Lecture 9-1

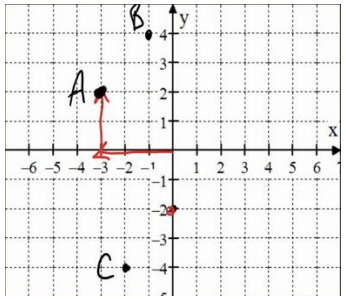
Object-Oriented Programming II

Lecture plan

BankAccount

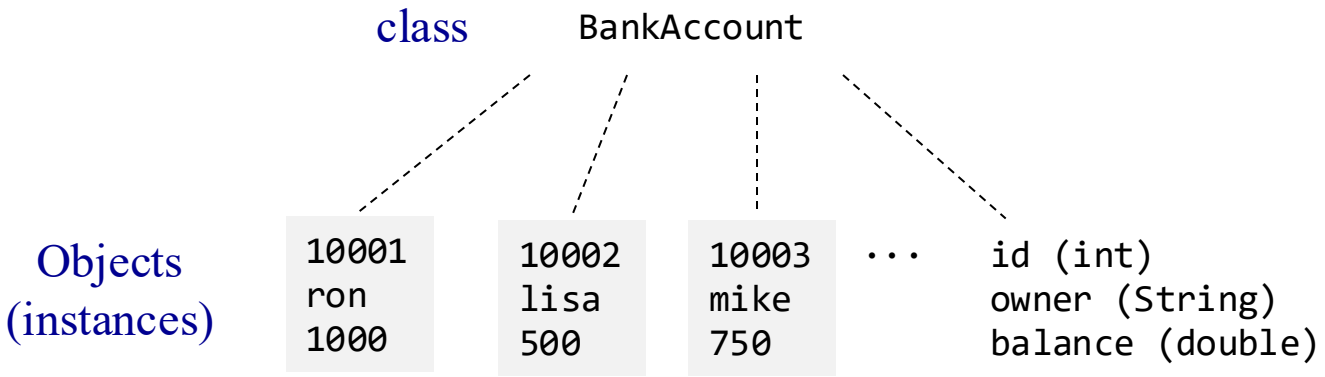
$$\frac{1}{2} + \frac{1}{3}$$

Fraction



Point

Bank account



Abstraction (Client view): How to *use* `BankAccount` objects (previous lecture)

Implementation (Class view): How to *realize* the `BankAccount` class (this lecture)

Bank account: requirements

Goal: represent and manipulate *bank accounts*

Representation

Things we wish *to record* about bank accounts:

- account ID
- account owner
- account balance

…

Manipulation

Things we wish *to do* with bank accounts:

- *construct* a new account
- *display* the account
- *deposit* money in the account
- *withdraw* money from the account
- *transfer* money to another account

Determined by: Representative users, product managers, system architects

Key outcome: `BankAccount` API

Bank account: requirements

Goal: represent and manipulate *bank accounts*

Representation

Things we wish ***to record*** about bank accounts:

- account ID
- account owner
- account balance

…

fields

Manipulation

Things we wish ***to do*** with bank accounts:

- *construct* a new account
- *display* the account
- *deposit* money in the account
- *withdraw* money from the account
- *transfer* money to another account

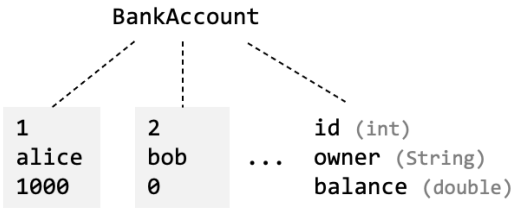
…

methods

Class API (public interface)

```
/** 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. */
    public BankAccount(String owner, double balance)

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

    /** 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 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()

    /** Returns the data of this bank account, as a string.  */
    public String toString()

    ...
}
```

Fields




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

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this bank account (fields)
    private int id;
    private String owner;
    private double balance;

    ...

    /** 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()

    /** Sets the balance of this account. */
    public void setBalance()

    ...

    // More BankAccount methods

    ...
```

Object fields

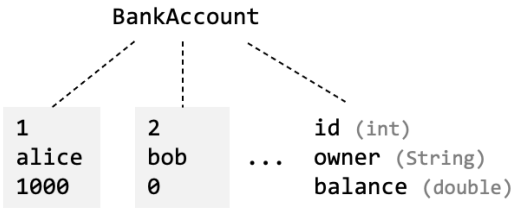
Also known as: *private variables*, *properties*, *attributes*

Each object of the class has a private set of field values

Each field data type can be ...

- Primitive
- Array
- Object
- Any possible data type.

Getters / Setters




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

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this bank account (fields)
    private int id;
    private String owner;
    private double balance;

    ...

    /** 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()

    /** Sets the balance of this account. */
    public void setBalance()

    ...

    // More BankAccount methods

    ...
```

Getter methods (AKA *accessors*)

Used to facilitate *reading* the data (fields) of objects

Setter methods (AKA *mutators*)

Used to facilitate *writing* the data (fields) of objects

```
// Client code:
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
System.out.println(aliceAcc.getBalance()); // prints 1000
bobAcc.setBalance(500);
```

How much object data to expose (class design issue)?

Consult the system's *requirements*;

Best practice: Expose as little as possible.

