

# CONCEPTS OF PROGRAMMING LANGUAGES

## Chapter 11

### Abstract Data Types and Encapsulation Concepts



ROBERT W. SEBESTA

12/E

ISBN 0-321-49362-1

# Chapter 1 1 Topics

---

- 추상화의 개념
- 데이터 추상화의 소개
- 추상 데이터 타입의 설계 고려사항
- 언어 예제
- 매개변수 추상 데이터 타입
- 캡슐화 구조
- 캡슐화 명칭

# 추상화의 개념

---

- 추상화(*abstraction*)는 중요한 속성만을 포함하는 개체의 표현이다.
- 추상화의 개념은 프로그래밍에서 기본이다.
- 대부분의 프로그래밍언어는 부프로그램으로 프로세스 추상화를 지원
- 1980년대 이래 설계된 대부분의 언어는 데이터 추상화 지원

# 데이터 추상화 소개

---

- 추상 데이터 타입(*abstract data type*)은 다음 두 조건을 만족시키는 사용자-정의 데이터 타입:
  1. **정보 은폐** : 타입의 객체 표현은 그 타입을 사용하는 프로그램에게 은폐. 따라서 타입의 정의에서 제공되는 연산만이 수행 가능함
  2. 타입의 선언과 타입의 객체에 수행되는 연산은 **단일 구문 단위에 포함됨**

# 데이터 추상화의 장점

---

- 첫 조건의 장점

- 신뢰성—데이터 표현의 은폐로 사용자 코드는 타입의 객체를 직접 참조할 수 없음; 사용자 코드에 영향을 주지 않고 변경가능
  - ※ 데이터 멤버를 접근 방법 → 접근자 매소드(getter, setter)
- 프로그래머가 알아야 하는 변수와 코드의 범위 감소
- 이름 충돌 감소

- 둘째 조건의 장점

- 프로그램의 논리적인 조직화
- 수정력에 도움(데이터 구조와 연관된 모든 것이 함께)
- 분리 컴파일

# ADT를 위한 언어 요구사항

---

- ADT 정의를 캡슐화 해주는 구문 단위
- 타입 이름과 부프로그램 헤더를 작성하는 방법
- 기본적인 연산은 언어 처리기로 구축됨

# 언어예제 : C++

---

- **C struct type**과 **Simula 67 class**에 기반
- 클래스는 캡슐화 장치
- 클래스는 타입
- 한 클래스의 모든 사례들은 멤버 함수의 단일 카피를 공유한다.
- 한 클래스의 각 사례는 클래스 데이터 멤버들 고유의 복사본을 갖는다.
- 사례들은 정적, 스택 동적, 혹은 힙 동적임

# 언어예제 : C++ (continued)

---

- 정보 은폐
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance (Chapter 12)



# 언어예제 : C++ (continued)

---

- 생성자(Constructor):
  - 사례들의 데이터 멤버들을 초기화하는 함수 (객체를 생성하는 것은 아님)
  - 만약 객체의 일부가 힙-동적이면 기억공간 할당할 수 있음
  - 객체의 매개변수화를 제공하기 위한 매개변수를 포함할 수 있음
  - 사례가 생성될 때 묵시적으로 호출
  - 명시적으로 호출 가능
  - 이름은 클래스 이름과 동일

# 언어예제 : C++ (continued)

---

- 소멸자(Destructor)

- 사례가 소멸된 이후 클린업을 위한 함수; 보통 힙 공간 해제를 위한 용도
- 객체의 존속기간 종료 시 묵시적으로 호출
- 명시적으로 호출가능
- 이름은 클래스 이름 앞에 **tilde(~)**를 선행

# 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 number) {
            if (topSub == maxLen)
                cerr << "Error in push - stack is full\n";
            else stackPtr[++topSub] = number;
        };
        void pop () {...};
        int top () {...};
        int empty () {...};
}
```

# A Stack class header file

---

