arithmetic methods

The methods that we saw so far (constructors, `toString`, accessors) typically appear in *any* class;

We now turn to illustrate some *domain-specific* methods.

Fraction methods

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    ...
}
```

Fraction abstraction

Fraction – arithmetic methods

Observations (from this API):

- Objects can be passed to methods as arguments
- Objects can be returned by methods as return values
- What is the meaning of “this fraction”?

Using methods (a client perspective)

`Fraction` API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the
    public Fraction multiply(Fract

     /** Returns the inverse of this frac
     public Fraction invert()

    ...
}
```

```
// client code (in any class)
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
Fraction sum = a.add(b);
System.out.println(a + " + " +  b + " = " + sum);
...
```

```
% java FractionDemo

1/3 + 1/2 = 5/6
```

Using methods (a client perspective)

Fraction API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the
    public Fraction multiply(Fract

     /** Returns the inverse of this frac
     public Fraction invert()

    ...
}
```

In the class documentation, “this” refers to the object on which the method was called.

```
// client code (in any class)
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
Fraction sum = a.add(b);
System.out.println(a + " + " +  b + " = " + sum);
...
```

```
% java FractionDemo

1/3 + 1/2 = 5/6
```

Method calling:

objectVariable`.`*methodName*(*arguments*)

Using methods (a client perspective)

Fraction API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the
    public Fraction multiply(Frac

     /** Returns the inverse of this frac
     public Fraction invert()

    ...
}
```

```
// client code (in any class)
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
Fraction sum = a.add(b);
System.out.println(a + " + "  + b + " = " + sum);
...
```

```
% java FractionDemo

1/3 + 1/2 = 5/6
```

Anatomy of `Fraction sum = a.add(b)`

1. Calls the `add` method on object `a`

2. The `add` method operates on the `this` object (here, `a`), and on the `other` object (here, `b`)

3. The `add` method returns a `Fraction` object

`sum` ends up pointing to the returned object.

Using methods (a client perspective)

`Fraction` API / class skeleton

API

```
/** Represents a signed fraction, like 2/3 or -1/5. */
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inv
     public Fraction

    ...
}
```

Another example

```
// client code (in any class)
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(3,7);
Fraction c = new Fraction(2,5);
// Computes a * (b + c):
Fraction d = a.multiply(b.add(c));
System.out.println(a + " * (" + b + " + " + c + ") = " + d);
...
```

```
% java FractionDemo

1/3 * (3/7 + 2/5) = 29/105
```

Recap: Class design

`Fraction` API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */                                API
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    ...
}
```

<u>Software engineering</u>

The architect plans a class design / API that meets the domain requirements

Recap: Class design

`Fraction` API / class skeleton

```
/** Represents a signed fraction, like 2/3 or -1/5. */                    API
public class Fraction {

    /** Constructs a fraction from the two integers */
    public Fraction(int numerator, int denominator)

    ...

    /** Returns a fraction which is the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns a fraction which is the product of this fraction and the other one. */
    public Fraction multiply(Fraction other)

     /** Returns the inverse of this fraction. */
     public Fraction invert()