Static variables

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

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this bank account (fields)
    private int id;
    private String owner;
    private double balance;

    ...

    /** 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()

    /** Sets the balance of this account. */
    public void setBalance()

    ...

    // More BankAccount methods

    ...
```

Static variables

Class-level, global variables

`static`: Unlike fields, not associated with any particular object

Some classes use statics, some don't.

Constructing new objects

```
/** 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. */
    public BankAccount(String owner, double balance)

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

    /** 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 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()

    /** Returns the data of this bank account, as a string. */
    public String toString()

    ...
}
```

Constructing new objects (client view)

`Foo v = new Foo(`*arguments*`)`

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
```

Constructing new objects

```
/** 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. */
    public BankAccount(String owner, double balance)

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

    /** 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 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()

    /** Returns the data of this bank account, as a string. */
    public String toString()

    ...
}
```

Constructing new objects (client view)

`Foo v = new Foo(`*arguments*`)`

1. "`Foo v`" declares a new variable `v` of type `Foo`. The result is an object variable, designed to refer to `Foo` *objects*
2. `new` calls a `Foo` class constructor, that creates a new object of type `Foo`
3. When the constructor terminates, it returns the address of the new object;
4. `v` ends up referring to the new object.

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
```

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 {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Sets id to the running id number, and then increments it
        this.id = nextAccountId++;
        // Sets the fields of the newly created object to the given arguments
        this.owner = owner;
        this.balance = balance;
    }

    /** Constructs a new bank account with the given owner and zero balance. */
    public BankAccount(String owner) {
        this.id = nextAccountId++;
        this.owner = owner;
        this.balance = 0;
    }

    ...
}
```

Constructors

Special methods, used for:

- Creating new objects
- Initializing the newly created object

The constructor's name = class name

Constructors are often overloaded.

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
```

The current object (`this`)

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

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Sets id to the running id number, and then increments it
        this.id = nextAccountId++;
        // Sets the fields of the newly created object to the given arguments
        this.owner = owner;
        this.balance = balance;
    }

    /** Constructs a new bank account with the given owner and zero balance. */
    public BankAccount(String owner) {
        this.id = nextAccountId++;
        this.owner = owner;
        this.balance = 0;
    }

    ...
}
```

The current object

`this`: a local variable that holds the address of the current object

`this.fieldName`: Accessing a field of the current object

In the context of constructors:

- `this` refers to the object that the constructor constructed
- When a constructor terminates, it does an implicit `return this`
- Client view: The constructor call `new` is replaced with the address of the new object.

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
```

Static methods

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

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Sets id to the running id number, and then increments it
        this.id = generateId();
        // Sets the fields of the newly created object to the given arguments
        this.owner = owner;
        this.balance = balance;
    }

    /** Constructs a new bank account wit
    public BankAccount(String ow
        this.id = generateId();
        this.owner = owner;
        this.balance = 0;
    }

    // Generates an account id. Used by the class constructors.
    private static int generateId() {
        return nextAccountId++;
    }
    ...
}
```

Different constructor implementation (example of using a static method):

Uses a helper method for generating new account IDs.

Static / helper methods

Methods that help other methods

`static`: Not associated with any particular object

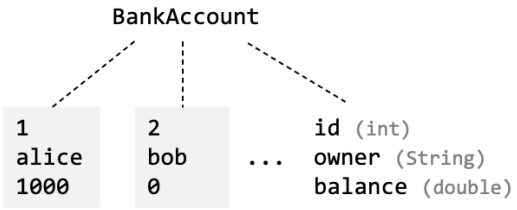
Some classes use static methods, some don't.

In Java: All methods, by default, are non-static

Methods are designed to operate on the *current object*.

If you want to create a method that *does not* operate on the current object, you must declare it `static`.

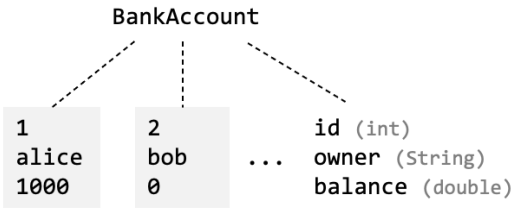
Methods



Methods operate on objects.

```
/** 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. */
    public BankAccount(String owner, double balance)

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

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

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

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

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

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

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

    /** Returns the data of this bank account (id, owner, balance), as a string. */
    public String toString()

    ...
}
```

Methods

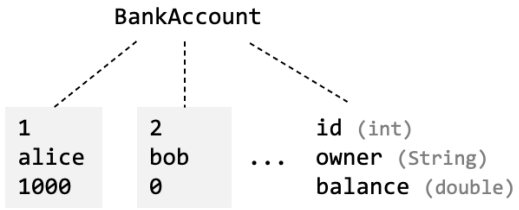


Methods operate on objects.

```
/** 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. */
    public BankAccount(String owner, double balance)

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

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

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

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

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

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

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

    /** Returns the data of this bank account (id, owner, balance), as a string. */
    public String toString()

    ...
}
```

Methods

```
/** 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. */
    public BankAccount(String owner, double balance)

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

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

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

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

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

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

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

    /** Returns the data of this bank account (id, owner, balance), as a string. */
    public String toString()

    ...
}
```



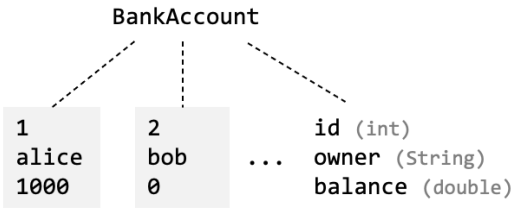
Methods operate on objects.

