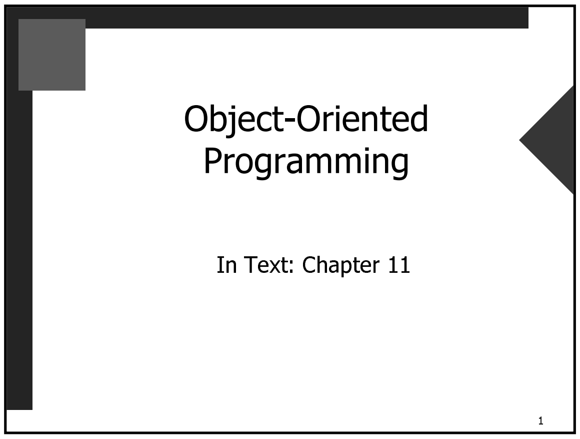
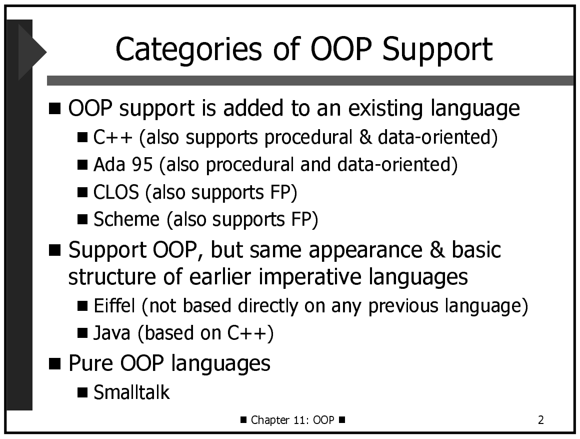


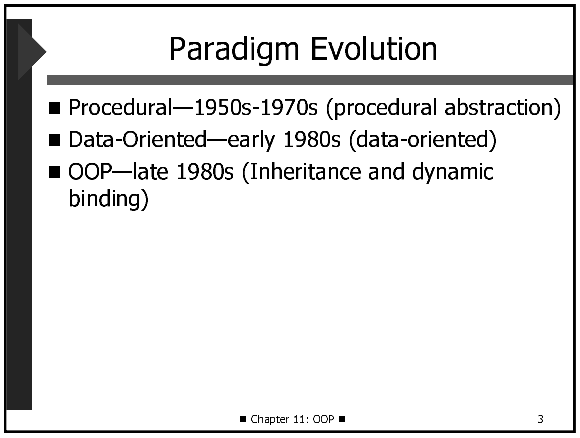
Object-Oriented Programming

In Text: Chapter 11



Categories of OOP Support

- OOP support is added to an existing language
  - C++ (also supports procedural & data-oriented)
  - Ada 95 (also procedural and data-oriented)
  - CLOS (also supports FP)
  - Scheme (also supports FP)
- Support OOP, but same appearance & basic structure of earlier imperative languages
  - Eiffel (not based directly on any previous language)
  - Java (based on C++)
- Pure OOP languages
  - Smalltalk



Paradigm Evolution

- Procedural—1950s-1970s (procedural abstraction)
- Data-Oriented—early 1980s (data-oriented)
- OOP—late 1980s (Inheritance and dynamic binding)

Origins of Inheritance

Observations of the mid-late 1980s:

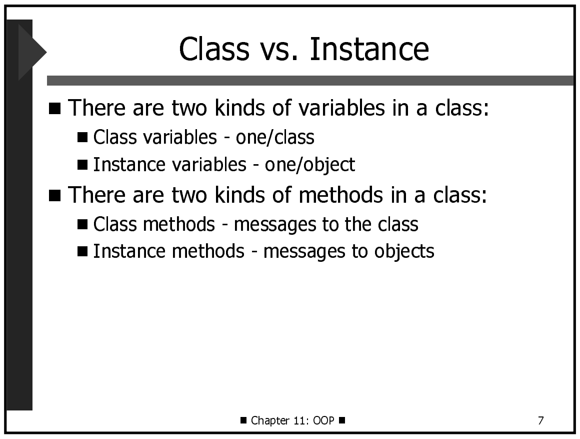
- Productivity increases can come from reuse
- Unfortunately:
  - ADTs are difficult to reuse—never quite right
  - All ADTs are independent and at the same level
- Inheritance solves both—reuse ADTs after minor changes and define classes in a hierarchy

