

# Encapsulation

Programming Languages  
CS 214



The root of all software jokes is a lack of proper testing.

# Review

A *type* consists of two things:

- a set of *values*; and
- a set of *operations* onto those values.

An *abstract data type (ADT)* consists of:

- a collection of *data*:  $\text{type}_1 \times \text{type}_2 \times \dots \times \text{type}_n$
- a collection of *operations* on the data:

$F(\text{type}_1 \times \text{type}_2 \times \dots \times \text{type}_n) \rightarrow \text{ADT}$

$G(\text{ADT}) \rightarrow \text{type}_1$

...

$H(\text{ADT}) \rightarrow \emptyset$

Obviously, these two are related...



# Encapsulation

An encapsulation mechanism is a language's construct for “wrapping” data and operations into a *single syntactic structure*.

Three different encapsulation mechanisms have evolved:

- The *class*, a mechanism that lets programmers create new types that encapsulate data and operations; and
- The *module* (aka *package*), a mechanism that lets programmers store new types and their operations in a distinct container.
- The *Function* within Functional Languages, through closures, Functions encapsulate data and functionality, exposing them to functions called from within that function

The evolutionary history of these provides useful context, so we'll examine the history of each separately...



# ADTs

In the early 1970s, *imperative* programming languages had evolved to the point where much of programming was building *abstract data types (ADTs)*, which consisted of an abstract *type* and its supported *operations*.

For example, a *Stack* ADT consists of:

- *Initialization(&Stack)*  $\rightarrow \emptyset$
- *push(value  $\times$  &Stack)*  $\rightarrow \emptyset$
- *isEmpty(Stack)*  $\rightarrow bool$
- *pop(&Stack)*  $\rightarrow \emptyset$
- *isFull(Stack)*  $\rightarrow bool$
- *top(Stack)*  $\rightarrow value$

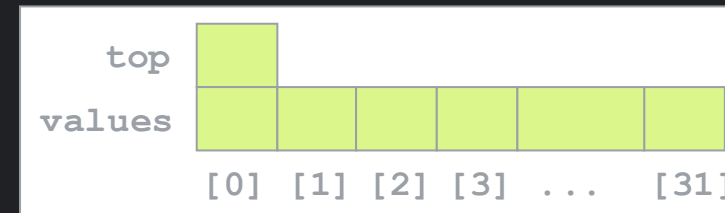
The ADT's operations make up its *programmer's interface*, through which users are to interact with the ADT.



# ADT Example in C

```
/* IntStack.h (minus precondition checks)*/
```

```
#define STACK_MAX 32
typedef struct StackStruct {
    int myTop;
    int myValues[STACK_MAX];
} IntStack;
```



```
void init(IntStack* sRef) { sRef->myTop = -1; }
```

```
int isEmpty(IntStack s) { return s.myTop < 0; }
```

```
int isFull(IntStack s) {return s.myTop == STACK_MAX-1;}
```

```
void push(int value, IntStack* sRef) {
    sRef->myTop++;
    sRef->myValues[sRef->myTop] = value;
}
```

```
void pop(IntStack* sRef) { sRef->myTop--; }
```

```
int top(IntStack s) { return s.myValues[s.myTop]; }
```



# Problem 1

Nothing prevents a programmer from *bypassing* the interface:

Instead of writing:  
a programmer  
can write:

```
while ( !isFull(s) )  
    // ... do something with s
```

```
while ( s.myTop < STACK_MAX )  
    // ... do something with s
```

If we upgrade our  
array-based  
stack to a linked  
stack:

```
/* IntStack.h ... */
```

```
typedef struct Node {  
    int value;  
    struct Node * next;  
} IntStackNode;
```

```
typedef struct StackStruct {  
    IntStackNode * top;  
} IntStack;
```

then the programmer's code *breaks* (↑ maintenance costs)!



# Problem 2

Nothing prevents the programmer from *adding operations that violate the design of the ADT...*

```
#include "IntStack.h"

int peekUnderTop(IntStack s)
{
    return s.values[s.top-1];
}
```

Such operations will also “break” if we change the implementation details that underlie the *Stack* ADT (↑ maintenance costs)!

