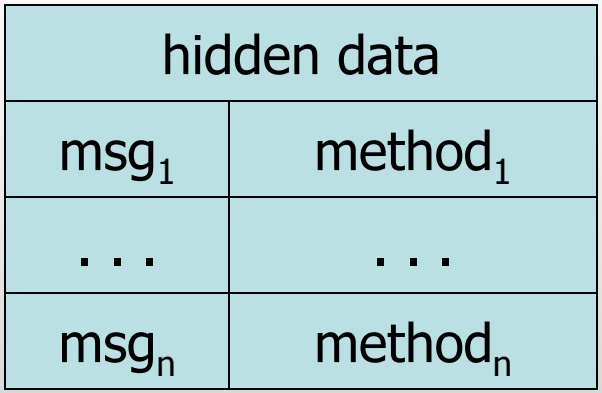
Objects

- An object consists of
  - hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - public operations
    - methods or member functions
    - can also have public variables in some languages
- Object-oriented program:
  - Send messages to objects



Object-oriented programming

- Programming methodology
  - organize concepts into objects and classes
  - build extensible systems
- Language concepts
  - encapsulate data and functions into objects
  - inheritance allows reuse of implementation
  - Polymorphism

Object-Oriented languages

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., Ada and C++)
  - Some support functional program (e.g., CLOS)
  - Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
  - Some are pure OOP language (e.g., Smalltalk)

The Concept of Abstraction

- An *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
- Nearly all programming languages designed since 1980 support *data abstraction*

Introduction to Data Abstraction

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit
  - 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

Advantages of Data Abstraction

- Advantage of the first condition
  - Program organization, modifiability (everything associated with a data structure is together), and separate compilation
- Advantage the second condition
  - 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

Design Issues

- A syntactic unit to define an ADT
- Built-in operations
  - Assignment
  - Comparison
- Common operations
  - Iterators
  - Accessors
  - Constructors
  - Destructors
- Parameterized ADTs

Language Examples: 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

Language Examples: C++ (continued)

- 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 Examples: 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 tilde (~)

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 num) {…};
      void pop () {…};
      int top () {…};
      int empty () {…};
}
```

Evaluation of ADTs in C++ and Ada

- C++ support for ADTs is similar to expressive power of Ada
- Both provide effective mechanisms for encapsulation and information hiding
- Ada packages are more general encapsulations

Language Examples: C++ (continued)

- Friend functions or classes – to provide access to private members to some unrelated units or functions
  - 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

An Example in Java

```
class StackClass {
   private:
        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
- `struct`s are lightweight classes that do not support inheritance

Language Examples: C# (continued)

- Common solution to need for access 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 {degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements
- Also known as generic classes
- C++ and Ada provide support for parameterized ADTs
- Java 5.0 provides a restricted form of parameterized ADTs
- C# does not currently support parameterized classes

Parameterized ADTs in C++

- Classes can be somewhat generic by writing parameterized constructor functions

```
template <class type>
class stack {
…
 stack (int size) {
 stk_ptr = new int [size];
 max_len = size - 1;
  top = -1;
 };
 …
}

 stack stk(100);
```

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 Ada and Fortran 95

Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a *header file*
- Problem: the linker does not check types between a header and associated implementation
- `#include` preprocessor specification

Encapsulation in C++

- Similar to C
- Addition of *friend* functions that have access to private members of the friend class

Ada Packages

- Ada specification packages can include any number of data and subprogram declarations
- Ada packages can be compiled separately
- A package's specification and body parts can be compiled separately

C# Assemblies

- A collection of files that appear 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

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
- Ada Packages
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the `with` clause

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
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- Ada and C++ allow parameterized ADTs
- C++, C#, Java, and Ada provide naming encapsulation

Object-Oriented Programming

- Abstract data types
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- Polymorphism

Inheritance

- Productivity increases can come from reuse
  - ADTs are difficult to reuse
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are called *classes*
- Class instances are called objects
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*

Object-Oriented Concepts (continued)

- 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

Object-Oriented Concepts (continued)

- 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 *overriden*

Object-Oriented Concepts (continued)

- There are two kinds of variables in a class:
  - *Class variables* – one/class
  - *Instance variables* – one/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- Single vs. Multiple Inheritance
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance

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

- The Exclusivity of Objects
- Subclasses as Types
- Type Checking and Polymorphism
- Single and Multiple Inheritance
- Object Allocation and De-Allocation
- Dynamic and Static Binding
- Nested Classes

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

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

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

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 extremely convenient and valuable

Allocation and De-Allocation 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 thru 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?

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

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 C++

- General Characteristics:
  - Evolved from SIMULA 67
  - Most widely used OOP language
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities

Support for OOP in C++ (continued)

- 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)

Support for OOP in C++ (continued)

- 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 { ... };
//     In this one, b and y are protected and
//     c and z are public

class subclass_2 : private base_class { ... };
//    In this one, b, y, c, and z are private,
//    and no derived class has access to any
//    member of base_class
```

Reexportation 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;
        ...
}
```

Reexportation (continued)

- 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

Support for OOP in C++ (continued)

- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator

Support for OOP in C++ (continued)

- 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*

Support for OOP in C++ (continued)

- 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

- Because of its close relationship to C++, focus is on the differences from that language
- 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

Support for OOP in Java (continued)

- 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 interface Comparable {
        public int comparedTo (Object b);
}
```

  - Methods can be **`final`** (cannot be overriden)

Support for OOP in Java (continued)

- 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

Support for OOP in Java (continued)

- Several varieties of nested classes
- All can be hidden from all classes in their package, except for the nesting class
- Nested classes can be anonymous
- A local nested class is defined in a method of its nesting class
  - No access specifier is used

Support for OOP in Java (continued)

- 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#

- 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

Support for OOP in C# (continued)

- 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()
```

Support for OOP in C#

- 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`

Support for OOP in C# (continued)

- 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

Support for OOP in C#

- 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

Support for OOP in Ada 95 (continued)

- Dynamic Binding
  - Dynamic binding is done using polymorphic variables called classwide types
    - For the tagged type **`PERSON`**, the classwide type is **`PERSON'class`**
  - Other bindings are static
  - Any method may be dynamically bound
  - Purely abstract base types can be defined in Ada 95 by including the reserved word `abstract`

The Object Model of JavaScript

- General Characteristics of JavaScript
  - Little in common with Java
    - Similar to Java only in that it uses a similar syntax
  - Dynamic typing
  - No classes or inheritance or polymorphism
  - Variables can reference objects or can directly access primitive values

The Object Model of JavaScript

- JavaScript objects
  - An object has a collection of properties which are either data properties or method properties
  - Appear as hashes, both internally and externally
  - A list of property/value pairs
  - Properties can be added or deleted dynamically
  - A bare object can be created with new and a call to the constructor for Object

```
var my_object = new Object();
```

  - References to properties are with dot notation

JavaScript Evaluation

- Effective at what it is designed to be
  - A scripting language
- Inadequate for large scale development
- No encapsulation capability of classes
  - Large programs cannot be effectively organized
- No inheritance
  - Reuse will be very difficult

Implementing OO Constructs

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

Instance Data Storage

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient

Dynamic Binding of Methods Calls

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

Summary

- OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
- Major design issues: exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit de-allocation of objects, and nested classes
- Smalltalk is a pure OOL
- C++ has two distinct type system (hybrid)
- Java is not a hybrid language like C++; it supports only OO programming
- C# is based on C++ and Java
- JavaScript is not an OOP language but provides interesting variations