OOP Definitions

- 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
- 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

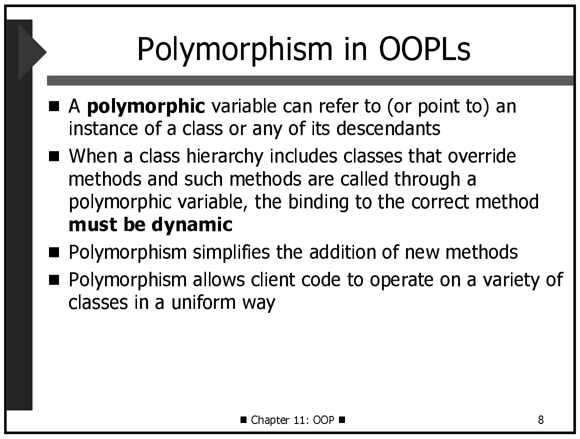
Inheritance

- In the simplest case, a class inherits all of the entities of its parent
- 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
- 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
- Single vs. multiple inheritance
- One **disadvantage** of inheritance for reuse: Creates **interdependencies among classes** that complicate maintenance



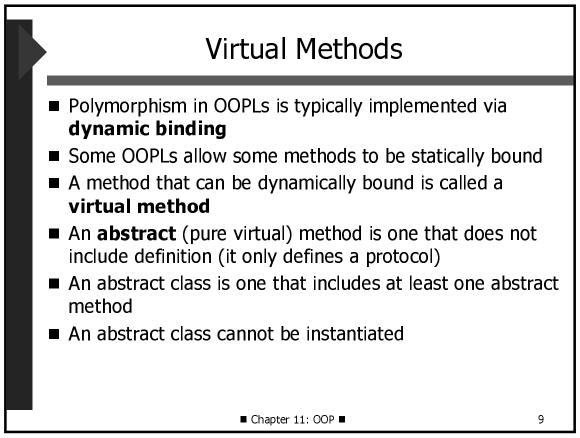
Class vs. Instance

- 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 - messages to the class
  - Instance methods - messages to objects



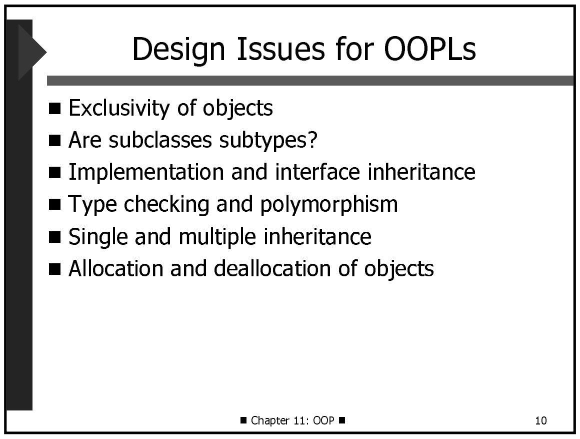
Polymorphism in OOPLs

- A **polymorphic** variable can refer to (or point to) an instance of a class or 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 **must be dynamic**
- Polymorphism simplifies the addition of new methods
- Polymorphism allows client code to operate on a variety of classes in a uniform way



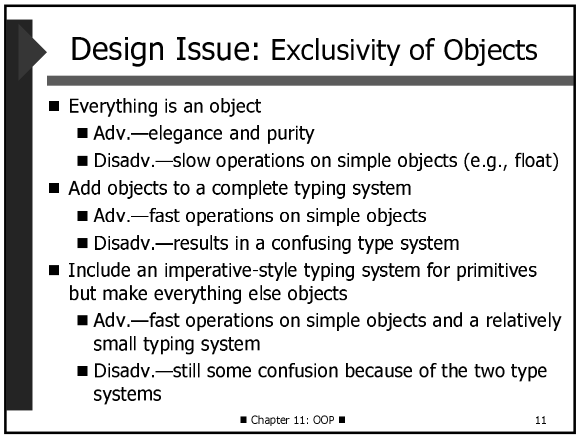
Virtual Methods

- Polymorphism in OOPLs is typically implemented via **dynamic binding**
- Some OOPLs allow some methods to be statically bound
- A method that can be dynamically bound is called a **virtual method**
- An **abstract** (pure virtual) method is one that does not include definition (it only defines a protocol)
- An abstract class is one that includes at least one abstract method
- An abstract class cannot be instantiated