Recall: High maintenance costs led industry from *spaghetti coding* to *structured programming*.

Eliminating the maintenance costs of “broken” code led industry from *structured programming* to *modular programming*.



Writing code is like humor

if you have to explain it, it's probably not that good.

# Modules and Packages

The problem with structured programming was that it did nothing to hide an ADT's *implementation details*.

In 1977, Wirth designed *Modula* with a new “container” construct in which a type and its operations could be stored.

- Wirth called this container the *module*.
- Rather than thinking of a type as values and operations, Wirth considered a type to be *just values* (i.e., data).
- The *module* was Wirth's construct for “wrapping” a type and its operations together (i.e., building an ADT).
- Fortran (90 and later) also provides a *Module* construct

In the 1980s, Ada adopted a similar approach for ADTs, but called their container the *package* instead of the module.



# ADT Example in Ada

```
-- IntStackPackage.ads is the IntStackPackage specification  
-- the Ada equivalent of a C header file
```

```
package IntStackPackage is  
  type IntStack is private;  
  
  procedure init(s: in out IntStack);  
  function isEmpty(s: in IntStack) return Boolean;  
  function isFull(s: in IntStack) return Boolean;  
  procedure push(value: in Integer; s: in out IntStack);  
  procedure pop(s: in out IntStack);  
  function top(s: in IntStack) return Integer;  
  
  private  
    STACK_MAX: constant Integer := 32;  
  
  type IntStack is  
    record  
      myTop: Integer;  
      myValues: array(1..STACK_MAX) of Integer;  
    end Stack;  
  
end IntStackPackage;
```

All declarations before *private* are visible externally; those after *private* are local to the package.



# ADT Example in Ada (ii)

```
-- IntStackPackage.adb is the IntStackPackage body
-- the Ada equivalent of a C implementation file

package body IntStackPackage is
  procedure init(s: in out IntStack) is
  begin
    s.myTop := 0;
  end init;

  function isEmpty(s: in IntStack) return Boolean is
  begin
    return s.myTop < 1;
  end isEmpty;

  procedure push(value: in Integer; s: in out IntStack) is
  begin
    s.myTop := s.myTop + 1; s.myValues(s.myTop) := value;
  end;

  -- ... definitions of isFull(), pop(), top(), ...

end IntStackPackage;
```



# Package Specifications

The “public section” of the specification creates its *interface*

- Nothing else in the package is accessible;
- If a programmer wishes to use the ADT, they must do so using the declarations in its interface.

An Ada package specification differs from a C header file:

- Its private section allows the package to *hide implementation details* from the programmer (everything in a header file is public).
- The specification file *must be compiled* before it can be used (and before the package body can be compiled).

By separating the ADT’s *public interface* from its *private implementation*, a programmer cannot write programs that depend upon the ADT’s implementation details.



# Example Usage

Given such an ADT, a programmer can write:

```
-- IntStackTest.adb
with TextIO, IntStackPackage; use TextIO, IntStackPackage;

procedure IntStackTest is
  s1, s2: IntStack;
  i: Integer;
begin
  init(s1);

  while (not isFull(s1)) loop
    get(i); push(i, s1);
  end loop;

  s2:= s1;

  while (not isEmpty(s2)) loop
    put(top(s2)); pop(s2);
  end loop;

end IntStackTest;
```

In modular programming, ADT operations are subprograms that receive the ADT as an argument.

Used in this way, a package/module is a container in which an ADT can be “wrapped”.



# Modules/Packages As “Objects”

Modules/packages can also be used as “objects”:

```
-- IntStack.ads is the IntStack specification

package IntStack is
  function isEmpty() return Boolean;
  function isFull() return Boolean;
  procedure push(value: in Integer);
  procedure pop;
  function top() return Integer;
end IntStack;
```

All of these identifiers are public (there is no *private* section).

The operations do *not* receive the ADT via a parameter.

There is no *init* subprogram in this kind of module/package (we’ll see why shortly).



