CS320 Principles of Programming Languages

Week 8: Object-Oriented Programming

Jingke Li

Portland State University

Fall 2017
Object-Oriented Programming

Using *data abstraction* rather than function abstraction for organizing computation, and encapsulating the state of the program in objects.

*Fundamental Concepts:*

- **Encapsulation** — The data in an object are not visible directly, but can be accessed only through operations defined on the object.

- **Inheritance** — Allows a new class be defined as an extension of an existing class, inheriting the data and method definitions of the base class.

- **Subtype Polymorphism** — Allows a subclass object to be used whenever a parent class object is expected.

- **Dynamic Dispatch** — It ensures that the code that gets executed is always consistent with the type of the object; very useful in dealing with family-of-class type polymorphism.
Object-Oriented Languages

- **Simula 67** ('60s, Dahl and Nygaard in Norway)
  - First OO language, introduces classes, objects, inheritance, virtual functions, coroutines (*Missing features*: encapsulation, class variables)

- **Smalltalk** ('70s, Alan Jay et. al. at Xerox PARC)
  - The quintessential OO language — everything is an object, including an integer or a class; all operations are “messages to objects”

- **C++** ('80s, Bjarne Straustrup at Bell Labs)
  - Provides OO features without compromising efficiency; very complicated design, needs disciplined usage

- **Java** ('90s, James Gosling at SUN)
  - Provides safety, portability, and simplicity; efficiency is secondary

- **Ruby** ('90s, Matsumoto in Japan)
  - A dynamic OO language; supports multiple programming paradigms

- ...
Encapsulation

It means the implementation details of an object are hidden from the user:

- User *does not need to* know the hidden information in order to use the object — allows the implementation to be handled separately
- User *is not permitted to* directly use or manipulate the hidden information even if desiring to do so — protect the integrity of the data in the object

*Example:*

```java
class IntStack {
    // hidden information
    private int top;
    private int[] storage;
    // interface
    public IntStack() { top = 0; storage = new int[100]; }
    public void push(int i) { storage[top++] = i; }
    public int pop() { return storage[--top]; }
}
```
Inheritance

Inheritance allows a new class to be defined as an extension of an existing class, inheriting its data and method definitions.

```
In Java:
class Point {
    protected int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void print() { ... }
}
class ColorPoint extends Point {
    private Color color;
    public ColorPoint(int x, int y, int color) {
        super(x,y); this.color = color;
    }
}

ColorPoint inherits all data and operations from its base class:

ColorPoint cp = New ColorPoint(10, 20, Red);
cp.print();
```
Inheritance

There are multiple forms of inheritance.

- **Inheritance of (Full) Specification**: The base class has both data and method definitions, and the subclass inherits them all. In this case, the subclass is a special case of the base class.
  
  *e.g.* Previous example: `ColorPoint` is a special case of `Point`.

- **Inheritance of Implementation**: The base class has method definitions that the subclass likes to inherit. In this case, the subclass is not necessarily related to the base class conceptually.
  
  *e.g.* `Colorpoint` may also want to inherit from the following class:

```java
class ColorObject {
    private Color color;
    ColorObject(Color c) { color = c; }
    Color getColor() { return color; }
    void setColor(Color newColor) { color = newColor; }
    void toGreyScale() { ... }
}
```
Inheritance

- **Inheritance of Interface**: The base class defines a set of data items and method studs, and the subclass provides the actual values for the data items and the implementation for the methods.

```java
interface ForeignCurrency {
    String country; // country of origin
    double rating; // stability rating
    double toUSD(double); // conversion to USD
    void showSample(); // display a sample bill
}

class BritishPound implements ForeignCurrency { ... }
class JapaneseYen implements ForeignCurrency { ... }
```

Each subclass must implement all the methods defined in the interface.
Inheritance

- **Inheritance of Type (Only):** The base class is mostly empty, the subclass provides all the details. In this case, the base class serves as a type container for the actual instances that are defined by the subclasses.

```java
abstract class Statement {
    abstract void print();
}

class Assign extends Statement {
    Exp lhs, rhs;
    void print() { ... }
    ...
}
class If extends Statement {
    Exp cond;
    Statement s1, s2;
    void print() { ... }
    ...
}
```
Abstract Classes and Abstract Methods