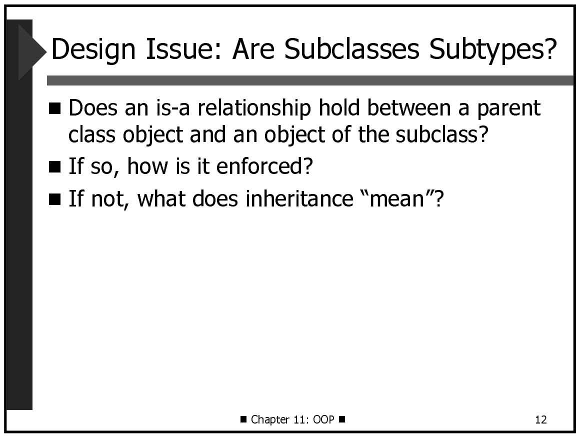
Design Issues for OOPLs

- Exclusivity of objects
- Are subclasses subtypes?
- Implementation and interface inheritance
- Type checking and polymorphism
- Single and multiple inheritance
- Allocation and deallocation of objects



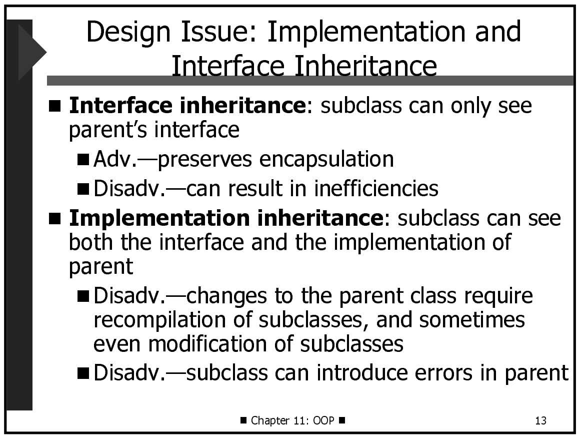
Design Issue: Exclusivity of Objects

- Everything is an object
  - Adv.—elegance and purity
  - Disadv.—slow operations on simple objects (e.g., float)
- Add objects to a complete typing system
  - Adv.—fast operations on simple objects
  - Disadv.—results in a confusing type system
- Include an imperative-style typing system for primitives but make everything else objects
  - Adv.—fast operations on simple objects and a relatively small typing system
  - Disadv.—still some confusion because of the two type systems



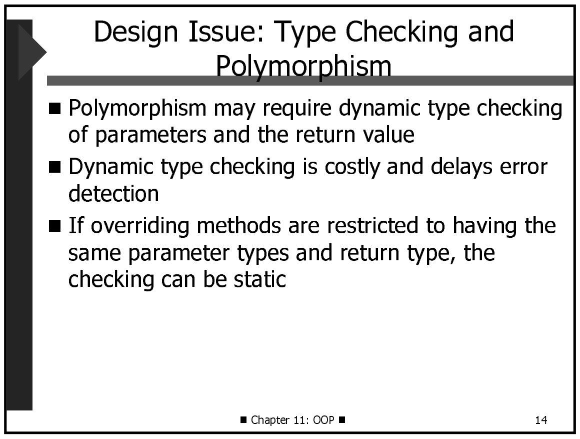
Design Issue: Are Subclasses Subtypes?

- Does an is-a relationship hold between a parent class object and an object of the subclass?
- If so, how is it enforced?
- If not, what does inheritance "mean"?



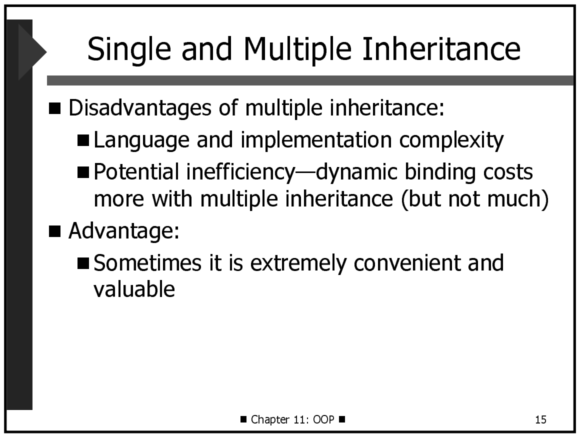
Design Issue: Implementation and Interface Inheritance