For example, consider the toString() method:

Designed to supply a textual description of the current object

A common method, essential for debugging / displaying.

Methods




```
/** 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. */
    public BankAccount(String owner, double balance)

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

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

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

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

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

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

    /** Returns the balance of this bank accoun
    public double getBalance()

    /** Returns the data of this bank account (id
    public String toString()

    ...
}
```

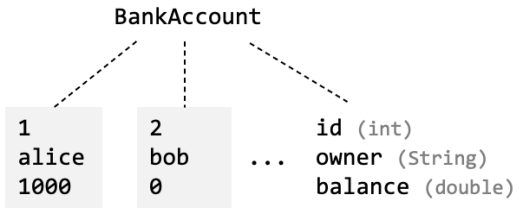
Applying `toString` to different objects:

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

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
System.out.println(aliceAcc.toString());
System.out.println(bobAcc.toString());
```

```
// Output:
1 Alice 1000
2 Bob 0
```

Methods




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

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate an id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    /** Returns the data of this bank account (id, owner, balance), as a string */
    public String toString() {
        return this.id + " " + this.owner + " " + this.balance;
    }
    ...
}
```

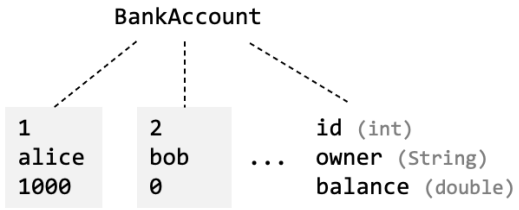
Applying `toString` to different objects:

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

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAccount("Bob");
...
System.out.println(aliceAcc.toString());
System.out.println(bobAcc.toString());
```

```
// Output:
1 Alice 1000
2 Bob 0
```

Methods




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

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate an id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    /** Returns the data of this bank account (id, owner, balance), as a string */
    public String toString() {
        return this.id + " " + this.owner + " " + this.balance;
    }
    ...
}
```

`this` refers to the object on which the method was called.

Applying `toString` to different objects:

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

```
// Client code
BankAccount aliceAcc = new BankAccount("Alice", 1000);
BankAccount bobAcc = new BankAc
...
System.out.println(aliceAcc.toString());
System.out.println(bobAcc.toString());
```

`this` will refer to `aliceAcc`

`this` will refer to `bobAcc`

```
// Output:
1 Alice 1000
2 Bob 0
```

Class design

```
/** Represents an unsafe  bank account.
public class UnsafeBankAccount {

    // Data of this unsafe bank account (fields)
    public int id;
    public String owner;
    public double balance;

    /** Constructs a new unsafe bank account. */
    public UnsafeBankAccount(String owner, double balance) {
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }
    ...
    // No need for getters and setters, the fields can be accessed directly
    ...
}
```

How to design an unsafe class

- Declare the object fields *public* (instead of *private*)
- Don't write getters and setters
- Result: Clients have unrestricted access to the object's data

  (public fields / public methods can be accessed by any class).

Accessing an object's public field:

objectVariable.*fieldName*

```
// Unsafe client code, can appear in any class:
UnsafeBankAccount bobAcc = new UnsafeBankAccount("Bob", 1000);

// Gets Bob's balance
System.out.println(bobAcc.balance);

// Gives Bob a nice present:
bobAcc.balance = 1000000000;

// Changes the account owner and ID:
bobAcc.owner = "Zzzoro";
bobAcc.id = -17;
System.out.println(bobAcc);
```

```
// Output:
1000
-17 Zzzoro 1000000000
```

Class design (back to safety)

```
/** Represents a bank account. */
public class BankAccount {
    ...
    // data of this bankAccount
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account */
    public BankAccount(String owner, double balance) {
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    /** Returns the balance of this bank account. */
    public double getBalance() {
        return this.balance;
    }

    /** Sets the balance of this bank account. */
    public void setBalance(double balance) {
        this.balance = balance;
    }

    // More getter and setters follows.

    // More BankAccount methods.

}
```

private fields (accessible only by methods of this class)

public getter (accessible by any class)

public getter (accessible by any class)

Best Practice

1. Protect the objects' data by making their fields `private`
2. Facilitate *managed access* to the object's fields, selectively, by writing public getters/setters

Perspective

In this simple example, the getters and setters don't do much;

In a real banking app, these methods are used for …

- Checking login credentials
- Verifying data integrity
- Recording transaction history
- Etc.

Class design

```
/** Represents a bank account.
public class BankAccount {

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

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

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

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

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

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

    /** Returns the data of this bank account, as a string  */
    public String toString()

    ...
}
```

Class design

```
/** Represents a bank account.
public class BankAccount {

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

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

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

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

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

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

    /** Returns the data of this bank account, as a string  */
    public String toString()

    ...
}
```

Commonly-used methods

- constructor(s)
- `toString()`
- getters / setters, as needed

In addition:

The class typically features domain-specific methods, as needed;

In this example: methods that perform *banking services*.

