#### Week 4

Classes

#### Last week

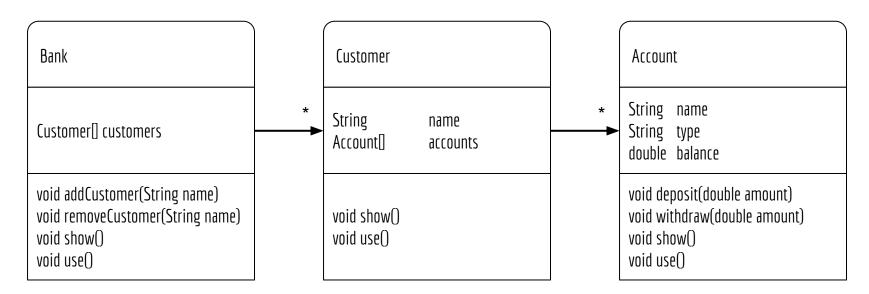
- We broke down a small program into a set of methods.
- This is "procedural programming"

```
public static void main(String[] args) {
        showMatchingWords(readSentence());
                                                                                          return In.nextLine();
public static void showMatchingWords(String sentence) {
         System.out.println("Matching words = " + matchingWords(sentence);
public static int matchingWords(String sentence) {
        int count = 0;
                                                                                          return false:
         for (String word : sentence.split(" +"))
                 if (anyVowels(word))
                          count++:
        return count:
```

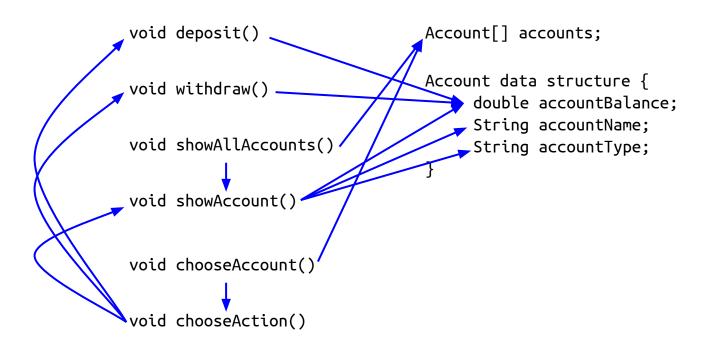
```
public static String readSentence() {
         System.out.print("Sentence: ");
public static boolean anyVowels(String word) {
         for (int i = 0; i < word.length(); i++)
                  if (isVowel(word.charAt(i)))
                           return true:
public static boolean isVowel(char c) {
         return "aeiou".contains(String.valueOf(c));
```

#### This week

- We break down a larger program into classes of objects.
- This is object oriented programming.



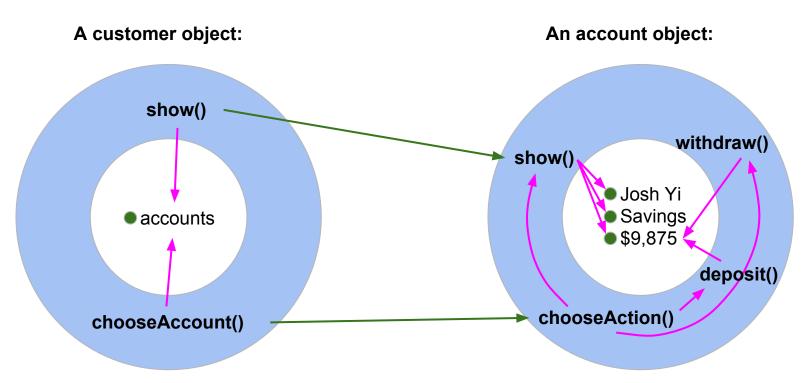
## Procedural Bank Program



### Procedural Bank Program

- Each program element is global.
- Dependencies between elements are not well-structured / unclear.
- Not clear how to subdivide the work among programmers in a team.
- Not clear what impact a change will have since any element can be accessed by any other element.