- **Interface inheritance**: subclass can only see parent's interface
  - Adv.—preserves encapsulation
  - Disadv.—can result in inefficiencies
- **Implementation inheritance**: subclass can see both the interface and the implementation of parent
  - Disadv.—changes to the parent class require recompilation of subclasses, and sometimes even modification of subclasses
  - Disadv.—subclass can introduce errors in parent



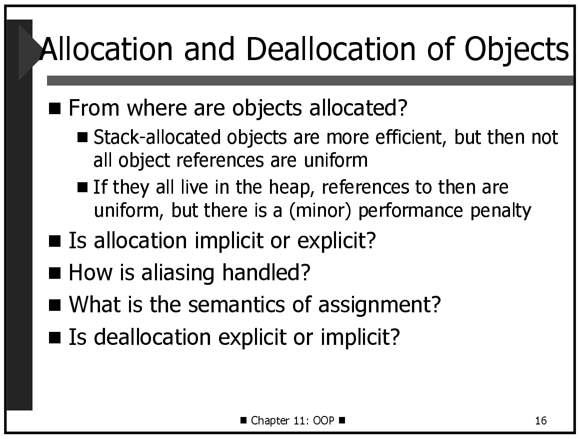
Design Issue: Type Checking and Polymorphism

- 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



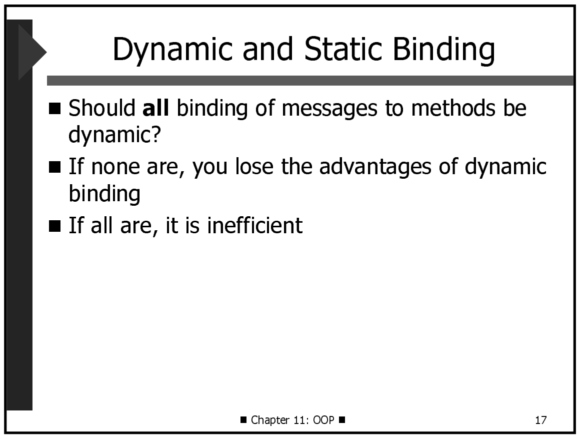
Single and Multiple Inheritance

- Disadvantages of multiple inheritance:
  - Language and implementation complexity
  - Potential inefficiency—dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is extremely convenient and valuable



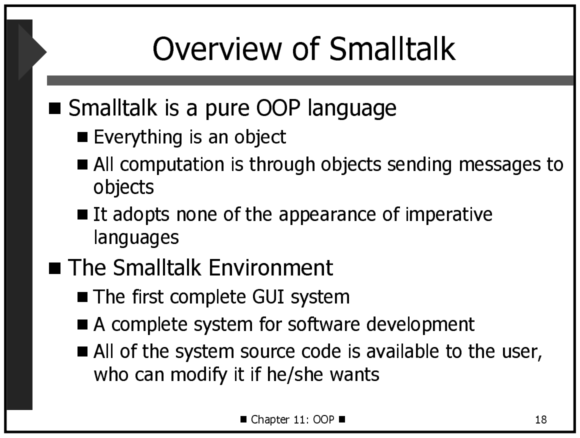
Allocation and Deallocation of Objects

- From where are objects allocated?
  - Stack-allocated objects are more efficient, but then not all object references are uniform
  - If they all live in the heap, references to then are uniform, but there is a (minor) performance penalty
- Is allocation implicit or explicit?
- How is aliasing handled?
- What is the semantics of assignment?
- 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



Overview of Smalltalk

- Smalltalk is a pure OOP language
  - Everything is an object
  - All computation is through objects sending messages to objects
  - It adopts none of the appearance of imperative languages
- The Smalltalk Environment
  - The first complete GUI system
  - A complete system for software development
  - All of the system source code is available to the user, who can modify it if he/she wants

Introduction to Smalltalk

Expressions:

- Literals (numbers, strings, and keywords)
- Variable names (all variables are references)
- Message expressions
- Block expressions

Smalltalk Message Expressions

- Two parts: the receiver object and the message itself
- The message part specifies the method and possibly some parameters
- Replies to messages are objects
- Three message forms: unary, binary, and keyword

Smalltalk Message Forms

- Unary (no parameters)

```
myAngle sin
```

(receiver = myAngle, message = sin)

- Binary (one parameter, an object)

```
12 + 17
```

(receiver=12, message=+, param=17)

- Keyword (use keywords to organize params)

```
myArray at: 1 put: 5
```

(receiver=myArray, message=at:put:, params=1, 5)

- Multiple messages to the same object can be strung together, separated by semicolons

Smalltalk Methods

- General form:

```
message_pattern [| temps |] statements
```

- A message pattern is like the formal parameters of a subprogram
  - For a unary message, it is just the name
  - For others, it lists keywords and formal names
  - temps are just names—Smalltalk is typeless!

