Topics

- Parameterized Abstract Data Types
- Encapsulation Constructs
- Naming Encapsulations

Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements – only an issue for static typed languages
- Also known as generic classes
- C++, Java 5.0, and C# 2005 provide support for parameterized ADTs

Parameterized ADTs in 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 in C++ (continued)

- The stack element type can be parameterized by making the class a templated 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 in Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as `LinkedList` and `ArrayList`
- Eliminate the need to cast objects retrieved
- Eliminate the problem of having multiple types in a structure
- Users can define generic classes
- Generic collection classes cannot store primitives
- Indexing is not supported
- Example of the use of a predefined generic class:

```
ArrayList <Integer> myArray = new ArrayList <Integer> ();
myArray.add(0, 47);  // Put an element with subscript 0 in it
```

Parameterized Classes in Java 5.0 (continued)

```
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 in C# 2005

- Similar to those of Java 5.0, except no wildcard classes
- Predefined for Array, List, Stack, Queue, and Dictionary
- Elements of parameterized structures can be accessed through indexing

Encapsulation Constructs

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of 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*

Nested Subprograms

- Organizing programs by nesting subprogram definitions inside the logically larger subprograms that use them
- Nested subprograms are supported in Python, JavaScript, and Ruby

Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a *header file*
- Problem 1: the linker does not check types between a header and associated implementation
- Problem 2: the inherent problems with pointers
- `#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 (prototypes)
  - The member definitions are defined in a separate file
- *Friend*s provide a way to grant access to private members of a class

C# Assemblies

- A collection of files that appears to application programs to be a single dynamic link library 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
  - Can place each library in its own namespace and qualify names used outside with the namespace
  - C# also includes namespaces
- If you don't specify `using namespace std;` in C++, you must use `std::cout`

Naming Encapsulations (continued)

- Java Packages
  - Packages can contain more than one class definition; classes in a package are *partial* friends
  - Clients of a package can use fully qualified name or use the ***import*** declaration
  - If you don't specify `import java.util.*;` or import `java.util.ArrayList;` in Java, you must use `java.util.ArrayList`

Naming Encapsulations (continued)

- *Ruby Modules:*
  - Ruby classes are name encapsulations, but Ruby also has modules
  - Typically encapsulate collections of constants and methods
  - Modules cannot be instantiated or subclassed, and they 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

Ruby Module Example

```
module MyStuff
  PI = 3.14159265
  def MyStuff.mymethod1(p1)
  . . .
  end
  def MyStuff.mymethod2(p2)
  . . .
  end
end
```

`MyStuff` is defined in a file named `myStuffMod`

```
require 'myStuffMod'
. . .
MyStuff.mymethod1(x)
. . .
```