    ...
}
```

Software engineering

The architect plans a class design / API that meets the domain requirements

Domain requirements in this example:

Handling fractions

Fractions = the set of all pairs a/b so that a and b are integers

The fractions set is closed under addition, multiplication, and inversion

Design decisions

The `Fraction` constructor is designed to take any two integers, and return a `Fraction` object

The `add`, `multiply`, and `invert` methods are designed to operate on `Fraction` objects, and return `Fraction` objects.

Recap: key OO concepts

Objects

Instances of a class

For example, 1/2 and 2/3 are *instances* of the `Fraction` class

Object variable: a variable that refers to an object, acting as the object's "handle"

Methods: Operate on objects (unlike *functions*, that operate on no particular object)

- Constructors: Create new objects
- `toString`: Returns a textual representations of the given object
- Getters / accessors: Provide access to the object's data
- Other methods: Capture the domain requirements.

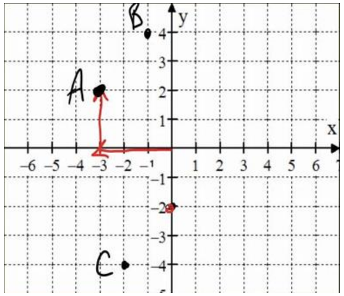
Class design is an *acquired art*;

It takes experience, and seeing many class examples (*design patterns*)

Lecture plan

`Fraction`

`BankAccount`



`Point`

BankAccount

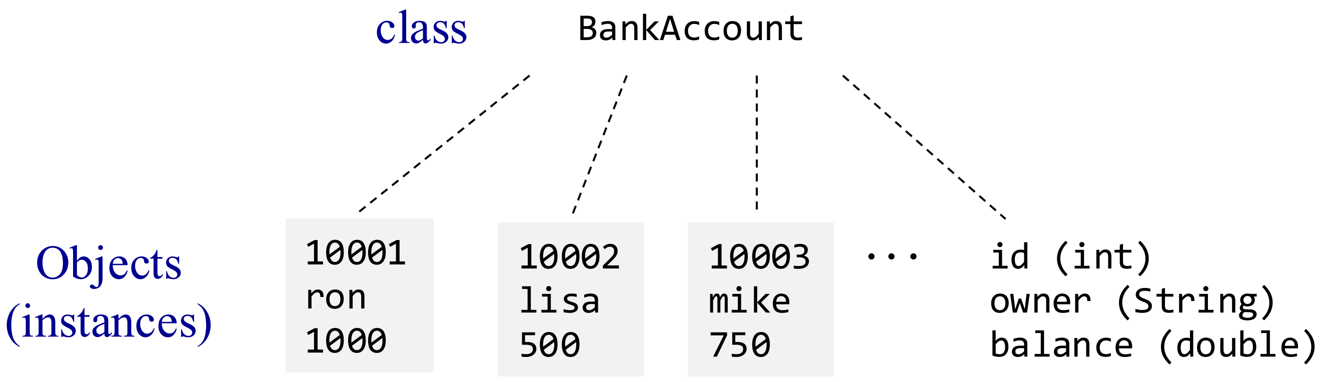
Bob: I just started working here... My boss wants me to develop a program that finds bank customers who have negative account balances. How do I get started?

Alice: Well, around here we all work with a class named `BankAccount`. All bank accounts are objects of this class.

BankAccount

Bob: I just started working here... My boss wants me to develop a program that finds bank customers who have negative account balances. How do I get started?

Alice: Well, around here we all work with a class named `BankAccount`. All bank accounts are objects of this class.



Bob: Thanks! So How do I get started?

Alice: Take a look at the `BankAccount` API, and take it from there.

Bank account abstraction

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *   The account id is generated automatically by the constructor.
     *   The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Constructors

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Constructors

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    // More BankAccount methods
    ...
```

```
// Client code (can appear in any class, like BankAccountDemo)
...
// Alice opens an account with 1000 balance
BankAccount aliceAcc = new BankAccount("Alice", 1000);

// Bob opens an account with 0 balance
BankAccount bobAcc = new BankAccount("Bob");

System.out.println(aliceAcc);
System.out.println(bobAcc);
...
```

```
% java BankAcountDemo

1 Alice 1000
2 Bob 0
```

<u>Constructor overloading</u>

- Common OOP practice
- Typically, we need more than one way to create objects
- Each way can be supported by a different constructor.

Bank account abstraction

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Accessors / getters

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Accessors / getters

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    ...
}
```

```
// Client code
...
// Ben opens an account with 5000 balance
BankAccount benAcc = new BankAccount("Ben", 5000);
...
System.out.println("Current balance of Ben: " + benAcc.getBalance());
...
```

```
% java BankAcountDemo
Current balance of Ben: 5000
```

Bank account abstraction

API

```java
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Banking methods

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is generated automatically by the constructor.
     *  The first constructed account has id=1, the second id=2, and so on. */
    public BankAccount(String owner, double balance)

    /** Constructs a new bank account with the given owner and a zero balance. */
    public BankAccount(String owner)

    /** Returns the id of this account. */
    public int getId()

    /** Returns the owner of this account. */
    public String getOwner()

    /** Returns the balance of this account. */
    public double getBalance()

    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)