Smalltalk Assignments

- Simplest Form:
- name1 <- name2
- It is simply a pointer assignment
- RHS can be a message expression
- index <- index + 1

Smalltalk Blocks

- A sequence of statements, separated by periods, delimited by brackets

  [index <- index + 1. sum <- sum + index]

- A block specifies something, but doesn't do it
- To request the execution of a block, send it the unary message, `value`
- e.g., […] `value`
- If a block is assigned to a variable, it is evaluated by sending value to that variable
- e.g.,

```
addIndex <- [sum <- sum + index]
 …
addIndex value
```

Blocks with Parameters

- Blocks can have parameters

```
[:x :y | statements]
```

- If a block contains a relational expression, it returns a Boolean object, true or false
- The objects true and false have methods for building control constructs

Smalltalk Iteration

- The method **whileTrue:** from Block is used for pretest logical loops. It is defined for all blocks that return Boolean objects

```
[count <= 20]
    whileTrue: [sum <- sum + count.
              count <- count + 1]
```

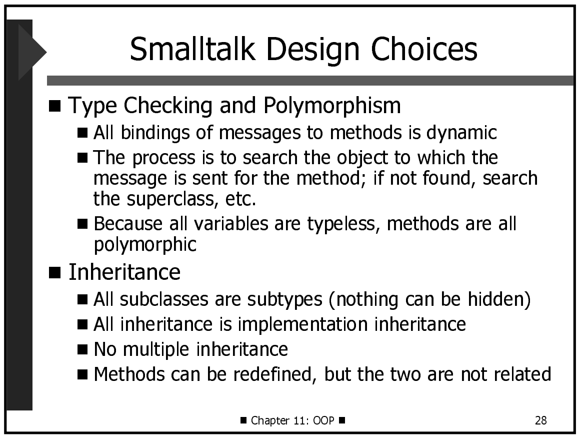
- **timesRepeat:** is defined for integers and can be used to build counting loops

```
xCube <- 1.
3 timesRepeat: [xCube <- xCube * x]
```

Smalltalk Selection

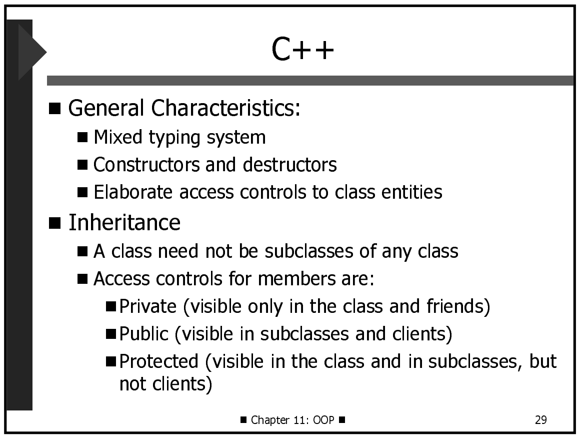
- The Boolean objects have the method **ifTrue:ifFalse:**, which can be used to build selection

```
total = 0
  ifTrue: […]
  ifFalse: […]
```



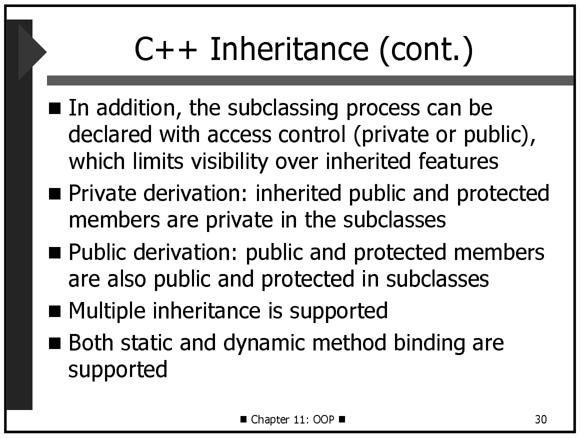
Smalltalk Design Choices

- Type Checking and Polymorphism
  - All bindings 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.
  - Because all variables are typeless, methods are all polymorphic
- Inheritance
  - All subclasses are subtypes (nothing can be hidden)
  - All inheritance is implementation inheritance
  - No multiple inheritance
  - Methods can be redefined, but the two are not related



C++

- General Characteristics:
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities
- Inheritance
  - A class need not be subclasses of any class
  - Access controls for members are:
    - Private (visible only in the class and friends)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)



C++ Inheritance (cont.)

