Null Object
Null Object

NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing
Null Object & Binary Search Tree

Node

BinaryNode

NullNode

10

5

8

20

Null Node

Null Node

Null Node

Null Node
Comparing Normal Tree vs Tree with Null Nodes

Normal BST

```java
public class BinaryNode {
    Node left;
    Node right;
    int key;

    public boolean includes( int value ) {
        if (key == value)
            return true;
        else if (value < key && left == null )
            return false;
        else if (value < key)
            return left.includes( value );
        else if (right == null)
            return false;
        else
            return right.includes(value);
    }
    etc.
}
```

With Null Nodes

```java
public class BinaryNode extends Node {
    Node left = new NullNode();
    Node right = new NullNode();
    int key;

    public boolean includes( int value ) {
        if (key == value)
            return true;
        else if (value < key )
            return left.includes( value );
        else
            return right.includes(value);
    }
    etc.
}
```

```java
public class NullNode extends Node {
    public boolean includes( int value ) {
        return false;
    }
    etc.
}
```
Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way
Applicability - When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way
Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable
Implementation

Too Many classes

Multiple Do-nothing meanings

Try Adapter pattern

Transformation to RealObject

Try Proxy pattern
Refactoring: Introduce Null Object

You have repeated checks for a null value

Replace the null value with a null object

```java
if (customer == null)
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();
```
Create Null Subclass

```java
public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}

boolean isNull() { return true;}
```

Compile
Replace all nulls with null object

class SomeClassThatReturnCustomers {

    public Customer getCustomer() {
        if (_customer == null )
            return Customer.newNull();
        else
            return _customer;
    }
    etc.
}

Compile
Replace all null checks with isNull()

```java
if (customer == null)
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();

if (customer.isNull())
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();
```

Compile and test
Find an operation clients invoke if not null
Add Operation to Null class

```java
if (customer.isNull())
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();
```

class NullCustomer {
    public BillingPlan getPlan() {
        return BillingPlan.basic();
    }
}
```
Remove the Condition Check

```java
if (customer.isNull())
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();
```

Compile & Test
Repeat last two slides for each operation
clients check if null
Special Case
Special Case

Represent special cases by a subclass

Use when multiple places that have same behavior

After conditional check for particular class instance

Or same behavior after a null check
Object-Oriented Recursion
A method polymorphically sends its message to a different receiver. Eventually a method is called that performs the task. The recursion then unwinds back to the original message send.
Without tail recursion doing this on a long linked list could cause a stack overflow. So while it may not be a good idea to do this on a linked list it does provide a simple example to explain the idea.
class HeadNode {
    public void add(int value) {
        next.add(value);
    }
}

class Node {
    public void add(int value) {
        if (element > value)
            prependNode(value);
        else
            next.add(value);
    }
}

class TailNode {
    public void add(int value) {
        prependNode(value);
    }
}
Decorator
Prime Directive
Data + Operations
Decorator Pattern
Adds responsibilities to individual objects

Dynamically
Transparency
import java.io.*;
import sdsu.io.*;

class ReadingFileExample
{
    public static void main( String args[] ) throws Exception
    {
        FileInputStream inputFile;
        BufferedInputStream bufferedFile;
        ASCIIInputStream cin;

        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );

        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );
    }
}
ConcreteDecoratorB
Component
operation()
Decorator
operation()
component
ConcreteComponent
operation()
ConcreteDecoratorA
ConcreteDecoratorB
Decorator forwards all component operations
Favor Composition over Inheritance
Refactoring: Move Embellishment to Decorator

Client \rightarrow aBinaryTree
\hspace{2cm} \text{toArray}

Client \rightarrow anOddValueDecorator \rightarrow aBinaryTree
\hspace{2cm} \text{toArray}

\hspace{2cm} \text{toArray}
Benefits & Liabilities

Benefits

Simplifies a class
Distinguishes a class's core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together
Pipes and Filters
Pipes & Filters

ls | grep -i b | wc -l

**Context**
Processing data streams

**Problem**
Building a system that processes or transforms a stream of data

**Forces**
Small processing steps are easier to reuse than large components

Non-adjacent processing steps do not share information

System changes should be possible by exchanging or recombining processing steps, even by users

Final results should be presented or stored in different ways
Solution

Divide task into multiple sequential processing steps or filter components

Output of one filter is the input of the next filter

Filters process data incrementally

Filter does not wait to get all the data before processing
Solution Continued

Data source – input to the system

Data sink – output of the system

Pipes - connect the data source, filters and data sink

Pipe implements the data flow between adjacent processes steps

Processing pipeline – sequence of filters and pipes

Pipeline can process batches of data
Python Interpreter

- Python Program File
  - Lexer
    - (Python Tokenizer)
  - Token Struct
  - Parser
  - Abstract Syntax Tree
  - Compiler
  - Intermediate Byte Code
  - Code Evaluator

http://wiki.cs.uiuc.edu/cs427/Python+-+Batch+Sequential
Intercepting Filter - Problem

Preprocessing and post-processing of a client Web request and response

A Web request often must pass several tests prior to the main processing

- Has the client been authenticated?
- Does the client have a valid session?
- Is the client's IP address from a trusted network?
- Does the request path violate any constraints?
- What encoding does the client use to send the data?
- Do we support the browser type of the client?

Nested if statements lead to fragile code
Intercepting Filter - Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.

Centralization of common logic is desired.

Services should be easy to add or remove unobtrusively without affecting existing components, so that they can be used in a variety of combinations, such as

Logging and authentication

Debugging and transformation of output for a specific client

Uncompressing and converting encoding scheme of input