    /** Returns the data of this bank account
    public String toString()
    ...
}
```

Banking methods

Banking methods

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {
    // Constructors, getters, setters, typically come here
    /** Handles a deposit of sum to this account. */
    public void deposit(double sum)

    /** Handles a withdrawal of sum from this account. */
    public void withdraw(double sum)

    /** Handles a transfer of sum from this account to the other account. */
    public void transferTo(BankAccount other, double sum)
    ...
}
```

Banking methods

```
// Typical banking scenario 1:
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
System.out.println(aliceAcc); System.out.println(bobAcc);

// Typical banking scenario 2:
aliceAcc.withdraw(200);
bobAcc.deposit(500);
System.out.println(aliceAcc); System.out.println(bobAcc);

// Typical banking scenario 3:
aliceAcc.transferTo(bobAcc, 400);
System.out.println(aliceAcc); System.out.println(bobAcc);
```

```
% java BankAcountDemo
1 Alice 1000
2 Bob 0

1 Alice 800
2 Bob 500

1 Alice 400
2 Bob 900
```

Banking methods

API

```
/** Represents a bank account.
 *  A bank account has an id (an int), an owner (a string), and a balance (a double). */
public class BankAccount {
       // Constructors, getters, setters, typically come here
       /** Handles a deposit of sum to this account. */
       public void deposit(double sum)

       /** Handles a withdrawal of sum from this account. */
       public void withdraw(double sum)

       /** Handles a transfer of sum from this account to the other account. */
       public void transferTo(BankAccount other, double sum)
       ...
}
```

Technically, such methods are called *mutators*, since they mutate (change) the *state* (data) of objects;

We say that `BankAccount` objects are *mutable*

Should a class be *mutable*? *immutable*?

Decided by the class architect, according to the domain requirements

For example:

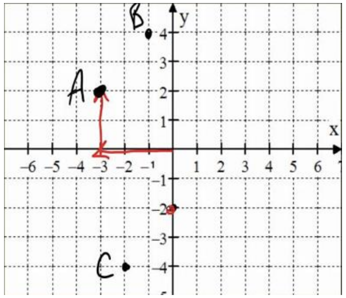
- Fractions are *immutable*: Once a fraction is created, it's impossible to change it
- Bank accounts are *mutable*: We have to change their balances,
  and other account data.

Lecture plan

$$\frac{1}{2} + \frac{1}{3}$$

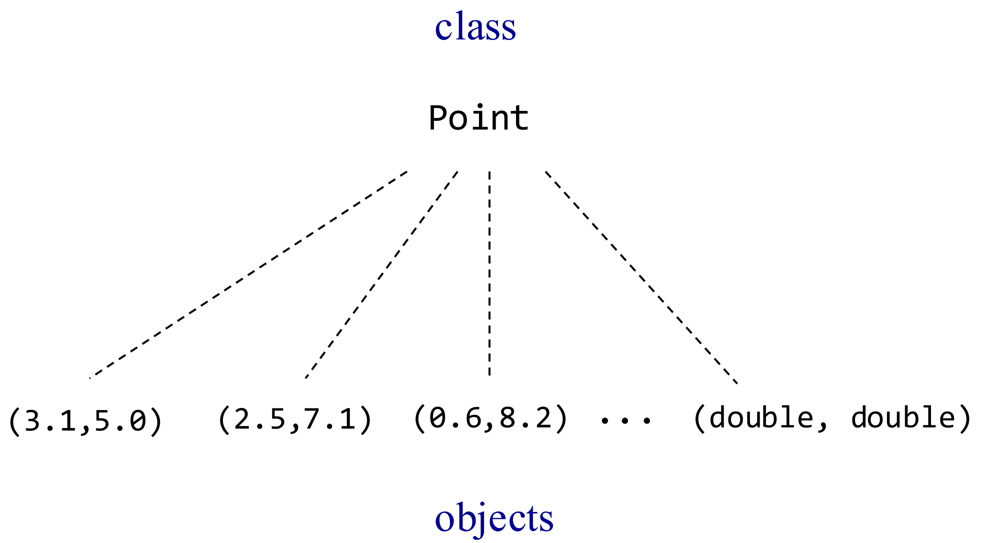
Fraction

BankAccount



Point

Point



class

Point

(3.1,5.0) (2.5,7.1) (0.6,8.2) ... (double, double)

objects

Point: abstraction (API) and usage

API

```
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

    /** Constructs a point from the two doubles */
    public Point(double x, double y)

    /** Returns a textual representation of this point as "(x,y)" */
    public String toString()

    /** Returns the Euclidean distance between this point
     *  and the other one */
    public double distanceTo(Point other) {

    /** Returns a point which is the vector
     *  addition of this point and the other one */
    public Point add(Point other) {

    /** Draws this point in a graphical 2D plain */
    public void draw()

    /** Draws a line between this point and the other one */
    public void drawLineTo(Point other)

    ...
}
```

