Change is Bad

The Joy of Immutability

Chris Okasaki

United States Military Academy
(West Point)

[Ctrl-L for full-screen mode]
Bugs, bugs, bugs

Scenario 1: The Rabbit of Caerbannog
Bugs, bugs, bugs

Scenario 1: The Rabbit of Caerbannog

- You pass object to innocent-looking method
Bugs, bugs, bugs

Scenario 1: The Rabbit of Caerbannog

- You pass object to innocent-looking method
- It changes object without your consent
Bugs, bugs, bugs

Scenario 1: The Rabbit of Caerbannog

- You pass object to innocent-looking method
- It changes object without your consent
- You don’t even notice changes until later
Bugs, bugs, bugs

Scenario 2: Dude, Where’s My Car?
Bugs, bugs, bugs

Scenario 2: Dude, Where’s My Car?

- You put an object in a hashtable
Bugs, bugs, bugs

Scenario 2: Dude, Where’s My Car?

- You put an object in a hashtable
- Somebody changes the object in a way that changes the hashkey
Bugs, bugs, bugs

Scenario 2: Dude, Where’s My Car?

- You put an object in a hashtable
- Somebody changes the object in a way that changes the hashkey
- Now you can’t find the object!
Bugs, bugs, bugs

Scenario 3: Dude, Where’s My Car? (With Threads)
Bugs, bugs, bugs

Scenario 3: Dude, Where’s My Car? (With Threads)

- You put an object on a stack
Scenario 3: Dude, Where’s My Car? (With Threads)

- You put an object on a stack
- Some other thread pops the stack
Bugs, bugs, bugs

Scenario 3: Dude, Where’s My Car? (With Threads)

- You put an object on a stack
- Some other thread pops the stack
- You pop the stack, expecting to get back original object
Bugs, bugs, bugs

Change is Bad!
Immutability to the Rescue
Immutability to the Rescue

- An immutable object cannot be changed.
Immutability to the Rescue

- An **immutable** object cannot be changed. Example: a set \(\{1,2,3\}\) contains the elements 1,2,3 forever. They cannot be deleted and new elements cannot be added.
Immutability to the Rescue

- An immutable object cannot be changed. Example: a set \{1,2,3\} contains the elements 1,2,3 forever. They cannot be deleted and new elements cannot be added.

- If want to change object, make new object instead.
Immutability to the Rescue

• An **immutable** object cannot be changed. Example: a set \{1,2,3\} contains the elements 1,2,3 forever. They cannot be deleted and new elements cannot be added.

• If want to change object, make new object instead. Example: when delete 2 from \{1,2,3\}, get new set \{1,3\} but old set still exists.
Immutability to the Rescue

- An **immutable** object cannot be changed.
  Example: a set \{1,2,3\} contains the elements 1,2,3 forever. They cannot be deleted and new elements cannot be added.

- If want to change object, make new object instead.
  Example: when delete 2 from \{1,2,3\}, get new set \{1,3\} but old set still exists.

- Eliminates all previous bugs
Immutability in Interfaces

class Stack {
    // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}

vs

class Stack {
    // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}
Immutability in Interfaces

class Stack { // Mutable
    int size();
    void push(Element x);
    Element pop();
    Element top();
    ...
}

VS

class Stack { // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}
Immutability in Interfaces

```java
class Stack {  // Mutable
    int size();
    void push(Element x);
    Element pop();
    Element top();
    ...
}

vs

class Stack {  // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}
```

Change is Bad – p.9/25
Immutability in Interfaces

class Stack { // Mutable
    int size();
    void push(Element x);
    Element pop();
    Element top();
    ...
}

vs

class Stack { // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}
Immutability in Interfaces

class Stack { // Mutable
    int size();
    void push(Element x);
    Element pop();
    Element top();
    ...
}

vs

class Stack { // Immutable
    int size();
    Stack push(Element x);
    Stack pop();
    Element top();
    ...
}
Undo – Traditional

Commonly implemented using the *Command Pattern*

- Keep a stack of objects (commands)
- Each command knows how to undo itself
- To undo, pop latest command and undo it

Problem: Undoing some commands is tricky
Undo – Immutable

Alternative, use immutable objects

- Keep a stack of objects (state of system)
- To undo, pop previous state

Easy!
Symbol Table – Traditional

Maintain information about identifiers in compiler

```java
void visit(AST ast, MutableEnv env) {
    ...process this node...
    env.add(ast.id);
    visit(ast.left, env);
    visit(ast.right, env);
    env.remove(ast.id);
}
```
Symbol Table – Immutable

Now, make the symbol table immutable

```java
void visit(AST ast, ImmutEnv env) {
    ...process this node...
    ImmutEnv newEnv = env.add(ast.id);
    visit(ast.left, newEnv);
    visit(ast.right, newEnv);
    // discard newEnv
    // no need to remove id!
}
```
Symbol Table – Comparison

env.add(ast.id);
visit(ast.left, env);
visit(ast.right, env);
env.remove(ast.id);

vs

ImmutEnv newEnv = env.add(ast.id);
visit(ast.left, newEnv);
visit(ast.right, newEnv);
Costs vs Benefits

Costs

- Hard to program? Unfamiliar
- Efficiency? Both time and space
- Language issue: multiple return values
- Language issue: garbage collection
Costs vs Benefits

Benefits

- Fewer bugs!
- Less synchronization
- More sharing
- Less copying
  - Less defensive copying
  - Less copying in distributed system
How-To

- Easiest with node-based structures (lists, trees)
- Once node is created, never changed
- To update, copy node and make changes in copy
- Called “path-copying”
Binary Search Trees

\[ t' = \text{insert}(8, t) \]

(Before)  \hspace{1cm}  (After)

```
(Before)  \hspace{1cm}  (After)
```

Change is Bad – p.20/25
Binary Search Trees

class Tree {
    int elem; Tree left, right;
}

Tree insert(int x, Tree t) {
    if (t == null) {
        return new Tree(x, null, null);
    }
    if (x < t.elem) {
        t.left = insert(x, t.left);
    }
    return t;
    ...
    }

Binary Search Trees

class Tree {
    int elem; Tree left, right;
}

Tree insert(int x, Tree t) {
    if (t == null) {
        return new Tree(x, null, null);
    }

    if (x < t.elem)
        return new Tree(t.elem, insert(x, t.left), t.right);

    ...
}

Binary Search Trees

Essential difference:

```java
    t.left = insert(x, t.left);
    return t;
```

vs

```java
    return new Tree(t.elem,
                    insert(x, t.left),
                    t.right);
```
Difficulties

Some objects hard to make immutable
Difficulties

Some objects hard to make immutable

- Random access
  - Arrays
  - Hashtables
Difficulties

Some objects hard to make immutable

- Random access
  - Arrays
  - Hashtables
- Cycles
  - Doubly-linked lists
Difficulties

Some objects hard to make immutable

- Random access
  - Arrays
  - Hashtables
- Cycles
  - Doubly-linked lists
- Multiple entrypoints
  - Union-find
Conclusion

Today

- Mutable is default
- Immutable in (rare) special occasions
Conclusion

Today

• Mutable is default
• Immutable in (rare) special occasions

Tomorrow?

• Immutable is default
• Mutable only when good reason