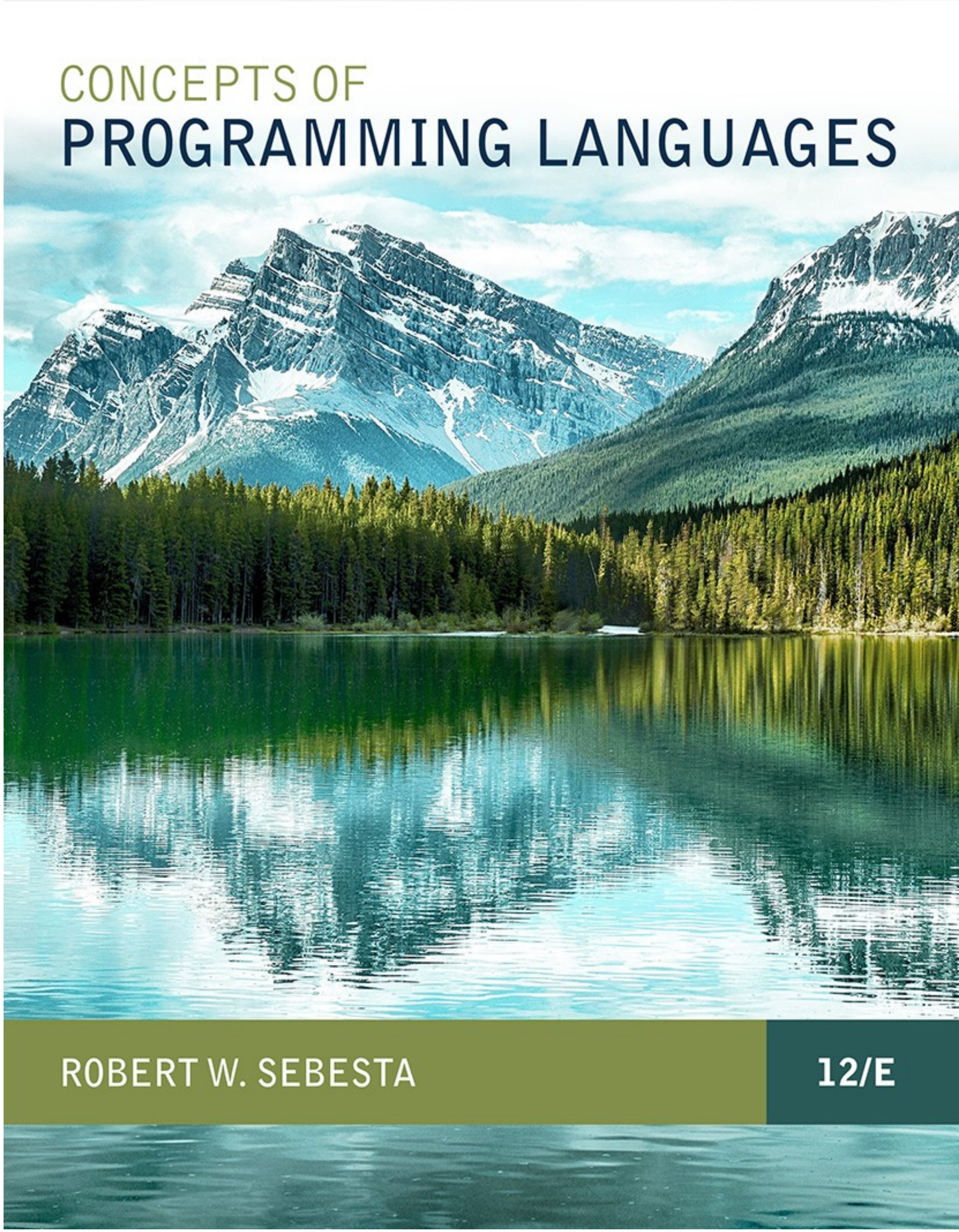
Chapter 11

Abstract Data Types and Encapsulation Concepts



ISBN 0-321-49362-1

Chapter 11 Topics

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

The Concept of Abstraction

- ***Abstraction***:
  - a view or representation of an entity that includes only the most significant attributes
  - **Bird** : two wings, two legs, one tail, able to fly
  - E.g., Crows, robins, sparrows
- ***Data Abstraction:***
  - almost all programming languages designed since 1980 support *data abstraction*

Introduction to Data Abstraction

- *Abstract Data Type* (ADT)
  - **user-defined data type** satisfies the following two conditions:
    - The *data representation* of the object type is **hidden** from the program using the object; only operations are provided in the type's definition
    - The *declarations* of the type and the *protocols* of the operations in the object type are contained in a single syntactic unit (e.g., a C++/Java class).
  - Other programs are allowed to create variables of the defined type.