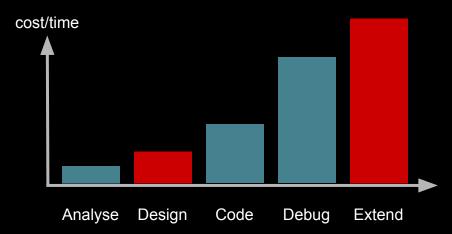
## Object-Oriented Bank Program



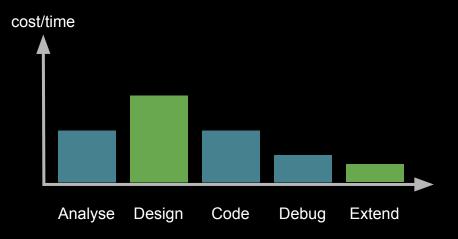
## Advantages of Object-Oriented Programming

- Each kind of object has separate concerns. Different programmers can code different kinds of object without stepping on each other's toes.
- Dependencies between objects are few and easy to manage. Most dependencies are isolated within an object.
- Objects export an interface and hide the implementation details. The programmer of one object can change its internal details without bothering the programmers of other objects.
- Object structures simplify naming. e.g. if accountBalance is inside an account object, just name it balance.
- Objects better map onto the way the real world works. The real world has objects.









#### What is a class?

A class is a template for creating objects.

The members of a class are **fields** and **methods**.

**Read**: Each account has a name, type and balance. You can do these things with an account:

- deposit
- withdraw
- show
- use

#### Class diagram



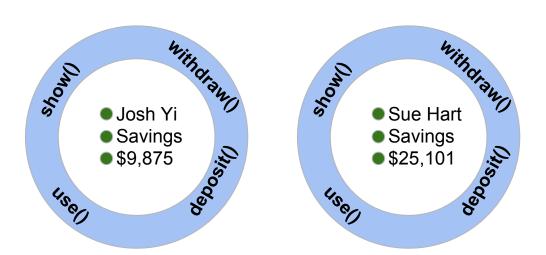
#### Account

String name String type double balance

void deposit(double amount)
void withdraw(double amount)
void show()
void use()

## What is an object?

An object is an instance of a class. Each object gets its own copy of the members.



Account

String name
String type
double balance

void deposit(double amount)
void withdraw(double amount)
void show()
void use()

#### Instance vs Static

Static members:

```
private static int x;
public static void foo() { ... }
```

Instance members:

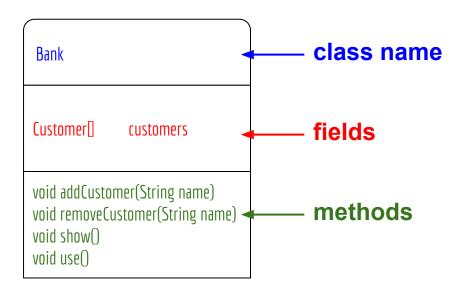
```
private int x;
public void foo() { ... }
```

Only instance members are copied into each object.

Therefore, don't use static in objects.

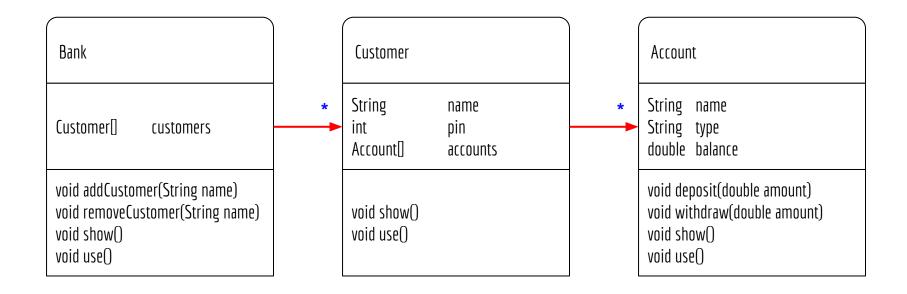
## Class diagrams

- Class diagrams help us to sketch and evaluate OO program designs.
- A class is depicted as a box with class name / fields / methods



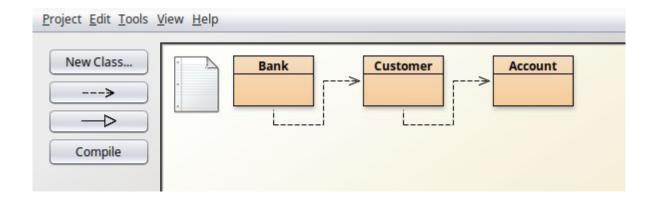
### Class diagrams

- An arrow indicates one class uses another class.
- \* indicates multiplicity. E.g. a Customer uses/has many accounts.