# “Object” Package Bodies

```
-- IntStack.adb is the IntStack body

package body IntStack is

    STACK_MAX: constant Integer := 32;
    myTop: Integer;
    myValues: array(1..STACK_MAX) of Integer;

    function isEmpty() return Boolean is
    begin
        return myTop < 1;
    end;

    procedure push(value: in Integer) is
    begin
        myTop:= myTop+1; myValues(myTop) := value;
    end push;

    -- ... definitions of isFull, pop, top, ...
begin
    myTop:= 0;
end IntStack;
```

Note 1: All of the implementation details are here in the body, making them *private*.

Note 2: A package may have an *initialization block* that is executed when a program using the package is run...



# “Object” Modules in Use

Such modules/packages can be used in an object-like way:

```
-- IntStackTest.adb tests the IntStack package
```

```
with Text_IO, IntStack; use Text_IO;

procedure IntStackTest is
  i: Integer;
begin

  while (not IntStack.isFull()) loop
    get(i);
    IntStack.push(i);
  end loop;

  while (not IntStack.isEmpty()) loop
    put(IntStack.top());
    IntStack.pop;
  end loop;

end IntStackTest;
```

In this approach, a module/package superficially resembles an OO-language object (created from a class)...

But a module/package is *not a type*, so it cannot be used to create variables.

→ *Only one such “object” can exist at a time.*



# Generic Packages

Ada also allows packages to be given *type-parameters*, providing a way to circumvent the “one object” problem:

```
-- Stack.ads is the generic Stack specification
generic
  type Item is private;          -- Item is a type parameter
  Integer size;                 -- size is a data parameter
package Stack is
  function isEmpty() return boolean;
  function isFull() return boolean;
  procedure push(v: in Item);
  procedure pop();
  function top() return Item;
end Stack;
```

Ada’s keyword *generic* tells the compiler that the parameters *Item* and *size* will be supplied by the ADT’s user (ideal for containers).

Such a *Stack* stores “generic” *Items*, instead of “hardwired” *Integers*.



# Generic Package Bodies

```
-- Stack.adb is the generic Stack body

package body Stack is

  myCapacity: Integer := size;
  myTop: Integer;
  myValues: array(1..myCapacity) of Item;

  function isEmpty() return Boolean is
  begin
    return myTop < 1;
  end;

  procedure push(value: in Item) is
  begin
    myTop:= myTop+1; myValues(myTop) := value;
  end push;

  -- ... definitions of isFull, pop, top, ...
begin
  myTop:= 0;
end Stack;
```

This far more elegant than C++ *templates*: the compiler already knows that *Stack* is a generic package because the spec. is compiled first.

Recent versions of Fortran's *module* have added this *generic* mechanism.



# Generic Instantiation

Now we can dynamically create multiple *Stack* “objects”:

```
-- StackTest.adb tests the generic Stack package
```

```
with Text_IO, Stack; use Text_IO;
procedure StackTest is
  i: Integer;
  package intStack1 is new Stack(integer, 8);
  package intStack2 is new Stack(integer, 8);
begin
  while (not intStack1.isFull()) loop
    get(i);
    intStack1.push(i);
  end loop;

  intStack2:= intStack1;

  while (not intStack2.isEmpty()) loop
    put(intStack2.top());
    intStack2.pop;
  end loop;
end StackTest;
```

This permits generic Ada packages to be constructed and operated on in a way similar to objects.



Why do programmers always mix up Halloween and Christmas?

Because Oct 31 == Dec 25.

There are 2 hard problems in computer science:

- cache invalidation
- naming things
- off-by-1 errors

# History: Simula

Back in 1967, *Dahl & Nygaard* noted that at its simplest, *programming consists of operating on variables*.

That is, variables are *typed objects*, and since types consist of values and operations, programming can be reduced to operating on variables.

Dahl & Nygaard were working on constructs to simplify the representation of “*real world*” objects in software, so that real-world processes could be more easily *simulated*.

Their language was *Simula* (*Simple universal language*) and it provided useful *Simulation* and *Process* constructs...



# Simula Classes

Dahl & Nygaard reasoned that if types are *values plus operations*, then a language should provide a syntactic structure that explicitly combines data and operations.

