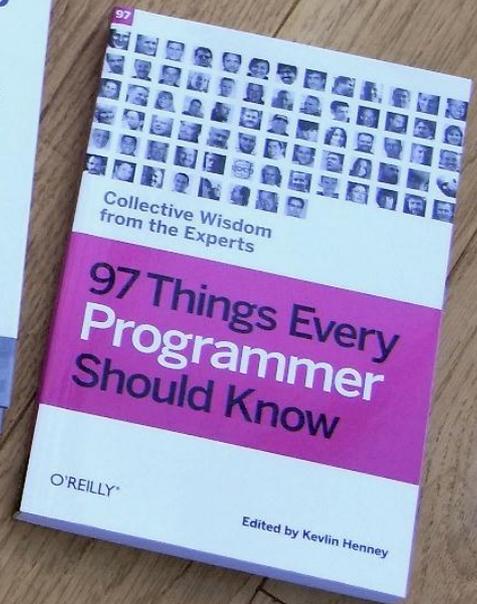
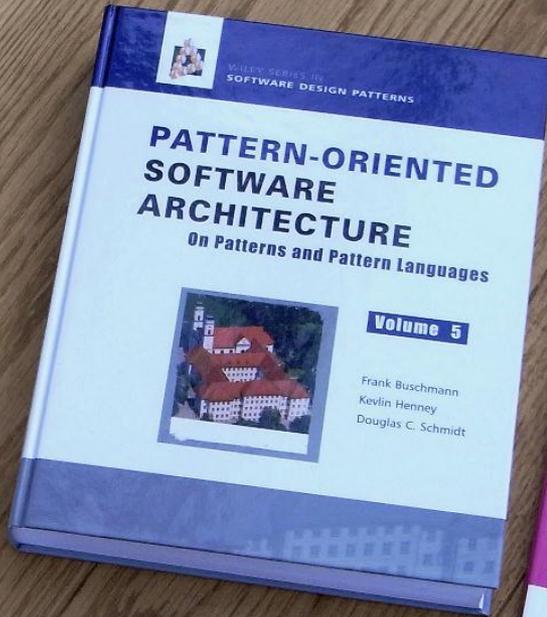
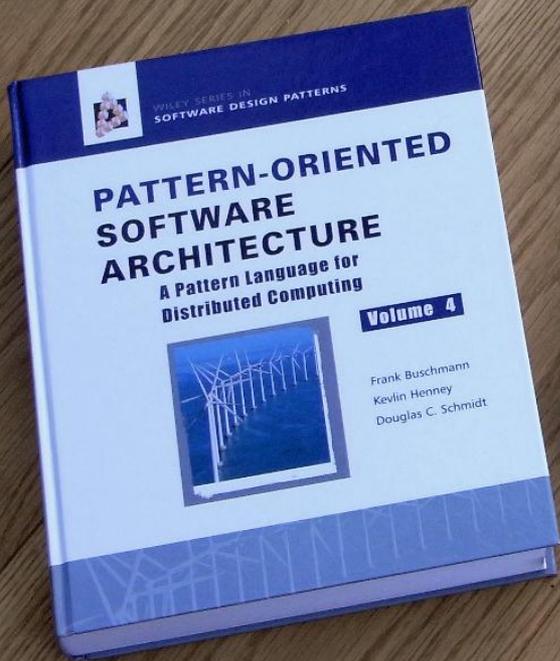


It Is Possible to Do  
Object-Oriented  
Programming in Java

Kevlin Henney  
*kevlin@curbralan.com*  
*@KevlinHenney*



The Java programming language platform provides a *portable, interpreted, high-performance, simple, object-oriented* programming language and supporting run-time environment.

<http://java.sun.com/docs/white/langenv/Intro.doc.html#318>

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

Alan Kay

Ignorance

Apathy

Selfishness

Encapsulation

Inheritance

Polymorphism

Encapsulation

Polymorphism

Inheritance

**Encapsulation**

Polymorphism

Inheritance

*encapsulate* enclose (something) in or as if in a capsule.

