CSSE 220 Day 4

Implementing Classes in Java, using

- Documented Stubs
- Test-First Programming

Check out *BankAccount* and *WordGames* from SVN

Questions?

Today

- Encapsulation
- Java classes:
  - Implementation details
  - “How To” example
  - Practice in **WordGames** project

Encapsulation in Object-Oriented Software

- *Encapsulation*—separating implementation details from how an object is used
  - Client code sees a *black box* with a known *interface*
  - Implementation can change without changing client

	Functions	Objects
Black box exposes	Function signature	Constructor and method signatures
Encapsulated inside the box	Operation implementation	Data storage and operation implementation

Q1, 2

Bank Account Example

- Essentially based on *Big Java*
  - But using explicit `this` references
  - And putting fields at the top of the class
- Comparing and contrasting with Python
  - Source code with Python examples is in SVN for reference
- Next slide shows the entire class
  - Subsequent slides discuss it piece by piece

The *BankAccount* class

```java
/**
 * A BankAccount has a balance that can be
 * changed by deposits and withdrawals.
 *
 * @author Cay Horstmann.
 */
public class BankAccount {
    private double balance;

    /**
     * Constructs a bank account
     * with a zero balance.
     */
    public BankAccount() {
        this.balance = 0.0;
    }

    /**
     * Constructs a bank account with a
     * given initial balance.
     *
     * @param initialBalance
     *            the initial balance
     */
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    /**
     * Deposits money into the bank account.
     *
     * @param amount
     *            the amount to deposit
     */
    public void deposit(double amount) {
        double newBalance = this.balance + amount;
        this.balance = newBalance;
    }

    /**
     * Withdraws money from the bank account.
     *
     * @param amount
     *            the amount to withdraw
     */
    public void withdraw(double amount) {
        double newBalance = this.balance - amount;
        this.balance = newBalance;
    }

    /**
     * Returns the current balance.
     *
     * @return the current balance
     */
    public double getBalance() {
        return this.balance;
    }
}
```

A class has 3 parts after its header: *fields*, *constructors* and *methods*.

Class Definitions

Javadoc comment precedes the class definition

Name of class, follows the *class* keyword

Java

```
/** javadoc… */
public class BankAccount {
    ...
}
```

Python

```
class BankAccount:
    """docstring..."""
    …
```

Access specifier (aka *visibility*), one of:

- `public`,
- `protected`,
- `private`, or
- default (i.e., no specifier, called *package* visibility)

Java classes are usually declared `public`

Method Definitions

Javadoc comment precedes the method definition (always if the method is public, optionally if the method is private)

```
/** javadoc… */
public void deposit(double amount) {
    ...
}
```

Access specifier

Return type
- *void* means nothing returned

Parameters with types
- Do not list "self" as in Python

Java methods usually are a mix of `public` (when used by objects of other classes) and `private` (when used only within this class).

```
def deposit(self, amount):
    """docstring..."""
    ...
```

Java

Python

Constructor Definitions

Javadoc comment precedes the constructor definition

```
/** javadoc… */
public BankAccount() {
    ...
}

/** javadoc… */
public BankAccount(double initAmount)
{
    ...
}
```

Access specifier

Parameters with types
- Do not list "self" as in Python

No explicit return type
- If you accidentally put a return type, it is a weirdly named method, not a constructor!

Constructor name is *always* the same as the class name

Use *overloading* to handle default argument values

```
def __init__(self,
             initAmt=0.0):
    """docstring..."""
    ...
```

Java

Java constructors are almost always `public`

Python

Public Interface

- The *public interface* of an object:
  - Is the inputs and outputs of the black box
  - Defines how we access the object as a user
  - Consists of:
    - `public` constructors of its class, plus
    - `public` methods of its class
- The *private implementation* of an object consists of:
  - Its (private) instance fields
  - Definitions of its constructors and methods

BankAccount
BankAccount() **BankAccount**(double initAmount) void **deposit**(double amount) void **withdraw**(double amount) double **getBalance**()

The above shows the public interface of BankAccount objects. The next slides show their private implementation.

