Del Bit al String

CONTENIDO

Hola!

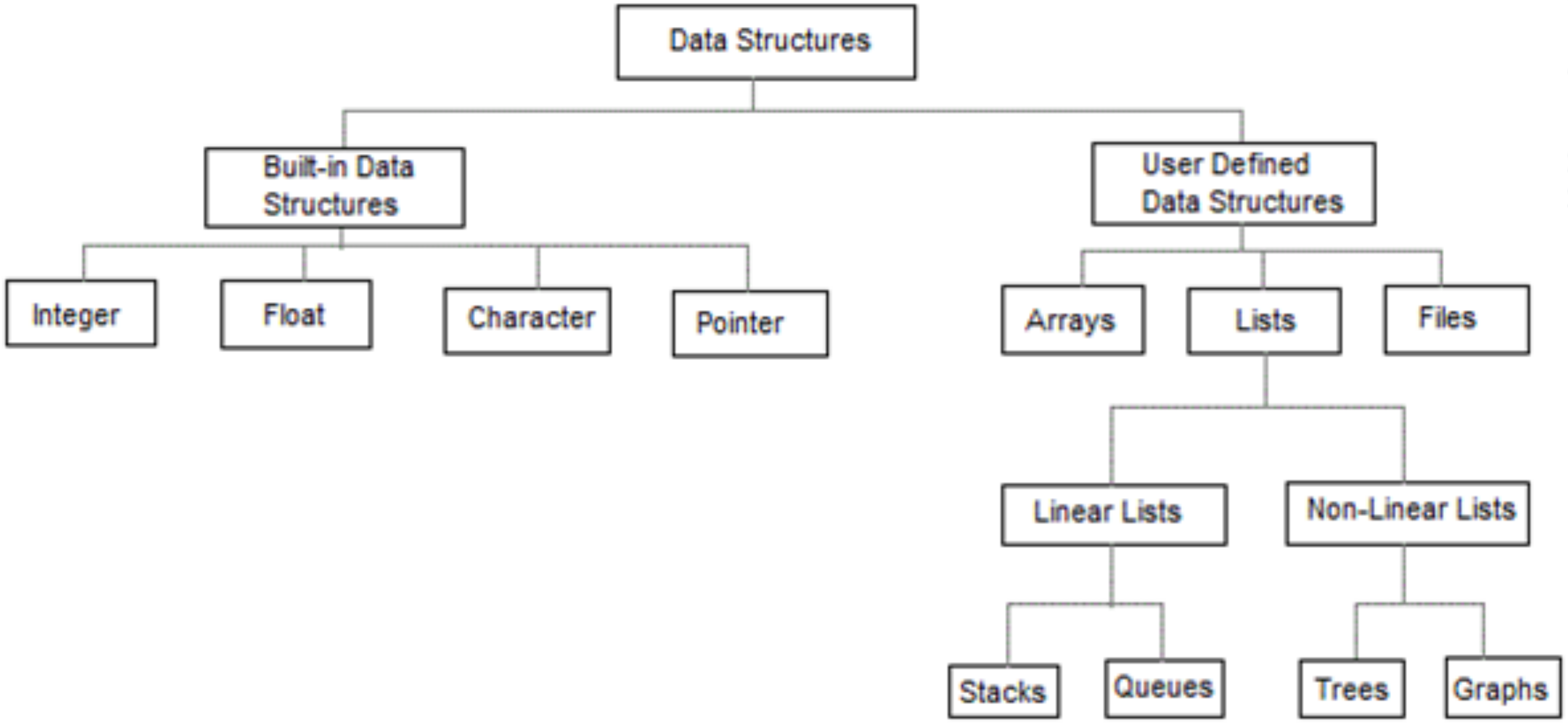
Carlos Muiño

Fullstack Developer at @String-Projects.

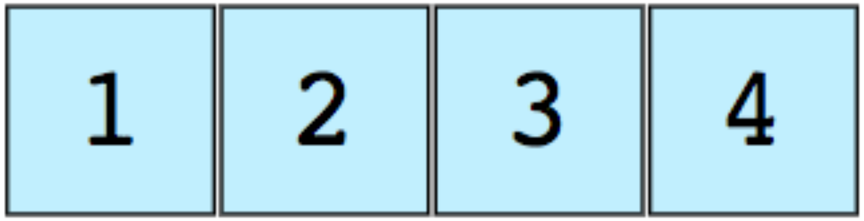
@camumino

Conceptos Básicos del Desarrollo Software

- Estructuras de datos
- Algoritmia
- Refactorizacion

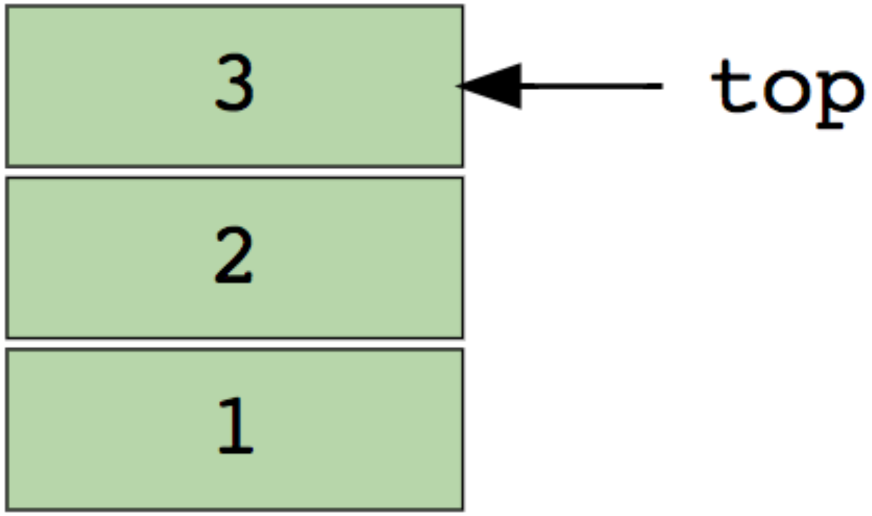

Data Structures
Built-in Data Structures
User Defined Data Structures
Integer
Float
Character
Pointer
Arrays
Lists
Files
Linear Lists
Non-Linear Lists
Stacks
Queues
Trees
Graphs