An Example in C++

```
class Stack {
   private:
        int *stackPtr, maxLen, topPtr;          // data members
   public:
        Stack() { // constructor
               stackPtr = new int [100];
               maxLen = 99;
               topPtr = -1;
        };
        ~Stack () {delete [] stackPtr;};        // destructor
        void push (int number) {
          if (topSub == maxLen)
            cerr << "Error in push - stack is full\n";
          else stackPtr[++topSub] = number;
        };
        void pop () {…};                        // member functions
        int top () {…};
        int empty () {…};
}
```

Advantages of Data Abstraction

- **Advantages** of the first condition
  - Reliability--by hiding the data representations,
    - user code cannot directly access objects of the type
    - allow the representation to be changed without affecting user code
  - Reduces the range of code and variables of which the programmer must be aware
  - Name conflicts are less likely
- **Advantages** of the second condition
  - Provides a method of program organization
  - Aids modifiability (everything associated with a data structure is together)
  - Separate compilation

Language Requirements for ADTs

- Information Hiding
  - A **method** of **type names** and **subprogram headers visible to clients**, while hiding actual definitions
  - e.g., Java Interface; C++ header file
- Encapsulation
  - A **syntactic unit encapsulates** the type definition

Design Issues

- Can abstract types be **parameterized**?
- What access controls are **provided**?
- Is the specification of the type physically **separate** from its **implementation**?

Language Examples: C++

- **class** is the encapsulation device in C++
- A class is a type
- All of the **class** instances of a class share a single copy of the member functions
- Each instance of a class—Object, has its own copy of the data members
- Class Instances can be:
  - static: e.g., int CSCI_6221[8];
  - stack dynamic: referred by variable; e.g. pass-by-reference
  - heap dynamic: **using new(),** referred by a pointer

An Example in C++

```
class Stack {
  private:
      int *stackPtr, maxLen, topPtr;          // data members
  public:
      Stack() { // constructor
              stackPtr = new int [100];
              maxLen = 99;
              topPtr = -1;
      };
      ~Stack () {delete [] stackPtr;};        // destructor
      void push (int number) {
         if (topSub == maxLen)
           cerr << "Error in push - stack is full\n";
         else stackPtr[++topSub] = number;
      };
      void pop () {…};                        // member functions
      int top () {…};
      int empty () {…};
}
```

Language Examples: C++

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

  - E.g., class CSCI {

```
public:      int  publicCSCIMember;
protected:   int  protectedCSCIMember;
private:     int  privateCSCIMember;
};
```

Language Examples: C++

- **Constructors**:
  - Functions to initialize the **data members** of instances (they *do not* create the objects)
  - May 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 (a public function)
  - Name is the same as the class name

Language Examples: C++

- **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 (~)
  - The destructor of a class will run when its lifetime is over. If you want its memory to be freed and the destructor run, you have to delete it if it was allocated on the **heap**. If it was allocated on the **stack** this happens automatically (i.e. when it goes out of scope.). If it is a member of a class (not a pointer, but a full member), then this will happen when the containing object is destroyed.
  - Ref: http://stackoverflow.com/questions/677653/does-delete-call-the-destructor

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 member functions are visible to clients
  Stack(); //** constructor
  ~Stack(); //** 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() { //** constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Language Examples: C++

- Friend functions or classes – to provide access to private members to some unrelated units or functions
  - Necessary in C++

Language Examples – Objective-C

- Interface container   // class definition

```
@interface class-name: parent-class {
   instance variable declarations
 }
   method prototypes
@end
```

- Implementation container

```
@implementation class-name
 method definitions
@end
```

- Classes are types

Language Examples – Objective-C

```
// stack.m - interface and implementation for a simple stack
#import <Foundation/Foundation.h>
@interface Stack: NSObject {
  int stackArray[100], *stackPtr, maxLen, topSub;}
  -(Stack *) initWith;                    // constructor
  -(void) push: (int) number;             // - xxx : instance method
  -(void) pop;
  -(int) top;
  -(int) empty;
@end
@implementation Stack
  -(Stack *) initWith {
    maxLen = 100;
    topSub = -1;
    stackPtr = stackArray;
    return self;   }
@end
```

Language Examples – Objective-C

- Method prototypes form

  (+ | -) (return-type) method-name [: (formal-parameters)];

  - Plus (+) indicates a class method
  - Minus (-) indicates an instance method
  - Parameter list format is different
    - No parameter (method name is `meth0`)

      `-(void) meth0;`
    - One parameter (method name is `meth1:`)

      `-(void) meth1: (int) x;`
    - Two parameters: (method name is `meth2:second:`)