Class design

```
/** Represents a bank account.
public class BankAccount {

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

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

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

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

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

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

    /** Returns the data of this bank account, as a string  */
    public String toString()

    ...
}
```

```
// Client code
BankAccount aliceAcc =
            new BankAccount("Alice");
BankAccount bobAcc =
            new BankAccount("Bob", 500);
...
// Bob withdraws 200
bobAcc.withdraw(200);
...
// Alice deposits 1000
aliceAcc.deposit(1000);
...
// Alice transfers 400 to Bob
aliceAcc.transferTo(bobAcc, 400);
...
System.out.println(aliceAcc);
System.out.println(bobAcc);
```

```
// Output:
1 Alice 600
2 Bob 700
```

Class design

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    // Various constructors and methods (code omitted)
    ...

    /** Handles a deposit of sum to this bank account. */
    public void deposit(double sum) {
        this.balance = this.balance + sum;
    }

    /** Handles a withdrawal of sum from this bank account.*/
    public void withdraw(double sum) {
        this.balance = this.balance - sum;
    }

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

```
// Client code
BankAccount aliceAcc =
            new BankAccount("Alice");
BankAccount bobAcc =
            new BankAccount("Bob", 500);
...
// Bob withdraws 200
bobAcc.withdraw(200);
...
// Alice deposits 1000
aliceAcc.deposit(1000);
...
```

Class design

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    // Various constructors and methods (code omitted)
    ...
    /** Handles a deposit of sum to this bank account. */
    public void deposit(double sum) {
        this.balance = this.balance + sum;
    }

    /** Handles a withdrawal of sum from this bank account.*/
    public void withdraw(double sum) {
        this.balance = this.balance - sum;
    }

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

```
// Client code
BankAccount aliceAcc =
            new BankAccount("Alice");
BankAccount bobAcc =
            new BankAccount("Bob", 500);
...
// Bob withdraws 200
bobAcc.withdraw(200);
...
// Alice deposits 1000
aliceAcc.deposit(1000);
...
// Alice transfers 400 to Bob
aliceAcc.transferTo(bobAcc, 400);
...
System.out.println(aliceAcc);
System.out.println(bobAcc);
```

```
// Output:
1 Alice 600
2 Bob 700
```

Class design

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    // Various constructors and methods (code omitted)
    ...

    /** Handles a deposit of sum to this bank account. */
    public void deposit(double sum) {
        this.balance = this.balance + sum;
    }

    /** Handles a withdrawal of sum from this bank account.*/
    public void withdraw(double sum) {
        this.balance = this.balance - sum;
    }

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

Methods can call other methods, for their effect.

Best practice

Whenever possible, call existing methods. If there is a method that does the job, use it.

We could have used:

```
this.balance = this.balance – sum
other.balance = other.balance + sum;
```

But, in a real banking system, the `withdraw` and `deposit` methods will provide data verification and security services. Let them do their job!

Constructors (revisited)

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate a unique id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    /** Constructs a new bank account with a zero balance. */
    public BankAccount(String owner) {
        this.id = generateId();
        this.owner = owner;
        this.balance = 0;
    }
    ...
   // More BanAccount methods
}
```

Constructors (revisited)

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate a unique id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    /** Constructs a new bank account with a zero balance. */
    public BankAccount(String owner) {
        this.id = generateId();
        this.owner = owner;
        this.balance = 0;
    }
    ...
   // More BanAccount methods
}
```

Same action, better style

```
/** Constructs a new bank account with a zero balance. */
public BankAccount(String owner) {
    this(owner, 0);
}
```

Compilation convention I:
A constructor can call another constructor in the same class using the syntax:
`this(`*arguments*`)`

Best practice

Reuse existing code, as much as possible.

The current object (revisited)

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate a unique id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    ...

    /** Returns the data of this bank account (id, owner, balance), as a string */
    public String toString() {
        return this.id + " " + this.owner + " " + this.balance;
    }
    ...
}

}
```

The current object (revisited)

```
/** Represents a bank account. */
public class BankAccount {

    // Stores the next account ID
    private static int nextAccountId = 1;

    // Data of this BankAccount (fields)
    private int id;
    private String owner;
    private double balance;

    /** Constructs a new bank account with the given owner and balance.
     *  The account id is a running integer, generated automatically. */
    public BankAccount(String owner, double balance) {
        // Calls a helper function to generate a unique id
        this.id = generateId();
        this.owner = owner;
        this.balance = balance;
    }

    ...

    /** Returns the data of this bank account (id, owner, balance), as a string */
    public String toString() {
        return this.id + " " + this.owner + " " + this.balance;
    }
    ...
}
```

Same action, better style

```
return id + " " + owner + " " + balance;
```

Compilation convention II:

Inside a method, each variable `foo` which is neither a local variable, nor a parameter, nor a static variable, is treated by the compiler by default as the field `this.foo`

Best practice

To reduce clutter, drop the `this` prefix (unless it's needed).

Class design: Recap

```
/** Represents a bank account.
public class BankAccount {

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

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

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

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

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

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

    /** Returns the data of this bank account, as a string  */
    public String toString()

    ...
}
```

A class API is like a contract

- In the banking example, the API creates a domain language for banking operations
- It supports the work of many developers
- It enforces consistency, modularity, and safety

Class design: Recap