- *express the essential feature of (someone or something) succinctly.*
- *enclose (a message or signal) in a set of codes which allow use by or transfer through different computer systems or networks.*
- *provide an interface for (a piece of software or hardware) to allow or simplify access for the user.*

**The New Oxford Dictionary of English**



**A distinction between inheritance and subtyping is not often made: classes are often equated directly with types. From a behavioural point of view a type defines characteristics and a class defines an implementation of these characteristics.**

**Kevlin Henney**

*Distributed Object-Oriented Computing:  
The Development and Implementation of an Abstract Machine*



In many object-oriented programming languages the concept of *inheritance* is present, which provides a mechanism for sharing code among several classes of objects. Many people even regard inheritance as the hallmark of object-orientedness in programming languages. We do not agree with this view, and argue that the essence of object-oriented programming is the encapsulation of data and operations in objects and the protection of individual objects against each other. [...]

The author considers this principle of protection of objects against each other as the basic and essential characteristic of object-oriented programming. It is a refinement of the technique of abstract data types, because it does not only protect one type of objects against all other types, but one object against all other ones. As a programmer we can consider ourselves at any moment to be sitting in exactly one object and looking at all the other objects from outside.

*Pierre America*

"A Behavioural Approach to Subtyping in Object-Oriented Programming Languages"

**Object-oriented programming does not have an exclusive claim to all these good properties. Systems may be modeled by other paradigms [...]. Resilience can be achieved just as well by organizing programs around abstract data types, independently of taxonomies; in fact, data abstraction alone is sometimes taken as the essence of object orientation.**

**Martín Abadi and Luca Cardelli**  
*A Theory of Objects*

## **abstraction, n. (*Logic*)**

- the process of formulating a generalized concept of a common property by disregarding the differences between a number of particular instances. On such an account, we acquired the concept of *red* by recognizing it as common to, and so abstracting it from the other properties of, those individual objects we were originally taught to call red.
- an operator that forms a class name or predicate from any given expression.

E J Borowski and J M Borwein  
**Dictionary of Mathematics**

$\forall T \bullet \exists \text{RecentlyUsedList} \bullet$   
{  
  new : RecentlyUsedList[T],  
  isEmpty : RecentlyUsedList[T]  $\rightarrow$  Boolean,  
  size : RecentlyUsedList[T]  $\rightarrow$  Integer,  
  add : RecentlyUsedList[T]  $\times$  Integer  $\rightarrow$  RecentlyUsedList[T],  
  get : RecentlyUsedList[T]  $\times$  Integer  $\rightarrow$  T,  
  equals : RecentlyUsedList[T]  $\times$  RecentlyUsedList[T]  $\rightarrow$  Boolean  
}

```
class RecentlyUsedList
{
    private ...
    public boolean isEmpty() ...
    public int size() ...
    public void add(String toAdd) ...
    public String get(int index) ...
    public boolean equals(RecentlyUsedList other) ...
}
```

```
class RecentlyUsedList
{
    private ...
    public boolean isEmpty() ...
    public int size() ...
    public void add(String toAdd) ...
    public String get(int index) ...
    public boolean equals(RecentlyUsedList other) ...
    public boolean equals(Object other) ...
}
```

```

class RecentlyUsedList
{
    private List<String> items = new ArrayList<String>();
    public boolean isEmpty()
    {
        return items.isEmpty();
    }
    public int size()
    {
        return items.size();
    }
    public void add(String toAdd)
    {
        items.remove(toAdd);
        items.add(toAdd);
    }
    public String get(int index)
    {
        return items.get(size() - index - 1);
    }
    public boolean equals(RecentlyUsedList other)
    {
        return other != null && items.equals(other.items);
    }
    public boolean equals(Object other)
    {
        return
            other instanceof RecentlyUsedList &&
            equals((RecentlyUsedList) other);
    }
}

