Object Oriented Programming: Class Design
Learning Objectives

- Create abstractions for entities you want programs to model
- Use encapsulation to define safe ways of using objects you create
- Recognize the public interface created by a class
- Evaluate the effectiveness of a class' design choices
Creating Abstractions

- Objects in Java know some things and they can do some stuff.
- When designing an object (real or software) for someone to use, the object should include only the properties & behaviors it needs for a given user to use it!
When you drive a car, you're shown information on speed, fuel, and RPMs. You have the choice of actions like *drive*, *park*, *reverse*, etc.

As the driver of a car, you are **not** concerned with the crankshaft & spark plugs & cylinders & valves & ....
Class Design as Abstraction

- Decide what you want a user to be able to do with the objects that you are creating.
  - These will be the methods of the class

- Identify all of the properties such an object must have to be able to implement those desired behaviors
  - These will be the instance variables of the class
  - Some properties can be exposed to the user, some are only necessary for internal use

- Implement the public methods to expose important properties and perform essential behaviors
We're tasked with designing a Bank Account object that a team at TD Bank will use in their software systems.

What are the essential operations of a Bank Account object?
What information does a Bank Account need to store in order to perform these operations?
Bank Account Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>check balance</td>
<td>none</td>
<td>amount</td>
</tr>
<tr>
<td>deposit</td>
<td>amount</td>
<td>none (?)</td>
</tr>
<tr>
<td>withdraw</td>
<td>amount</td>
<td>none (?)</td>
</tr>
</tbody>
</table>

Note: it's not that a Bank Account *couldn't* do more. But from our stated problem, this is all a Bank Account must do at a minimum.
Core Operations → Class Skeleton

```java
public class BankAccount {
    public double checkBalance() {
        return 0.0; // TODO
    }

    public void deposit(double amount) { // TODO
    }

    public void withdraw(double amount) { // TODO
    }
}
```
Adding in the Instance Variables

To check an account's balance, make a deposit to the account, or withdraw an amount from the account, we need to know:

- the account balance

That's it! An account could store information about its owner, account type, interest rates, withdrawal limits, etc. But all of that is extraneous for the operations we chose.
public class BankAccount {
    public double balance; // public for now (bad), but will change!
    public BankAccount(double startingAmount) {
        // TODO
    }
    public double checkBalance() {
        return 0.0; // TODO
    }
    public void deposit(double amount) {
        // TODO
    }
    public void withdraw(double amount) {
        // TODO
    }
}
Implementing the Class: a First Pass

Now we can move to implement our methods!

- An important first step is to formalize the requirements by writing some kind of tests.
  - unit tests or just rudimentary `main` method tests are both OK
- Writing the tests first requires us to understand what the expected behavior should be before we implement the method.
  - Pretty important to know what you intend to do before doing it!
  - (Yes, the tests will fail at first because there is no implementation.)
Writing a User Story Test

A **user story** is a narrativized description of how a user will use the program you're writing.

"Customers should be able to create a new Bank Account with an initial deposit of $10. They should be able to deposit $20 and see the change reflected in the account afterwards."
Writing a User Story Test (in main)

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(10.0);
    double startingBalance = harrysAccount.getBalance();
    System.out.println("Harry's new account has a balance of " + startingBalance);
    System.out.println("(Balance should be $10)");

    harrysAccount.deposit(20);
    double newBalance = harrysAccount.getBalance();
    System.out.println("Harry's account has a balance of " + newBalance);
    System.out.println("(Balance should be $30)");
}
```
Writing a User Story Test (in JUnit)

```java
@Test
public void testUserStoryOne() {
    BankAccount harrysAccount = new BankAccount(10.0);
    double actualBalance = harrysAccount.getBalance();
    double expectedBalance = 10
    assertEquals(expectedBalance, actualBalance, 0.01);