```
/** Represents a bank account.
public class BankAccount {

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

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

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

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

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

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

    /** Returns the data of this bank account, as a string  */
    public String toString()

    ...
}
```

What would it take to develop an ATM system?

- Server-side logic (key element: the `BankAccount` class)

Plus (not discussed in this lecture):

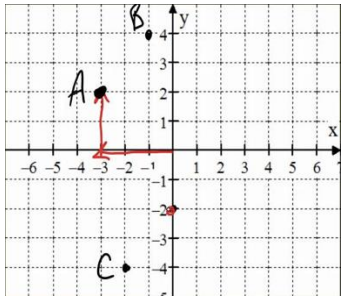
- Server-side database (to persist objects)
- Client-side user interfaces (ATM, web, app, ...)
- Communications / transaction processing / multi-threading
- Security.

Lecture plan

`BankAccount`

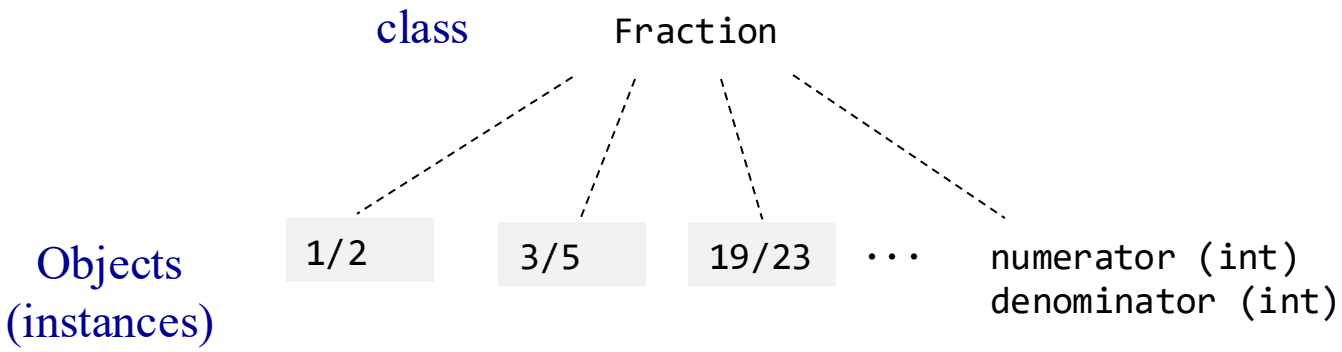
$$\frac{1}{2} + \frac{1}{3}$$

`Fraction`



`Point`

Fractions



Abstraction (Client view): How to *use* `Fraction` objects (previous lecture)

Implementation (Class view): How to *realize* the `Fraction` class (this lecture)

Class API

```
/** Represents a signed fraction like 1/2 or -2/3.
 *  A fraction is characterized by a numerator and a denominator. */
public class Fraction {

    /** Constructs a fraction. For example,
    *  given 3 and 5, constructs the fraction 3/5. */
    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 the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns 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 string representation of this fraction
    / * in the form "numerator/denominator" */
    public String toString()

    // More Fraction methods
}
```

Creating and displaying objects

```
/** Represents a signed fraction like 1/2 or -2/3.
 *  A fraction is characterized by a numerator and a denominator. */
public class Fraction {

    /** Constructs a fraction. For example,
     *  given 3 and 5, constructs the fraction 3/5. */
    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 the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns 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 string representation of this fraction
     * in the form "numerator/denominator" */
    public String toString()

    // More Fraction methods
}
```

```
// Client code, in any class:
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
System.out.println(a);
System.out.println(b);
...
```

```
// Output:
1/3
1/2
```

Creating and displaying objects

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

    private int numerator;
    private int denominator;

    /** Constructs a fraction. */
    public Fraction(int numerator, int denominator)
        // Initializes the object's fields
        this.numerator = numerator;
        this.denominator = denominator;
    }

    ...

    /** Returns a string representation of this fraction
    / * in the form "numerator/denominator" */
    public String toString() {
        return numerator + "/" + denominator;
    }

    // More Fraction methods
}
```

```
// Client code, in any class:
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
System.out.println(a);
System.out.println(b);
...
```

```
// Output:
1/3
1/2
```

Fraction Constructor

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

    private int numerator;
    private int denominator;

    /** Constructs a fraction. */
    public Fraction(int numerator, int denominator)
        // Initializes the object's fields
        this.numerator = numerator;
        this.denominator = denominator;
    }

    ...

    /** Returns a string representation of this fraction
    / * in the form "numerator/denominator" */
    public String toString() {
        return numerator + "/" + denominator;
    }

    // More Fraction methods
}
```

```
// Client code, in any class:
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
System.out.println(a);
System.out.println(b);
Fraction c = new Fraction(-2,-3);
Fraction d = new Fraction(4,-5);
Fraction e = new Fraction(8,16);
System.out.println(c);
System.out.println(d);
System.out.println(e);
...
```