```

```
typedef struct RecentlyUsedList RecentlyUsedList;
```

```
RecentlyUsedList * create();
```

```
void destroy(RecentlyUsedList *);
```

```
bool isEmpty(const RecentlyUsedList *);
```

```
int size(const RecentlyUsedList *);
```

```
void add(RecentlyUsedList *, int toAdd);
```

```
int get(const RecentlyUsedList *, int index);
```

```
bool equals(const RecentlyUsedList *, const RecentlyUsedList *);
```

```
struct RecentlyUsedList
{
    int * items;
    int length;
};
```

```

RecentlyUsedList * create()
{
    RecentlyUsedList * result = (RecentlyUsedList *) malloc(sizeof(RecentlyUsedList));
    result->items = 0;
    result->length = 0;
    return result;
}

void destroy(RecentlyUsedList * self)
{
    free(self->items);
    free(self);
}

bool isEmpty(const RecentlyUsedList * self)
{
    return self->length == 0;
}

int size(const RecentlyUsedList * self)
{
    return self->length;
}

static int indexOf(const RecentlyUsedList * self, int toFind)
{
    int result = -1;
    for(int index = 0; result == -1 && index != self->length; ++index)
        if(self->items[index] == toFind)
            result = index;
    return result;
}

static void removeAt(RecentlyUsedList * self, int index)
{
    memmove(&self->items[index], &self->items[index + 1], (self->length - index - 1) * sizeof(int));
    --self->length;
}

void add(RecentlyUsedList * self, int toAdd)
{
    int found = indexOf(self, toAdd);
    if(found != -1)
        removeAt(self, found);
    self->items = (int *) realloc(self->items, (self->length + 1) * sizeof(int));
    self->items[self->length] = toAdd;
    ++self->length;
}

int get(const RecentlyUsedList * self, int index)
{
    return self->items[self->length - index - 1];
}

bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
{
    return lhs->length == rhs->length && memcmp(lhs->items, rhs->items, lhs->length * sizeof(int)) == 0;
}

```

```
struct RecentlyUsedList
{
    std::vector<int> items;
};
```

```
extern "C"
{
    RecentlyUsedList * create()
    {
        return new RecentlyUsedList;
    }

    void destroy(RecentlyUsedList * self)
    {
        delete self;
    }

    bool isEmpty(const RecentlyUsedList * self)
    {
        return self->items.empty();
    }

    int size(const RecentlyUsedList * self)
    {
        return self->items.size();
    }

    void add(RecentlyUsedList * self, int toAdd)
    {
        std::vector<int>::iterator found =
            std::find(self->items.begin(), self->items.end(), toAdd);
        if(found != self->items.end())
            self->items.erase(found);
        self->items.push_back(toAdd);
    }

    int get(const RecentlyUsedList * self, int index)
    {
        return self->items[self->items.size() - index - 1];
    }

    bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
    {
        return lhs->items == rhs->items;
    }
}
```

OO  $\equiv$  ADT?

OO ≠ ADT

# Autognosis

An object can only  
access other  
objects through  
public interfaces

```
class RecentlyUsedList
{
    ...
    public boolean equals(RecentlyUsedList other)
    {
        return other != null && items.equals(other.items);
    }
    public boolean equals(Object other)
    {
        return
            other instanceof RecentlyUsedList &&
            equals((RecentlyUsedList) other);
    }
}
```

```
bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
{
    return
        lhs->length == rhs->length &&
        memcmp(lhs->items, rhs->items, lhs->length * sizeof(int)) == 0;
}
```

```
extern "C"
{
    ...
    bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
    {
        return lhs->items == rhs->items;
    }
}
```

**Reflexivity:** I am me.

**equals**

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references.

- It is *reflexive*: for any non-null reference value *x*, *x.equals(x)* should return true.
- It is *symmetric*: for any non-null reference values *x* and *y*, *x.equals(y)* should return true if and only if *y.equals(x)* returns true.
- It is *transitive*: for any non-null reference values *x*, *y*, and *z*, if *x.equals(y)* returns true and *y.equals(z)* returns true, then *x.equals(z)* should return true.
- It is *consistent*: for any non-null reference values *x* and *y*, multiple invocations of *x.equals(y)* consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value *x*, *x.equals(null)* should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values *x* and *y*, this method returns true if and only if the objects *x* and *y* have the same value (i.e., *x == y* has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

**Symmetry:** If you're the same as me, I'm the same as you.

**Transitivity:** If I'm the same as you, and you're the same as them, then I'm the same as them too.

**Consistency:** If there's no change, everything's the same as it ever was.

**Null inequality:** I am not nothing.

**No throw:** If you call, I won't hang up.

**Hash equality:** If we're the same, we both share the same magic numbers.

Here are four common pitfalls that can cause inconsistent behavior when overriding *equals*:

1. Defining *equals* with the wrong signature.
2. Changing *equals* without also changing *hashCode*.
3. Defining *equals* in terms of mutable fields.
4. Failing to define *equals* as an equivalence relation.