Generally no Javadoc here, since you should choose variable names that are self-documenting.

Instance Field Definitions

```
/** javadoc as needed… */
private double balance;
```

Access specifier

Type

Name

No instance field definitions in Python

Java instance fields should almost always be `private`

An object is an *instance* of a class

When do you need a field?

Answer: Whenever you have data that is associated with the object, that needs to remain alive as long as the object remains alive.

Java

Python

Constructor Implementation

```java
/** javadoc… */
public BankAccount(double initAmount) {
    this.balance = initAmount;
}
```

```python
def __init__(self,
             initAmt=0.0):
    """docstring..."""
    self.balance = initAmt
```

Use the **this** keyword inside constructors and methods to refer to the implicit argument

Java

Python

Method Implementation

```java
/** javadoc… */
public double getBalance() {
    return this.balance;
}

/** javadoc… */
public void deposit(double amount) {
    double newBalance =
        this.balance + amount;
    this.balance = newBalance;
}
```

```python
def getBalance(self):
    """docstring..."""
    return self.balance


def deposit(self, amount):
    """docstring..."""
    newBal =
        self.balance
        + amount
    self.balance = newBal
```

The deposit method has a **parameter variable** *(amount)*, a **local variable** *(newBalance)*, and a reference to a **field** *(this.balance)*.

- Do you see the difference between these types of variables?

Java

Can omit *return* for *void* methods

Python

Q13,14

The *BankAccount* class (summary)

```
/**
 * A BankAccount has a balance that can b
 * changed by deposits and withdrawals.
 *
 * @author Cay Horstmann.
 */
public class BankAccount {
    private double balance;

    /**
     * Constructs a bank account
     * with a zero balance.
     */
    public BankAccount() {
        this.balance = 0.0;
    }

    /**
     * Constructs a bank account with a
     * given initial balance.
     *
     * @param initialBalance
     *            the initial balance
     */
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    /**
     * Deposits money into the bank account.
     *
     * @param amount
     *            the amount to deposit
     */
    public void deposit(double amount) {
        double newBalance = this.balance + amount;
        this.balance = newBalance;
    }

    /**
     * Withdraws money from the bank account.
     *
     * @param amount
     *            the amount to withdraw
     */
    public void withdraw(double amount) {
        double newBalance = this.balance - amount;
        this.balance = newBalance;
    }

    /**
     * Returns the current balance.
     *
     * @return the current balance
     */
    public double getBalance() {
        return this.balance;
    }
}
```