Client code (PointDemo)

```
...
// Creates two points and prints their addition
Point p1 = new Point(0.1,0.1);
Point p2 = new Point(0.2,0.2);
System.out.println(p1 + " + " + p2 +
                   " = " + p1.add(p2));
```

output:

```
(0.1,0.1) + (0.2,0.2) = (0.3,0.3)
```

Point: abstraction (API) and usage

API

```java
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

    /** Constructs a point from the two doubles */
    public Point(double x, double y)

    /** Returns a textual representation of this point as "(x,y)" */
    public String toString()

    /** Returns the Euclidean distance between this point
     *   and the other one */
    public double distanceTo(Point other) {

    /** Returns a point which is the vector
      *  addition of this point and the other one */
    public Point add(Point other) {

    /** Draws this point in a graphical 2D plain */
    public void draw()

    /** Draws a line between this point and the other one */
    public void drawLineTo(Point other)

    ...
}
```

Client code (PointDemo)

```java
...
// Creates two points, draws them, and draws
// a line that connects them
Point p1 = new Point(0.1,0.1);
Point p2 = new Point(0.8,0.8);
p1.draw();
p2.draw();
p1.drawLineTo(p2);
```

Point: abstraction (API) and usage

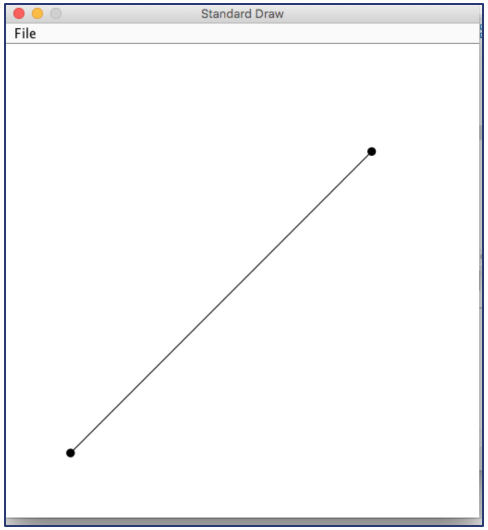
API

```
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

        /** Constructs a point from the two doubles */
       public Point(double x, double y)

       /** Returns a textual representation of this point as "(x,y)" */
       public String toString()

       /** Returns the Euclidean distance between this point
        *   and the other one */
       public double distanceTo(Point other) {

       /** Returns a point which is the vector
          *  addition of this point and the other one */
       public Point add(Point other) {

       /** Draws this point in a graphical 2D plain */
       public void draw()

       /** Draws a line between this point and the other one */
       public void drawLineTo(Point other)

       ...
}
```

Client code (PointDemo)

```
...
// Creates two points, draws them, and draws
// a line that connects them
Point p1 = new Point(0.1,0.1);
Point p2 = new Point(0.8,0.8);
p1.draw();
p2.draw();
p1.drawLineTo(p2);
```

output



(It looks like the Point class uses the services of a class like StdDraw).