## BlueJ class diagrams

- BlueJ shows simplified class diagrams
- Fields and methods not shown
- Multiplicity not shown



# Design Rules

### Design rules

This week we will use design rules to write good object-oriented code.

#### Design rules govern:

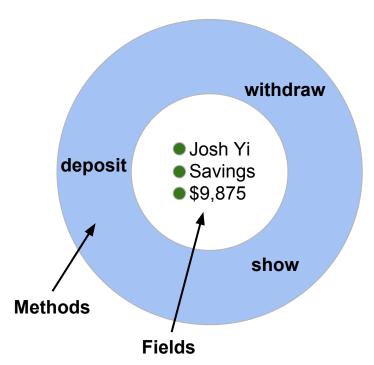
- How code should be split into separate classes and methods
- How and to what extent code should interact

### Design rule #1: Encapsulation

#### **Encapsulation**:

- Fields are **hidden** behind methods
  - o fields are always private
  - methods may be public
- An **object** encapsulates <u>related</u> fields+methods.
- **Rule:** If a method uses a field, it is defined in the same class.

#### An account object:



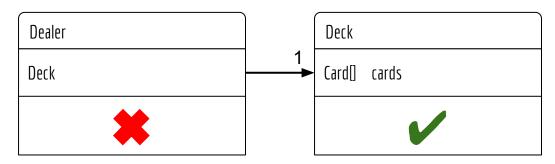
### Design rule #2: Push it right

**Goal**: Shuffle a deck of standard playing cards.

**Question**: Which class is responsible?

- a) The dealer should shuffle the deck. (The cards are private inside the deck)
- b) The deck should shuffle itself. (YES: the deck has direct access the cards)

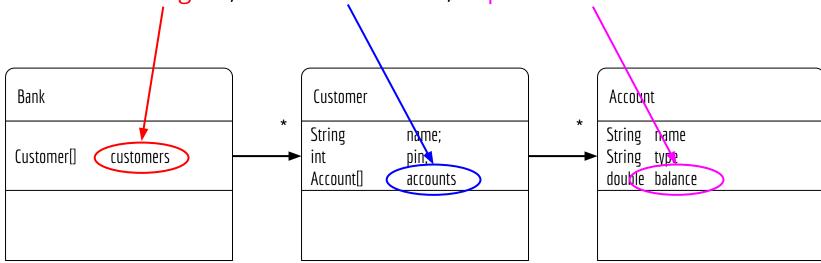
**Payoff**: The deck is more useful. The shuffle method is more reusable.



### Design rule #3: Spread plans across classes

**Goal**: Use a customer's account at the bank.

Scenario: User logs in, selects an account, deposits or withdraws.

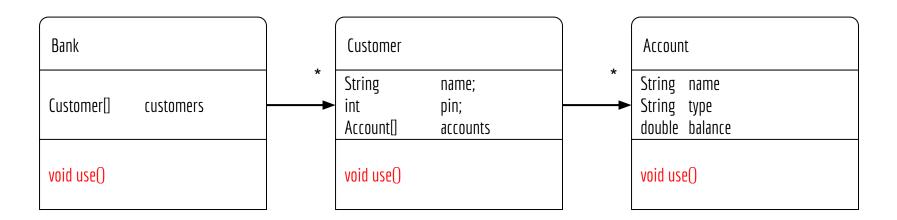


Question: Which class is responsible? Answer: ALL classes are responsible!

### Design rule #3: Spread plans across classes

**Goal**: Use a customer's account at the bank.

Convention: Use the same method name across classes for the same goal.



## Design rule #4: Hide by default

- Make everything private unless there is a reason to make it public.
- Make all fields private.
- Make methods private if no other class needs to use them.
- Make methods public only if other classes need to use them.