private field

Constructor

Reference to the field, using the *this* keyword

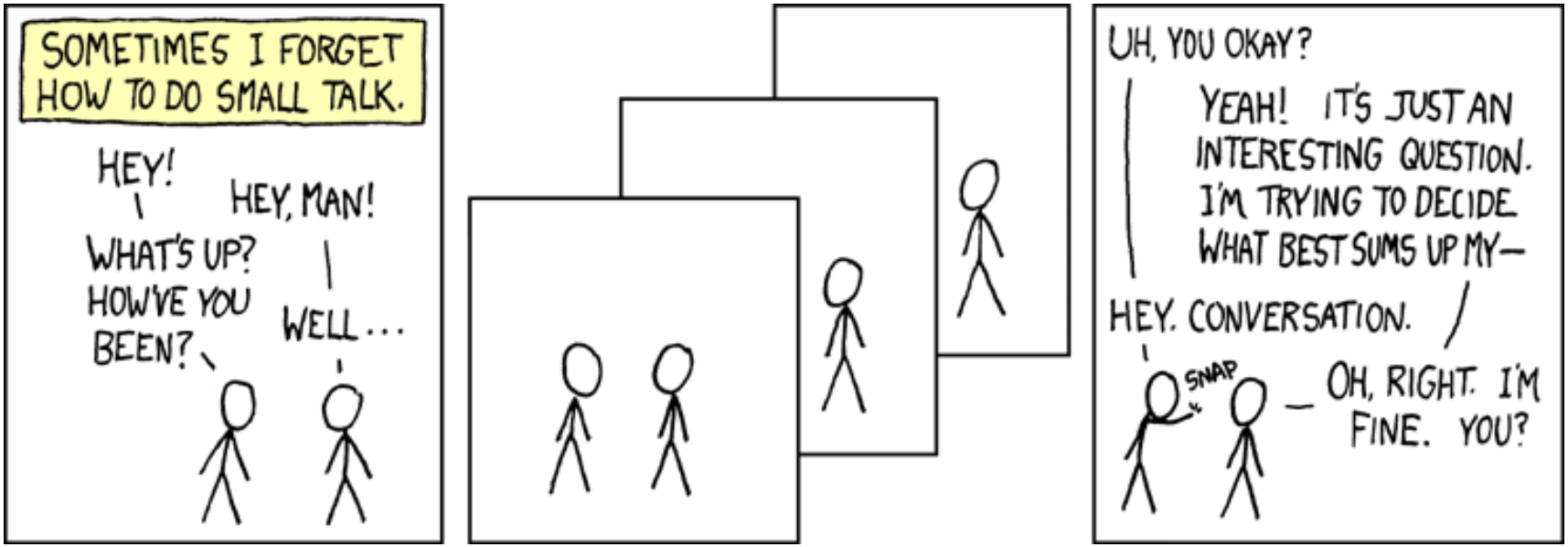
deposit method. Note the use of a parameter, local variable and field.

Withdraw method

A *getter* method that preserves the encapsulation of the private field.

Another constructor. Note *overloading*.

How To: Do Small Talk



But surely I owe you an accurate answer!

How To: Implement a Class

1. **Create the** (initially empty) **class**
   - File ⇒ New ⇒ Class
2. Write ***documented stubs*** for the public interface of the class
   - Find out which methods you are asked to supply
     - If the class ***implements*** an ***interface***, then the interface tells you exactly which methods you must implement
     - And Eclipse volunteers to type their ***stubs*** for you!
   - *Documented stubs* means that you write the documentation at this step (BEFORE fully implementing the constructors and methods, that is, while they are only stubs)
3. **Impleme**
   - Determ
   - Implem ds and addition
4. **Test the**

3. Test and implement each constructor and method
   - Write the test cases BEFORE implementing the constructor/method

Live Coding

The BankAccount project that you checked out of SVN has the code that we just discussed. Examine it at your leisure.

Turn now to the WordGames project that you checked out of SVN. Let's together:

- Study the *StringTransformable* interface.
- Write a *Shouter* class that implements *StringTransformable*. Its *transform* method should return its given String transformed into all UPPER-CASE ("shouting").
  1. Create the (initially empty) class
  2. Write documented stubs (use Quick Fix!)
  3. Write tests, then implement and test the class
  4. Commit your work
- When you are done with Shouter, continue per the WordGames instructions (linked from Homework 4).

Shouter – After Eclipse writes stubs for you

```
1  /**
2   * TODO Put here a description of this class: what its objects are and/or do.
3   *
4   * @author mutchler. Created Dec 4, 2009.
5   */
6  public class Shouter implements StringTransformable {
7
8      @Override
9      public String transform(String stringToTransform) {
10         return null;
11         // TODO Replace this auto-generated method stub by working code.
12     }
13 }
```

Step 1: Create the (initially empty) **class**

- File ⇒ New ⇒ Class

Step 2: Write ***documented stubs*** for the *public interface* of the class

Do you understand what it means to ***implement an interface*** ?

Do you see what a ***stub*** is?

Did you see how Eclipse offered to write the stubs for you?

Note the TODO's: The above is not yet a *documented* stub – see the next slide for that.

