
■ Chapter 10

Object-Oriented Programming

Introduction

- Classification of programming languages according to its support of Object-Orientation:
 - ❑ Programming languages that support procedural and data-oriented, in addition to object-oriented programming. **C++** and **Ada95** are examples of such programming languages.
 - ❑ Programming languages that were designed to support object-oriented programming and do not support other programming paradigms, but still employ some of the basic imperative structures. **Java** and **C#** are among such languages.
 - ❑ Programming languages that offer complete support for object-oriented programming. **Smalltalk** is the only pure object-oriented language.

Object-Oriented Programming

- The roots of the Object-Oriented programming concept refer to **SIMULA67**. However, the Object-Oriented concept was not fully developed until the evolution of **Smalltalk 80**.
 - An Object-Oriented language must support three language features: abstract data types, inheritance, and dynamic binding
-

Inheritance

- One of the most important opportunities for increasing software development productivity is the concept of software reuse. Software reuse is greatly facilitated with using **inheritance**, by which a new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify some of those entities and add new entities.

Inheritance Terminology

- ❑ **Class:** the abstract data type in Object-Oriented languages.
- ❑ **Object:** A class instance.
- ❑ **Derived class (subclass):** A class that is defined through inheritance from another class.
- ❑ **Parent class (superclass):** A class from which a new class is derived.
- ❑ **Methods:** the subprograms that define the operations on objects of a class.
- ❑ **Messages:** the calls to methods. Messages have two parts--a method name and the destination object.
- ❑ **Message Protocol (Message Interface):** the entire collection of an object's methods.

