Another Design Pattern

- Suppose I had the following code:

```java
public static void main(String[] args){
    BankAccount myBankAccount;
    myBankAccount.credit(200);
}
```

- What happens if I try to execute this code?
Another Design Pattern

- I get a NullPointerException and my program stops.

- All objects are null until they are instantiated!

Another Design Pattern

- In past examples, we have implemented a “default” constructor that took no parameters and created an account with no account number and a zero balance.

- Could there be any problems with this?
Another Design Pattern

- This implementation could lead to potential problems in the program.

- I could mistakenly create lots of “blank” accounts, and since they work like any real amount I could add or remove money from them.

Moving on...

- So, I want to avoid unnecessary NullPointerExceptions in my program, but I also don’t want to open up the misuse of my BankAccount class.

- If only there were a way to create a Null Bank Account, that looked like a real bank account, but wouldn’t function like one.
The Null Object design pattern

- **Problem:**
  - Checking to see if an instance of my class is null before calling a method can add a lot of complexity, and must be handled in many locations in the code.

- **Applicability:**
  - The Null Object pattern can be used when you want to provide "do nothing" functionality for a class in a single place.
  - This can also be used to create a stub of a class for unit testing purposes.

---

The Null Object design pattern

- **Motivation / Value:**
  - Allows me to implement a consistent response in the case that an object is null.
  - Simplifies code elsewhere because all instances can be processed without concern of a null pointer.
  - Reduces the complexity of code by removing the need for checking for null values and responding accordingly.
The Null Object design pattern

Solution:
- Create a NullObject class that implements all of the methods by either doing nothing, or some other default functionality.

OR
Back to our example....

- Now we can do this:
  ```java
  public static void main(String[] args)
  BankAccount myBankAccount = new NullBankAccount();
  myBankAccount.credit(200);
  ```

- And our object will do nothing:
  ```java
  public class NullBankAccount() extends BankAccount{
    . . .
    public void credit(double amount){
      //do nothing.
    }
  }
  ```

An example....

- Implement a NullDeck class for a DeckOfCards interface.
  ```java
  public interface DeckOfCards{
    public IPlayingCard drawCard() throws OutOfCardsException;
    public void add(IPlayingCard cardToAdd);
    public void add(IPlayingCard[] cardsToAdd);  // Future: Method required.
    public void add(IDeckOfCards cardsToAdd);
    public void shuffle();
    public IHand[] dealCards(int numOfHands) throws OutOfCardsException;
    public IHand[] dealCards(int numOfHands, int numOfCards) throws OutOfCardsException;
    public boolean isEmpty();
    public void reset();
  }
  ```
public class NullDeck implements IDeckOfCards{
    // Constructor
    public NullDeck(){
        // Do nothing.
    }

    public IPlayingCard drawCard() throws OutOfCardsException{
        throw new OutOfCardsException();
    }

    public void addCardToADD()
    // Do nothing
    }

    public void addCard(IPlayingCard[] cardsToAdd)
    // Do nothing
    }

    public IHand[] dealCards(int numOfHands) throws OutOfCardException{
        throw new OutOfCardsException();
    }

    public IHand[] dealCards(int numOfHands, int numOfCards)
        throws OutOfCardException{
            throw new OutOfCardsException();
        }
Your Turn....

```java
public boolean isEmpty()
{
    return true;
}

public void reset()
{
    //do nothing
}
}
```

Another Design Pattern

- **Composite Design Pattern.**
  - We’ve talked about composition before, where one object exists inside of another. (For example: We have a PlayingCard object inside of a StandardDeck object.)
  - This pattern uses a recursive object design.
Recursion

■ To understand recursion, you must first understand recursion.

■ What is a recursive method?
  - A method that calls itself.

■ What is a recursive class?
  - A class that is composed of the same class.

The Composite design pattern

■ Problem:
  - In some situations an object may represent a group of objects and another may represent an individual item. However, we want to treat the individual object and the group (composition) object the same.

■ Applicability:
  - The Composite pattern can be used when we want to interact with objects and group of objects with the same interface.
  - A group can contain an individual item or other groups.
  - The group and the individual share a common interface.
The Composite design pattern

- Solution:
  - Create a class that is a composite of it’s parent.

- Motivation / Value:
  - Allows us to implement code with a consistent interface. I can collaborate with an individual object and a composition of objects the same way.
  - What data structure does this pattern model?

A Tree!
A Composite Example

- Suppose we needed to create a system to run a jobset in a batch environment that runs a series of processes in a sequence.
- Some processes are a single step, while other processes are a series of steps.
- Our system needs to be able call all of the processes with a common interface.
  - This sounds remarkably like the problem the composite design pattern solves.

A Solution with Composite
Design a Binary Tree

- Using these Null Object and Composite design patterns, design a BinaryTree that can: that can store integers.

- The tree should be structured such that:
  - There are no duplicate numbers in the tree
  - The left subtree contains all the values less than the number stored at the node.
  - The right side contains all the values greater than the number stored at the node.

- The target interface should be:

```java
<<interface>>
IntegerTree

+ add(int) : void
+ clear() : void
+ exists(int) : boolean
+ height() : boolean
+ getRootValue() : int
+ isEmpty() : boolean
```