Shouter – After you DOCUMENT your stubs

```
1  /**
2   * A Shouter "shouts". That is, given blah, it produces the result of changing
3   * all the characters in blah to upper-case.
4   *
5   * @author David Mutchler. Created December 4, 2009.
6   */
7  public class Shouter implements StringTransformable {
8
9      /**
10      * "Shouts". That is, given blah, returns the result of changing all the
11      * characters in blah to upper-case.
12      *
13      * @param stringToTransform
14      * @return the result of changing all the characters in the given String to
15      *         upper-case.
16      */
17     @Override
18     public String transform(String stringToTransform) {
19         return null;
20         // TODO Replace this auto-generated method stub by working code.
21     }
22 }
```

Do you see the form for Javadoc comments? For their tags?

The form for a class?

Step 1: Create the (initially empty) **class**

- File ⇒ New ⇒ Class

Step 2: Write ***documented stubs*** for the *public interface* of the class

Do you understand what it means to use ***documented stubs*** ?

Do you know what you must document? (Answer: anything *public*.)

ShouterTest

```
12 public class ShouterTest {
13     private Shouter shouter;
14
15     /**
16      * Runs before each test, constructing for each test
17      *
18      * @throws java.lang.Exception
19      */
20     @Before
21     public void setUp() throws Exception {
22         this.shouter = new Shouter();
23     }
24
25     /**
26      * Test method for {@link Shouter#transform(java.lang.String)}. Tests that a
27      * string in all upper case stays that way.
28      */
29     @Test
30     public void testAllUpperCase() {
31         String upperCase = "CAPS LOCK IS CRUISE CONTROL";
32
33         assertEquals(upperCase, this.shouter.transform(upperCase));
34     }
```

Do you understand why you ***write tests before implementing*** ?

Do you see what a ***field*** is? Why one is used here? (Answer: so the Shouter can be reused in all the tests. It would also be OK to construct a new Shouter for each test.)

Did you see how the *assertEquals* method works? How you specify a test? How the @Before and @Test annotations work?

Look at the (many) tests we supplied in *ShouterTest*. Are they a good set of tests, with good *coverage*? Could we test how *fast* Shouter's *transform* runs?

Step 1: Create the (initially empty) **class**

Step 2: Write ***documented stubs*** for the *public interface* of the class

Step 3a: We provided ***some JUnit tests*** for the *transform* method of each class.

Shouter – After you implement it

```java
/**
 * A Shouter "shouts". That is, given blah, it produces the result of changing
 * all the characters in blah to upper-case.
 *
 * @author David Mutchler. Created December 4, 2009.
 */
public class Shouter implements StringTransformable {

    /**
     * "Shouts". That is, given blah, returns the result of changing all the
     * characters in blah to upper-case.
     *
     * @param stringToTransform
     * @return the result of changing all the characters in the given String to
     *         upper-case.
     */
    @Override
    public String transform(String stringToTransform) {
        return stringToTransform.toUpperCase();
    }
}
```

Do you understand how Eclipse helps you find the right method to apply to the *stringToTransform*? (Pause after typing the dot.)

Do you see why you don't need a local variable?

Do you know Java's 1st dirty little secret about constructors? (Namely, that Java inserted a do-nothing constructor for you! More on this later.)

Censor

- **Censor**: given ***blah***, produces the result of replacing each occurrence of the *character* (not *string*) ***foo*** in ***blah*** with an asterisk, where ***foo*** is the character that the particular Censor censors.
- How do you deal with *foo*?
  - Can it be a parameter of *transform*?
    - No, that violates the StringTransformable interface
  - Can it be a local variable of *transform*?
    - No, it needs to live for the entire lifetime of the Censor.
  - What's left?
    - Answer: It is a ***field***! (What is a sensible name for the field?)
- How do you initialize the field for *foo*?
  - Answer: by using Censor's constructors!