- An abstract class cannot be instantiated, i.e. no object can be created for an abstract class. This is the same as for an interface.
- However, unlike interface, an abstract class can have instance variables, regular methods, and even constructors!

```java
abstract class Statement {
    protected int linenum;
    public void showLinenum() {
        System.out.println(linenum);
    }
    public Statement(int n) { linenum = n; }
}

class Assign extends Statement {
    public Assign(int n) { super(n); }  // using base constructor
    ...
}
```

- An abstract method declaration provides only a signature. Its implementation must be provided by every (non-abstract) subclass.
Multiple Inheritance?

Some languages (e.g. C++) permit *multiple* inheritance. So a class can have both specification-based and implementation-based inheritance. In these languages, we could declare:

```
class ColorBook: public Publication, public ColorObject {
    ...
}
```

```
class ColorMagazine: public Publication, public ColorObject {
    ...
}
```

However, supporting multiple inheritance adds complexity to compiler and overhead to target code.
Multiple Inheritance?

- Java has only single inheritance, but it has a notion of *interfaces*, which are like abstract class descriptions with no variables or method implementations at all. To achieve the same effect, we could declare

```java
class ColorBook extends ColorObject implements Publication {
    ...
}

class ColorMagazine extends ColorObject implements Publication {
    ...
}
```
Polymorphism

Polymorphism provides the ability to perform same or similar actions on data of different types, hence avoiding writing the same code multiple times (while maintaining type safety and efficiency).

There are three types of polymorphism:

- **Overloading (Ad-hoc Polymorphism)** — the same symbol means different things depending on the types to which it is applied
  
  *e.g.* operator overloading and function overloading

- **Parametric Polymorphism** — the same code behavior (*i.e.* semantics) applies to different types of data
  
  *e.g.* polymorphic functions and parameterized modules

- **Subtype Polymorphism** — an object of subtype of $T$ can be safely used in any context where an object of $T$ is expected
  
  *e.g.* subclasses
Polymorphism

These types of polymorphism are fundamentally different.

- Overloading is essentially a notational trick — the same notation is used to represent one of several different things depending on the underlying types

- In contrast, a polymorphic function/module typically has a single implementation regardless of the types of its parameters

- Subtype polymorphism is generally supported through subtyping, i.e., objects of different types are entirely substitutable for objects of their base type
Operator Overloading

Most languages provide some form of operator overloading, e.g., overloading of arithmetic operators to work on either integers or reals. Some languages (e.g., Ada, C++) allow user-defined overloading, normally for user-defined types:

class Complex {
  private:
    int real, imag;
  public:
    Complex(int r, int i) {real=r; imag=i;}
    Complex operator + (Complex const &obj) {
      Complex res;
      res.real = real + obj.real;
      res.imag = imag + obj.imag;
      return res;
    }
  };

int main() {
  Complex c1(10,5), c2(2,4);
  Complex c3 = c1 + c2;
}
Function Overloading

Another form of overloading is function overloading:

```java
class MyClass {
    int i;
    // constructor overloading
    public MyClass(void) { i = 0; }
    public MyClass(int val) { i = val; }
    ...
    // method overloading
    public void print(double d) { ... }
    public void print(int i, int j) { ... }
}
```

- There are two versions of the constructor `MyClass`
- There are two versions of the function `print`
Function Overloading

Functions can be overloaded in two cases:

- Functions have different types of arguments:

```java
class MyClass {
    ...
    public void f(int i)     { ... }
    public void f(String s) { ... }
    public void f(Boolean b) { ... }
}
```

- Functions have different number of arguments:

```java
class MyClass {
    ...
    public void f()       { ... }
    public void f(int i)  { ... }
    public void f(int i, int j) { ... }
}
```

**Note:** The overloaded functions each have a separate implementation, and these implementations need not to be related.
Function Overloading

Question: What if two functions have the same number of arguments with matching types, but different return types?

```
Java

class MyClass {
    ...
    public void f(int i) { ... }
    public int  f(int i) { ... }
}
```

Answer: This is not allowed — it’s a type error!