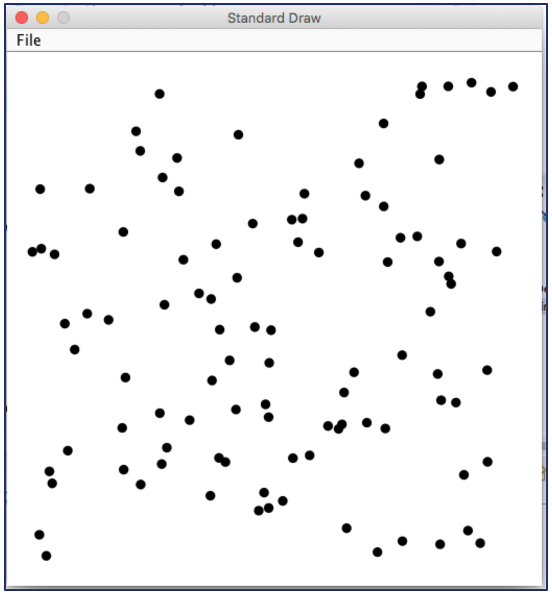
Point: abstraction (API) and usage

API

```
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

     /** Constructs a point from the two doubles */
     public Point(double x, double y)

     /** Returns a textual representation of this point as "(x,y)" */
     public String toString()

     /** Returns the Euclidean distance between this point
      *   and the other one */
     public double distanceTo(Point other) {

     /** Returns a point which is the vector
       *  addition of this point and the other one */
     public Point add(Point other) {

     /** Draws this point in a graphical 2D plain */
     public void draw()

     /** Draws a line between this point and the other one */
     public void drawLineTo(Point other)

     ...
}
```

Client code (PointDemo)

```
...
// Creates an array of random points
int N = 100;
Point[] points = new Point[N];
for (int i = 0; i < N; i++)
    points[i] = new Point(Math.random(),
                          Math.random());
// Draws the points
for (int i = 0; i < N; i++)
    points[i].draw();
```



(Here we make a huge assumption about the canvas, assuming that it is a square whose edge length is 1.0)

Point: abstraction (API) and usage

API

```
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

    /** Constructs a point from the two doubles */
    public Point(double x, double y)

    /** Returns a textual representation of this point as "(x,y)" */
    public String toString()

    /** Returns the Euclidean distance between this point
     *   and the other one */
    public double distanceTo(Point other) {

    /** Returns a point which is the vector
     *   addition of this point and the other one */
    public Point add(Point other) {

    /** Draws this point in a graphical 2D plain */
    public void draw()

    /** Draws a line between this point and the other one */
    public void drawLineTo(Point other)

    ...
}
```

Client code (PointDemo)

```
...
// Creates an array of random points
int N = 100;
Point[] points = new Point[N];
for (int i = 0; i < N; i++)
    points[i] = new Point(Math.random(),
                          Math.random());
// Draws the points
for (int i = 0; i < N; i++)
    points[i].draw();
```

Observations (from the client code):

- Point is a *type*
- Just like we can create and use arrays of *primitive* data types, we can create and use arrays of *object* types.

Point: abstraction (API) and usage

API

```
/** Represents a point in a plain.
 *  A point has x and y coordinates (double values).
 *  Provides algebraic and graphical operations. */
public class Point {

    /** Constructs a point from the two doubles */
    public Point(double x, double y)

    /** Returns a textual representation of this point as "(x,y)" */
    public String toString()

    /** Returns the Euclidean distance between this point
     *   and the other one */
    public double distanceTo(Point other) {

    /** Returns a point which is the vector
     *   addition of this point and the other one */
    public Point add(Point other) {

    /** Draws this point in a graphical 2D plain */
    public void draw()

    /** Draws a line between this point and the other one */
    public void drawLineTo(Point other)

    ...
}
```

Client code (PointDemo)

```
...
// Creates an array of random points
int N = 100;
Point[] points = new Point[N];
for (int i = 0; i < N; i++)
    points[i] = new Point(Math.random(),
                          Math.random());
// Draws the points
for (int i = 0; i < N; i++)
    points[i].draw();
```

Observations (from the class API):

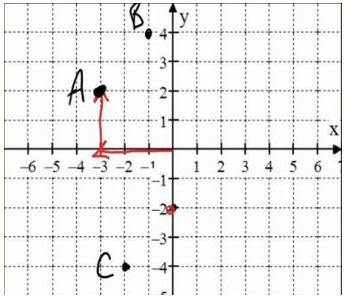
- Confusing class design
- Mixes representation and algebraic operations (abstract, mathematical issues) with drawing issues (specific)
- Perhaps it's better to factor the drawing operations to a separate class.

Recap

$\frac{1}{2} + \frac{1}{3}$

Fraction

BankAccount



Point

This lecture

Abstraction: How to *use objects* in programs

Next lecture

Implementation: How to *build classes* that represent these objects