### Access modifiers

Class members may be declared with an access modifier.

private: can be accessed only within the class.
 private double readBalance()

no modifier: can also be accessed within the package.
 String getPassword()

protected: can also be accessed by subclasses.
 protected int width;

public: can also be accessed by other classes.
 public void deposit()

## Class format

#### Import statements

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
```

#### Class declaration

```
public class Account {
    ...
}
```

Class names begin with an uppercase letter.

#### Fields

```
public class Account {
    private String name;
    private String type;
    private double balance;
}
```

Fields begin with a lowercase letter. Fields are always private.

#### Constants

#### Constructors

**Goal**: Initialise a new object.

```
public class Account {
    ...
    public Account() {
        name = ...;
        type = ...;
        balance = ...;
    }
}
```

Constructors are named after the class. Constructors have no return type. Constructors initialise the fields of a newly created object.

## Constructors approach #1: initialise from literals

**Goal**: Initialise a new object with literal values.

```
public class Account {
    ...
    public Account() {
        name = "Default name";
        type = "Savings;
        balance = 0.0;
    }
}
```

Initialise with default values.

### Constructors approach #2: initialise from user

**Goal**: Initialise a new object with values read from the user.

```
public class Account {
    ...
    public Account() {
        name = readName();
        type = readType();
        balance = readBalance();
    }
}
Use the read pattern.
```

### Constructors approach #3: initialise from params

**Goal**: Initialise a new object from parameters.

```
public class Account {
    ...
    public Account(String name, String type, double balance) {
        this.name = name;
        this.type = type;
        this.balance = balance;
    }
    Parameters are named after fields.
    Use this.<name> to refer to a field.
    Use <name> to refer to a parameter.
```

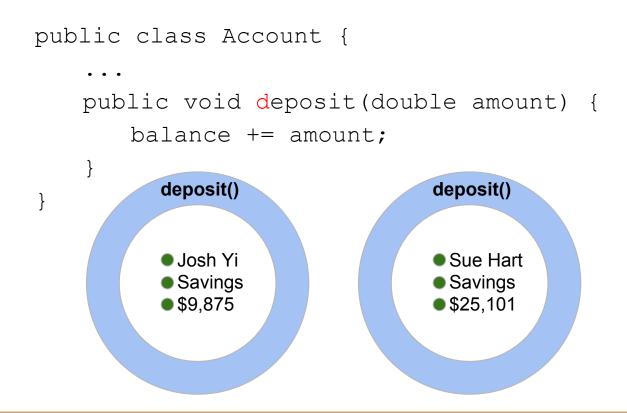
### The "this" keyword

- this is a reference to the current object.
- this.<member> accesses a member of the current object.
- <member> also accesses a member of the current object.unless there is a local variable or parameter with the same name. Then you must use this.<member>

#### e.g.

- janesAccount.deposit(10.0); // call deposit on another object
- this.deposit(10.0); // deposit on myself
- deposit (10.0); // call deposit on myself

#### Methods



Methods begin with a lowercase letter.

Use **instance** methods NOT **static** methods.

Each instance of class Account will have its own **deposit** method.

## Design rule #4 (again): Hide by default

```
public class Account {
                                     The readName() method is only used
   public Account() {
                                     within class Account.
       name = readName();
                                     No outside class needs it.
       type = readType();
                                     Make it private.
       balance = readBalance();
   private String readName() {
       System.out.print("Account name: ");
       return In.nextLine();
```

## toString method

```
public class Account {
    ...
    @Override
    public String toString() {
        return "The account has $" + balance;
    }
}
```

Returns a string representation of the object.

This is a standard method of all classes and we override the default behaviour.

### Format to 2 decimal places - pattern

**Goal**: Show to two decimal places.

