ABSTRACT DATA TYPES

The Concept of Abstraction
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 while those designed since 1980 also support *data abstraction*

Introduction to Data Abstraction
An *abstract data type* is a user-defined data type that satisfies the following two conditions:
i). The representation of, and operations on, objects of the type are defined in a single syntactic unit
ii). 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
Generally, the concept of ADTs and their use in program design was a milestone in the development of languages

Advantages of Data Abstraction
i). The advantage of the first condition is that program organization, modifiability (everything associated with a data structure is together), and separate compilation
ii). The advantage the second condition is 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
i). Can abstract types be parameterized?
ii). What access controls are provided?
iii). What is the form of the container for the interface to the type?

Language Examples: C++
- C++ is 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
- 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 although they can be explicitly called
  - Name is the same as the class name
- 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 (~)

Example of code 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 () {…};
}
```

Evaluation of ADTs in C++

- C++ has strong support for ADTs
- It provides effective mechanisms for encapsulation and information hiding
- Classes are types
- Friend functions or classes are used to provide access to private members to some unrelated units or functions and are necessary in C++

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

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 () {…};
}
```

Language Examples: C#

- Based on C++ and Java
- Adds two access modifiers, *internal* and *protected internal*
- 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
- In most common solutions there is need for access to data members through accessor methods (getter and setter), however, 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;
   ...
}
public class TEstWeather {
   static void Main(){
      ...
      Weather w = new Weather();
      int degreeDaysToday, oldDegreeDays;
      ...
      w.DegreeDays = degreeDaysToday;
      ...
      oldDegreeDays = w.DegreeDays;
      ...
   }
}
```

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 while Limited private types have NO built-in operations
- 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)
- Having part of the implementation details (the representation) in the spec package and part (the method bodies) in the body package is not good *One solution: make all ADTs pointers*

Problems:

i). Difficulties with pointers
ii). Object comparisons

iii). Control of object allocation is lost

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
```

Parameterized Abstract Data Types
Parameterized ADTs allow designing an ADT that can store any type elements (among other things). They are also known as generic classes

Parameterized ADTs in C++
Classes can be somewhat generic by writing parameterized constructor functions. E.g. the stack element type can be parameterized by making the class a templated class as shown below

```
template <class Type>
class stack {
     private:
     Type *stackPtr;
     const int maxLen;
     int topPtr;
     public:
     stack() {
          stackPtr = new Type[100];
          maxLen = 99;
          topPtr = -1;
     }
     …
}
```

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

Parameterized Classes in C# 2005
- Similar to those of Java 5.0
- Elements of parameterized structures can be accessed through indexing

Object Oriented Programming

Inheritance is the central theme in OOP and languages that support it and productivity increases can come from code reuse

- ADTs are difficult to reuse—always need changes
- All ADTs are independent and at the same level

Generally inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts and it is able to address the above two concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are usually called classes and class instances are called objects
- A class that inherits is a derived class or a subclass and the class from which another class inherits is a parent class or superclass
- Subprograms that define operations on objects are called *methods* and calls to methods are called messages
- The entire collection of methods of an object is called its message protocol or message interface
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent, however, inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses
  - A class can hide entities from its clients
  - A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one *overrides* the inherited one
  - The method in the parent is *overridden*
- There are two kinds of variables in a class:
  - *Class variables* – one per class
  - *Instance variables* – one per object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- One disadvantage of inheritance for reuse is that it creates interdependencies among classes that complicate maintenance

Polymorphism in OOPLs (Dynamic Binding)

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated

Design Issues for OOP Languages

i). The Exclusivity of Objects
ii). Are Subclasses Subtypes
iii). Implementation and Interface Inheritance
iv). Type Checking and Polymorphism
v). Single and Multiple Inheritance
vi). Object Allocation and DeAllocation
vii). Dynamic and Static Binding
viii). Nested Classes

1) **The Exclusivity of Objects**
   - Everything is an object
     - Advantage - elegance and purity
     - Disadvantage - slow operations on simple objects
   - Add objects to a complete typing system
     - Advantage - fast operations on simple objects
     - Disadvantage - results in a confusing type system (two kinds of entities)
   - Include an imperative-style typing system for primitives but make everything else objects
     - Advantage - fast operations on simple objects and a relatively small typing system
     - Disadvantage - still some confusion because of the two type systems

2) **Are Subclasses Subtypes?**
   - Does an "is-a" relationship hold between a parent class object and an object of the subclass?
     - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
   - A derived class is a subtype if it has an is-a relationship with its parent class
     - Subclass can only add variables and methods and override inherited methods in "compatible" ways

3) **Implementation and Interface Inheritance**
   - If only the interface of the parent class is visible to the subclass, it is interface inheritance
     - Disadvantage - can result in inefficiencies
   - If both the interface and the implementation of the parent class is visible to the subclass, it is implementation inheritance
     - Disadvantage - changes to the parent class require recompilation of subclasses, and sometimes even modification of subclasses