Note: Overloading is resolved statically; that is

- for operator overloading, the compiler selects the appropriate version of the operator based on operands’ type
- for function overloading, the compiler selects the appropriate version of the function based on the number of arguments, or the types of arguments (This should explain the answer above.)
Polymorphic Functions

Functions that can be executed with arguments of different types.

**Example:** The C library function `qsort`

```c
void qsort(void *base, size_t nitems, size_t size,
           int (*compar)(const void *, const void *))
```

- The type of the array’s elements and the type of the comparison function’s arguments are both specified by the generic `void`* type.
- They can be replaced by different concrete types:

```c
static int icompare(int *i, int *j) { return (*i - *j); }
static int dcompare(double *x, double *y) { return (*x - *y); }
int main() {
    int a[10];
    double b[10];
    qsort(a, 10, sizeof(int), icompare);  // sort an int array
    qsort(b, 10, sizeof(double), dcompare);  // sort a double array
}
```
Polymorphic Functions

Example: A list length function in ML:

```ml
fun length (l: 'a list) : int =
  case l of nil => 0
  | (h::t) => (length t) + 1;

length ["sun", "mon", "tue"] => 3
length [10, 9, 8, 7] => 4
```

Note:

- From the implementation point of view, there is a fundamental difference in comparison to function overloading — typically there is a single implementation for a polymorphic function, shared by all the uses of it.

- The languages that choose to support polymorphic functions (e.g. C, ML) typically do not support function overloading; and vice versa (e.g. C++, Java)
Subtype Polymorphism

In general, a subclass instance may be used wherever a base class instance is wanted:

```java
Point[] pa = new Point[10];
pa[0] = new Point(10, 10);
pa[1] = new Point(20, 20);
pa[2] = new ColorPoint(30, 30, Red);
...
```

- However, the extra data or methods defined in the subclass cannot be accessed through a reference to the base class type:

  ```
  ... pa[2].col ... // not allowed
  ```

- Unless casting is used:

  ```
  ... ((ColorPoint) pa[2]).col ... // OK
  ```
Subtype Polymorphism

In C++, the situation is slightly different, since C++ manipulates objects directly, instead of via references:

```
class BaseClass { ... }
class SubClass : public BaseClass { ... }

BaseClass base;
SubClass sub;
...
base = sub;
```

- The assignment works by truncating the SubClass object to fit in the space of a BaseClass object. The extra subclass data would be lost. (Hence it’s preferable to manipulate objects through pointers.)
Method Overriding

If two methods have exactly the same type signature, one is defined in the base class, and the other in the subclass, then the latter overrides the former.

**Example: A simple case**

```java
class BaseClass {
    ...
    public void f() { ... }
    public int g(int i) { ... }
}

class SubClass {
    ...
    public void f() { ... }
    public int g(int i) { ... }
}
```

- Both functions `f` and `g` in `SubClass` override their counterpart in `BaseClass`
Example: A more complex case

```java
class BaseClass {
    
    public void f() { ... }
    public int g() { ... }
    public int g(int i) { ... }

}

class SubClass {
    
    public void f() { ... }
    public void g(String s) { ... }

}
```

- Function `f` in `SubClass` still overrides its counterpart in `BaseClass`
- But function `g` now has an overloading relationship with the two `g` functions in `BaseClass`; both are available in `SubClass`'s scope through inheritance
Method Overriding

If two methods have the same type signature except for their return types, it is illegal overriding! (Recall the overloading example.)

```java
class BaseClass {
    ...
    public void f(int i) { ... }
}

class SubClass {
    ...
    public int f(int i) { ... }
}
```

Note: The selection of a method definition among overriding ones is based on the language’s method-binding rules, which could lead to a runtime resolution.
Method Binding

If a subclass redefines a base class method, which version will be invoked?

```java
class Point {
    public void draw() { ... } // draw at x,y in b&w
}

class ColorPoint extends Point {
    public void draw() { ... } // draw at x,y with color
}

class Test {
    ColorPoint cp = new ColorPoint(30,40,Red);
    Point p = new Point(10,20);
    Point q = cp;

    p.draw(); // draw in b&w
    cp.draw(); // draw in color
    q.draw(); // draw in ??
}
```
Method Binding

- **Static Binding**: Uses the object’s static type information to determine method binding at *compile-time*. (It is C++’s default binding rule.)
- **Dynamic Binding**: Uses the actual type of the object at the time of invocation to determine method binding. (It is Java’s binding rule.)