```
// Output:
1/3
1/2
-2/-3     // can be improved
4/-5      // can be improved
8/16      // can be improved
```

Fraction Constructor: Improved

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

    private int numerator;
    private int denominator;

    /** Constructs a fraction, and reduces it if necessary. */
    public Fraction(int numerator, int denominator)
        this.numerator = numerator;
        this.denominator = denominator;
        // Reduces this fraction
        this.reduce();
    }
    ...

    //  Reduces this fraction by dividing its numerator and
    //  denominator by their GCD
    private void reduce() {
        int gcd = gcd(Math.abs(numerator),
                      Math.abs(denominator));
        numerator = numerator / gcd;
        denominator = denominator / gcd;
    }

    // Returns the greatest common divisor of two positive numbers.
    private static int gcd(int x, int y) {
        int rem;
        while (y != 0) {
            rem = x % y;
            x = y;
            y = rem; }
        return x;
    }

    // More Fraction methods
}
```

```
// Client code, in any class:
...
Fraction a = new Fraction(1,3);
Fraction b = new Fraction(1,2);
System.out.println(a);
System.out.println(b);
Fraction c = new Fraction(-2,-3);
Fraction d = new Fraction(4,-5);
Fraction e = new Fraction(8,16);
System.out.println(c);
System.out.println(d);
System.out.println(e);
...
```

```
// Output:
1/3
1/2
-2/-3     // can be improved
4/-5      // can be improved
1/2
```

Fraction Constructor: Improved

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

    private int numerator;
    private int denominator;

    /** Constructs a fraction, and reduces it if necessary. */
    public Fraction(int numerator, int denominator)
        this.numerator = numerator;
        this.denominator = denominator;
        // Reduces this fraction
        this.reduce();
    }
    ...

    //  Reduces this fraction by dividing its numerator and
    //  denominator by their GCD
    private void reduce() {
        int gcd = gcd(Math.abs(numerator),
                      Math.abs(denominator));
        numerator = numerator / gcd;
        denominator = denominator / gcd;
    }

    // Returns the greatest common divisor of two positive numbers.
    private static int gcd(int x, int y) {
        int rem;
        while (y != 0) {
            rem = x % y;
            x = y;
            y = rem; }
        return x;
    }

    // More Fraction methods
}
```

Note 1: Minimize code clutter

In the constructor's code:
"`this.reduce();`" can be replaced with
"`reduce();`"

Explanation: we are calling a method in the same class, on the current object.

Note 2: (refactoring)

Best practice: Move the `gcd` function to a `MyMath` class, make it public, and use it by calling `MyMath.gcd(int,int);`

Rationale: The `gcd` function can serve many needs, not only those of the `Fraction` class

And... It simply does not belong to a class that represents fractions.

Fraction Constructor: Final

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

    private int numerator;
    private int denominator;

    /** Constructs a reduced fraction from the given numerator and
     *  denominator. For example, given 8 and 16, constructs the
     *  fraction 1/2. If the denominator is negative, inverts the signs
     *  of both the numerator and the denominator.
     *  For example, 2/-3 becomes -2/3, and -2/-3 becomes 2/3.
     *  @param numerator     (can be signed)
     *  @param denominator (can be signed, must be non-zero)
     *  @return a reduced, properly signed fraction
     *  @pre denominator != 0 (not checked)
     */
    public Fraction(int numerator, int denominator)
        // Handles the signs of the numerator and denominator
        if (denominator < 0) {
            numerator = -1 * numerator;
            denominator = -1 * denominator;
        }
        // Initializes the fields of this fraction
        this.numerator = numerator;
        this.denominator = denominator;
        // Divides the numerator and the denominator of this
        // fraction by their greatest common divisor
        reduce();
    }
    ...
    // More Fraction methods
}
```

```
// Client code, in any class:
...
Fraction c = new Fraction(-2,-3);
Fraction d = new Fraction(4,-5);
Fraction e = new Fraction(8,16);
System.out.println(c);
System.out.println(d);
System.out.println(e);
...
```

```
// Output:
2/3
-4/5
1/2
```

Fraction Constructor: Final

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

    private int numerator;
    private int denominator;

    /** Constructs a reduced fraction from the given numerator and
     *  denominator. For example, given 8 and 16, constructs the
     *  fraction 1/2. If the denominator is negative, inverts the signs
     *  of both the numerator and the denominator.
     *  For example, 2/-3 becomes -2/3, and -2/-3 becomes 2/3.
     * @param numerator     (can be signed)
     * @param denominator (can be signed, must be non-zero)
     * @return a reduced, properly signed fraction
     * @pre denominator != 0 (not checked)
     */
    public Fraction(int numerator, int denominator)
        // Handles the signs of the numerator and denominator
        if (denominator < 0) {
            numerator = -1 * numerator;
            denominator = -1 * denominator;
        }
        // Initializes the fields of this fraction
        this.numerator = numerator;
        this.denominator = denominator;
        // Divides the numerator and the denominator of this
        // fraction by their greatest common divisor
        reduce();
    }
    ...
    // More Fraction methods
}
```

Note 1: Javadoc

A documentation tool that uses annotations (like `/** comment */` and `@`) for generating HTML documentation pages from the source code.

Helps creating standardized, easy-to-read documentation for Java classes, methods, and fields.

Note 2: Exceptions

What to do when `denominator == 0` ?

Solution: Throwing an exception (later).

Fraction arithmetic methods

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

    /** Constructs a fraction. For example,
   * given 3 and 5, constructs the fraction 3/5. */
    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 the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

    /** Returns 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 string representation of this fraction
    / * in the form "numerator/denominat
    public String toString()

    // More Fraction methods
}
```

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

```
// Output:
1/3 + 3/7 = 16/21
1/3 * 3/7 = 1/7
```

Fraction arithmetic methods

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

    private int numerator;
    private int denominator;

    /** Constructs a reduced fraction from the given num. and denom. */
    public Fraction(int numerator, int denominator) {
        // Code omitted
    }

    ...