4) **Type Checking and Polymorphism**
   - Polymorphism may require dynamic type checking of parameters and the return value
     - Dynamic type checking is costly and delays error detection
   - If overriding methods are restricted to having the same parameter types and return type, the checking can be static

5) **Single and Multiple Inheritance**
   - Multiple inheritance allows a new class to inherit from two or more classes
   - Disadvantages of multiple inheritance:
     - Language and implementation complexity (in part due to name collisions)
     - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
     - Advantage: Sometimes it is quite convenient and valuable

6) **Allocation and DeAllocation of Objects**
   - From where are objects allocated?
     - If they behave line the ADTs, they can be allocated from anywhere
   - Allocated from the run-time stack
   - Explicitly create on the heap (via new)
     - If they are all heap-dynamic, references can be uniform through a pointer or reference variable
   - Simplifies assignment - dereferencing can be implicit
     - If objects are stack dynamic, there is a problem with regard to subtypes
   - Is deallocation explicit or implicit?

7) **Dynamic and Static Binding**
   - Should all binding of messages to methods be dynamic?
     - If none are, you lose the advantages of dynamic binding
     - If all are, it is inefficient
   - Allow the user to specify

8) Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

Support for OOP in Smalltalk

- Smalltalk is a pure OOP language
  - Everything is an object
  - All objects have local memory
  - All computation is through objects sending messages to objects
  - None of the appearances of imperative languages
  - All objected are allocated from the heap
  - All deallocation is implicit
- Type Checking and Polymorphism
  - All binding of messages to methods is dynamic
- The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc. up to the system class which has no superclass
  - The only type checking in Smalltalk is dynamic and the only type error occurs when a message is sent to an object that has no matching method
- Inheritance
  - A Smalltalk subclass inherits all of the instance variables, instance methods, and class methods of its superclass
  - All subclasses are subtypes (nothing can be hidden)
  - All inheritance is implementation inheritance and does not allow multiple inheritance
- Evaluation of Smalltalk
  - The syntax of the language is simple and regular
  - Good example of power provided by a small language
  - Slow compared with conventional compiled imperative languages
  - Dynamic binding allows type errors to go undetected until run time
  - Introduced the graphical user interface
  - It has the greatest impact of advancement of OOP

Support for OOP in C++

- The following are the general characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities
- Inheritance
  - A class need not be the subclass of any class
  - Access controls for members are
    - Private (visible only in the class and friends) (disallows subclasses from being subtypes)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)
- In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses
  - Private derivation - inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses

Inheritance Example in C++

```
class base_class {
     private:
          int a;
          float x;
     protected:
          int b;
          float y;
     public:
          int c;
          float z;
};

class subclass_1 : public base_class { … };
// b and y are protected and c and z are public
class subclass_2 : private base_class { … };
// b, y, c, and z are private, and no derived class has access to
//any member of base_class
```

Re exportation in C++

- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (**::**), e.g.,

```
class subclass_3 : private base_class {
     base_class :: c;
     …
}
```

- One motivation for using private derivation
  - A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition
- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator
- Dynamic Binding
  - A method can be defined to be virtual, which means that they can be called through polymorphic variables and dynamically bound to messages
  - A pure virtual function has no definition at all
  - A class that has at least one pure virtual function is an *abstract class*
- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
- Static binding is faster
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Support for OOP in Java

- Generally Java has a close relationship to C++ and the differences from that C++ language will be discussed
- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value

  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with new
  - A finalize method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object
- Inheritance
  - Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (interface)
  - An interface can include only method declarations and named constants, e.g.,

```
public class Clock extends Applet
          implements Runnable
```

  - Methods can be **final** (cannot be overriden)
- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is final (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the methods is static or private both of which disallow overriding Several varieties of nested classes and nested classes can be anonymous. A local nested class is defined in a method of its nesting class
- Encapsulation
  - Two constructs, classes and packages
  - Packages provide a container for classes that are related
  - Entities defined without an scope (access) modifier have package scope, which makes them visible throughout the package in which they are defined
  - Every class in a package is a friend to the package scope entities elsewhere in the package
- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as ―normal‖ way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

Support for OOP in C#

- The following are the general characteristics
  - Support for OOP similar to Java
  - Includes both classes and structs
  - Classes are similar to Java‘s classes
  - structs are less powerful stack-dynamic constructs (e.g., no inheritance)
- Inheritance
  - Uses the syntax of C++ for defining classes
  - A method inherited from parent class can be replaced in the derived class by marking its definition with new
  - The parent class version can still be called explicitly with the prefix base: `base.Draw()`
- Dynamic binding
  - To allow dynamic binding of method calls to methods:
- The base class method is marked virtual
- The corresponding methods in derived classes are marked override
  - Abstract methods are marked abstract and must be implemented in all subclasses
  - All C# classes are ultimately derived from a single root class, Object
- Nested Classes
  - A C# class that is directly nested in a nesting class behaves like a Java static nested class
  - C# does not support nested classes that behave like the non-static classes of Java
- Evaluation
  - C# is the most recently designed C-based OO language
  - The differences between C#‘s and Java‘s support for OOP are relatively minor