Back to the Example:

```java
class Test {
    ColorPoint cp = new ColorPoint(30,40,Red);
    Point p = new Point(10,20);
    Point q = cp;
    ...
    q.draw(); // draw in ??
}
```

- With static method binding, q’s declared type is Point, hence Point’s version is selected by compiler; q.draw() draws in b&w
- With dynamic method binding, at the time of invocation, q is an object of ColorPoint, so q.draw() draws in color
Dynamic Binding in C++

By default, C++ methods use static binding (because it’s less expensive). To use dynamic binding, methods must be explicitly marked as `virtual`:

```cpp
class Point {
public:
    int x, y;
    Point(int x0, int y0) : x(x0), y(y0) {}
    virtual void draw() { ... } // draw at x,y in b&w
};
class ColorPoint: public Point {
public:
    Color col;
    ColorPoint(int x0, int y0, Color c0) : Point(x0,y0), col(c0) {}
    void draw() { ... } // draw at x,y with color
};

ColorPoint *cp = new ColorPoint(30,40,Red);
Point *p = new Point(10,20), *q = cp;
cp->draw(); // draws in color
p->draw(); // draws in color
q->draw(); // draws in color
```
Static Dispatch vs. Dynamic Dispatch

- **Static Dispatch**: To handle method overloading and static method binding, compiler maintains a table of methods for each class. When a method call is encountered, compiler uses its type signature to select (dispatch) a method from the table.

- **Dynamic Dispatch**: At runtime, a method invocation from an object always passes the object itself as the 0th argument of the method, e.g., `q.draw()` is actually implemented as `draw(q)`
  
  `a.f(i,j)` is actually implemented as `f(a,i,j)`

Dynamic binding’s implementation bears a similarity to that of static binding: when a method call is encountered, runtime system uses the method’s name and first argument to select (dispatch) a method from a method table.
A Review Example

Inheritance, overloading, and dynamic binding mixing together can complicate a program:

```java
import java.lang.Math;

class Point {
    // ... properties and constructors...

    public boolean equal(Point p) {
        return (p.x==x && p.y==y);
    }
}

class ColorPoint extends Point {
    // ... properties and constructors...

    public boolean equal(ColorPoint cp) {
        // ... implementation...
    }
}
...
ColorPoint cp = new ColorPoint(10,20,Red);
ColorPoint cq = new ColorPoint(10,20,Blue);
Point p = new Point(10,20), q = cq;
q.equal(p);       // (a)
q.equal(cp);      // (b)
cq.equal(cp);     // (c)
cq.equal(p);      // (d)
cq.equal(q);      // (e)
```

Questions:
1. What’s the relationship between the two definitions of `equal`?
2. Which `equal` method is used in each of the five cases?
A Review Example

Answer:

1. The two versions of `equal` are overloaded; there is no overriding.
2. See below:

```java
class Point {
    public boolean equal(Point p) { return (p.x==x && p.y==y); }
}
class ColorPoint extends Point {
    public boolean equal(ColorPoint cp) { ... }
}
...
ColorPoint cp = new ColorPoint(10,20,Red);
ColorPoint cq = new ColorPoint(10,20,Blue);
Point p = new Point(10,20), q = cq;

q.equal(p); // (a) Point's
q.equal(cp); // (b) Point's, with use of subtype polymorphism
cq.equal(cp); // (c) ColorPoint's
cq.equal(p); // (d) Point's
cq.equal(q); // (e) Point's
```
Summary

- Object-oriented programming views programs as collections of stateful objects that interact by sending messages.

- Four concepts are central to object-oriented programming: encapsulation, inheritance, polymorphism, and dynamic dispatch.

- Overloading is a form of compile-time polymorphism; overloaded functions are resolved by compiler.

- Method overriding is also related to polymorphism; it arises when there are multiple definitions for the same method in a hierarchy family of classes.

- Depending on the language and the method declaration, method overriding is either resolved at compile time (static dispatch) or at runtime (dynamic dispatch).