They took the *record* construct, extended it to store *subprograms* as well as *data* and christened it the *class* (from mathematics).

Subprograms (operations)
Data (state information)

Their *class* construct provided for creating *new types*, which could then be used to *declare variables*.

In time, variables were replaced by *objects* and object-oriented programming (OOP), culminating in *Smalltalk*.



# Example: An *IntStack* Class

```
! IntStack.sim (minus precondition checks);

Class IntStack(Size);           ! Classes can have parameters;
  Integer Size;                 ! Params precede 'Begin';
Begin

                                ! Attribute variables;
  Integer myTop;                ! Data encapsulated;
  Integer Array myValues(1:Size); ! but not hidden

                                ! Methods;
                                ! Operations encapsulated;
  Procedure Init;               ! Initialization;
    myTop:= 0;                  ! 1-line methods need no 'end';

  Boolean Procedure IsEmpty;    ! Functions are typed procs;
    IsEmpty:= myTop < 1;       ! Assign RV to function name;

  Boolean Procedure IsFull;
    IsFull:= myTop >= Size;

! ... continued on next page ... ;
```



# Example: An *IntStack* Class (ii)

```
! IntStack.sim (continued)
```

```
Procedure Push(Value);      ! Methods can have parameters;  
  Integer Value;           ! Parameters precede 'Begin';  
Begin                       ! Methods with multiple statements;  
  myTop:= myTop + 1;       ! must be 'wrapped' in a block;  
  myValues(myTop) := Value;  
End of Push;
```

```
Procedure Pop;  
  myTop:= myTop - 1;
```

```
Integer Procedure Top;  
  Top:= myValues(myTop);
```

```
Init;                       ! Life: code following methods;  
                             ! is executed on object creation;  
End of IntStack;
```

Simula-67 did not hide the data: nothing prevented code from:

- *accessing the class's implementation details*; or
- *violating the intent of the creator of the class*.



# Simula: Using the *IntStack* Class

To use the class, we write something like this:

```
! IntStackTest.sim  
  
Begin  
  Ref(IntStack) S1, S2;      ! Reference (pointer) variables;  
  Integer value;           ! Normal variable;  
  
  S1 :- New IntStack(8);     ! Special reference assignment;  
  
  While Not S1.IsFull Do Begin ! Dot-notation for messages;  
    Value:= InInt;         ! no args, no parentheses;  
    S1.Push(Value);        ! args require parentheses;  
  End;  
  
  S2 :- S1;                 ! Reference assignment  
  
  While Not S2.IsEmpty Do Begin  
    OutInt(S2.Top);  
    S2.Pop;  
  End;  
End of Program;
```



Nurse: Doctor, there's a patient on line 1 who says he's invisible!

Doctor: ...

Well, tell him I can't see him right now!

# C++ Classes

In 1986, Stroustrup added the *class* construct to C...

```
// IntStack.h

class IntStack {
public:
    IntStack(int size);
    bool isEmpty() const;
    bool isFull() const;
    void push(int value);
    void pop();
    int top() const;
private:
    int myTop, myCapacity;
    int * myValues;
};
```

```
// IntStack.h (cont'd)

inline IntStack::IntStack(int size) {
    myTop = -1;
    myCapacity = size;
    myValues = new int[size];
}

inline
bool IntStack::isEmpty() const {
    return myTop < 0;
}

inline void
IntStack::push(int value) const {
    myValues[++myTop] = value;
}

// ... other operation definitions ...
```

More complex operations can be separately compiled...



# Using C++ Classes

C++ objects can be either statically or dynamically

```
// IntStackTest1.cpp
#include "IntStack.h"

int main() {
    IntStack s(8);
    int aValue;

    while ( !s.isFull() ) {
        cin >> aValue;
        s.push(aValue);
    }

    while ( !s.isEmpty() ) {
        cout << s.top() << ' ';
        s.pop();
    }
}
```

```
// IntStackTest2.cpp
#include "IntStack.h"

int main() {
    IntStack* s = new IntStack(8);
    int aValue;

    while ( !s->isFull() ) {
        cin >> aValue;
        s->push(aValue);
    }

    while ( !s->isEmpty() ) {
        cout << s->top() << ' ';
        s->pop();
    }
}
```

There are languages where objects *must* be dynamically allocated.