Live Coding

Let's together:

- Write a *Censor* class that implements *StringTransformable*. Its *transform* method should return the result of replacing each occurrence of the *character* (not *string*) ***foo*** in ***blah*** with an asterisk, where ***foo*** is the character that the particular Censor censors.
  1. Create the (initially empty) class
  2. Write documented stubs (use Quick Fix!)
  3. Write tests, then implement and test the class
  4. Commit your work
- When you are done with Censor, continue per the WordGames instructions (linked from Homework 4).

Censor – After Eclipse writes stubs for you

```
1  /**
2   * TODO Put here a description of this class: wha
3   *
4   * @author mutchler. Created Dec 7, 2009.
5   */
6  public class Censor implements StringTransformable {
7
8      /**
9       * TODO Put here a description of what this constructor does.
10      */
11     public Censor() {
12         // TODO
13     }
14
15     /**
16      * TODO Put here a description of what this constructor does.
17      * @param characterToCensor
18      */
19     public Censor(char characterToCensor) {
20         // TODO
21     }
22
23     @Override
24     public String transform(String stringToTransform) {
25         return null;
26         // TODO Replace this auto-generated method stub by working code.
27     }
28 }
```

Step 1: Create the (initially empty) **class**

Step 2: Write ***documented stubs*** for the *public interface* of the class

Do you see why you need stubs for the two Censor constructors? (See the calls to them in the CensorTest class.)

Do you understand what it means to ***implement an interface*** ?

Do you see what a ***stub*** is? Did you see how Eclipse offered to write the stubs for you?

Note the TODO's: ***The above is not yet a documented stub*** – see the next slide for that.

```java
/**
 * A Censor "censors". That is, given blah, it produces the result of replacing
 * each occurrence of the character (not string) foo in blah with an asterisk,
 * where foo is the character that the particular Censor censors.
 *
 * @author David Mutchler. Created December 4, 2009.
 */
public class Censor implements StringTransformable {

    /**
     * Sets 'e' as the default character to censor.
     */
    public Censor() {
        // TODO
    }

    /**
     * Sets the given character as the character to censor.
     *
     * @param characterToCensor
     */
    public Censor(char characterToCensor) {
        // TODO
    }

    /**
     * Returns the result of replacing each occurrence of the character (not
     * string) foo in the given String with an asterisk, where foo is the
     * character that this particular Censor censors.
     *
     * @param stringToTransform
     * @return the result of replacing each occurrence of the character (not
     *         string) foo in the given String with an asterisk, where foo is
     *         the character that this particular Censor censors.
     */
    @Override
    public String transform(String stringToTransform) {
        return null;
        // TODO Replace this auto-generated method stub by working code.
    }
}
```

```
private char characterToCensor;
```

```
this.characterToCensor = 'e';
```

```
this.characterToCensor =
        characterToCensure;
```

Censor – After you DOCUMENT your stubs

Step 1: Create the (initially empty) **class**

Step 2: Write ***documented stubs*** for the *public interface* of the class

Do you understand what it means to use ***documented stubs*** ?

Do you know what you must document? (Answer: anything *public*.)

```java
public class Censor implements StringTransformable {

    private char characterToCensor;

    /**
     * Sets 'e' as the default character to censor.
     */
    public Censor() {
        this.characterToCensor = 'e';
    }

    /**
     * Sets the given character as the character to censor.
     *
     * @param characterToCensor
     */
    public Censor(char characterToCensor) {
        this.characterToCensor = characterToCensor;
    }

    /**
     * Returns the result of replacing each occurrence of the character (not
     * string) foo in the given String with an asterisk, where foo is the
     * character that this particular Censor censors.
     *
     * @param stringToTransform
     * @return the result of replacing each occurrence of the character (not
     *         string) foo in the given String with an asterisk, where foo is
     *         the character that this particular Censor censors.
     */
    @Override
    public String transform(String stringToTransform) {
        return stringToTransform.replace(this.characterToCensor, '*');
    }
}
```

Censor final version

Do you see why Censor needs a field? How the field is initialized? How the field is referenced (using *this*)?

How Censor has two constructors? How those constructors are called in CensorTest?

Should we have made a field for the '*' constant? (Probably.)