```
@Override
public String toString() {
    return "The account has $" + formatted(balance);
}
private String formatted(double value) {
    DecimalFormat f = new DecimalFormat("###,##0.00");
    return f.format(value);
}
```

0 means always show a digit. # means show a digit if needed.

Package to import: java.text.\*

#### Getter and setter methods

```
public class Account {
   private String name;
   public String getName() {
       return name;
   public void setName(String name) {
      this.name = name;
```

- A getter returns a field
   The name is get<Field>
- A setter sets a field
   The name is set<Field>

# Design rule #4 (again!): Hide by default

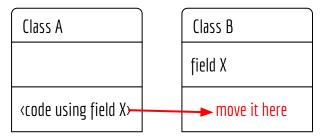
- Getters and setters export a field.
- Almost like making a field public.
- Avoid using getters and setters.

#### There is usually a better way!

If code in class A needs to get access to a field in class B, consider moving

the code into class B.

See design rule #2: "Push it right"



# Creating an object

Creating an object using a constructor:

Account janesAccount = new Account ("Jane Knowles", "Savings", 25283.21);

new returns the memory address of the new object.
 The memory address is also known as: "reference" / "pointer".

The variable janesAccount stores a pointer to the object,

NOT the object itself.



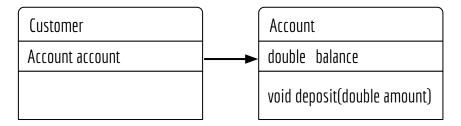
# Passing an object

Passing an object actually passes the memory address.

```
Account janesAccount = new Account ("Jane Knowles", "Savings", 25283.21) ;
use(janesAccount)
private void use(Account account) {
    ... use account here ...
Both variables refer to the same object:
   A062ED10
                    A062ED10
 janesAccount
                  account
```

# Client/supplier interactions

A client class uses a supplier class. e.g. A Customer uses an Account:



Client code:

Supplier code:

```
public class Account {
          ...
          public void deposit(double amount)
{
          balance += amount;
     }
          ...
```

# Using a toString method

Using another object's toString method

```
Explicitly: System.out.println(janesAccount.toString());
Implicitly: System.out.println(janesAccount);
```

Using this object's toString method

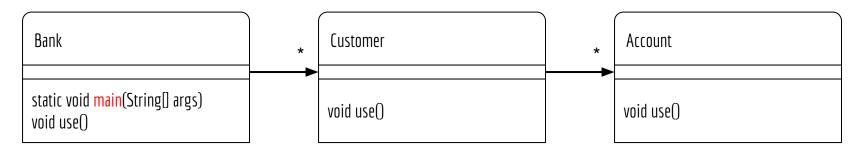
```
Explicitly without this: System.out.println(toString());
Explicitly with this: System.out.println(this.toString());
Implicitly with this: System.out.println(this);
```

### The "main" method

From this week, the main method is the **only** static method. It creates and uses the first object.

```
public static void main(String[] args) {
    Bank bank = new Bank();
    bank.use();
}
```

The main method is defined in the main (left-most) class.



# Process

#### Process: words-to-code

**Analysis**: Read the specification. Analyse the words to guide your design.

- A noun may be a class, a field or a function.
- A verb is a procedure
- An adjective is a boolean field or function.
- <noun 1> has <noun 2> suggests <noun 2> is a field of class <noun 1>
- <noun 1> of <noun 2> suggest <noun 1> is a field of class <noun 2>

# Process steps

#### Analysis/Design

- Read the specification
- Identify the classes and fields (analyse the **nouns**)
- Identify the constructors (look for these words: initial, create, add)
- Identify the goals (analyse the verbs)
- Write these down on a class diagram, following design rules

#### Coding

- Code the classes and fields
- Code the constructors
- Write a plan for each goal (patterns and key code)
- Code the goals as methods
- Add the main method

# Specification

A customer has one bank account. The initial balance is read in.

The customer can deposit, withdraw, and show the balance with two decimal places.

The amounts to deposit and withdraw are read in.

# NOUNS (classes and fields)

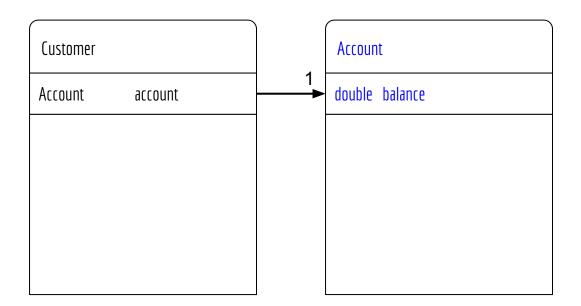
# Nouns

A customer has one bank account.

Customer	
Account	account

### Nouns

The initial balance (of the account) is read in.



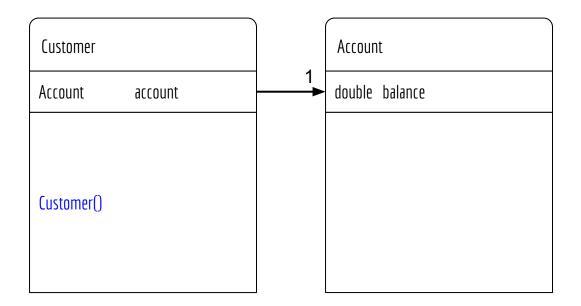
# Code