Martin Odersky, Lex Spoon and Bill Venners  
"How to Write an Equality Method in Java"  
<http://www.artima.com/lejava/articles/equality.html>

Here are four common pitfalls that can cause inconsistent behavior when overriding `equals`:

1. Defining `equals` with the wrong signature.
2. Changing `equals` without also changing `hashCode`.
3. Relying on `equals` and `hashCode` to be invariant when they depend on mutable fields.
4. Failing to define `equals` as an equivalence relation.

Here are four common pitfalls that can cause inconsistent behavior when overriding `equals`:

1. Defining `equals` with the wrong signature.
2. Changing `equals` without also changing `hashCode`.
3. Failing to define `equals` as an equivalence relation.
4. Relying on `equals` and `hashCode` to be invariant when they depend on mutable fields.

```
bool equals(  
    const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)  
{  
    bool result = size(lhs) == size(rhs);  
    for(int index = 0; result && index != size(lhs); ++index)  
        result = get(lhs, index) == get(rhs, index);  
    return result;  
}
```

```
extern "C" bool equals(  
    const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)  
{  
    bool result = size(lhs) == size(rhs);  
    for(int index = 0; result && index != size(lhs); ++index)  
        result = get(lhs, index) == get(rhs, index);  
    return result;  
}
```

```
class RecentlyUsedList
{
    ...
    public boolean equals(RecentlyUsedList other)
    {
        boolean result = other != null && size() == other.size();
        for(int index = 0; result && index != size(); ++index)
            result = get(index).equals(other.get(index));
        return result;
    }
    public boolean equals(Object other)
    {
        return
            other instanceof RecentlyUsedList &&
            equals((RecentlyUsedList) other);
    }
}
```

One of the most pure object-oriented programming models yet defined is the Component Object Model (COM). It enforces all of these principles rigorously. Programming in COM is very flexible and powerful as a result. There is no built-in notion of equality. There is no way to determine if an object is an instance of a given class.

William Cook

"On Understanding Data Abstraction, Revisited"

```
class RecentlyUsedList
{
    ...
    public boolean equals(RecentlyUsedList other)
    {
        boolean result = other != null && size() == other.size();
        for(int index = 0; result && index != size(); ++index)
            result = get(index).equals(other.get(index));
        return result;
    }
    public boolean equals(Object other)
    {
        return
            other instanceof RecentlyUsedList &&
            equals((RecentlyUsedList) other);
    }
}
```

```
class RecentlyUsedList
{
    ...
    public boolean equals(RecentlyUsedList other)
    {
        boolean result = other != null && size() == other.size();
        for(int index = 0; result && index != size(); ++index)
            result = get(index).equals(other.get(index));
        return result;
    }
}
```

In a purist view of object-oriented methodology, dynamic dispatch is the only mechanism for taking advantage of attributes that have been forgotten by subsumption. This position is often taken on abstraction grounds: no knowledge should be obtainable about objects except by invoking their methods. In the purist approach, subsumption provides a simple and effective mechanism for hiding private attributes.

*Martín Abadi and Luca Cardelli, A Theory of Objects*

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $S$  is a subtype of  $T$ .

*Barbara Liskov*

"Data Abstraction and Hierarchy"

It is possible to  
do Object-Oriented  
programming in  
Java

Object-Oriented  
subset of Java:  
class name is  
only after “new”

```
interface RecentlyUsedList
{
    boolean isEmpty();
    int size();
    void add(String toAdd);
    String get(int index);
    boolean equals(RecentlyUsedList other);
}
```

```
class RecentlyUsedListImpl
    implements RecentlyUsedList
{
    private List<String> items = ...;
    public boolean isEmpty() ...
    public int size() ...
    public void add(String toAdd) ...
    public String get(int index) ...
    public boolean equals(RecentlyUsedList other) ...
    public boolean equals(Object other) ...
}
```

```
class ArrayListBasedRecentlyUsedList
    implements RecentlyUsedList
{
    private List<String> items = ...;
    public boolean isEmpty() ...
    public int size() ...
    public void add(String toAdd) ...
    public String get(int index) ...
    public boolean equals(RecentlyUsedList other) ...
    public boolean equals(Object other) ...
}
```

