Object-Orientation

- Orthogonal to other paradigms
  - OO in imperative, functional and other languages
  - Most commonly imperative
- Evolved from SIMULA-67
  - Smalltalk-80 first fully realized OO language
- Three main features
  - Data abstraction
  - Inheritance
  - Polymorphism via dynamic method binding

Inheritance

- Code reuse
  - Class is logical unit for reuse
  - Class often not exactly what is wanted
    - Requires minor modifications
- Classification
  - Often classes are related in some way
  - Hierarchical classification
- Inheritance
  - New type can inherit attributes and behaviors from another type
  - Can add attributes and behaviors
  - Can modify behaviors
  - New type is a specialized version of type it inherits from
- Class is an ADT
  - Objects are instances of classes
- Derived class or subclass (child)
  - Class defined via inheritance from other class
- Base class or superclass (parent)
  - Class on which another is based
- E.g.
  - Vehicle/Truck
  - Pointing Device/Mouse
  - Bird/Robin
- Class variables vs instance variables
- Class methods vs instance methods

- Methods
  - The behaviors defined by a class
  - Complete set is message protocol or message interface or interface
- Message passing
  - Invoking a behavior
  - Method performed on (by) object and access attributes
    - Object is an implicit parameter
- Overriding
  - Method in subclass with same signature replaces method from superclass
- Single inheritance vs multiple inheritance
- Disadvantage of inheritance
  - Creates strong dependency between classes
  - Alternative is delegation

**Dynamic Binding**

- Subclass usually considered a subtype
  - Can assign subclass object to superclass variable
- Subclass may override behavior (method)
  - Specialized version for this type
- Polymorphism
  - Behavior is that defined by the type of the object not the type of the variable
  - achieved by dynamic binding (dispatch)
- E.g.
  - Use
    - Design
    - Specialization of subclass
    - Maintenance
    - E.g. add Pen as a pointing device
- Abstract class
  - Generalization for which objects never created
  - May specify but not implement methods
  - Abstract method
Design Issues

- Exclusivity of objects
  - Including primitive types
  - Operations are messages
  - Generally slower
  - Alternatives
    - Add classes to existing type structures
    - Complicates type structure
    - Primitives as values rest as classes
    - Value vs reference semantics
    - Wrapper classes
  - Subclass is subtype (is-a)
    - Requires/guarantees substitutability
    - Subclass must provide all methods of superclass with same semantics
    - Subclass could differ by
      - More/less methods
      - Different method signatures
      - Different visibility

- Multiple inheritance (vs single inheritance)
  - Both parents define a method with same name
  - Especially if need to override
  - Parents have common root
  - Inherit singly or doubly
  - Adds complexity
  - Allocation/deallocation
    - Stack vs heap
    - Implicit vs explicit deallocation

- Dynamic vs static binding (methods)
  - Speed
  - Nested classes
    - Visibility
  - Object initialization
    - Explicit vs implicit
Smalltalk

- Prototypical OO language
- Exclusively objects
  - Infix operators syntactic sugar for message passing
- Reply to message is an object
- Dynamic allocation and implicit deallocation
- Constructors explicitly called
  - E.g.
  - Subclass is subtype
    - Inherits all methods and attributes
    - Add methods and attributes
    - Override if same signature
    - Single inheritance
- Dynamic dispatch
  - Search up class hierarchy
- Dynamic type checking
  - Object cannot respond to message
- Evaluation
  - Small and elegant
  - Pure OO
  - Slow
  - Dynamic type binding
  - User interface
    - Windows, mouse, pop-up & pull-down menus

C++
- First widely used OO language
- Backward compatibility to C
  - Retained C's type system
  - Added classes
  - Objects static, stack dynamic or heap-dynamic
    - Explicit deallocation
Inheritance

- No common superclass
- Derived class
  - Inherit members
  - Can add data and function members
  - Can override function members
    - Same profile
    - Return type must be subtype
- Implicit initialization
- Visibility
  - public, protected, private

- Derivation mode
  - public
    - Inherited members have same visibility
    - Is a subtype
  - private
    - Inherited members are private
    - E.g.
    - E.g. linked list implementation of stack & queue
- Multiple inheritance
  - Reduces needs for friends

```java
class derived_class : derivation_mode base_class_name
| data member and member function declarations |
{
  class base_class
  {
    private:
      int a;
      float x;
    protected:
      int b;
      float y;
    public:
      int c;
      float z;
  };
  class subclass_1 : public base_class { ... };
  class subclasses_2 : private base_class { ... };
  class subclass_3 : private base_class
  {
    base_class :: c;
    ...;
  }
}
```
class single_linked_list {
    private:
    class node {
        node *next;
        int contents;
    };
    node *head;
    single_linked_list() { head = 0; }
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();
    int empty();
};

class stack : public single_linked_list {
    public:
    stack() {};
    void push(int value) {
        insert_at_head(value);
    }
    int pop() {
        return remove_at_head();
    }
};

class queue : public single_linked_list {
    public:
    queue() {};
    void enqueue(int value) {
        insert_at_tail(value);
    }
    int dequeue() {
        remove_at_head();
    }
};

class stack_2 : private single_linked_list {
    public:
    stack_2() {};
    void push(int value) {
        single_linked_list :: insert_at_head(value);
    }
    int pop() {
        return single_linked_list :: remove_at_head();
    }
    single_linked_list :: empty();
};

class queue_2 : private single_linked_list {
    public:
    queue_2() {};
    void enqueue(int value) {
        single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
        single_linked_list :: remove_at_head();
    }
    single_linked_list :: empty();
};
Dynamic Binding