      `-(int) meth2: (int) x second: (float) y;`

Language Examples – Objective-C

- Method call syntax

  [object-name method-name];

  Examples:

```
[myAdder add1: 7];          // 1 parameter
[myAdder add1: 7: 5: 3];   // 3 parameters
```

  – For the method:

```
-(int) meth2: (int) x second: (float) y;
```

  the call would be like the following:

```
[myObject meth2: 7 second: 3.2];
```

Language Examples – Objective-C

- Constructors:
  - are called *initializers* – all they do is *initialize variables*
  - Initializers can have any name – they are always called explicitly
  - Initializers always return `self`
- Objects:
  - are created by calling ***`alloc`*** and the constructor

```
Adder *myAdder = [[Adder alloc] init];
```

- All class instances are heap dynamic

Language Examples – Objective-C

```
// stack.m - interface and implementation for a simple stack
#import <Foundation/Foundation.h>
@interface Stack: NSObject {
  int stackArray[100], stackPtr, maxLen, topSub;}
  -(Stack *) initWith;         // constrcutor
  -(void) push: (int) number;
  -(void) pop;
  -(int) top;
  -(int) empty;
@end
@implementation Stack
  -(Stack *) initWith {
    maxLen = 100;
    topSub = -1;
    stackPtr = stackArray;
    return self;  }   ...
```

Language Examples – Objective-C

```
// stack.m - continued
  -(void) push: (int) number {
    if (topSub == maxLen)
      NSLog(@"Error in push - stack is full");
    else
      stackPtr[++topSub] = number;
  ...
}

...

@end
```

Language Examples – Objective-C

- An example of using of `stack.m`
  - Placed in the **`@implementation`** of `stack.m`

```
int main (int argc, char *argv[]) {
  int temp;
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  Stack *myStack = [[Stack alloc] initWith];
  [myStack push: 5];
  [myStack push: 3];
  temp = [myStack top];
  NSLog(@"Top element is: %i", temp);
  [myStack pop];
  temp = [myStack top];
  NSLog(@"Top element is: %i", temp);
  [myStack pop];
  [myStack release]; // give up the ownership of an object
  [pool drain];
  return 0;
}
```

Language Examples – Objective-C

- To **import** standard prototypes (e.g., i/o)

```
#import <Foundation/Foundation.h>
```

- The first thing a program must do is allocate and initialize a pool of storage for its data (pool's variable is `pool` in this case)

```
NSAutoreleasePool * pool =
            [[NSAutoreleasePool alloc] init];
```

- At the end of the program, the pool is released with:

```
[pool drain];
```

Language Examples – Objective-C

- Information Hiding
  - The directives `@private` and `@public` are used to specify the access of instance variables.
  - The default access is protected (private in C++)
  - There is **no** way to restrict access to methods
  - The name of a *getter* method is always the name of the instance variable
  - The name of a *setter* method is always the word set with the capitalized variable's name attached

Language Examples – Objective-C

- Getter & Setter
  - ```
    @interface MyClass : NSObject {
     NSString *finalGrade;  }  // variable
     -(NSString*) finalGrade;  // getter & setter
     -(void) setFinalGrade: (NSString *) gradeValue;
     @end
    ```
  - ```
    @interface MyClass : NSObject {
            @property NSString *finalGrade;   }
     @end

     @implementation MyClass
            @synthesize finalGrade;
            // create the same getter & setter above
     @end
    ```

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 (e.g., private:)
  - Implicit garbage collection of all objects
  - 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 int [] *stackRef;
        private int [] maxLen, topIndex;
        public StackClass() { // constructor
                stackRef = new int [100];
                maxLen = 99;
                topPtr = -1;
        };
        public void push (int num) {…};
        public void pop () {…};
        public int top () {…};
        public boolean empty () {…};
}
```

Language Examples: C#

- Based on C++ and Java
- Adds two more access modifiers, *internal* and *protected internal*
- *Ref: https://msdn.microsoft.com/en-us/library/7c5ka91b.aspx*
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- `structs` are lightweight classes that do not support inheritance

Language Examples: C#

- Common solution for accessing to data members: accessor methods (getter and setter)
- C# provides *properties* as a way of implementing getters and setters without requiring explicit method calls

C# Property Example

```
public class Weather {
   public int DegreeDays { //** DegreeDays is a property
      get {return degreeDays;}
      set {
        if (value < 0 || value > 30)
          Console.WriteLine(
               "Value is out of range: {0}", value);
        else degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Language Examples: Ruby

- Class:
  - Encapsulation construct; are dynamic
  - Class members can be marked private or public, with public being the default
- Variables:
  - 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`**