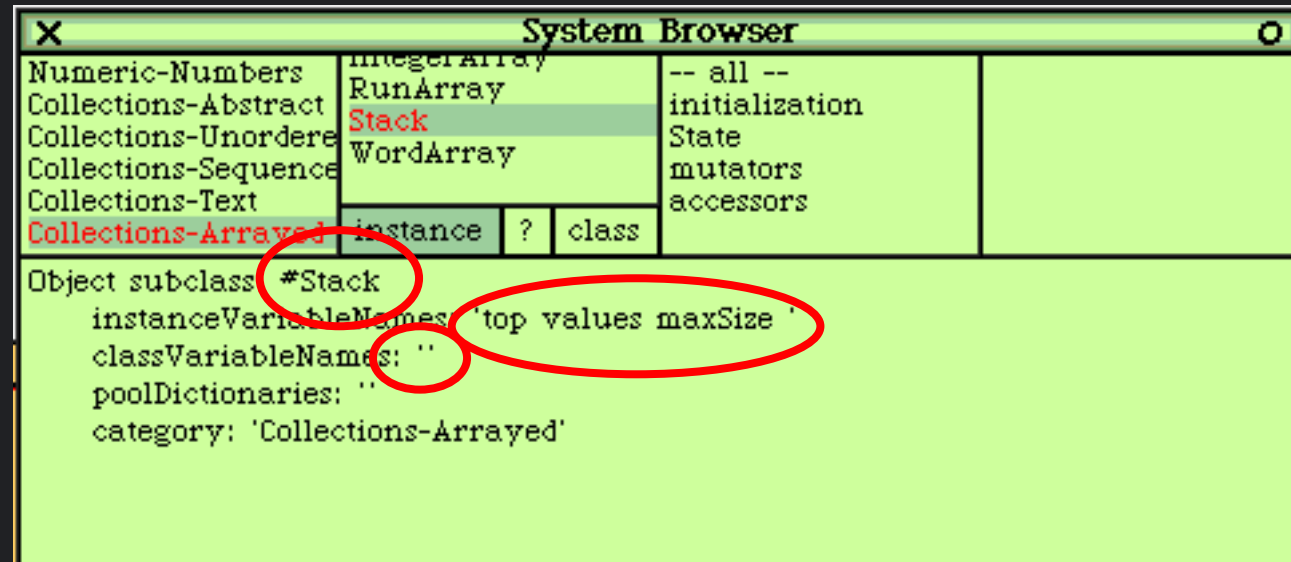


# Smalltalk

Smalltalk took the ideas of Simula and extended them:

- Everything (including programs) are *objects* or *methods*
- Attribute variables are *hidden* (private) by default
- A large predefined *class library* is provided
- A GUI *integrated development environment (IDE)* is provided

Building a Smalltalk *Stack* class is simple, as most of the syntax is auto-generated by the IDE:



The screenshot shows a 'System Browser' window with a table of classes and a detailed view of the '#Stack' class. The table lists classes like 'IntegerArray', 'RunArray', 'Stack', and 'WordArray'. The '#Stack' class is highlighted, and its definition is shown below, with red circles around the class name and instance variable names.

Class Name	Superclass	Category
Numeric-Numbers	IntegerArray	-- all --
Collections-Abstract	RunArray	initialization
Collections-Unordered	Stack	State
Collections-Sequence	WordArray	mutators
Collections-Text		accessors
Collections-Arrayed	instance ? class	

```
Object subclass: #Stack
  instanceVariableNames: 'top values maxSize'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Collections-Arrayed'
```



# Smalltalk Operations

Adding *Stack* operations to our class is also easy: as the IDE makes templates for us

Smalltalk *Arrays* store *Objects* (more later)...