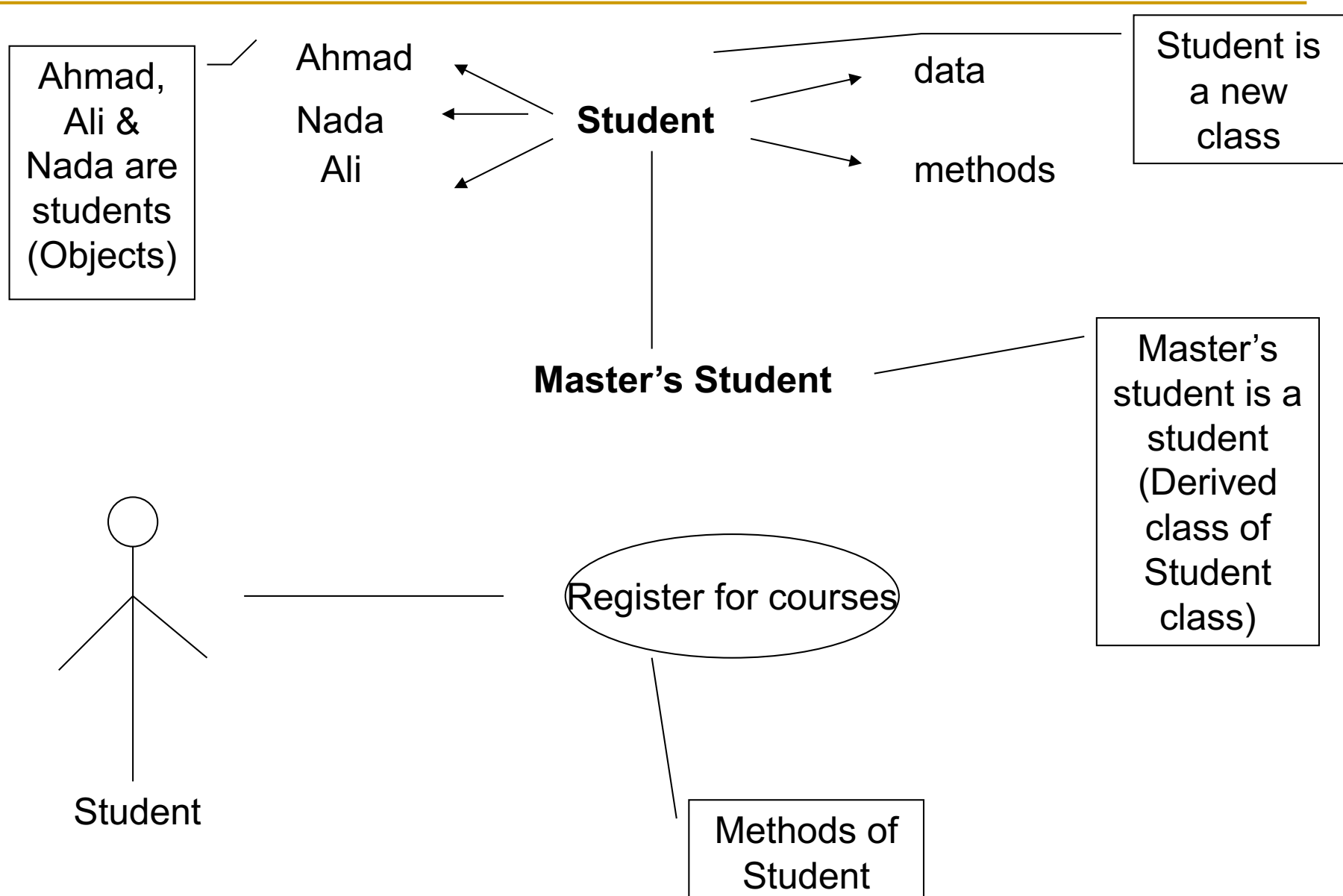
Inheritance Terminology

Classes can have two kinds of methods:

- **Instance Methods:** operate only on the objects of the class
- **Class Methods:** can perform operations on the class, and possibly on the objects of the class

Classes can also have two kinds of variables:

- **Instance Variables:** every object of a class has its own set of instance variables, which store the object's state.
 - **Class Variables:** belong to the class, so there is only one copy for the class.
-

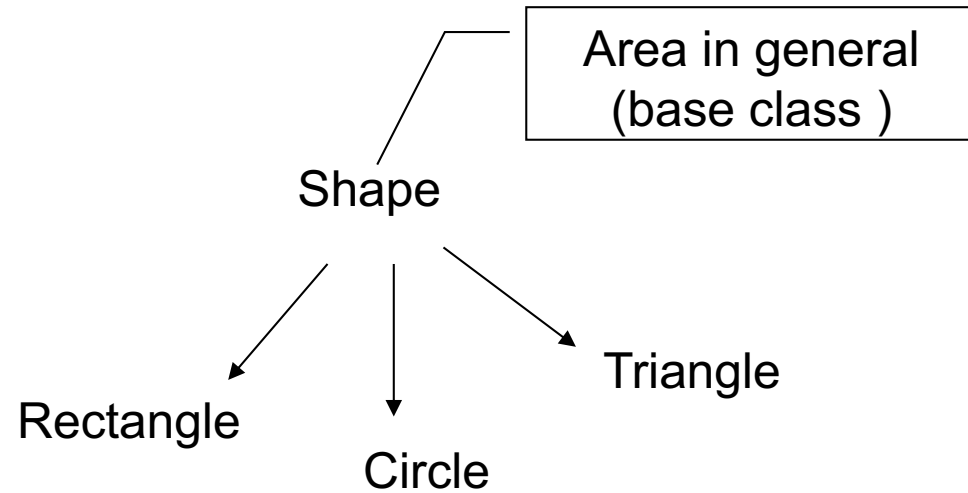


Inheritance Types

- **Single Inheritance:** if a new class is a subclass of a single parent class. Relationships among classes in this type of inheritance can be shown in a derivation tree.
 - **Multiple Inheritance:** if a new class has more than one parent class. Relationships among classes in this type of inheritance can be shown in a derivation graph.
-

Dynamic Binding

- In order to understand the concept of dynamic binding, let's consider the following situation: there are two classes; a base class A and a subclass B. Class A defines a method and class B overrides that method. If a client of A and B has a reference or pointer to A's objects. Then, that pointer or reference could also point at class B's objects, making it a polymorphic reference or pointer. If the method is called through the polymorphic reference or pointer, then it is the role of the system to determine during execution which method should be called.
-



Triangle, Circle and Rectangle (subclass) Each of them compute the Area with different way

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
-

Design Issues for Object-Oriented Languages – Exclusivity of Objects

- The exclusivity of objects refers to the exclusive use of objects in purest model of object-oriented computation, where all types are classes, and the classes are treated the same way, and all computation is performed through message passing.
 - For instance, in Smalltalk, adding 7 to the variable x is done by sending object 7 to the + method of the object x.
-

Design Issues for Object-Oriented Languages – Exclusivity of Objects

- The exclusivity of objects refers to the exclusive use of objects in purest model of object-oriented computation, where all types are classes, and the classes are treated the same way, and all computation is performed through message passing.
 - For instance, in Smalltalk, adding 7 to the variable x is done by sending object 7 to the + method of the object x.
-

The Exclusivity of Objects

- **Everything is an object**
 - ❑ Advantage - elegance and purity
 - ❑ Disadvantage - slow operations on simple objects
 - **Add objects to a complete imperative 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
-

Design Issues for OO Languages – Are Subclasses Subtypes?