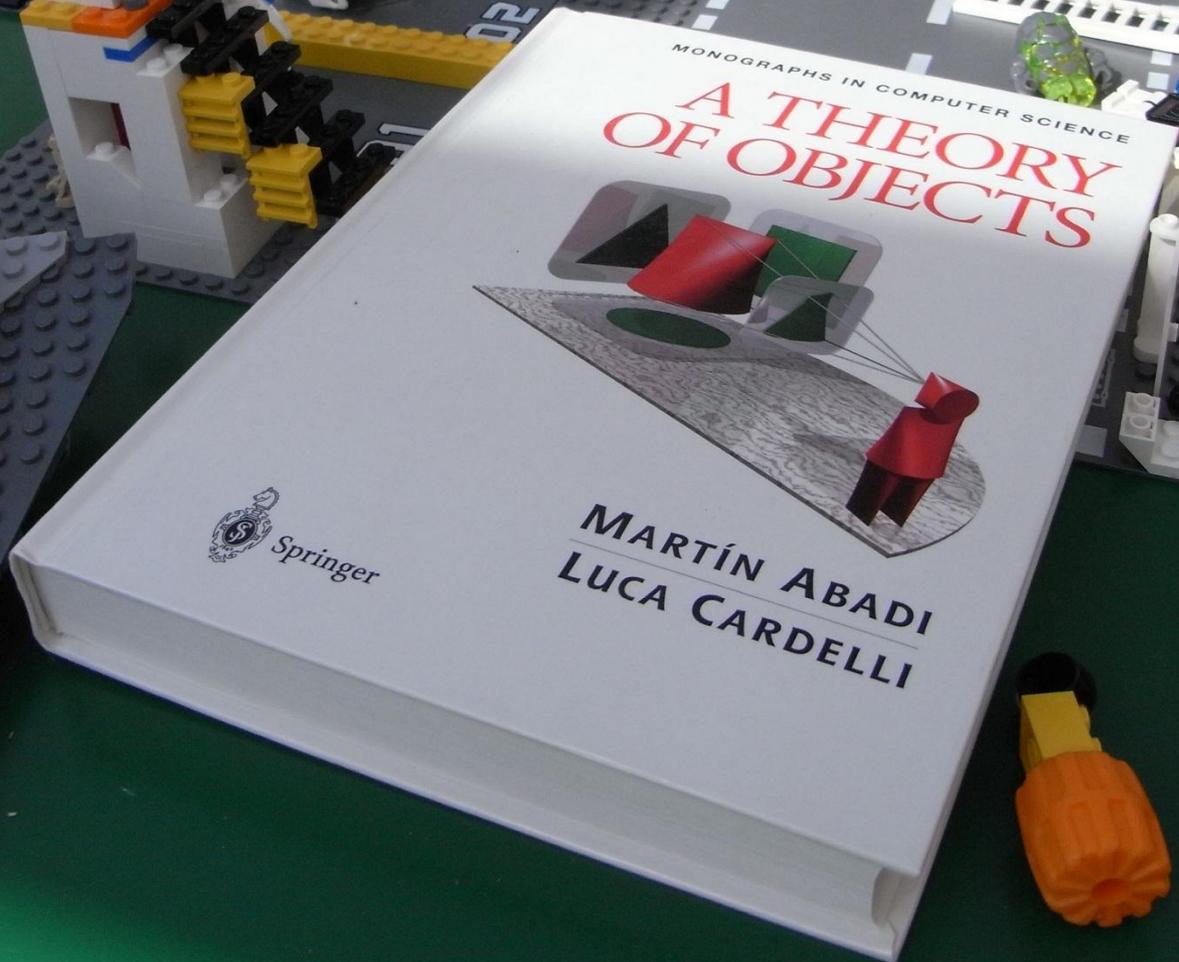
```
class RandomAccessRecentlyUsedList
    implements RecentlyUsedList
{
    private List<String> items = ...;
    public boolean isEmpty() ...
    public int size() ...
    public void add(String toAdd) ...
    public String get(int index) ...
    public boolean equals(RecentlyUsedList other) ...
    public boolean equals(Object other) ...
}
```

```
RecentlyUsedList list =  
    new RandomAccessRecentlyUsedList();
```

```
class RandomAccessRecentlyUsedList implements RecentlyUsedList
{
    ...
    public boolean equals(RecentlyUsedList other)
    {
        boolean result = other != null && size() == other.size();
        for(int index = 0; result && index != size(); ++index)
            result = get(index).equals(other.get(index));
        return result;
    }
    public boolean equals(Object other)
    {
        return
            other instanceof RecentlyUsedList &&
            equals((RecentlyUsedList) other);
    }
}
```

Here are five common pitfalls that can cause inconsistent behavior when overriding `equals`:

1. Defining `equals` with the wrong signature.
2. Changing `equals` without also changing `hashCode`.
3. Failing to define `equals` as an equivalence relation.
4. Defining `equals` in a class hierarchy where types and classes are not properly distinguished.
5. Relying on `equals` and `hashCode` to be invariant when they depend on mutable fields.