ARRAYS



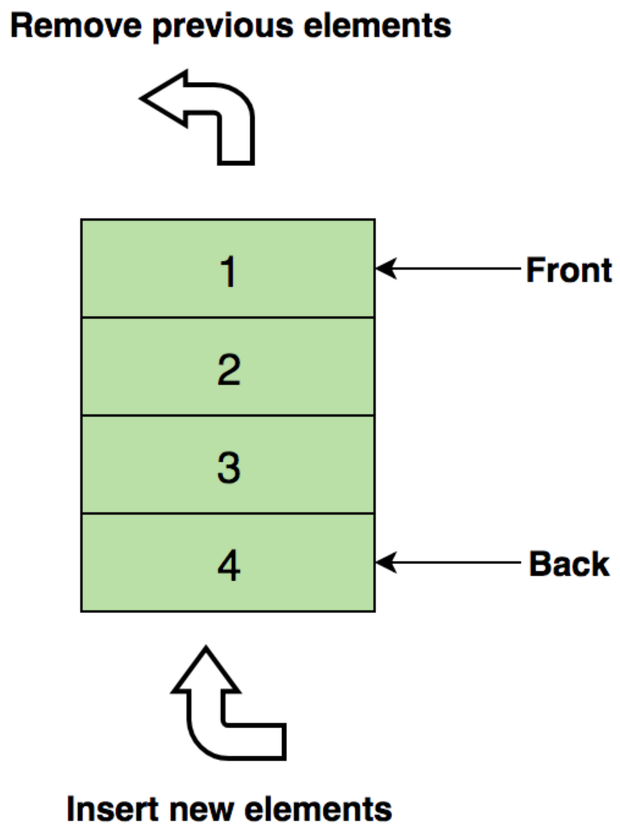
- Index
- Insert
- Get
- Delete
- Size

STACKS



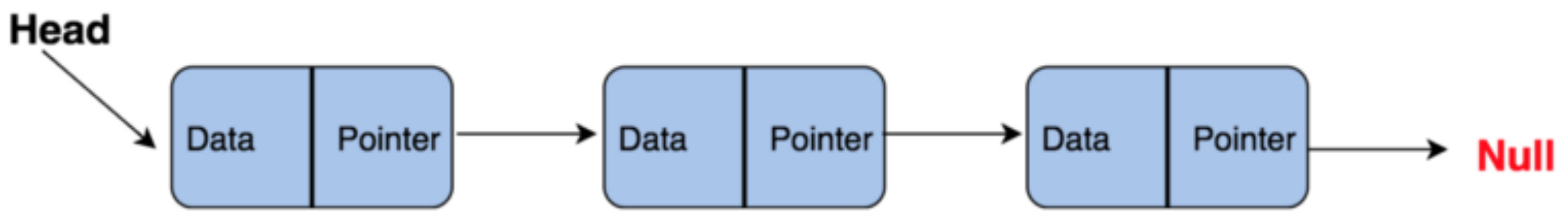
- Push
- Pop
- isEmpty
- Top

QUEUES



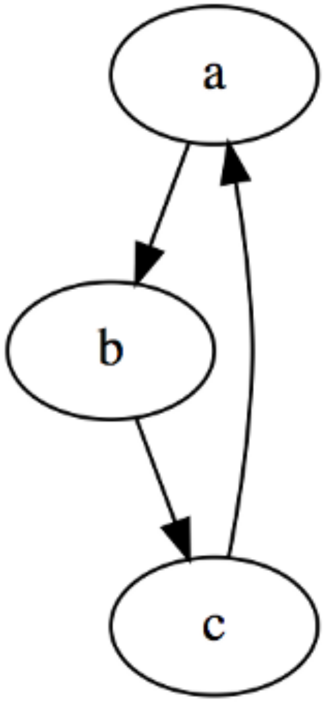
- Enqueue