- If an “is-a” relationship holds between a derived class and parent class, then this relationship guarantees that a variable of the derived class type could appear where a variable of the parent class type was legal, without causing a type error.
- Such a simple form of inheritance is apparent in Ada95 subtypes, as the following example clarifies:
Subtype Small_Int **is** Integer **range** -100...100
- The above example means that every Small-Int is an integer. In addition, Small_Int have all the operations of Integer variables, but can only store a subset of the possible values in Integer.

Design Issues - Type Checking and Polymorphism

- The following are the two kinds of type checking that must be performed between a message and a method:
 - The message's parameter type must be checked against the method's formal parameters, and the method's return type needs to be checked against the expected return type of the message.
 - An alternative is to delay type checking until the polymorphic variable is used to call a method. However, this alternative is expensive and delays type error detection.
-

Design Issues - Single and Multiple Inheritance

- The issue here is whether the language allows multiple inheritance in addition to single inheritance. The reasons for which the language designer does not include multiple inheritance even it is useful refer to the following:
 - ❑ Complexity: several problems arise here:
 - ❑ When the parent classes both define identically named methods and one or both of them need to be overridden in the subclass.
 - ❑ Diamond or Shared inheritance: this problem can be better illustrated with the following example: class C has both, A and B, as parent classes, where A and B are derived from the common class Z. The problem is: should C inherit which version of the variable inherited from Z?
 - ❑ Efficiency: the maintenance of systems that use multiple inheritance can be a serious problem because multiple inheritance leads to more complex dependencies among classes.
-

Design Issues - Allocation and Deallocation of Objects

- Two design questions arise regarding the allocation and deallocation of objects. These are:
 - The place from which objects are allocated. As it is known, objects could be allocated either from the run-time stack, or explicitly created on the heap with an operator or function, as **new**. Suppose that class B is a subclass of class A, and b1 the object of B needs to be assigned to an object a1 of A as:

$a1 = b1;$

- then, if a1 and b1 are references to heap-dynamic objects, there is no problem. However, if a1 and b1 are stack-dynamic, then the value of the object must be copied to the space of the target object. The problem arises when B adds a data field, then a1 will not have sufficient space for all of b1, resulting in a truncation for the excess in b1.
-

Design Issues - Allocation and Deallocation of Objects

- For objects allocated from the heap, the question that arises is whether deallocation is implicit, explicit, or both. If deallocation is implicit, then some implicit storage reclamation is required, such as reference counters or garbage collection. Explicit deallocation, on the other hand, raises the issue of whether dangling pointers or references can be created.
-

Design Issues - Dynamic and Static Binding

- dynamic binding is an essential part of object-oriented programming. The issue is whether the user is allowed to specify if the binding is static or dynamic.
 - The advantage obtained from this is that static binding is faster. So, if the binding needs not to be dynamic, then why to pay its price?
-

Design Issues – Nested Classes

- If a new class is needed by only one class, then it can be nested inside that class, which is called the nesting class. Two main issues appear here: Which of the nesting class facilities are visible in the nested class? And the opposite: Which of the nested class facilities are visible in the nesting class?
-

Support for Object-Oriented Programming in Smalltalk

- Smalltalk was the first language to include complete support for the object-oriented programming paradigm.
 - **General Characteristics**
 - ❑ **Objects:** virtually everything, from a small integer constant to a complex file handling system, is an object.
 - ❑ **Messages:** can be parameterized with variables that reference objects. Replies to messages have also the form of object.
 - ❑ **Allocation:** all Smalltalk objects are allocated from the heap and are referenced through reference variables.
 - ❑ **Deallocation:** all deallocation is implicit using a garbage collection
-

Support for OOP in Smalltalk - Type Checking

- In Smalltalk, type checking is only dynamic, and the only type error occurs when a message is sent to an object that has no matching method. The goal of type checking in Smalltalk is to ensure that a message matches some method.
 - Smalltalk variables are not typed (i.e., any name can be bound to any object). The meaning of an operation on a variable is determined by the variable's class to which the variable is currently bound.
-

Support for OOP in Smalltalk - Inheritance

- A Smalltalk subclass can inherit its superclass, instance variables, instance and class methods. A subclass can also have its own instance variables. The subclass can, in addition, define new methods and redefine methods of an ancestor class. If a subclass has a method with a name and protocol similar to that of the ancestor class, the subclass hides the ancestor's method. In order to access the hidden method, the message is prefixed with the pseudovariable **super**.
 - Smalltalk supports single inheritance only.
-