    /** Returns the sum of this fraction and the other one */
    public Fraction add (Fraction other) {
        return new Fraction ( (numerator * other.denominator +
                               other.numerator * denominator),
                              denominator * other.denominator );
    }

    /** Returns the product of this fraction and the other one */
    public Fraction multiply (Fraction other) {
        return new Fraction (numerator * other.numerator,
                             denominator * other.denominator);
    }

    // More Fraction methods
}
```

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

```
// Output:
1/3 + 3/7 = 16/21
1/3 * 3/7 = 1/7
```

Fraction arithmetic methods

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

    private int numerator;
    private int denominator;

    /** Constructs a reduced fraction from the given num. and denom. */
    public Fraction(int numerator, int denominator) {
        // Code omitted
    }

    ...

    /** Returns the sum of this fraction and the other one */
    public Fraction add (Fraction other) {
        return new Fraction ( (numerator * other.denominator +
                               other.numerator * denominator),
                              denominator * other.denominator );
    }

    /** Returns the product of this fraction and the other one */
    public Fraction multiply (Fraction other) {
        return new Fraction (numerator * other.numerator,
                             denominator * other.denominator);
    }

    // More Fraction methods
}
```

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

```
// Output:
1/3 + 3/7 = 16/21
1/3 * 3/7 = 1/7
1/3 * ( 3/7 + 2/5 ) = 29/105
```

Fraction arithmetic

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

    /** Constructs a fraction. For example */
    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 the sum of this fraction and the other one. */
    public Fraction add(Fraction other)

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

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

    /** Returns the division of this fraction and the other one */
    public Fraction divide()

    // More Fraction methods
}
```

```
// Client code, in any class:
System.out.print("2/3 : 4/5 = ");
System.out.println((new Fraction(2,3)).divide(new Fraction(4,5)));
```

```
// output:
2/3 : 4/5 = 5/6
```

Fraction arithmetic

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

    private int numerator;
    private int denominator;

    /** Constructs a reduced fraction from the given num. and denom. */
    public Fraction(int numerator, int denominator) {
        // Code omitted
    }

    ...

    /** Returns the product of this fraction and the other one */
    public Fraction multiply (Fraction other) { // Code omitted }

    /** Returns the reciprocal of this fraction. */
    public Fraction invert() {
        return new Fraction(denominator, numerator);
    }

    /** Returns the division of this fraction and the other one */
    public Fraction divide (Fraction other) {
        return multiply(other.invert());
    }

    // More Fraction methods
}
```

```
// Client code, in any class:
System.out.print("2/3 : 4/5 = ");
System.out.println((new Fraction(2,3)).divide(new Fraction(4,5)));
```

```
// output:
2/3 : 4/5 = 5/6
```

Recap: The Fraction class

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

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

    /** Returns 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 string representation of this fraction
    / * in the form "numerator/denominator" */
    public String toString()

    // More Fraction methods
}
```

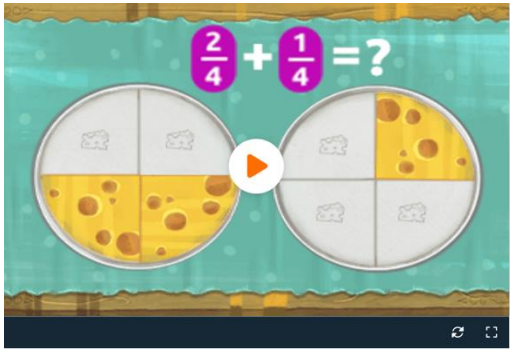
The Fraction class

Extends Java's type system

Provides accurate fraction representation

Possible applications

- Scientific / engineering programs that require high-precision computations
- Finance, music, games, cooking apps (1/2 cup of sugar...), etc.
- Programs that teach fraction arithmetic.

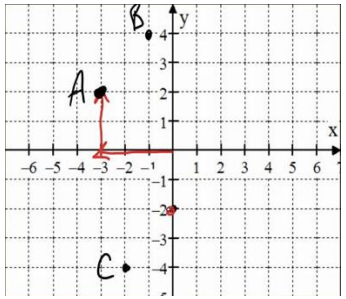


Lecture plan

`BankAccount`

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

`Fraction`



`Point`

Abstraction (Client view): How to *use* `Point` objects (previous lecture)

Implementation (Class view): How to *realize* the `Point` class (this lecture)

Class abstraction (API)

```
/** Represents a point in a plain. A mix of math and drawing methods. */
public class Point {

    /** Constructs a point */
    public Point(double x, double y)

    /** Returns the addition of this and the other point, as a Point object. */
    public Point add(Point other)

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

    /** Returns a textual representation of this point. */
    public String toString()

    /** Draws this point. */
    public void draw()

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

```
// Client code
...
// 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)
```

Class implementation

```
/** Represents a point in a plain. A mix of math and drawing methods. */
public class Point {

    // The coordinates of this point
    private double x;
    private double y;

    /** Constructs a point */
    public Point(double x, double y) {
       this.x = x;
       this.y = y;
    }

    ...

    /** Returns the addition of this and the other point, as a Point object. */
    public Point add(Point other) {
       return new Point((x + other.x),(y + other.y));
    }

    ...
```

```
// Client code
...
// 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));
```