- Dequeue
- isEmpty
- Top

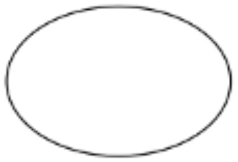
LINKED LIST

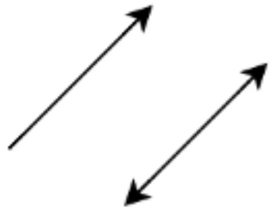


- InsertAtEnd
- InsertAtHead
- Delete
- DeleteAtHead
- Search
- isEmpty

GRAPHS



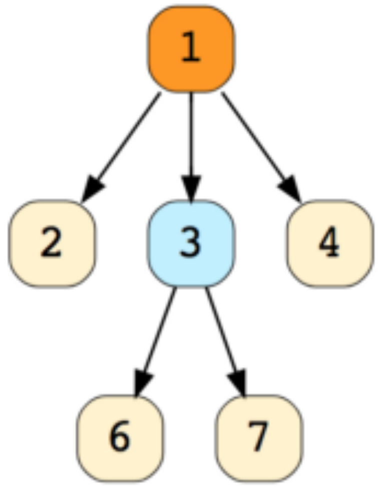




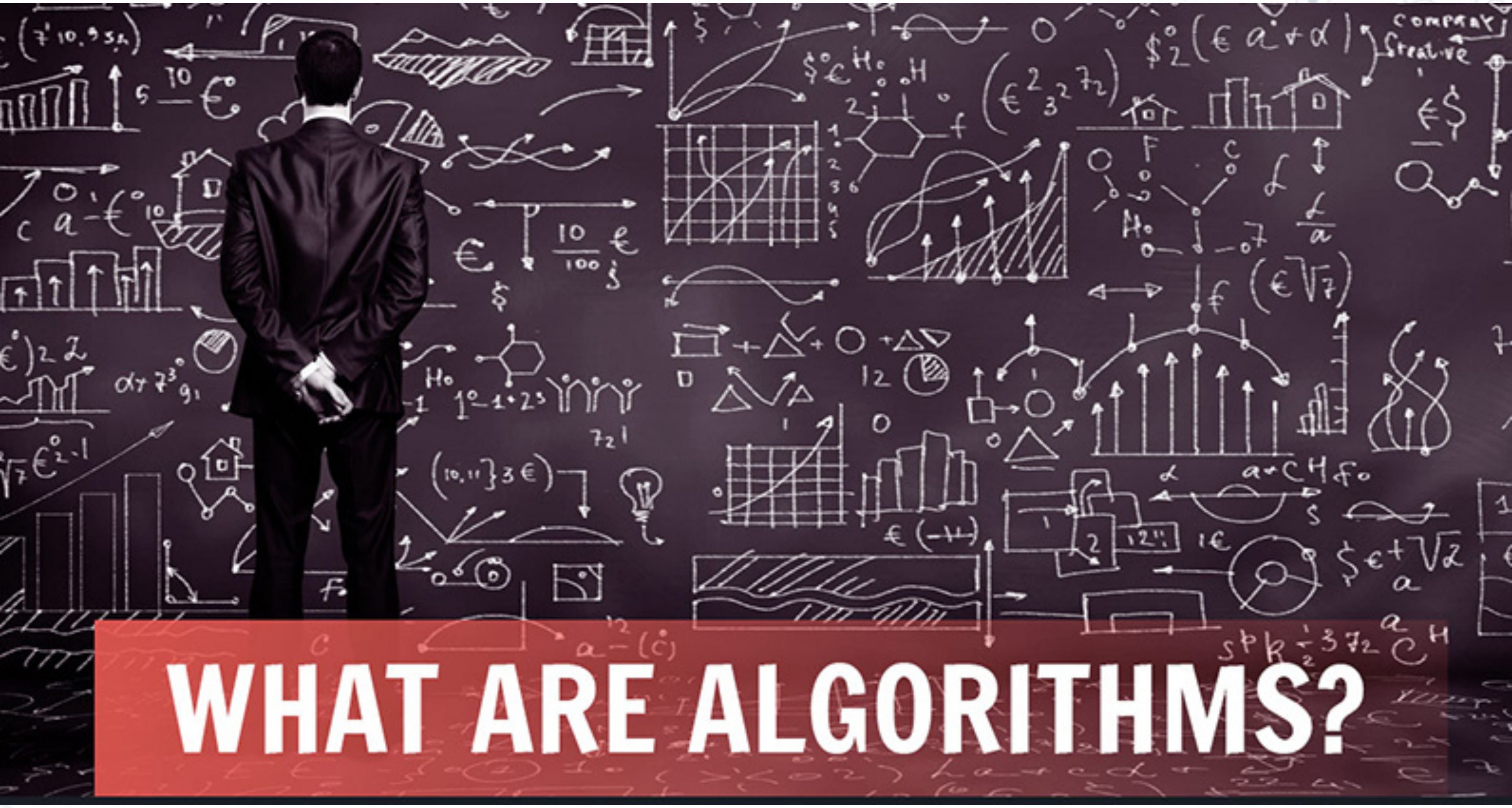
Vertex

Edge

TREES

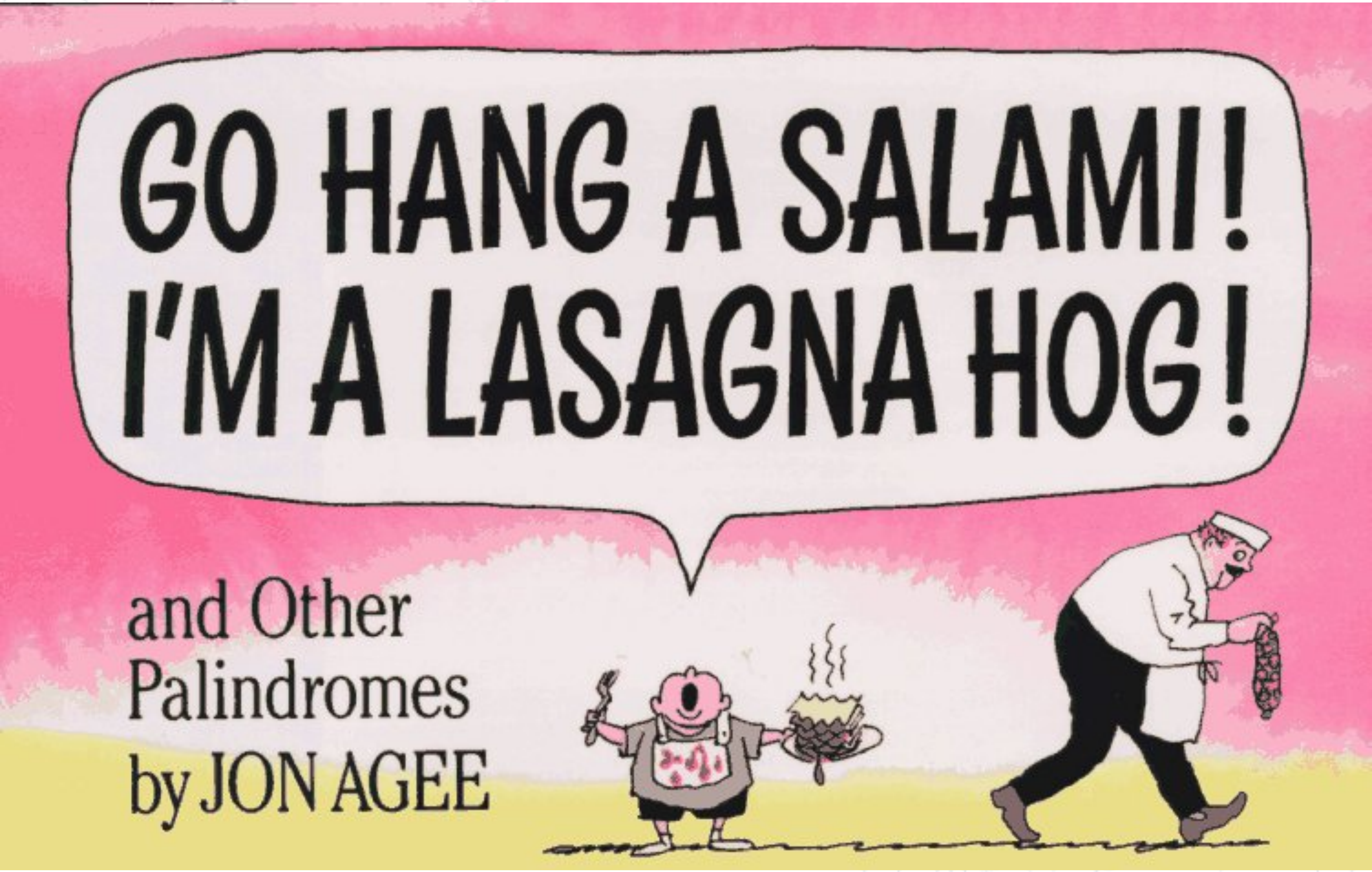


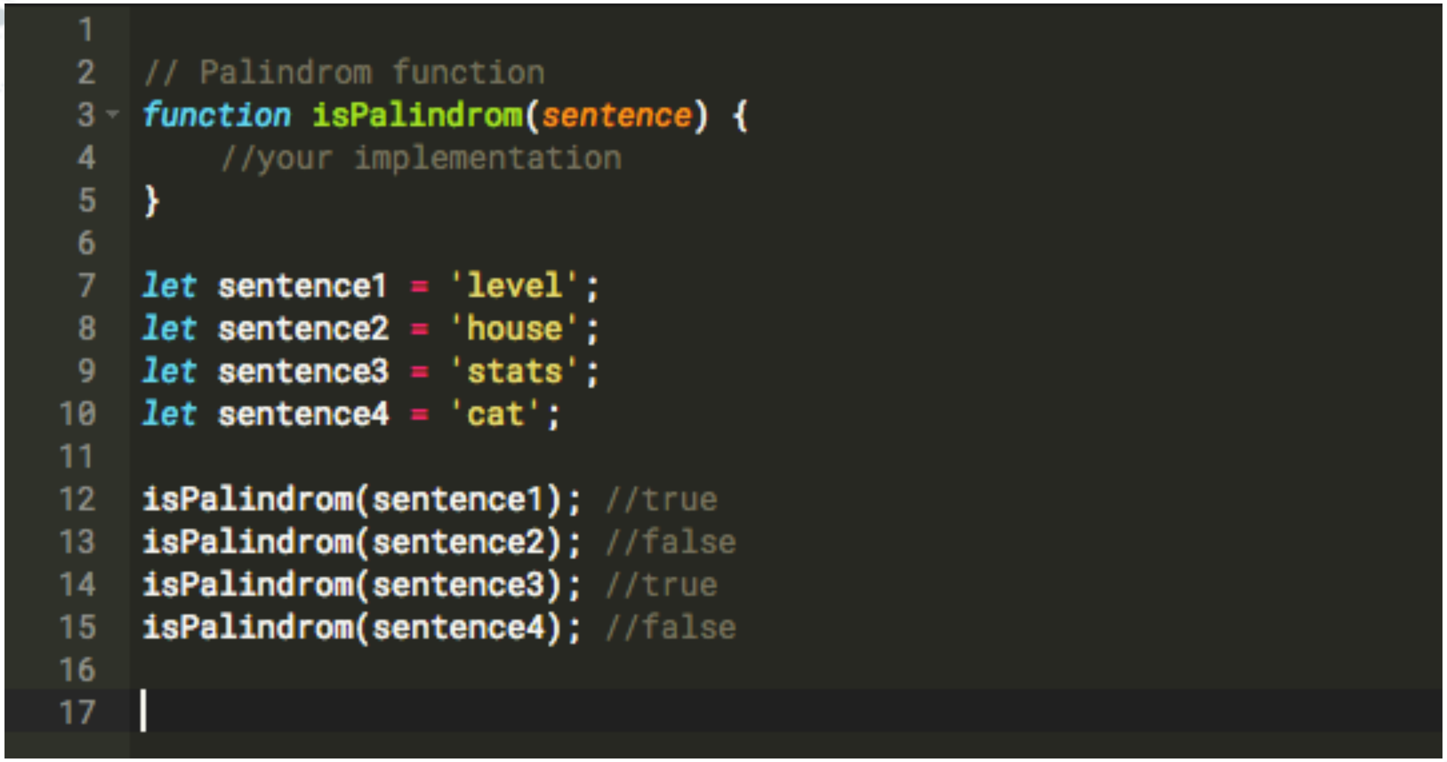
Root	1	
Parent	1	3
Child	2 3 4	6 7
Leaf	2 4 6 7	
Sibling	2 3 4	6 7



