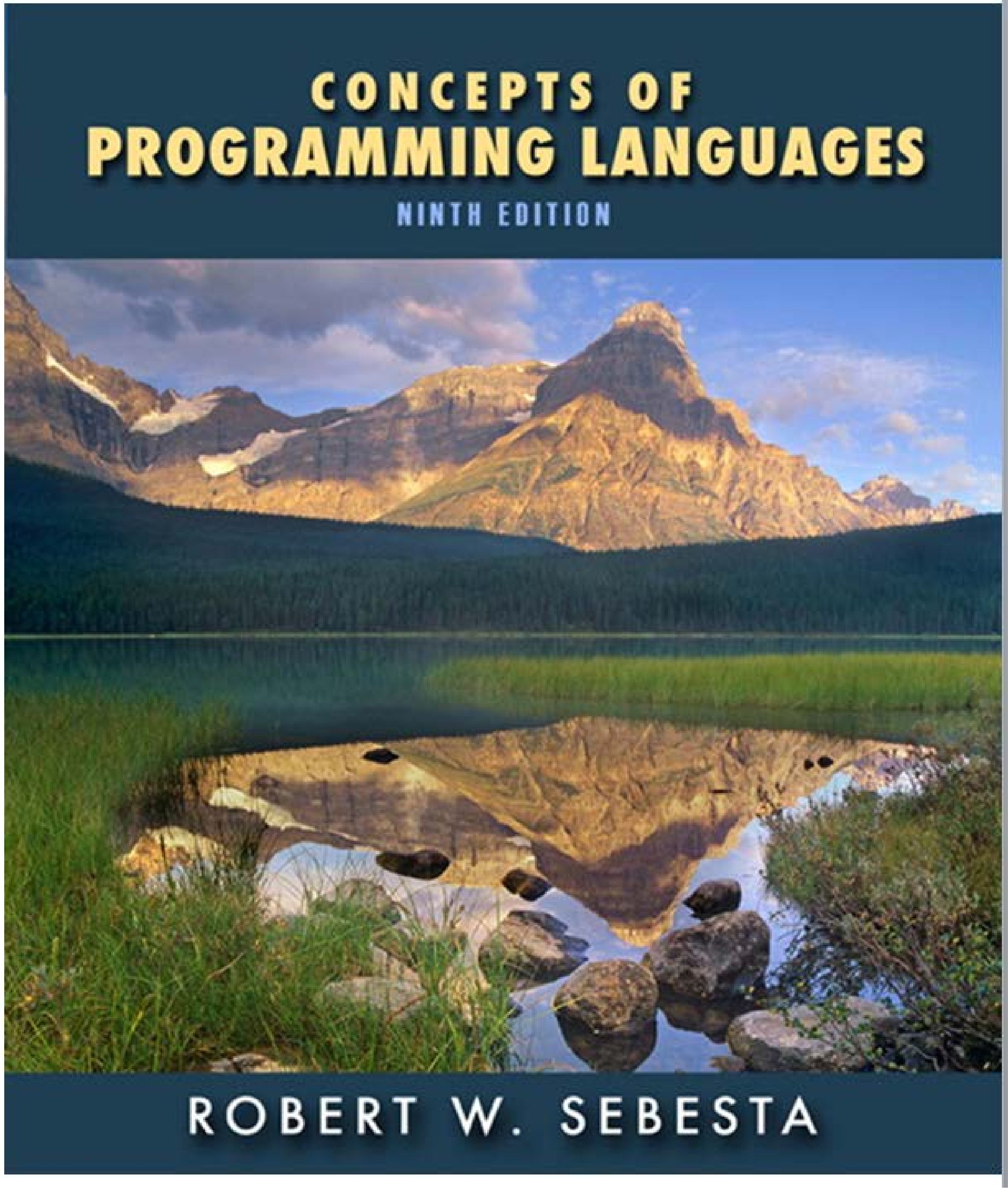
INF 212
ANALYSIS OF PROG. LANGS
ABSTRACT DATA TYPES

Instructors: Crista Lopes



CHAPTER 11

Abstract Data Types and Encapsulation Concepts

Chapter 11 Topics

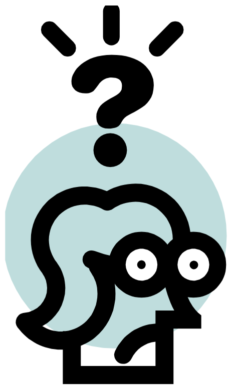
- The Concept of Abstraction
- Introduction to Data Abstraction
- Design Issues for Abstract Data Types
- Language Examples
- Parameterized Abstract Data Types
- Encapsulation Constructs
- Naming Encapsulations

The Concept of Abstraction

- An *abstraction* is a view or representation of an entity that includes only the most significant attributes
- The concept of *abstraction* is fundamental in programming (and computer science)
- Nearly all programming languages support *process abstraction* with subprograms
- Nearly all programming languages designed since 1980 support *data abstraction*

Introduction to Data Abstraction

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition



Advantages of Data Abstraction

- Advantage of the first condition
  - Program organization, modifiability (everything associated with a data structure is together), and separate compilation
- Advantage the second condition
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code

Language Requirements for ADTs

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

Design Issues

- What is the form of the container for the interface to the type?
- Can abstract types be parameterized?
- What access controls are provided?

