

A 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

```
1 /**
2  * A BankAccount has a balance that can be
3  * changed by deposits and withdrawals.
4  *
5  * @author Cay Horstmann.
6  */
7 public class BankAccount {
8     private double balance;
9
10    /**
11     * Constructs a bank account
12     * with a zero balance.
13     */
14    public BankAccount() {
15        this.balance = 0.0;
16    }
17
18    /**
19     * Constructs a bank account with a
20     * given initial balance.
21     *
22     * @param initialBalance
23     *         the initial balance
24     */
25    public BankAccount(double initialBalance) {
26        this.balance = initialBalance;
27    }
```

```
29 /**
30  * Deposits money into the bank account.
31  *
32  * @param amount
33  *         the amount to deposit
34  */
35    public void deposit(double amount) {
36        double newBalance = this.balance + amount;
37        this.balance = newBalance;
38    }
39
40    /**
41     * Withdraws money from the bank account.
42     *
43     * @param amount
44     *         the amount to withdraw
45     */
46    public void withdraw(double amount) {
47        double newBalance = this.balance - amount;
48        this.balance = newBalance;
49    }
50
51    /**
52     * Returns the current balance.
53     *
54     * @return the current balance
55     */
56    public double getBalance() {
57        return this.balance;
58    }
59 }
```

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

Javadoc comment precedes the class definition

Name of class, follows the *class* keyword

Class Definitions

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

```
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

Java

Python

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

Method Definitions

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

Access specifier

Return type
• *void* means nothing returned

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..."""  
    ...
```

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

Java

Python

Javadoc comment precedes the constructor definition

Constructor Definitions

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

Access specifier

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

No explicit return type

- If you accidentally put a return type, it is a weirdly named method, not a constructor!

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

Parameters with types

- Do not list "self" as in Python

Use *overloading* to handle default argument values

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

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
<pre>BankAccount () BankAccount (double initAmount) void deposit (double amount) void withdraw (double amount) double getBalance ()</pre>

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

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.

No instance field definitions in Python

Java

Python

Constructor Implementation

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

```
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

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

```
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

The *BankAccount* class (summary)

```
1 /**
2  * A BankAccount has a balance that can k
3  * changed by deposits and withdrawals.
4  *
5  * @author Cay Horstmann.
6  */
7 public class BankAccount {
8     private double balance;
9
10    /**
11     * Constructs a bank account
12     * with a zero balance.
13     */
14    public BankAccount() {
15        this.balance = 0.0;
16    }
17
18    /**
19     * Constructs a bank account with a
20     * given initial balance.
21     *
22     * @param initialBalance
23     *         the initial balance
24     */
25    public BankAccount(double initialBalance) {
26        this.balance = initialBalance;
27    }
```

private field

Constructor

Reference to the field,
using the *this* keyword

```
29 /**
30  * Deposits money into the bank account.
31  *
32  * @param amount
33  *         the amount to deposit
34  */
35 public void deposit(double amount) {
36     double newBalance = this.balance + amount;
37     this.balance = newBalance;
38 }
39
40 /**
41  * Withdraws money from the bank account.
42  *
43  * @param amount
44  *         the amount to withdraw
45  */
46 public void withdraw(double amount) {
47     double newBalance = this.balance - amount;
48     this.balance = newBalance;
49 }
50
51 /**
52  * Returns the current balance.
53  *
54  * @return the current balance
55  */
56 public double getBalance() {
57     return this.balance;
58 }
59 }
```

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*.

Live Coding

WordGames Shouter

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     }
}
```

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.

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?

Shouter – After you implement it

```
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 stringToTransform.toUpperCase(); ←
20    }
21 }
```

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 a string *inputString*, produces a new string by replacing each occurrence of `charToCensor` with a “*” (an asterisk).
- ▶ How do you deal with `charToCensor` ?
 - 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 `charToCensor` ?
 - Answer: by using Censor’s constructors!

Live Coding

WordGames Censor

Censor – After Eclipse writes stubs for you

```
1 /**
2  * TODO Put here a description of this class: what
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.

Censor – After you DOCUMENT your stubs

```
1 /**
2  * A Censor "censors". That is, given blah, it produces the result of replacing
3  * each occurrence of the character (not string) foo in blah with an asterisk,
4  * where foo is the character that the particular Censor censors.
5  *
6  * @author David Mutchler. Created December 4, 2009.
7  */
8 public class Censor implements StringTransformable {
9
10     /** ← private char characterToCensor;
11      * Sets 'e' as the default character to censor.
12      */
13     public Censor() {
14         // TODO ← this.characterToCensor = 'e';
15     }
16
17     /**
18      * Sets the given character as the character to censor.
19      *
20      * @param characterToCensor
21      */
22     public Censor(char characterToCensor) {
23         // TODO ← this.characterToCensor =
24                 characterToCensor;
25     }
26
27     /**
28      * Returns the result of replacing each occurrence of the character (not
29      * string) foo in the given String with an asterisk, where foo is the
30      * character that this particular Censor censors.
31      *
32      * @param stringToTransform
33      * @return the result of replacing each occurrence of the character (not
34      *         string) foo in the given String with an asterisk, where foo is
35      *         the character that this particular Censor censors.
36      */
37     @Override
38     public String transform(String stringToTransform) {
39         return null;
40         // TODO Replace this auto-generated method stub by working code.
41     }
42 }
```

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*.)

```

8 public class Censor implements StringTransformable {
9
10     private char characterToCensor;
11
12     /**
13      * Sets 'e' as the default character to censor.
14      */
15     public Censor() {
16         this.characterToCensor = 'e';
17     }
18
19     /**
20      * Sets the given character as the character to censor.
21      *
22      * @param characterToCensor
23      */
24     public Censor(char characterToCensor) {
25         this.characterToCensor = characterToCensor;
26     }
27
28     /**
29      * Returns the result of replacing each occurrence of the character (not
30      * string) foo in the given String with an asterisk, where foo is the
31      * character that this particular Censor censors.
32      *
33      * @param stringToTransform
34      * @return the result of replacing each occurrence of the character (not
35      *         string) foo in the given String with an asterisk, where foo is
36      *         the character that this particular Censor censors.
37      */
38     @Override
39     public String transform(String stringToTransform) {
40         return stringToTransform.replace(this.characterToCensor, '*');
41     }
42 }

```

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.)