*new:* is a message we send the class (i.e., a *class method*):

The screenshot shows the System Browser window with the `Stack` class selected. The `initialize:` method is highlighted, and its implementation is shown below:

```
initialize: size
  myTop ← 1
  myValues ← Array new: size.
  myCapacity ← size
```

The `Array new: size.` line is circled in red.

The screenshot shows the System Browser window with the `Stack` class selected. The `new:` method is highlighted, and its implementation is shown below:

```
new: size
  ↑ super new initialize: size
```

The `↑ super new initialize: size` line is circled in red.



# Smalltalk Operations (ii)

Messages that are sent to an object are *instance methods*; messages that are sent to the class are *class methods*.

Mutators like *push* and *pop* are equally easy:

The screenshot shows the System Browser window with the **Stack** class selected. The class hierarchy is shown on the left, with **Stack** highlighted in red. The right pane shows the class methods: `-- all --`, `initialization`, `State`, `mutators`, and `accessors`. The `mutators` category is selected, showing the `push:` message. The bottom pane shows the implementation of the `push:` message:

```
push: value
myValues at: myTop put: value.
myTop + myTop + 1
```

Other *Stack* operations (*isEmpty*, *isFull*, *top*) are just as easy...

The screenshot shows the System Browser window with the **Stack** class selected. The class hierarchy is shown on the left, with **Stack** highlighted in red. The right pane shows the class methods: `-- all --`, `initialization`, `State`, `mutators`, and `accessors`. The `accessors` category is selected, showing the `top` message. The bottom pane shows the implementation of the `top` message:

```
top
+ myValues at: myTop - 1
```