```
public class Customer {
    private Account account;
    private double balance;
}
```

# Initialisation (constructors)

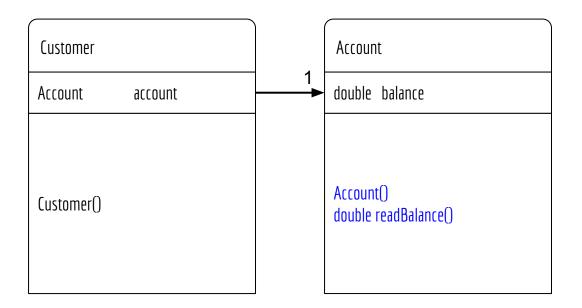
## Initialisation

A customer has one bank account.



### Initialisation

The initial balance (of the account) is read in.



#### Code

```
public class Customer {
    private Account account;

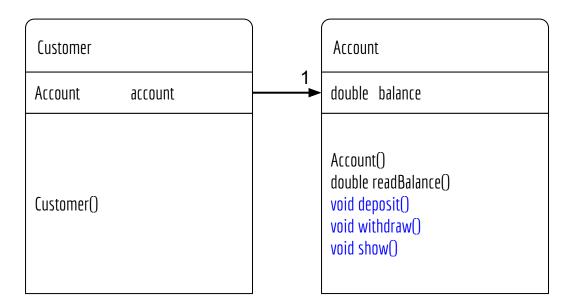
    public Customer() {
        account = new Account();
    }
}
```

```
public class Account {
    private double balance;
    public Account() {
        balance = readBalance();
    private double readBalance() {
         System.out.print("Balance: $");
         return In.nextDouble();
```

# VERBS (Goals)

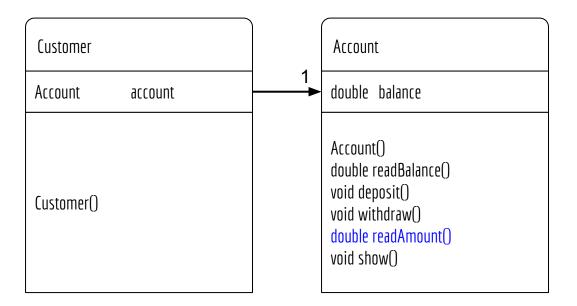
## Verbs

The customer can <u>deposit</u>, <u>withdraw</u>, and <u>show</u> the balance



# Verbs

The amounts to deposit and withdraw are <u>read</u> in.



# Goal analysis

#### List the goals:

- deposit: read in the amount
- withdraw: read in the amount
- show: the balance, formatted

Devise a plan for each goal

# Words to code

Goal	Plan / key code
deposit	<pre>balance += readAmount();</pre>
withdraw	balance -= readAmount();
show	toString, formatted

#### Code

```
public class Account {
   public void deposit() {
      balance += readAmount("deposit");
   public void withdraw() {
      balance -= readAmount("withdraw");
   private double readAmount(String action) {
      System.out.print("Amount to "
               + action + ": $");
      return In.nextDouble();
```