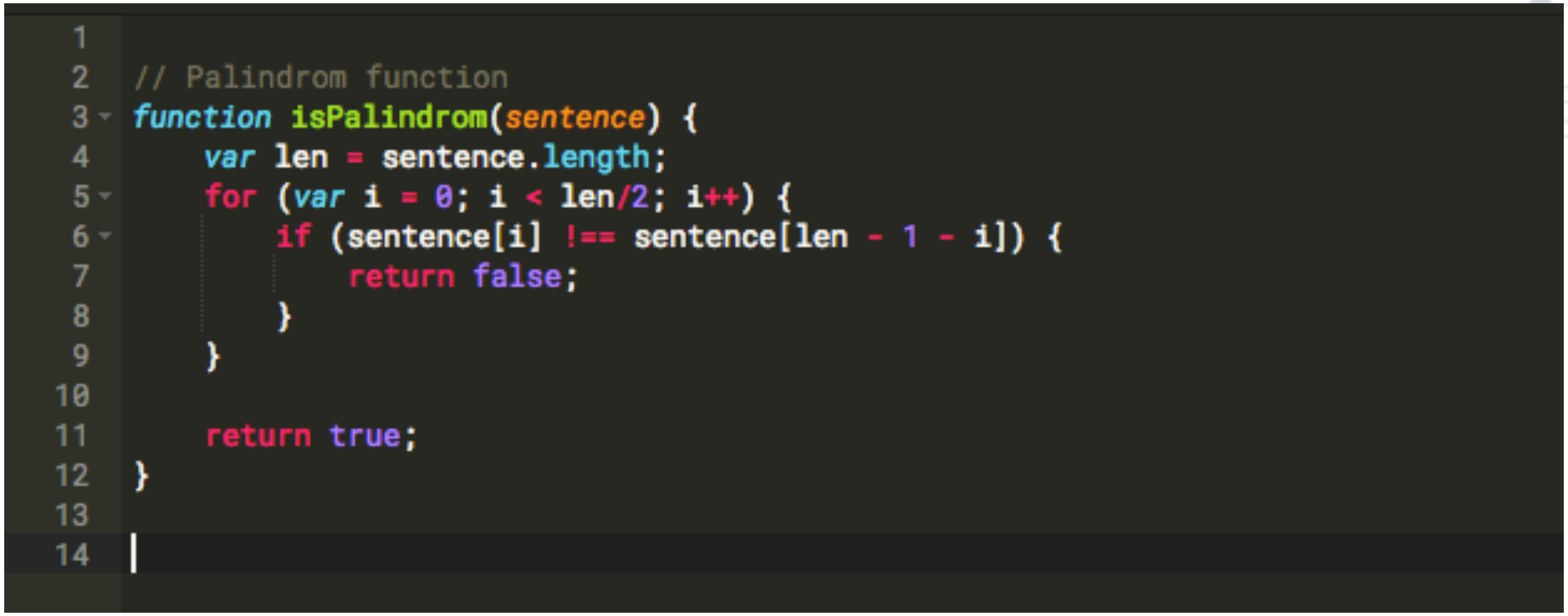
https://www.youtube.com/watch?v=6hfOvs8pY1k

