Chapter 11
Abstract Data Types
The Concept of Abstraction

- The concept of abstraction is fundamental in programming

- Nearly all programming languages support process abstraction with subprograms

- Nearly all programming languages designed since 1980 have supported data abstraction with some kind of module
Encapsulation

- **Original motivation:**

- Large programs have two special needs:
  1. Some means of organization, other than simply division into subprograms
  2. 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)

- These are called *encapsulations*
Encapsulation (Continued)

- Examples of Encapsulation Mechanisms

1. Nested subprograms in some ALGOL-like languages (e.g., Pascal, JavaScript, etc.)

2. FORTRAN 77 and C - Files containing one or more subprograms can be independently compiled
Introduction to Data Abstraction

- Definition: An *abstract data type* is a user-defined data type that satisfies the following two conditions:

1. The representation of and operations on objects of the type are defined in a single syntactic unit; also, other units can create objects of the type.

2. 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.
- **Advantage of Restriction 1:**
  
  - Same as those for encapsulation: program organization, modifiability (everything associated with a data structure is together), and separate compilation

- **Advantage of Restriction 2:**

  - **Reliability**—by hiding the data representations, user code cannot directly access objects of the type. User code cannot depend on the representation, allowing the representation to be changed without affecting user code.
Introduction to Data Abstraction (Continued)

- **Built-in types are abstract data types**

  For example the int type in most languages (e.g. Java)
  - The representation is hidden
  - Operations are all built-in
  - User programs can define objects of int type

- User-defined abstract data types must have the same characteristics as built-in abstract data types

- This distinction of thinking of primitives as ADTs seems funny, but consider does the CPU care what is in a memory location?
  - Does type matter to it?
  - Is there anything other than numeric or plain Boolean type in hardware?
Design Issues

- **Language Requirements to support abstract data types:**

1. A syntactic unit in which to encapsulate the type definition.

2. A method of making type names and subprogram headers visible to clients, while hiding actual definitions.

3. Some primitive operations must be built into the language processor (usually just assignment and comparisons for equality and inequality)
   - Some operations are commonly needed, but must be defined by the type designer
     - e.g., iterators, constructors, destructors
Design Issues (Continued)

- Language Design Issues:

1. Encapsulate a single type, or something more?
2. What types can be abstract?
3. Can abstract types be parameterized?
4. What access controls are provided?
Language Example C++

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

- **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 Example 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 tilda (~)
Language Example Java

A Related Language: 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