```
// Stack.h - the header file for the Stack class
#include <iostream>

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>
#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;
}
...
```

# 언어예제 : C++ (continued)

---

- 프렌드 함수 혹은 클래스 - 연관이 없는 단위나 함수에게 **private** 멤버 참조를 제공
  - Necessary in C++

# 언어예제 : Java

---

- Similar to C++, except:
  - 모든 사용자-정의 타입은 클래스
  - 모든 객체는 힙에 할당되고 참조 변수를 통해 접근
  - 클래스에 있는 개체들은 각기 접근수정자 (**private or public**)를 가짐
  - 묵시적인 **garbage collection**
  - Java는 **package scope**를 가짐
    - 사례변수나 매소드가 접근수정자를 갖지 않으면 패키지 전체에 걸쳐 가시적임

# An Example in Java

---

```
class StackClass {  
    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 () {...};  
}
```



# 언어예제 : C#

---

- C++ 과 Java에 기반
- 두개의 접근 수정자, *internal* 과 *protected internal* 추가됨
- 모든 클래스 사례는 힙 동적임
- 디폴트 생성자는 모든 클래스에 미리 정의됨
- 대부분 힙 객체를 가비지 수집하므로, 소멸자는 거의 사용되지 않음
- **Struct** 는 상속을 지원하지 않는다는 점에서 클래스와 다르다

# 언어 예제: C# (continued)

---

- 접근자 매소드 사용: **getter** 와 **setter**
- C# 은 명시적인 매소드 호출을 요구하지 않고 **getter**와 **setter**를 구현하기 위한 방법인 *properties*를 제공

# 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;
```

# 매개변수 ADT

---

- 매개변수 **ADT**는 어떤 타입의 요소들도 저장가능한 **ADT**를 설계할 수 있게 해줌-오직 정적 타입의 언어에만 해당
- 포괄 클래스(**generic class**)로 알려짐
- **C++**, **Java 5.0**, **C# 2005**은 매개변수 **ADT** 지원

# 매개변수 ADT : C++

---

- 스택 크기에서 포괄형 클래스로 만들기 위해서는 단지 생성자 함수를 다음과 같이 작성함

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

- 스택 객체의 선언:

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

# 매개변수 ADT : C++ (continued)

- 스택 원소타입을 템플릿 타입으로 만듦으로써 포괄형이 될 수 있음

```
template <typename 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;
        }
        ...
}
```

- 사례화: `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 that are removed
- 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> ();`



# 캡슐화 구조

---

- 대형 프로그램이 갖는 실질적인 문제점:
  - 단순히 부프로그램으로 분할하는 것 이상의 조직화의 문제
  - 프로그램 수정 후 재컴파일의 수단(전체 프로그램보다 더 작은 컴파일 단위)
- 명백한 해법: 논리적으로 연관된 부프로그램을 분리컴파일 단위로 그룹핑하는 것
- 그러한 집합을 캡슐화(*encapsulation*)라 함

# C의 캡슐화

---

- 1개 이상의 부프로그램을 포함하는 파일(library)들은 독립적으로 컴파일 가능
- 인터페이스는 *header file*에 위치
- 문제점 1: 링커(linker)는 헤더와 관련 구현함수들 간에 타입을 체크하지 않는다.
- 문제점 2: 포인터가 가진 고유의 문제
- `#include` 전처리기 명세는 헤더파일을 응용프로그램에 포함하기 위해 사용됨

# C++의 캡슐화

---

- C와 유사하게, 헤더와 코드 파일을 정의
- 혹은 클래스가 캡슐화를 위해 사용될 수 있음
  - 클래스는 인터페이스(프로토타입)로 사용가능
  - 멤버 정의는 별도 파일에 정의됨
- *Friend*는 클래스의 전용 멤버에 접근을 허용하는 방법을 제공

# 캡슐화 명칭 (*naming encapsulation*)

---

- 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

# 캡슐화 명칭(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

# Summary

---

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- C++, Java 5.0, and C# 2005 support parameterized ADTs
- C++, C#, Java, and Ruby provide naming encapsulations