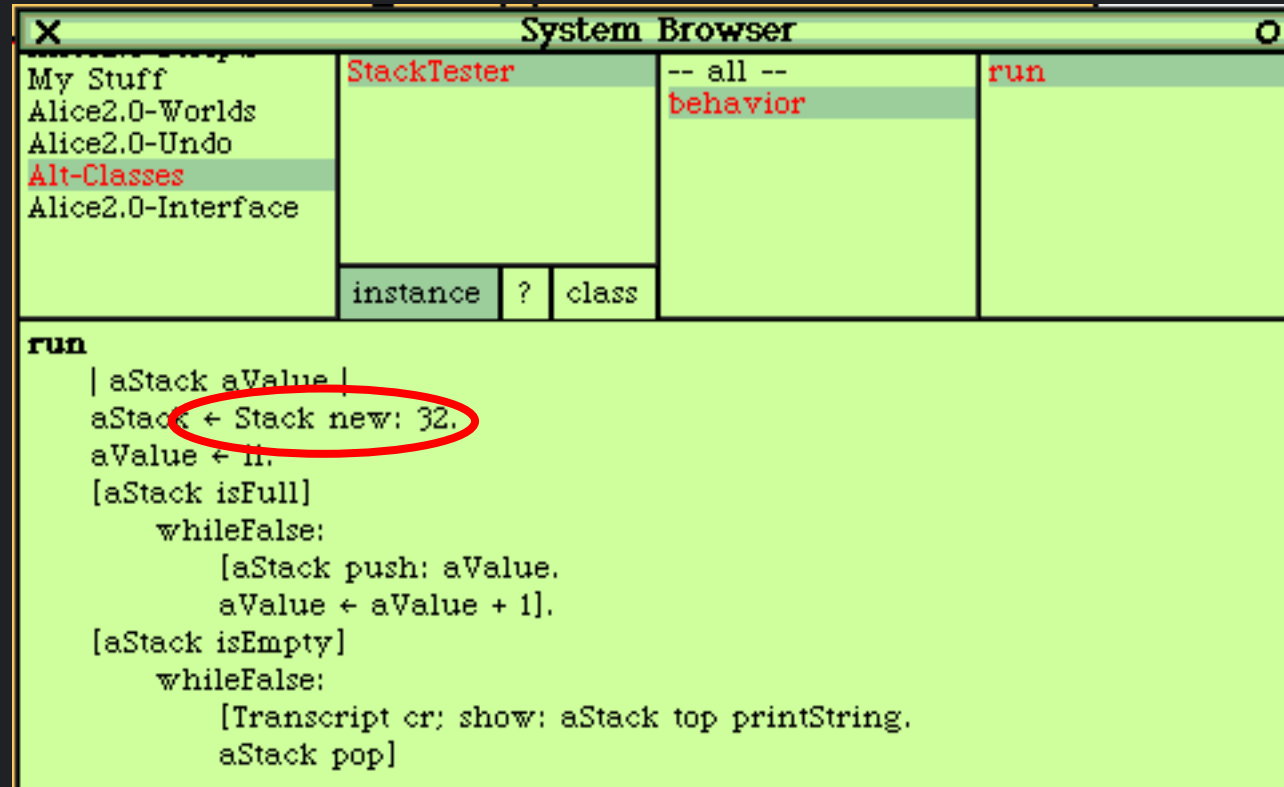


# Smalltalk "Programs"

"Programs" are classes with *new* and *run* methods:

Note that we send *new:* to the *class*, not an *instance* of the class (i.e., an *object*)...

To run such programs:

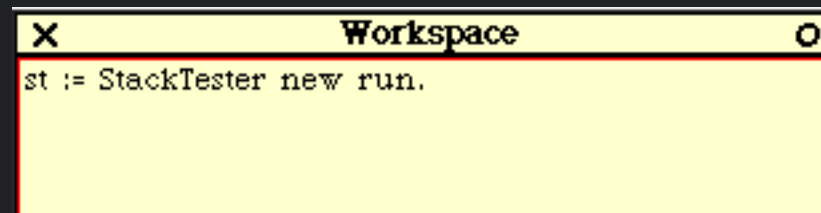


The screenshot shows the System Browser window with the following content:

My Stuff	StackTester	-- all --	run
Alice2.0-Worlds		behavior	
Alice2.0-Undo			
Alt-Classes			
Alice2.0-Interface			

Below the table, there are buttons for "instance", "?", and "class".

```
run
| aStack aValue |
aStack ← Stack new: 32.
aValue ← 1.
[aStack isFull]
  whileFalse:
    [aStack push: aValue.
     aValue ← aValue + 1].
[aStack isEmpty]
  whileFalse:
    [Transcript cr; show: aStack top printString.
     aStack pop]
```



The screenshot shows the Workspace window with the following code:

```
st := StackTester new run.
```



# Java Classes

In 1993, Gosling designed *Java*, based on C++ and Smalltalk.

```
// IntStack.java

public class IntStack {
    public IntStack(int size) {
        myTop = -1;
        myCapacity = size;
        myValues = new int[size];
    }

    public boolean isEmpty() {
        return myTop < 0;
    }

    public boolean isFull() {
        return myTop >= myCapacity;
    }
}
```

```
// IntStack.java (cont'd)

    public void push(int value) {
        myValues[++myTop] = value;
    }

    public void pop() {
        --myTop;
    }

    public int top() {
        return myValues[myTop];
    }

    private int    myTop,
                  myCapacity;
    private int [] myValues;
}
```

Java mixes C++ syntax with Smalltalk philosophy...



# Using Java Classes

Like Smalltalk,  
Java objects *must*  
be dynamically  
allocated:

Class variables are  
*reference variables*,  
so dot notation is  
used (vs. C++ ->)

Also like Smalltalk,  
every Java  
subprog. must be a  
member of a class.

```
// IntStackTest.java

import IntStack;

class IntStackTest {
    public static void main(String [] args) {
        IntStack s = new IntStack(8);
        int aValue = 11;

        while ( !s.isFull() ) {
            s.push(aValue);
            aValue++;
        }

        while ( !s.isEmpty() ) {
            System.out.println( s.top() );
            s.pop();
        }
    }
}
```

But most of Java's *syntax* is more similar to C++ than Smalltalk...