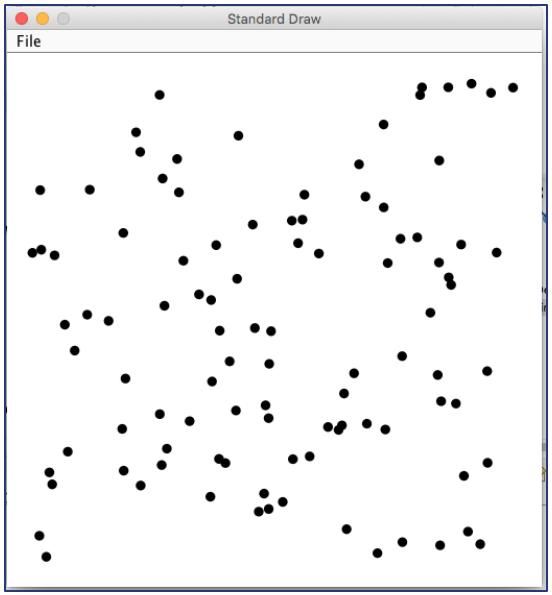
output

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

Class abstraction (API)

```
/** Represents a point in a plain. A mix of math and drawing methods. */
public class Point {

    /** Constructs a point */
    public Point(double x, double y)

    /** Returns the addition of this and the other point, as a Point object. */
    public Point add(Point other)

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

    /** Returns a textual representation of this point. */
    public String toString()

    /** Draws this point. */
    public void draw()

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

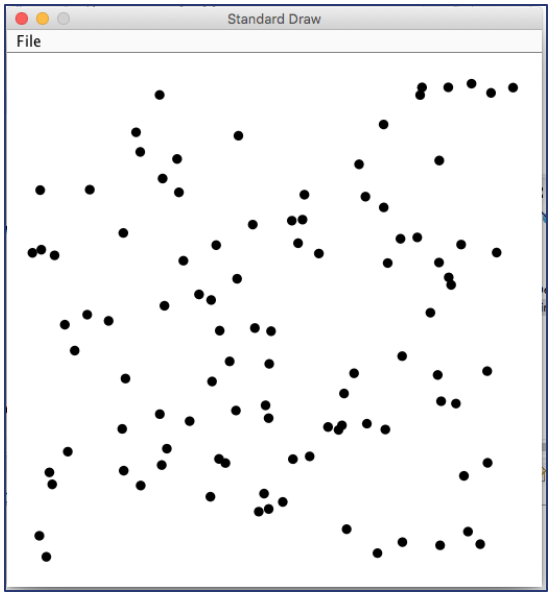
```
// Client code
...
// Creates an array of random points
int N = 100;
Point[] points = new Point[N];
for (int i = 0; i < N; i++)
    points[i] = new Point(Math.random(),
                          Math.random());
// Draws the points
for (int i = 0; i < N; i++)
    points[i].draw();
```



Class implementation

```
import stdLib.StdDraw;
/** Represents a point in a plain. A mix of math and drawing methods. */
public class Point {

   // The coordinates of this point
   private double x;
   private double y;

   /** Constructs a point */
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   ...

   /** Draws this point. */
   public void draw() {
      StdDraw.filledCircle(x, y, 0.01);
   }

   ...
```

```
// Client code
...
// 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();
```



Class implementation (complete)

```
import stdLib.StdDraw;
/** Represents a point in a plain. A mix of math and drawing methods. */
public class Point {
   // The coordinates of this point
   private double x;
   private double y;

   /** Constructs a point */
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   /** Returns the addition of this and the other point, as a Point object. */
   public Point add(Point other) {
      return new Point((x + other.x),(y + other.y));
   }

   /** Returns the distance between this and the other point. */
   public double distanceTo(Point other) {
      double dx = this.x - other.x;
      double dy = y - other.y;        // Better style
      return Math.sqrt(dx * dx + dy * dy);
   }

   /** Returns a textual representation of this point. */
   public String toString() {
      return ("(" + x + "," + y + ")");
   }

   /** Draws this point. */
   public void draw() {
      StdDraw.filledCircle(x, y, 0.01);
   }

   /** Draws a line between this and the other point. */
   public void drawLineTo(Point other) {
      StdDraw.line(x, y, other.x, other.y); }
}
```

Recap: Procedural programming VS OOP

Procedural programming

Class = library of static methods (*functions*)

(all the classes and methods in weeks / homework 1, 2, 3, ..., 7)

Object-Oriented programming

Class = Represents *objects*

(`BankAccount`, `Fraction`, `Point`, `User`, `Network`, ...)

Recap: Procedural programming VS OOP

Procedural programming

Class = library of static methods (*functions*)

(all the classes and methods in weeks / homework 1, 2, 3, ..., 7)

Static methods: Operate on the given arguments

No fields, no constructors, no notion of `this`

Object-Oriented programming

Class = Represents *objects*

(`BankAccount`, `Fraction`, `Point`, `User`, `Network`, ...)

Methods: Operate on objects (*and* on the given arguments)

Plus, optionally, some static methods

OOP = mix of OOP and Procedural programming