Palindrome

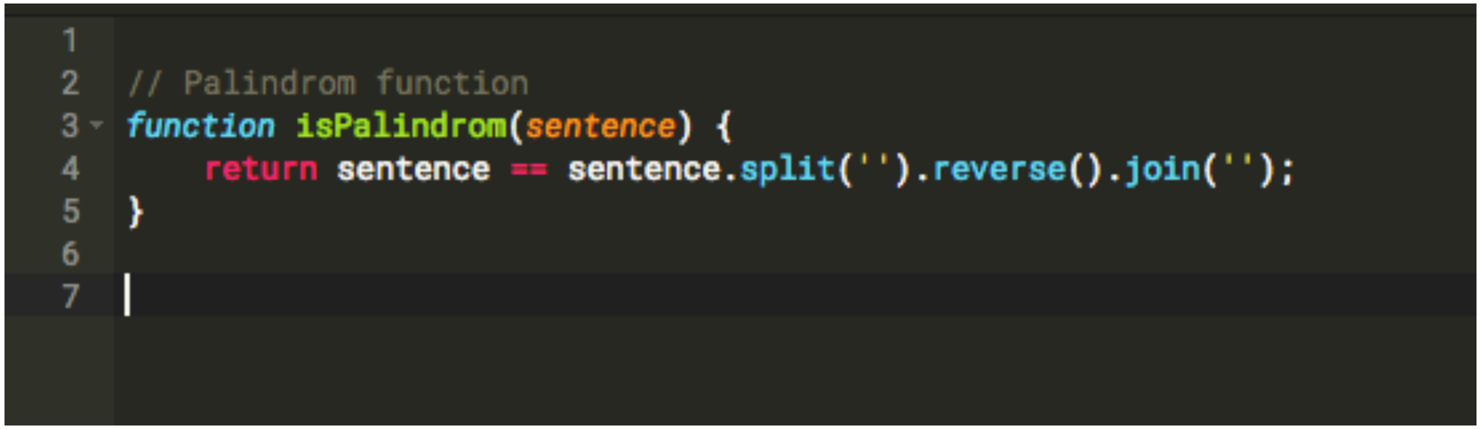

GO HANG A SALAMI! I'M A LASAGNA HOG!
and Other Palindromes
by JON AGEE



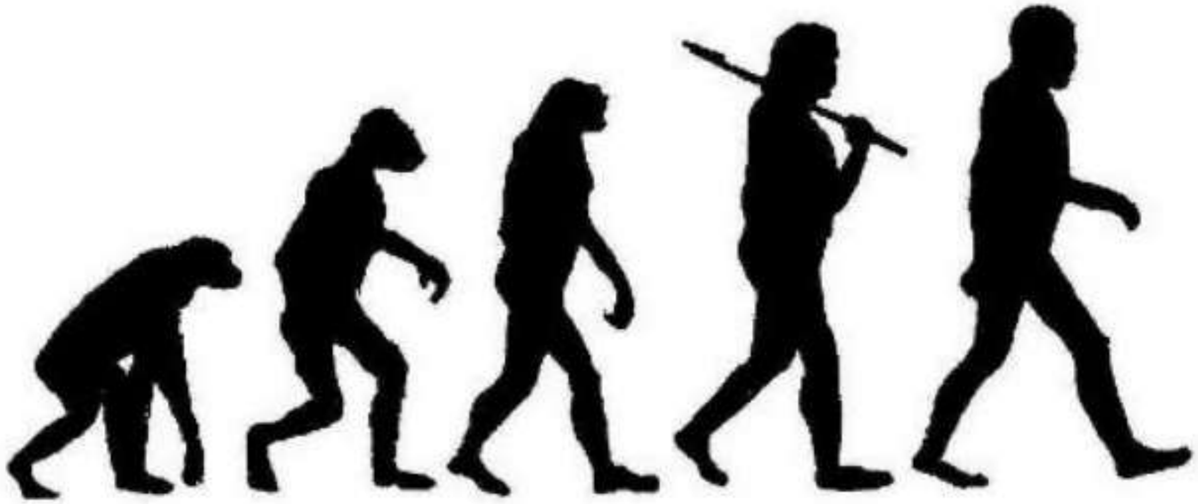
```

// Palindrom function
function isPalindrom(sentence) {
    //your implementation
}

let sentence1 = 'level';
let sentence2 = 'house';
let sentence3 = 'stats';
let sentence4 = 'cat';

isPalindrom(sentence1); //true
isPalindrom(sentence2); //false
isPalindrom(sentence3); //true
isPalindrom(sentence4); //false

```



```

// Palindrom function
function isPalindrom(sentence) {
    var len = sentence.length;
    for (var i = 0; i < len/2; i++) {
        if (sentence[i] !== sentence[len - 1 - i]) {
            return false;
        }
    }

    return true;
}

```



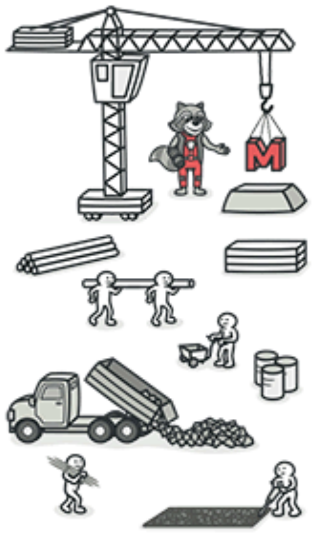
```

// Palindrom function
function isPalindrom(sentence) {
    return sentence == sentence.split('').reverse().join('');
}

```



Refactoring

Improving the Design of Existing Code

Composing Methods



Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

- § **Extract Method**
- § **Inline Method**
- § **Extract Variable**
- § **Inline Temp**
- § **Replace Temp with Query**
- § **Split Temporary Variable**
- § **Remove Assignments to Parameters**
- § **Replace Method with Method Object**
- § **Substitute Algorithm**

Extract Variable

Problem

You have an expression that's hard to understand.

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

Extract Variable

Solution

Place the result of the expression or its parts in separate variables that are self-explanatory.

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

Moving Features between Objects

Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.