Support for OOP in Smalltalk - Evaluation

- The syntax of Smalltalk is simple and very regular.
 - Smalltalk is built around a simple but powerful concept, which is that all programming can be done using only a class hierarchy that uses objects, inheritance, and message passing.
 - Compared to conventional compiled imperative language programs, equivalent Smalltalk programs are slower.
 - Dynamic binding in Smalltalk delays error detection until run time.
 - Smalltalk user interface has had its impact on computing, by being the first to use integrated windows, mouse-pointing devices, and pop-up or pull-down menus.
 - Smalltalk has the greatest impact on the advancement of object-oriented programming, which is the most common design and coding methodology.
-

Support for OOP in C++ - General Characteristics

- C++ is a **hybrid** language: it supports both; ***procedural*** and ***object-oriented*** programming.
- C++ **objects**: can be static, stack-dynamic, or heap dynamic.
- Explicit deallocation using the **delete** operator is required for heap dynamic objects.
- C++ **Constructor**: at least, one constructor is used in a C++ class to initialize the new object's data members. When an object is created, constructor methods are implicitly called. If a constructor is not defined, the compiler includes the default constructor, which calls *the parent class*' constructor.
- C++ **Destructor**: a destructor method is implicitly called when an object ceases to exist. The destructor main use is to delete heap-allocated data members. Also, it may be used to record part or all of the object's state for debugging purposes.

Support for OOP in C++ - Inheritance

- A C++ class can be derived from an existing class. C++ can also be, unlike Smalltalk, stand-alone without a parent class.
 - The derived class can inherit some or all of the base class' data members (data defined in a class) and member functions (functions defined in a class). In addition, the subclass can add new data members and member functions, and modify inherited data members and member functions.
-

Support for OOP in C++ - Inheritance

- The syntactic form of a derived class is as follows:

```
Class derived_class_name : access_mode  
    base_class_name  
    {data member and member function  
    declarations};
```

Support for OOP in C++ - Inheritance

```
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 , y are protected
c , z are public

- Class subclass_2 : private
base_class {...};

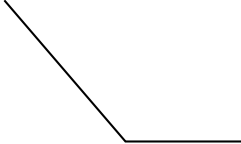
b , y ,c , z are
private

Support for OOP in C++ - Inheritance

Class A {...};

Class B {...};

Class c : public A , public B {...};



Class c inherits all of the members of both A and B.

If both A and B include the same name , they can be unambiguously .

Support for OOP in C++ - Dynamic Binding

- When a polymorphic variable is used to call a function in a derived class, the call must be dynamically bound to the correct function definition. Therefore, member functions that must be dynamically bounded are preceded with the keyword **virtual**.
-

Support for OOP in C++ - Dynamic Binding

```
public class shape {  
    public:  
        virtual void draw () = 0;  
        .....  
}  
public class circle : public shape {  
    public:  
        virtual void draw () {...};  
        .....  
}  
public class rectangle : public shape {  
    public:  
        virtual void draw () {...};  
        .....  
}  
public class square : public shape {  
    public:  
        virtual void draw () {...};  
        .....  
}
```

draw() = 0 used to indicate that this member function is *pure virtual function* : meaning that it has no body and cannot be called .

it must be redefined in derived classes. the purpose of a pure virtual function is to provide the interface of a function without giving any of its implementation . this allows every subclass to define its own version of the function .

any class include a pure virtual function is an *abstract class*. No object of an abstract class can be created .where the subclasses of such type can have objects.

Support for OOP in C++ - Evaluation

- Compared to Smalltalk:
 - ❑ C++ programmer has highly detailed control over member's access.
 - ❑ C++ allows multiple inheritance in addition to single inheritance.
 - ❑ C++ programmer can decide whether to use static or dynamic binding.
 - ❑ C++ provides static type checking.
 - ❑ C++ provides generic classes through its template facility.
 - ❑ C++ is a large and complex language.
 - ❑ Efficiency is a strong argument in favor of C++.
-

Support for Object-Oriented Programming in Java

General Characteristics

- ❑ Java does not use objects exclusivity
 - ❑ All Java classes must be subclasses of the root class, **Object**, or some class that is a descendent of Object.
 - ❑ All Java objects are explicit heap-dynamic. Most of objects are allocated with the **new** operator.
 - ❑ No explicit deallocation operator exists for Java; garbage collection is used.
-

Support for OOP in Java - Inheritance

- Directly, Java supports only single inheritance. However, it uses a kind of abstract class, called **interface**, which serves as a version of multiple inheritance.
-

Support for OOP in Java - Inheritance

- Directly, Java supports only single inheritance. However, it uses a kind of abstract class, called **interface**, which serves as a version of multiple inheritance.
 - A method in Java can be defined to be **final**, so that it can not be overridden in any descendent class. If this keyword specifies a class definition, then the class can not be the parent of any subclass.
-

Support for OOP in Java

Dynamic Binding

- ❑ All method calls in Java are dynamically bound unless the called method has been defined as **final**, where it can not be overridden and all bindings become static.

Nested Classes

- ❑ A nested class is defined in a method in a nesting class.
-

Support for OOP in Java - Evaluation

- Java does not support procedural programming.
 - Java does not allow parentless classes.
 - Dynamic binding is the normal way used to bind method calls to method definitions.
 - Java uses interface to support multiple inheritance.
-

Support for Object-Oriented in C#

General Characteristics

- C# includes both classes and structs.

Inheritance

- The following is C# syntax for defining classes:
public class NewClass : ParentClass {...}

Nested Classes

- C# supports static nested classes that are directly nested in a class.

Evaluation

- Since C# is based on its predecessors, from which designers learnt and improved, it is expected that some of the problems are remedied.
-

Support for Object-Oriented in C#

Dynamic Binding

In order to allow dynamic binding in C#, the base method must be marked with **virtual**, while the derived method is marked with **override**.