    harrysAccount.deposit(20);
    actualBalance = harrysAccount.getBalance();
    expectedBalance = 30;
    assertEquals(expectedBalance, actualBalance, 0.01);
}
```

JUnit requires that double comparisons are made with a "DELTA" value that represents an error tolerance.
"Customers should be able to create a new Bank Account with an initial deposit of $10. They should be able to withdraw $3, leaving a balance of $7."

"Customers should be able to create a new Bank Account with an initial deposit of $10. They may attempt to withdraw an amount greater than their balance, but this attempt should have no effect"

"Customers should be able to attempt to withdraw or deposit a negative amount, but these attempts should have no effect on their account balances"
@Test
class public void testUserStoryTwo() {
    BankAccount harrysAccount = new BankAccount(10.0);
harrysAccount.withdraw(3);
double actualBalance = harrysAccount.getBalance();
double expectedBalance = 7;
assertEquals(expectedBalance, actualBalance, 0.01);
}

@Test
class public void testUserStoryThree() {
    BankAccount harrysAccount = new BankAccount(10.0);
harrysAccount.withdraw(11);
double actualBalance = harrysAccount.getBalance();
double expectedBalance = 10;
assertEquals(expectedBalance, actualBalance, 0.01);
}
@Test  
public void testUserStoryFour() {
    BankAccount harrysAccount = new BankAccount(10.0);
    harrysAccount.withdraw(-3);
    double actualBalance = harrysAccount.getBalance();
    double expectedBalance = 10;
    assertEquals(expectedBalance, actualBalance, 0.01);
    harrysAccount.deposit(-1000);
    actualBalance = harrysAccount.getBalance();
    expectedBalance = 10;
    assertEquals(expectedBalance, actualBalance, 0.01);
}
public class BankAccount {
    public double balance; // public for now (bad), but will change!
    public BankAccount(double startingAmount) {
        if (startingAmount < 0) {
            balance = 0;
        } else {
            balance = startingAmount;
        }
    }
    public double checkBalance() {
        return balance
    }
    ...
}
public class BankAccount {
    ...

    public void deposit(double amount) {
        if (amount < 0) {
            return;
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount < 0 || amount > balance) {
            return;
        }
        balance -= amount;
    }
}
Encapsulation

Encapsulation refers to the process of hiding the data from the user and using methods to provide data access.

Why is this important?

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(10);
    BankAccount enemysAccount = new BankAccount(100);
    harrysAccount.balance = 1000000000
    enemysAccount.balance = -999999999
}
```
Public Instance Variables Violate Encapsulation

Make instance variables private in order to regulate access to important information!
Write getters & setters & other public methods that allow mediated access.

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(100);
    harrysAccount.withdraw(999999999); // refused!
}
```

Saved by a public method!
The Public Interface

An interface between two objects is the point at which they meet and interact.

- In Java, we'll see that the word has another meaning later

A class' public interface refers to the sum of all public instance variables and methods that can be interacted with in other classes.

- Effective class design exposes all necessary behaviors in the public interface
- Poor class design exposes unnecessary methods and sensitive data
private boolean isInvalidAmount(double amount) {
    return amount < 0;
}

Keep helper methods private!

- These are not intended for outside use, so nobody will miss them when using your object
- More public methods → more clutter for people trying to understand your classes--
Object Oriented Programming: Class Design
Learning Objectives

• Create abstractions for entities you want programs to model
• Use encapsulation to define safe ways of using objects you create
• Recognize the public interface created by a class
• Evaluate the effectiveness of a class' design choices
Creating Abstractions

- Objects in Java know some things and they can do some stuff.
- When designing an object (real or software) for someone to use, the object should include only the properties & behaviors it needs for a given user to use it!
When you drive a car, you're shown information on speed, fuel, and RPMs. You have the choice of actions like drive, park, reverse, etc.
As the driver of a car, you are not concerned with the crankshaft & spark plugs & cylinders & valves & ....
Class Design as Abstraction

- Decide what you want a user to be able to do with the objects that you are creating.
  - These will be the methods of the class
- Identify all of the properties such an object must have to be able to implement those desired behaviors
  - These will be the instance variables of the class
  - Some properties can be exposed to the user, some are only necessary for internal use
- Implement the public methods to expose important properties and perform essential behaviors
Bank Account Example

We're tasked with designing a Bank Account object that a team at TD Bank will use in their software systems.

What are the essential operations of a Bank Account object? What information does a Bank Account need to store in order to perform these operations?
Bank Account Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>check balance</td>
<td>none</td>
<td>amount</td>
</tr>
<tr>
<td>deposit</td>
<td>amount</td>
<td>none (?)</td>
</tr>
<tr>
<td>withdraw</td>
<td>amount</td>
<td>none (?)</td>
</tr>
</tbody>
</table>

Note: it's not that a Bank Account couldn't do more. But from our stated problem, this is all a Bank Account must do at a minimum.
public class BankAccount {
    public double checkBalance() {
        return 0.0; // TODO
    }

    public void deposit(double amount) {
        // TODO
    }

    public void withdraw(double amount) {
        // TODO
    }
}
Adding in the Instance Variables

To check an account's balance, make a deposit to the account, or withdraw an amount from the account, we need to know:

- **the account balance**

That's it! An account could store information about its owner, account type, interest rates, withdrawal limits, etc. But all of that is extraneous for the operations we chose.
public class BankAccount {
    public double balance; // public for now (bad), but will change!
    public BankAccount(double startingAmount) {
        // TODO
    }
    public double checkBalance() {
        return 0.0; // TODO
    }
    public void deposit(double amount) {
        // TODO
    }
    public void withdraw(double amount) {
        // TODO
    }
}
Implementing the Class: a First Pass

Now we can move to implement our methods!

- An important first step is to formalize the requirements by writing some kind of tests.
  - unit tests or just rudimentary main method tests are both OK
- Writing the tests first requires us to understand what the expected behavior should be before we implement the method.
  - Pretty important to know what you intend to do before doing it!
  - (Yes, the tests will fail at first because there is no implementation.)
Writing a User Story Test

A user story is a narrativized description of how a user will use the program you're writing.

"Customers should be able to create a new Bank Account with an initial deposit of $10. They should be able to deposit $20 and see the change reflected in the account afterwards."
Writing a User Story Test (in main)

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(10.0);
    double startingBalance = harrysAccount.getBalance();
    System.out.println("Harry's new account has a balance of "+ startingBalance);
    System.out.println("(Balance should be $10)"可谓);

    harrysAccount.deposit(20);
    double newBalance = harrysAccount.getBalance();
    System.out.println("Harry's account has a balance of "+ newBalance);
    System.out.println("(Balance should be $30)"可谓);
}
```
Writing a User Story Test (in JUnit)

```java
@Test
public void testUserStoryOne() {
    BankAccount harrysAccount = new BankAccount(10.0);
    double actualBalance = harrysAccount.getBalance();
    double expectedBalance = 10
    assertEquals(expectedBalance, actualBalance, 0.01);

    harrysAccount.deposit(20);
    actualBalance = harrysAccount.getBalance();
    expectedBalance = 30;
    assertEquals(expectedBalance, actualBalance, 0.01);
}
```