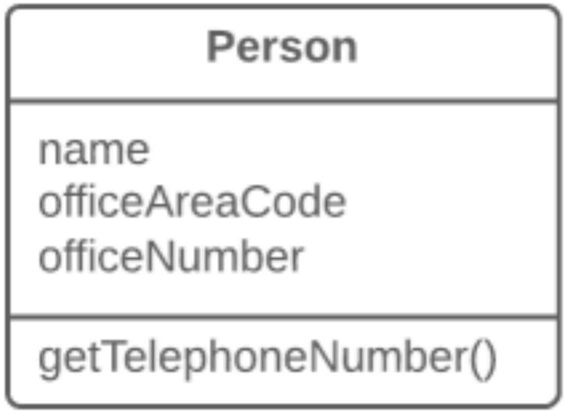
These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- § Move Method
- § Move Field
- § Extract Class
- § Inline Class
- § Hide Delegate
- § Remove Middle Man
- § Introduce Foreign Method
- § Introduce Local Extension

Extract Class

Problem

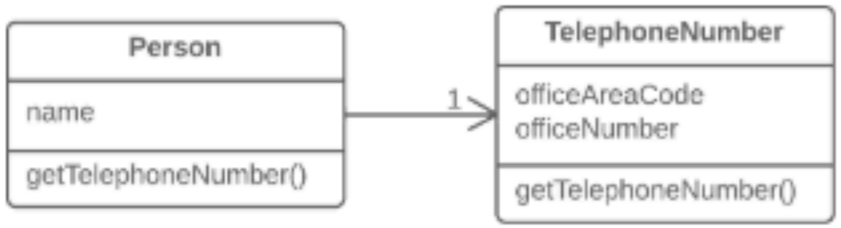
When one class does the work of two,
awkwardness results.



Extract Class

Solution

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



Organizing Data

These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.

- Change Value to Reference
- Change Reference to Value
- Duplicate Observed Data
- Self Encapsulate Field
- Replace Data Value with Object
- Replace Array with Object
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Encapsulate Field
- Encapsulate Collection
- Replace Magic Number with Symbolic Constant
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

Encapsulate Field

Problem

You have a public field.

```
class Person {
  public String name;
}
```

Encapsulate Field

Solution

Make the field private and create access methods for it.

```
class Person {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String arg) {
    name = arg;
  }
}
```

Simplifying Conditional Expressions

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

- § Consolidate Conditional Expression
- § Consolidate Duplicate Conditional Fragments
- § Decompose Conditional
- § Replace Conditional with Polymorphism
- § Remove Control Flag
- § Replace Nested Conditional with Guard Clauses
- § Introduce Null Object
- § Introduce Assertion

Consolidate Conditional Expression

Problem

You have multiple conditionals that lead to the same result or action.

```
double disabilityAmount() {
  if (seniority < 2) {
    return 0;
  }
  if (monthsDisabled > 12) {
    return 0;
  }
  if (isPartTime) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

Consolidate Conditional Expression

Solution

Consolidate all these conditionals in a single expression.

```
double disabilityAmount() {
  if (isNotEligableForDisability()) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

Consolidate Duplicate Conditional Fragments

Problem

Identical code can be found in all branches of a conditional.

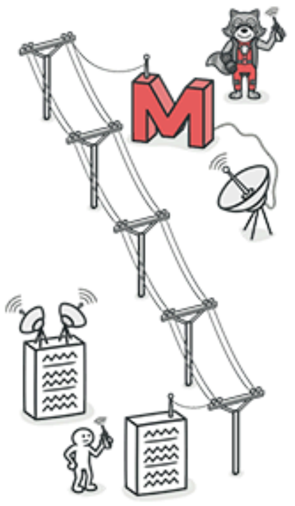
```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

Consolidate Duplicate Conditional Fragments

Solution

Move the code outside of the conditional.