- In addition, the subclassing process can be declared with access control (private or public), which limits visibility over inherited features
- 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
- Multiple inheritance is supported
- Both static and dynamic method binding are supported

Java

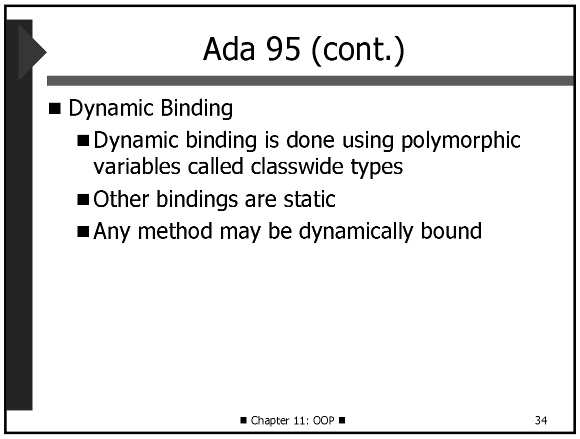
- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, accessed through reference variables, and most are allocated with new
- Inheritance
  - Single inheritance only, but there is an abstract class category (interfaces) that provides some of the benefits of multiple inheritance
  - An interface can include only method declarations and named constants (pure abstract class)
  - Methods can be final (cannot be overriden)

Java (cont.)

- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is final
- Encapsulation
  - Two constructs, classes and packages
  - Packages provide a container for classes that are related (can be named or unamed)
  - Entities defined without a scope (access) modifier are only visible within the package
  - Every class in a package is a friend to the package scope entities elsewhere in the package
  - Package scope is an alternative to the friends of C++

Ada 95

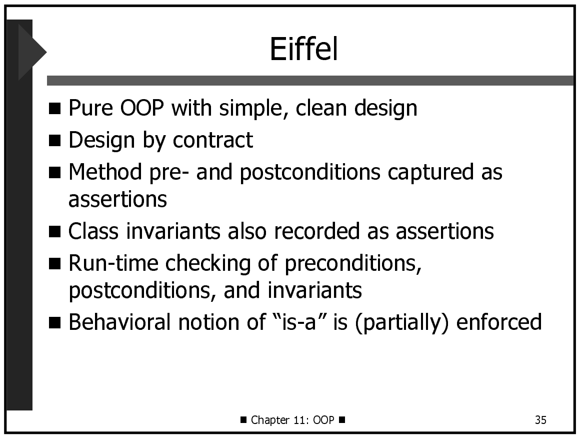
- General Characteristics
  - OOP was one of the most important extensions to Ada 83
  - Encapsulation container is a package that defines a tagged type
  - A tagged type is one in which every object includes a tag to indicate its type (at run-time)
  - Tagged types can be either private types or records
- Inheritance
  - Subclasses are derived from tagged types
  - New entities in a subclass are added in a record
  - All subclasses are subtypes
  - Single inheritance only, except through generics



Ada 95 (cont.)

- Dynamic Binding
  - Dynamic binding is done using polymorphic variables called classwide types
  - Other bindings are static
  - Any method may be dynamically bound

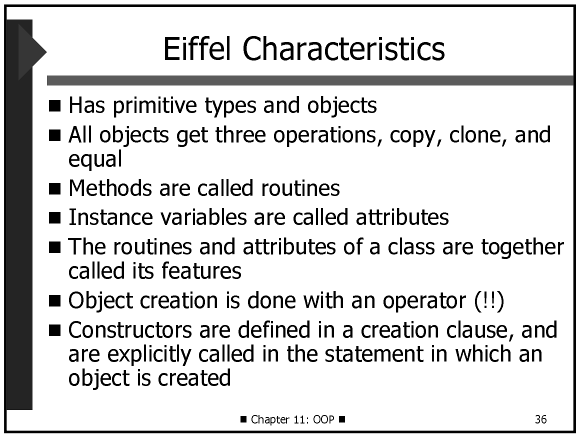
■ Chapter 11: OOP ■ 34



Eiffel

- Pure OOP with simple, clean design
- Design by contract
- Method pre- and postconditions captured as assertions
- Class invariants also recorded as assertions
- Run-time checking of preconditions, postconditions, and invariants
- Behavioral notion of "is-a" is (partially) enforced