```
public class Shape {  
    public:  
        virtual void Draw () {...}  
        .....  
}  
public class Circle : Shape {  
    public override void Draw () {...};  
    .....  
}  
public class Rectangle : Shape {  
    public override void Draw () {...};  
    .....  
}  
public class Square : Shape {  
    public override void Draw () {...};  
    .....  
}
```

Support for Object-Oriented Programming in Ada95

General Characteristics

- ❑ Ada95 classes form a new type category, which can be records or private types. Ada95 classes are defined in packages which allows them to be separately compiled.
 - ❑ Each object of a tagged type implicitly includes system-maintained tag that indicates its type.
 - ❑ Constructors and destructors are not called implicitly. They can be written and called explicitly.
-

Support for Object-Oriented Programming in Ada95

Inheritance

- ❑ Derived types in Ada95 are based on tagged types. New entities to be added to inherited methods are placed in a record definition. Ada95 derived classes are subtypes.
 - ❑ Deriving a class that does not include all the parent class' entities, child library packages are used, where a child library package is a package whose name is prefixed with that of a parent package.
 - ❑ Finally, multiple inheritance is not supported in Ada95.
-

Support for Object-Oriented Programming in Ada95

Dynamic Binding

- ❑ Dynamic binding is applied using a classwide type that represents all types in a class hierarchy rooted at a particular type. Every tagged type has implicitly a classwide type.

Child Packages

- ❑ Child packages are packages nested directly in other packages. A problem that appears with such a design is if a package has a large number of child packages, then the nesting package becomes large to be an effective compilation unit.

Evaluation

- ❑ Ada offers complete support for Object-Oriented programming.
 - ❑ Using child library units to control access to parent class' entities provides cleaner solution than friend functions of C++.
 - ❑ Each design decision need not be made with the root class' design.
 - ❑ Dynamic binding is not restricted to pointers and/or references.
-

The Object Model of JavaScript

- Even though JavaScript does not have classes and supports neither inheritance nor dynamic binding, it uses an object model that is based on C++ and Java objects

General Characteristics

- ❑ JavaScript is dynamically typed.
 - ❑ JavaScript does not have classes.
 - ❑ JavaScript cannot support class-based inheritance
 - ❑ JavaScript cannot support polymorphism.
 - ❑ JavaScript has two categories of variables; variables that directly store values of primitive types and variables that reference objects.
 - ❑ Variables can be declared but no type is specified; the type changes every time the variable is assigned a value.
-

The Object Model of JavaScript

JavaScript Objects

- Each object in JavaScript is a list of property-value pairs, where the property is a name and the value is a data value or reference to object.
- JavaScript properties are dynamic so that they can be added or deleted at any time.

Object Creation and Modification

- In JavaScript, objects are created with **new** operator, which creates a blank object (i.e. an object with no properties). For example, an object is created using the constructor for the predefined Object object as follows:

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

- The object's properties can be accessed using the dot notation, where the first word is the object name and the second word is the property name. The following is an example:

```
var my_car = new Object();  
my_car.make = "Ford";
```

- The **delete** operator is used to delete a property. The following is an example:

```
delete my_car.model;
```

The Object Model of JavaScript - Evaluation

- JavaScript is effective as a scripting language. However, its object model is not adequate for large software systems.

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
-