- Pointer to object (heap dynamic) provides polymorphism
  - Can reference any subtype of declared type
- Members that can be overridden must be declared virtual in base class
  - Declares that dynamic binding should occur
  - Pure virtual function
    ° Abstract class
- Static or stack dynamic objects cannot use dynamic binding
  - Not polymorphic
  - Assignment assigns data members and does not “change” type
    ° Possible slicing

```c++
class Thread { ... };
class Drawing { ... };
class DrawThread : public Thread, public Drawing { ... };
```

```c++
class Shape {  
public:  
    virtual void draw() = 0;
    ...  
};
class Circle : public Shape { 
public:  
    void draw() { ... }  
};
class Rectangle : public Shape { 
public:  
    void draw() { ... }  
};
class Square : public Rectangle { 
public:  
    void draw() { ... }  
};
```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;

ptr_shape = sq;  // Now ptr_shape points to a square object
ptr_shape->draw(); // Dynamically bound to the draw in the square class
rect->draw();  // Statically bound to the draw in the Rectangle class

Square sq;  // Allocate a Square object on the stack
Rectangle rect; // Allocate a Rectangle object on
// the stack
rect = sq;  // Untie the data members values from
// the Square object
rect.draw(); // Call the draw from the Rectangle object
Evaluation

- C++ more flexible
  - Many combinations of access control, derivation, bindings and friends
  - Fine control over execution overhead
  - Efficiency
- C++ more complex
  - Hybrid type system
  - Easy to misuse features
- Multiple inheritance
- Static type checking

Java

- Not pure OO
  - Primitives are values, rest are objects
  - Auto boxing & unboxing
- Common root class (Object)
- All objects are heap-dynamic
  - Garbage collection
  - "Does not reclaim locks or close files"
  - finalize method
- Supports nested classes

Inheritance

- final method cannot be overridden
- @Override annotation
- package private and private members
- Subclasses are subtypes
- Constructor
  - Automatically called
  - Parent constructor (super) must be called as first statement or default parent constructor called
- Single inheritance only
**Interface**
- public named constants and method signatures
- Specification of a class
  - Class implements an interface
    - No code-reuse
  - Defines a type
    - Implementing class is a subtype of the interface type
- Kind of multiple inheritance
  - Inherit from parent class and implement interface
    - Subtype of both
  - Can implement one or more interfaces
    - Name clashes
  - E.g. Comparable interface

**Abstract classes**
- Method can be defined abstract
- Class must be defined abstract

```java
public interface Comparable<T> {
    public int compareTo(T o);
}
```

**Dynamic Binding**
- Methods dynamically bound
  - Class methods (static) statically bound
  - final and private methods statically bound
    - cannot be overridden
Evaluation

- Good support for OO
  - More consistent than C++ since not hybrid
- Dynamic binding normal case
- Simpler access control mechanism than C++
  - Interfaces

Ruby

- Pure OO
- Executable class definitions
  - Can add members dynamically
    - Member definitions are executable
- Variables are typeless references to objects
  - Instance variables are private
    - Require accessor/setter methods
      - attr_reader, attr_writer
- Constructor called implicitly
  - super class constructor can be called via super

```ruby
class MyClass
  # A constructor
  def initialize
    @one = 1
    @two = 2
  end

  # A getter for @one
  def one
    @one
  end

  # A setter for @one
  def one=(my_one)
    @one = my_one
  end
end
```

```ruby
end # of class MyClass
```

```
attr_reader :one, :two
```
- Inheritance
  - Single inheritance
  - Subclass not necessarily subtype
    - Subtype can change access
    - Modules can be included in a class
      - Adds access to the functions of the modules (a mixin)
      - Module is a proxy superclass
- Dynamic binding
  - Variables are typeless
  - Method dispatch goes from class to superclass (like Smalltalk)
- Evaluation
  - Pure OO
  - Weaker access control
  - No abstract classes or interfaces (not really required)
  - Slow (interpreted)

Implementation of OO
- Instance variables
  - Class instance record (CIR)
    - Struct of instance variables
      - Efficient for inheritance (single)
      - Efficient access
        - Offsets known statically
      - Dynamic binding
        - CIR has reference to class
          - Virtual method table (vtable)
          - Subclass has copy of vtable
            - Extensions added
              - Overridden method pointers replaced
- Multiple inheritance
  - One superclass chosen and implemented as single inheritance
  - Other superclasses as separate entries in CIR
public class A {
    public int a, b;
    public void draw() { ... }
    public int area() { ... }
}

public class B extends A {
    public int c, d;
    public void draw() { ... }
    public void shift() { ... }
}

class A {
    public:
        int a;
        virtual void fun() { ... }
        virtual void init() { ... }
};
class B {
    public:
        int b;
};
class C : public A, public B {
    public:
        int c;
        virtual void fun() { ... }
        virtual void did() { ... }
};
Figure 12.8
An example of a reduction CSR with multiple parents.