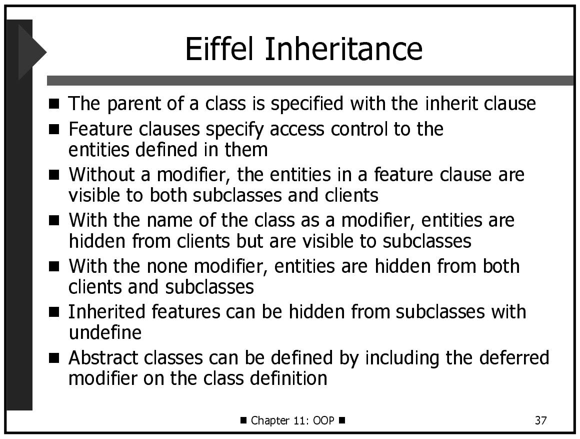
■ Chapter 11: OOP ■ 35



Eiffel Characteristics

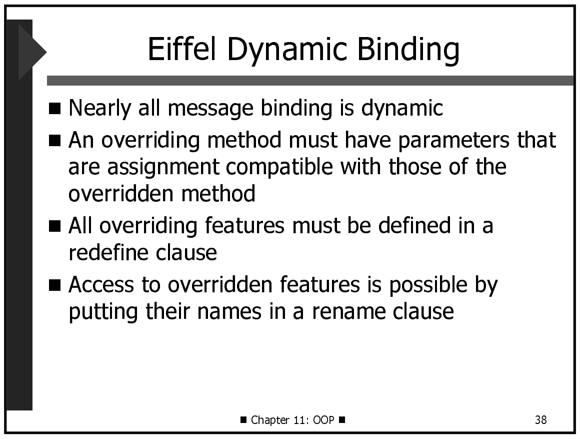
- Has primitive types and objects
- All objects get three operations, copy, clone, and equal
- Methods are called routines
- Instance variables are called attributes
- The routines and attributes of a class are together called its features
- Object creation is done with an operator (!!)
- Constructors are defined in a creation clause, and are explicitly called in the statement in which an object is created

■ Chapter 11: OOP ■ 36



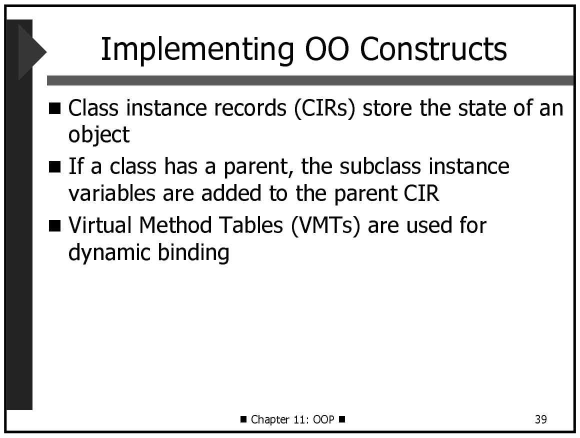
Eiffel Inheritance

- The parent of a class is specified with the inherit clause
- Feature clauses specify access control to the entities defined in them
- Without a modifier, the entities in a feature clause are visible to both subclasses and clients
- With the name of the class as a modifier, entities are hidden from clients but are visible to subclasses
- With the none modifier, entities are hidden from both clients and subclasses
- Inherited features can be hidden from subclasses with undefine
- Abstract classes can be defined by including the deferred modifier on the class definition



Eiffel Dynamic Binding

- Nearly all message binding is dynamic
- An overriding method must have parameters that are assignment compatible with those of the overridden method
- All overriding features must be defined in a redefine clause
- Access to overridden features is possible by putting their names in a rename clause



Implementing OO Constructs

- Class instance records (CIRs) store the state of an object
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Virtual Method Tables (VMTs) are used for dynamic binding