```
if (isSpecialDeal()) {
  total = price * 0.95;
}
else {
  total = price * 0.98;
}
send();
```



Simplifying Method Calls

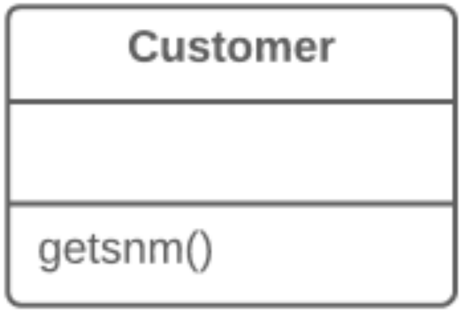
These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

- § Add Parameter
- § Remove Parameter
- § Rename Method
- § Separate Query from Modifier
- § Parameterize Method
- § Introduce Parameter Object
- § Preserve Whole Object
- § Remove Setting Method
- § Replace Parameter with Explicit Methods
- § Replace Parameter with Method Call
- § Hide Method
- § Replace Constructor with Factory Method
- § Replace Error Code with Exception
- § Replace Exception with Test

Rename Method

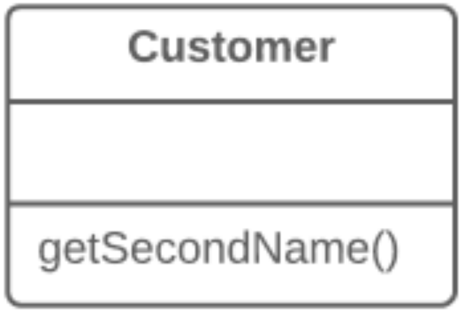
Problem

The name of a method doesn't explain what the method does.



Solution

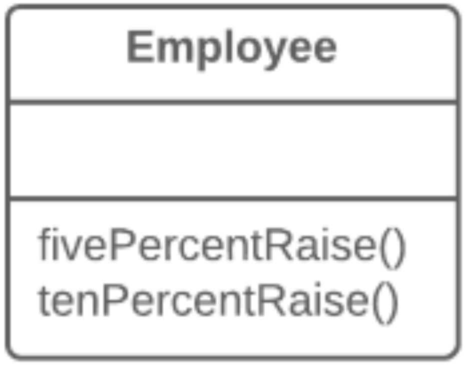
Rename the method.



Parameterize Method

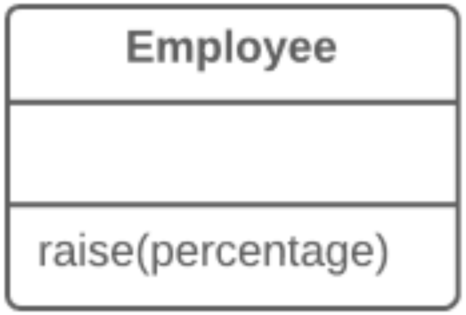
Problem

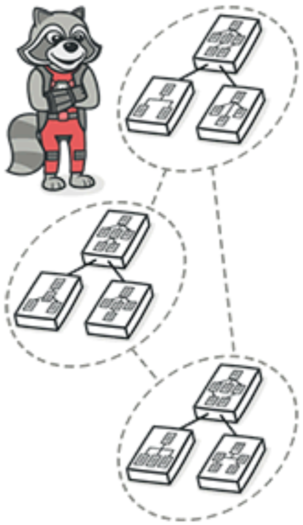
Multiple methods perform similar actions that are different only in their internal values, numbers or operations.



Solution

Combine these methods by using a parameter that will pass the necessary special value.





Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

§ Pull Up Field

§ Pull Up Method

§ Pull Up Constructor Body

§ Push Down Field

§ Push Down Method

§ Extract Subclass

§ Extract Superclass

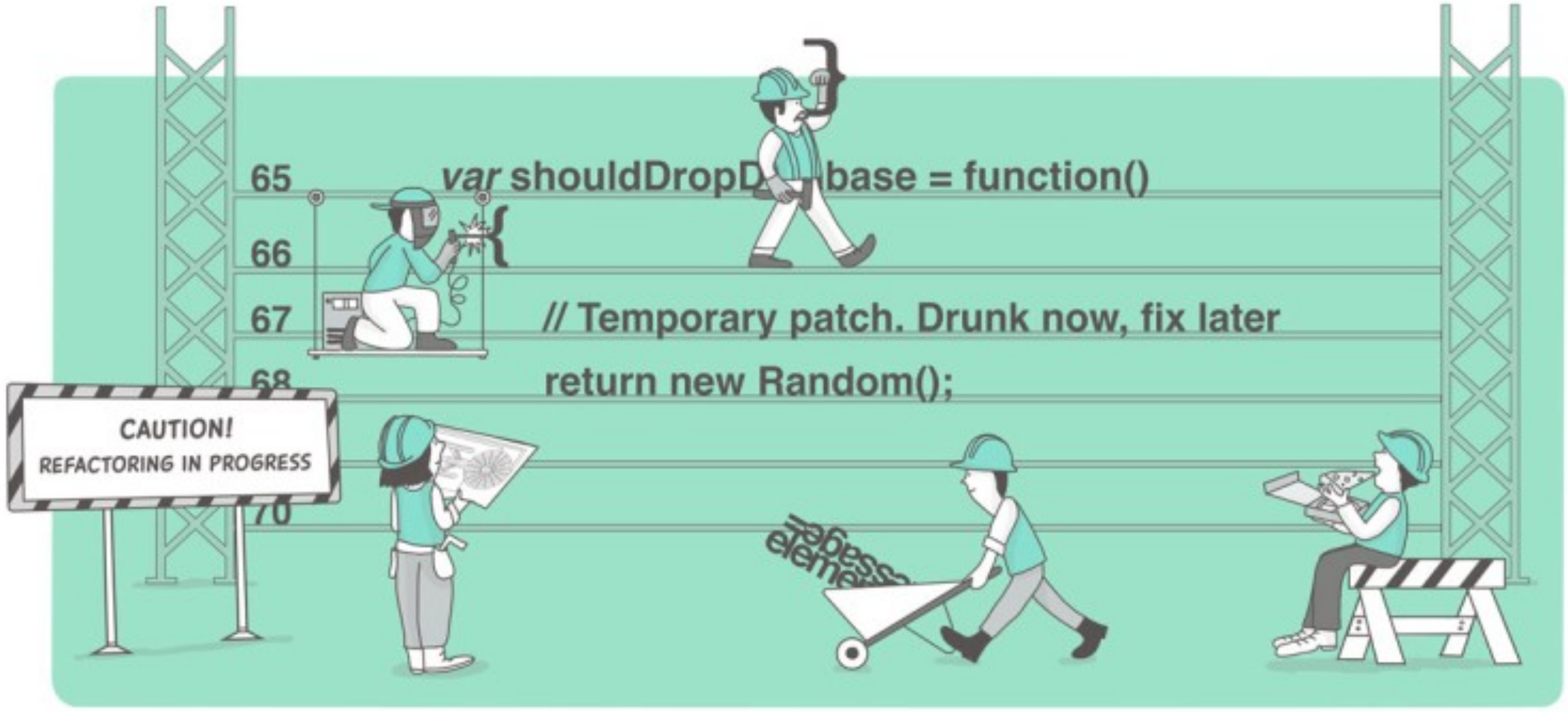
§ Extract Interface

§ Collapse Hierarchy

§ Form Template Method

§ Replace Inheritance with Delegation

§ Replace Delegation with Inheritance


65 var shouldDropDatabase = function()
66 {
67 // Temporary patch. Drunk now, fix later
68 return new Random();
70
CAUTION!
REFACTORING IN PROGRESS

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Gracias!

¿Preguntas?

@camumino

camumino@gmail.com