MONOGRAPHS IN COMPUTER SCIENCE

# A THEORY OF OBJECTS



 Springer

MARTÍN ABADI  
LUCA CARDELLI

Lambda-calculus  
was the first  
object-oriented  
language (1941)

```
newRecentlyUsedList =  
  λ • (let items = ref(⟨⟩) •  
    {  
      isEmpty = λ • #items = 0,  
      size = λ • #items,  
      add = λ x •  
        items := ⟨x⟩^⟨itemsy | y ∈ 0...#items ∧ itemsy ≠ x⟩,  
      get = λ i • itemsi  
    })
```

```
var newRecentlyUsedList = function() {  
  var items = []  
  return {  
    isEmpty: function() {  
      return items.length === 0  
    },  
    size: function() {  
      return items.length  
    },  
    add: function(newItem) {  
      (items = items.filter(function(item) {  
        return item !== newItem  
      })).unshift(newItem)  
    },  
    get: function(index) {  
      return items[index]  
    }  
  }  
}  
}
```

One of the most powerful mechanisms for program structuring [...] is the block and procedure concept. [...]

A procedure which is capable of giving rise to block instances which survive its call will be known as a *class*; and the instances will be known as *objects* of that class. [...]

A call of a class generates a new object of that class.

Ole-Johan Dahl and C A R Hoare  
"Hierarchical Program Structures" in *Structured Programming*

```
var newEmptyRecentlyUsedList = function() {
  return {
    isEmpty: function() {
      return true
    },
    size: function() {
      return 0
    },
    add: function(newItem) {
      var inserted = new InsertedRecentlyUsedList(newItem)
      this.isEmpty = inserted.isEmpty
      this.size = inserted.size
      this.add = inserted.add
      this.get = inserted.get
    },
    get: function(index) {
    }
  }
}
```

```
var newInsertedRecentlyUsedList = function(initialItem) {
  var items = [initialItem]
  return {
    isEmpty: function() {
      return false
    },
    size: function() {
      return items.length
    },
    add: function(newItem) {
      (items = items.filter(function(item) {
        return item !== newItem
      })).unshift(newItem)
    },
    get: function(index) {
      return items[index]
    }
  }
}
```

```
var newRecentlyUsedList = function() {  
  var items = []  
  return {  
    isEmpty: function() {  
      return items.length === 0  
    },  
    size: function() {  
      return items.length  
    },  
    add: function(newItem) {  
      (items = items.filter(function(item) {  
        return item !== newItem  
      })).unshift(newItem)  
    },  
    get: function(index) {  
      return items[index]  
    }  
  }  
}  
}
```

```
var newRecentlyUsedList = function() {
  var items = []
  return {
    ...
    supertypeOf: function(that) {
      return that &&
        that.isEmpty && that.size && that.add &&
        that.get && that.supertypeOf && that.equals
    },
    equals: function(that) {
      var result =
        this.supertypeOf(that) &&
        that.supertypeOf(this) &&
        this.size() === that.size()
      for(var i = 0; result && i !== this.size(); ++i)
        result = this.get(i) === that.get(i)
      return result
    }
  }
}
```

***Paradigms lost?***

**Or paradigms regained?**

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

---

# The Paradigms of Programming

Robert W. Floyd  
Stanford University

---



**Paradigm**(pæ·radim, -dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. *παράδειγμα* pattern, example, f. *παραδεικνύω* to exhibit beside, show side by side. . .]

1. A pattern, exemplar, example.

1752 J. Gill *Trinity* v. 91

The archetype, paradigm, exemplar, and idea, according to which all things were made.

From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the  $i$ th variable from the  $i$ th equation. Yet further decomposition would yield a fully detailed algorithm.

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing — is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil — objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened.

<http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html>