References

“Null Object”, Woolf, in Pattern Languages of Program Design 3, Edited by Martin, Riehle, Buschmann, Addison-Wesley, 1998, pp. 5-18

Special Case, Martin Fowler, http://martinfowler.com/eaaCatalog/specialCase.html

Patterns of Enterprise Application Architecture, Martin Fowler, Addision Wesley, 2003, pp. 496-498

Special Case

Java 8
Java 8

Target Release Date: March 18, 2014

New Features
   New Time, Date & Calendar classes
   Improvements to Cryptographic classes
   Nashorn JavaScript Engine
   Concurrency Improvements
      Accumulators, Adders
   Default Methods
   Functional language Features
      Lambda Expressions
      Collection Streams (internal iterators)
Lambda Expression

Anonymous Function

\[(\text{Integer } a, \text{Integer } b) \rightarrow a + b\]

Arguments |
---|

Body

\[(\text{Integer start, Integer stop}) \rightarrow \{
    \text{for (int } k = \text{start}; k < \text{stop}; k++)
    \text{System.out.println}(k);
\}\]
Short Version of Lambda Syntax

\[(\text{String } \text{text}) \rightarrow \text{text.length()};\]  
\[(\text{Integer } a, \text{Integer } b) \rightarrow a + b\]
Using Lambdas

Function<String,Integer> length = text -> text.length();
int nameLength = length.apply("Roger Whitney");

BiFunction<Integer,Integer,Integer> adder = (a, b) -> a + b;
int sum = adder.apply(1, 2);
Other Types of Lamdas

`Predicate<Integer> isLarge = value -> value > 100;`  
`if (isLarge.test(59))`  
`System.out.println("large");`  

`Consumer<String> print = text -> System.out.println(text);`  
`print.accept("hello World");`  

`int size = xxx;`  
`Supplier<List> listType = size > 100 ? () -> new ArrayList() : () -> new Vector();`  
`List elements = listType.get();`  
`System.out.println(elements.getClass().getName());`
Lambda Types

New - See java.util.function Interfaces

- Predicate\(<\text{T}>\) -- a boolean-valued property of an object
- Consumer\(<\text{T}>\) -- an action to be performed on an object
- Function\(<\text{T},\text{R}>\) -- a function transforming a \(\text{T}\) to a \(\text{R}\)
- Supplier\(<\text{T}>\) -- provide an instance of a \(\text{T}\) (such as a factory)
- UnaryOperator\(<\text{T}>\) -- a function from \(\text{T}\) to \(\text{T}\)
- BinaryOperator\(<\text{T}>\) -- a function from \((\text{T}, \text{T})\) to \(\text{T}\)

Pre-existing

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.beans.PropertyChangeListener
- etc.
Functional Interfaces

Interface with one method

Can be used to hold a lambda

java.langRunnable
  void run()
Runnable Example

Runnable test = () -> System.out.println("hello from thread");
Thread example = new Thread(test);
example.start();
OnClickListener Example

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View source) {
        makeToast();
    }
});

button.setOnClickListener(() -> makeToast());
String[] rawData = {"cat", "can", "bat", "rat"};
List<String> data = Arrays.asList(rawData);
data.forEach( word -> System.out.println(word) );
Stream

java.util.stream.Stream

Sequence of values

Operations on the values

Operations are chained together into pipelines
Example

String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
    .filter( s -> s.length() > 2)
    .filter( s -> s.charAt(0) == 'a')
    .map( s ->
        s.toUpperCase())
    .collect(Collectors.toList());
System.out.println(longWords);
Lazy Evaluation

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
    .filter(s -> s.length() > 2)
    .filter(s -> s.charAt(0) == 'a')
    .map(s -> s.toUpperCase())
    .collect(Collectors.toList());
System.out.println(longWords);
```

Only One pass of List to do all operations
ing Words

List<String> ingWords = heap.stream()
    .filter( s -> s.endsWith("ing"))
    .collect(Collectors.toList());
For More Information

State of the Lambda: Libraries Edition

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html

http://tinyurl.com/mshjfkj

State of the Lambda

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

http://tinyurl.com/kg5m9zu
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
  └── NullNode

10
  ├── 5
  │    └── Null Node
  └── 20
      └── Null Node

8
  ├── 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();
```

```java
plan = customer.getPlan();
```
public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}

boolean isNull() { return true;}

Compile
class SomeClassThatReturnCustomers {

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

}
Replace all null checks with `isNull()`

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

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

Compile and test

**What is the point of this step?**
Find an operation clients invoke if not null
Add Operation to Null class

```java
if (customer.isNullOrNull())
    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 TailNode {
    public void add(int value) {
        prependNode(value);
    }
}

class Node {
    public void add(int value) {
        if (element > value)
            prependNode(value);
        else
            next.add(value);
    }
}
Principles of OO Design, or Everything I Know About Programming, I Learned from Dilbert

Alan Knight
1. Never do any work that you can get someone else to do for you

Example 1 Total of bills that have been paid this quarter for a factory

```java
total = 0;
Vector billings = aFactory.billings();
for (Bill billing : billings)
  if ((billing.status() == "paid") && (billing.date() > startDate))
    total = total + billing.amount();
```

versus

```java
total = aFactory.totalBillingsPaidSince(startDate).
```
1. Never do any work that you can get someone else to do for you

Excuse me Smithers. I need to know the total bills that have been paid so far this quarter. No, don’t trouble yourself. If you’ll just lend me the key to your filing cabinet I’ll go through the records myself. I’m not that familiar with your filing system, but how complicated can it be? I’ll try not to make too much of a mess.

Verses

SMITHERS! I need the total bills that have been paid since the beginning of the quarter. No, I’m not interested in the petty details of your filing system. I want that total, and I’ll expect it on my desk within the next half millisecond.
1. Never do any work that you can get someone else to do for you

```
somebody.clients().add( new Client());

verses

somebody.addClient( new Client());
```

Less work

somebody just returns collection

Needs

addClient:
removeClient:
more?
Encapsulation & Responsibility

Encapsulation is about responsibility

Who does the work

Who should do the work
2. Avoid Responsibility

If you must accept a responsibility, keep it as vague as possible.

For any responsibility you accept, try to pass the real work off to somebody else.