Language Examples: Ruby

```
class StackClass
   def initialize                             # constructor
     @stackRef = Array.new                    # @xyz : instance variable
     @maxLen = 100
     @topIndex = -1
   end
   def push(number)
     temp = nil;
     if @topIndex == @maxLen
       puts "Error in push - stack is full"
     else
       @topIndex = @topIndex + 1
       @stackRef[@topIndex] = number
     end
   end
   def pop … end
   def top … end
   def empty … end
end
```

Parameterized Abstract Data Types

- Parameterized ADTs
  - allow designing an ADT that can store any **type** elements
  - also known as generic classes
- C++, Java 5.0, and C# 2005 support for parameterized ADTs

Parameterized ADTs: C++

- Classes:
  - can be somewhat generic by writing parameterized constructor functions

```
Stack (int size) {
  stk_ptr = new int [size];
  max_len = size - 1;
  top = -1;
};
```

A declaration of a stack object:

```
Stack stk(150);
```

Parameterized ADTs: C++

- The stack element type can be parameterized by making the class a template class

```
template <class Type>
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack() {  // Constructor for 100 elements
      stackPtr = new Type[100];
      maxLen = 99;
      topPtr = -1;
    }
    Stack(int size) {  // Constructor for a given number
      stackPtr = new Type[size];
      maxLen = size – 1;
      topSub = -1;
    }
  . . .
}
```

  - Instantiation: `Stack<int> myIntStack;`

Parameterized Classes: Java

- Generic parameters must be classes
- Most common generic types are the collection types, such as ***`LinkedList`*** and ***`ArrayList`***
- Users can define generic classes
- *Generic collection classes* cannot store primitives data types
- Indexing is not supported
- Example of the use of a predefined generic class:

```
ArrayList <Integer> myArray = new ArrayList <Integer> ();
myArray.add(0);  // Put an element
```

Parameterized Classes: Java

```
import java.util.*;
public class Stack2<T> {
  private ArrayList<T> stackRef;
  private int maxLen;
  public Stack2() {
    stackRef = new ArrayList<T> ();
    maxLen = 99;
  }
  public void push(T newValue) {
    if (stackRef.size() == maxLen)
      System.out.println(" Error in push - stack is full");
    else
      stackRef.add(newValue);
  ...
}
```

- Instantiation: `Stack2<String> myStack = new Stack2<String> ();`

Parameterized Classes: C#

- Similar to Java, except no wildcard classes
- Predefined for Array, List, Stack, Queue, and Dictionary
- Elements of parameterized structures can be accessed through indexing
  - Ref: https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx

```
e.g., class List<T> { … }
        List<String> stringList = new List<String>();
```

Encapsulation Constructs

- Large programs have two special needs:
  - organization, other than simply division into subprograms
  - partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
- Such collections are called *encapsulation*

Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a *header file*
- `#include` preprocessor specification – used to include header files in applications

Encapsulation in C++

- Can define header and code files, similar to those of C
- Or, classes can be used for encapsulation
  - The class is used as the interface
  - The member definitions are defined in a separate file
- *Friends* provide a way to grant access to private members of a class

Sample C++ Code

```
// header: test.h

#ifndef _test_h_
#define _test_h_
class test {
	private:
		int t;
	public:
		int getT();
		void setT(int);
};
#endif
```

```
// implementation: test.cc

#include ”test.h"
void test::setT(int y) {
	t=y;
};

int test::getT() {
	return t;
};
```

```
// main.cc

#include <iostream.h>
#include ”test.h"

int main() {
	test aTest = test();
	aTest.setT(3);
	cout << aTest.getT()
      << "\n";
};
```

C# Assemblies

- A collection of files that appears to application programs to be a single dynamic link library (DLL) or executable
- Each file contains a module that can be separately compiled
- A DLL is a collection of classes and methods that are individually linked to an executing program
- C# has an access modifier called `internal`; an `internal` member of a class is visible to all classes in the assembly in which it appears

Naming Encapsulations

- Large programs define many global names; need a way to divide into logical groupings
- A *naming encapsulation* is used to create a new scope for names
- C++ Namespaces
  - place each library in its own namespace and qualify names used outside with the namespace
  - C# also includes namespaces

Naming Encapsulations

- Java Packages
  - Packages can contain more than one class definitions
  - Classes in a package are *partial* friends
  - Clients of a package can use *fully qualified name* or use the ***`import`*** declaration

Naming Encapsulations

- Ruby classes are name encapsulations, but Ruby also has modules.
- Ruby Modules:
  - encapsulate collections of constants & methods
  - cannot be instantiated or subclassed
  - cannot define variables
  - methods defined in a module must include the module's name
  - access to the contents of a module is requested with the `require` method to load the module.