```
public void show() {
   System.out.println(this);
@Override
public String toString() {
   return "The account has $"
         + formatted(balance);
private String formatted(double amount) {
   return new DecimalFormat("###,##0.00")
         .format(amount);
```

# Specification

A customer has one bank account. The initial balance is read in.

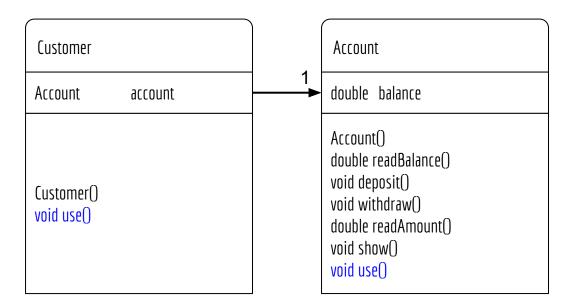
The customer can deposit, withdraw, and show the balance with two decimal places.

The amounts to deposit and withdraw are read in.

The customer uses the account by selecting deposit, withdraw or show from a menu, until the user selects exit.

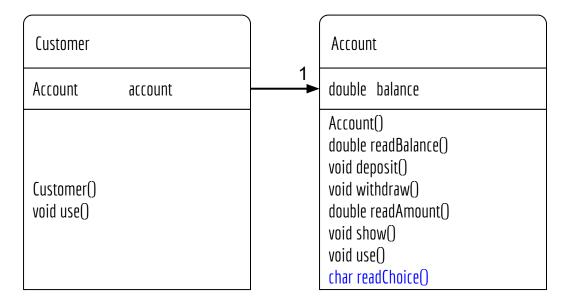
# Verbs

The customer <u>uses</u> the account ...



#### Verbs

The customer uses the account ... by <u>selecting</u> deposit, withdraw or show from a menu



# Sample I/O

```
Choice (d/w/s/x): d
Amount to deposit: $870
Choice (d/w/s/x): s
The account has $870.00
Choice (d/w/s/x): w
Amount to withdraw: $5
Choice (d/w/s/x): s
The account has $865.00
Choice (d/w/s/x): x
```

#### Menu solution

```
public class Account {
  public void use() {
   char choice;
   while ((choice = readChoice()) != 'x') {
      switch (choice) {
      case 'd': deposit(); break;
     case 'w': withdraw(); break;
     case 's': show() break;
     default: help(); break;
 private char readChoice() {
    System.out.print("Choice (d/w/s/x): ");
    return In.nextChar();
```

```
private void help() {
  System.out.println("The menu choices are:");
  System.out.println("d: deposit");
  System.out.println("w: withdraw");
  System.out.println("s: show");
 System.out.println("x: exit");
```

# Menu pattern

Read choice until exit

```
char choice;
while ((choice = readChoice()) != 'x')
```

Execute an action

```
switch (choice) {
   case 'd': deposit(); break;
   case 'w': withdraw(); break;
   case 's': show(); break;
```

One procedure for each action

```
private void deposit()
```

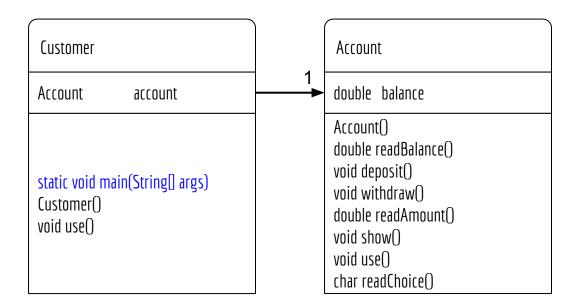
Exit is not a switch case.
 The end-of-input flag 'x' ends the loop.

# Design rule #2: push it right

```
public class Customer {
                                           public class Account {
                                             public void use() {
  public void use() {
                                               char choice;
    account.use();
                                               while ((choice = readChoice()) != 'x') {
                                                  switch (choice) {
                                                  case 'd': deposit(); break;
                                                  case 'w': withdraw(); break;
                                                  case 's': show() break;
                                                  default: help(); break;
```

#### The main method

Put the main method in the left-most (main) class.



#### Code: main method

```
public class Customer {
    public static void main(String[] args) {
        Customer customer = new Customer();
        customer.use();
    }
}
```

- The main method is the ONLY static method.
- The main method should always be two lines:
  - 1. Create the first object.
  - 2. Use the first object.