Language Examples: Ada

- The encapsulation construct is called a *package*
  - Specification package (the interface)
  - Body package (implementation of the entities named in the specification)
- Information Hiding
  - The spec package has two parts, public and private
  - The name of the abstract type appears in the public part of the specification package. This part may also include representations of unhidden types
  - The representation of the abstract type appears in a part of the specification called the *private* part
    - More restricted form with *limited private types*
      Private types have built-in operations for assignment and comparison

      Limited private types have NO built-in operations

Language Examples: Ada (continued)

- Reasons for the public/private spec package:

  1. The compiler must be able to see the representation after seeing only the spec package (it cannot see the private part)
  2. Clients must see the type name, but not the representation (they also cannot see the private part)

An Example in Ada

```
package Stack_Pack is
   type stack_type is limited private;
   max_size: constant := 100;
   function empty(stk: in stack_type) return Boolean;
   procedure push(stk: in out stack_type; elem:in Integer);
   procedure pop(stk: in out stack_type);
   function top(stk: in stack_type) return Integer;

   private  -- hidden from clients
   type list_type is array (1..max_size) of Integer;
   type stack_type is record
        list: list_type;
        topsub: Integer range 0..max_size) := 0;
   end record;
end Stack_Pack
```

Language Examples: C++

- Based on C `struct` type and Simula 67 classes
- The class is the encapsulation device
- All of the class instances of a class share a single copy of the member functions
- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic

Language Examples: C++ (continued)

- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance (Chapter 12)

Language Examples: C++ (continued)

- Constructors:
  - Functions to initialize the data members of instances (they *do not* create the objects)
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
  - Name is the same as the class name

Language Examples: C++ (continued)

- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class Stack {
  private:
      int *stackPtr, maxLen, topPtr;
  public:
      Stack() { // a constructor
            stackPtr = new int [100];
            maxLen = 99;
            topPtr = -1;
      };
      ~Stack () {delete [] stackPtr;};
      void push (int num) {…};
      void pop () {…};
      int top () {…};
      int empty () {…};
}
```

A Stack class header file

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private: //** These members are visible only to other
//** members and friends (see Section 11.6.4)
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
  Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

The code file for Stack

```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Language Examples: Java

- Similar to C++, except:
  - All user-defined types are classes
  - All objects are allocated from the heap and accessed through reference variables
  - Individual entities in classes have access control modifiers (private or public), rather than clauses
  - Java has a second scoping mechanism, package scope, which can be used in place of friends
    - All entities in all classes in a package that do not have access control modifiers are visible throughout the package

An Example in Java

```
class StackClass {
  private:
      private int [] *stackRef;
      private int [] maxLen, topIndex;
      public StackClass() { // a constructor
            stackRef = new int [100];
            maxLen = 99;
            topPtr = -1;
      };
      public void push (int num) {…};
      public void pop () {…};
      public int top () {…};
      public boolean empty () {…};
}
```

Abstract Data Types in Ruby

- Encapsulation construct is the class
- Local variables have “normal” names
- Instance variable names begin with “at” signs (`@`)
- Class variable names begin with two “at” signs (`@@`)
- Instance methods have the syntax of Ruby functions (`def … end`)
- Constructors are named `initialize` (only one per class)—implicitly called when `new` is called
  - If more constructors are needed, they must have different names and they must explicitly call `new`
- Class members can be marked private or public, with public being the default
- Classes are dynamic

Abstract Data Types in Ruby (continued)

```
class StackClass {
  def initialize
    @stackRef = Array.new
    @maxLen = 100
    @topIndex = -1
   end

   def push(number) … end
   def pop … end
   def top … end
   def empty … end
end
```

Summary

- The concept of ADTs and their use in progra milestone in the development of languages
- Two primary features of ADTs are the packa their associated operations and information hiding
- Ada provides packages that simulate ADTs
- **C++ data abstraction is provided by classes**
- **Java's data abstraction is similar to C++**
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs
- C++, C#, Java, Ada, and Ruby provide naming encapsulations

Reset

What's an Abstract Data Type?

- From http://stackoverflow.com/tags/abstract-data-type/info:
  "An abstract data type (ADT) is a specification for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects of those operations."

ABSTRACT

- Does not specify how the data type is implemented
- 1 ADT → multiple implementations

CONCRETE

- Has 1 implementation

What about subtyping?

- class A extends B {…}

- Doesn’t make A abstract
- Regular subtyping relation, see type systems

Specifying ADTs in Java

```
public interface Queue<E> extends Collection<E> {
        boolean add(E e);
        E element();
        boolean offer(E e);
        E peek();
        E poll();
        E remove();
}
```

Implementing ADTs in Java

```
public class LinkedList<E> implements Queue<E> {
	boolean add(E e) {…}
	E element() {…}
	boolean offer(E e) {…}
	E peek() {…}
	E poll() {…}
	E remove() {…}
}
```

```
public class PriorityQueue<E> implements Queue<E> {
	boolean add(E e) {…}
	E element();
	boolean offer(E e) {…}
	E peek() {…}
	E poll() {…}
	E remove() {…}
}
```

Are Java classes ADTs?

Are Java classes ADTs?

- No, unless they are marked abstract

- (yes, abstract Java classes are ADTs, like interfaces)

Are C++ classes ADTs?

Are C++ classes ADTs?

- No, unless they are marked abstract

Are Python classes ADTs?

- No, unless they are marked abstract.

Are classes ADTs?

- No, unless they are marked abstract.

Abstract Data Types

- What part of the word ABSTRACT is unclear?!?
- abstract = no concrete implementation