# C++ Templates

C++ classes can have *parameters* (→ Ada *generic packages*):

```
// Stack.h
template<class Item>
class Stack {
public:
    Stack(int size);
    bool isEmpty() const;
    bool isFull() const;
    void push(Item value);
    void pop();
    Item top() const;

private:
    int myTop, myCapacity;
    Item * myValues;
};

#include "Stack.hpp"
```

```
// Stack.hpp
template<class Item>
inline Stack<Item>::Stack(int size) {
    myTop = -1;
    myCapacity = size;
    myValues = new Item[size];
}

template<class Item>
inline bool Stack<Item>::isEmpty() {
    return myTop < 0;
}

template<class Item>
inline Item Stack<Item>::top() {
    return myValues[myTop];
}

// ... isFull, push, pop, ...
```

This does not feel as integrated as Ada's *generic* mechanism...



# C++ Template Instantiation

A *template* is a *factory* for creating classes...

```
// StackTest1.cpp

#include "Stack.h"

int main() {
    Stack<int> s(32);
    int aValue;

    while ( !s.isFull() ) {
        cin >> aValue;
        s.push(aValue);
    }

    while ( !s.isEmpty() ) {
        cout << s.top() << ' ';
        s.pop();
    }
}
```

```
// StackTest1.cpp

#include "Stack.h"
#include "Student.h"

int main() {
    Stack<Student> s(32);
    Student aValue;

    while ( !s.isFull() ) {
        cin >> aValue;
        s.push(aValue);
    }

    while ( !s.isEmpty() ) {
        cout << s.top() << ' ';
        s.pop();
    }
}
```

Templates are esp. useful in creating classes that *contain* other objects.



# Java Containers

Java 1.5 added *generics* and *auto-boxing/unboxing*:

```
// Stack.java

public class Stack<Item> {
    public Stack(int size) {
        myTop = -1;
        myCapacity = size;
        myValues = (Item[])
                    new Object[size];
    }
    public boolean isEmpty() {
        return myTop < 0;
    }
    public boolean isFull() {
        return myTop >= myCapacity-1;
    }
}
```

```
// Stack.java (cont'd)

public void push(Item value) {
    myValues[++myTop] = value;
}
public void pop() {
    --myTop;
}
public Item top() {
    return myValues[myTop];
}
private int      myTop,
                 myCapacity;
private Item []  myValues;
}
```

Such a *Stack* can still store anything derived from *Object*...



# Java: Using *Generic* Containers

Since `Stack<Item>` stores *Objects*...

1. We can't pass a primitive type arg (e.g., `int`), but we can pass a *wrapper-arg*.

2. *Auto-boxing* lets us pass primitive-type-values as args.

3. *Auto-unboxing* lets us retrieve *Items* as primitive type-vals.

```
// StackTester.java
import Stack;

class StackTester {
    public static void main(String [] args) {
        Stack<Integer> s = new Stack<>(8);
        int aValue = 11;

        while ( !s.isFull() ) {
            s.push( aValue ); // int auto-boxed
            aValue++;
        }

        while ( !s.isEmpty() ) {
            int anInt = s.top(); // auto-unboxed
            System.out.println(anInt);
            s.pop();
        }
    }
}
```



What's the difference between a poorly dressed man on a bicycle and a nicely dressed man on a tricycle?

Attire!

# Functional Encapsulation

In Clojure, `defn-` allows us to declare a method locally

```
⇒ (defn test []  
  (do  
    (defn- innerTest [] "testing")  
    (innerTest)  
  )  
)
```

```
⇒ (test)
```

```
"testing"
```

```
⇒ (innerTest)
```

```
ERROR - Cannot read properties of undefined (reading 'call')
```

Local functions and variables are a form of encapsulation,  
and the primary means of it in functional languages



# Summary

To achieve the goals of abstract data typing, we need:

- *Encapsulation*: data and operations in 1 syntactic unit;
- *Data hiding*: the ability to restrict access to ADT data.

Three different mechanisms have evolved for doing so:

- *The module*: a container for storing a data-type and its operations
- *The class*: a type-constructor that stores data and operations.
- *The Function*: the fulfillment of a contract, maintaining data within its own closure.

The class is better for *flexibility, reusability, and maintenance*,  
the module is better for *time-efficiency and performance*,  
Functions are better for *Multithreading and performance*.

*Containers* are packages or classes that store data-values.

- *Generic packages* and *templates* support strongly typed containers.
- *Generic containers* can be built in most modern languages.