JUnit requires that double comparisons are made with a "DELTA" value that represents an error tolerance.
Other Potential Stories to Test

"Customers should be able to create a new Bank Account with an initial deposit of $10. They should be able to withdraw $3, leaving a balance of $7."

"Customers should be able to create a new Bank Account with an initial deposit of $10. They may attempt to withdraw an amount greater than their balance, but this attempt should have no effect"

"Customers should be able to attempt to withdraw or deposit a negative amount, but these attempts should have no effect on their account balances"
@Test
public void testUserStoryTwo() {
    BankAccount harrysAccount = new BankAccount(10.0);
    harrysAccount.withdraw(3);
    double actualBalance = harrysAccount.getBalance();
    double expectedBalance = 7;
    assertEquals(expectedBalance, actualBalance, 0.01);
}

@Test
public void testUserStoryThree() {
    BankAccount harrysAccount = new BankAccount(10.0);
    harrysAccount.withdraw(11);
    double actualBalance = harrysAccount.getBalance();
    double expectedBalance = 10;
    assertEquals(expectedBalance, actualBalance, 0.01);
}
@Test
class UserStoryFour {
    public void testUserStoryFour() {
        BankAccount harrysAccount = new BankAccount(10.0);
        harrysAccount.withdraw(-3);
        double actualBalance = harrysAccount.getBalance();
        double expectedBalance = 10;
        assertEquals(expectedBalance, actualBalance, 0.01);

        harrysAccount.deposit(-1000);
        actualBalance = harrysAccount.getBalance();
        expectedBalance = 10;
        assertEquals(expectedBalance, actualBalance, 0.01);
    }
}
Writing Methods

```java
public class BankAccount {
    public double balance; // public for now (bad), but will change!
    public BankAccount(double startingAmount) {
        if (startingAmount < 0) {
            balance = 0;
        } else {
            balance = startingAmount;
        }
    }
    public double checkBalance() {
        return balance
    }
    ...
}
```
public class BankAccount {
    
    public void deposit(double amount) {
        if (amount < 0) {
            return;
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount < 0 || amount > balance) {
            return;
        }
        balance -= amount;
    }
}

CLASS DESIGN

Writing Methods
**Encapsulation**

**Encapsulation** refers to the process of hiding the data from the user and using methods to provide data access.

Why is this important?

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(10);
    BankAccount enemysAccount = new BankAccount(100);
    harrysAccount.balance = 10000000000
    enemysAccount.balance = -9999999999
}
```
**Public Instance Variables Violate Encapsulation**

Make instance variables private in order to regulate access to important information! Write getters & setters & other public methods that allow mediated access.

```java
public static void main(String[] args) {
    BankAccount harrysAccount = new BankAccount(100);
    harrysAccount.withdraw(999999999); // refused!
}
```

Saved by a public method!
The Public Interface

An **interface** between two objects is the point at which they meet and interact.

- In Java, we'll see that the word has another meaning later

A class' **public interface** refers to the sum of all public instance variables and methods that can be interacted with in other classes.

- Effective class design exposes all necessary behaviors in the public interface
- Poor class design exposes unnecessary methods and sensitive data
Helper Methods & Public Interface

Sometimes it's helpful to write a helper method when implementing your class:

```java
private boolean isInvalidAmount(double amount) {
    return amount < 0;
}
```

Keep helper methods private!

- These are not intended for outside use, so nobody will miss them when using your object
- More public methods → more clutter for people trying to understand your classes