

CS421 Lecture 15: Object-Oriented Languages¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

July 10, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Encapsulation

Inheritance

Polymorphism

Typing and Object-Orientation

Object-Based Languages

Objectives

By the end of this lecture, you should

- ▶ be familiar with the core concepts of object-oriented languages
- ▶ understand different design decisions made in designing OO languages
- ▶ understand different methods of typing object-oriented languages, including some of the theory behind why this is challenging
- ▶ know the difference between object (or prototype) languages and class-based, object-oriented languages

Object-Oriented Languages

Object-oriented (OO) languages are those based around the concepts of:

- ▶ The **class**, an abstract entity which specifies data and functionality related to that data; and
- ▶ the **object**, an instance of a particular class which actually contains the specified data and provides something for the specified functionality to work on.

OO languages can be seen as an extension of ADT mechanisms, but with some additional features.

OO Languages – Core Concepts

For a language to be an OO language it generally must support the following three concepts:

- ▶ **Encapsulation:** The language should have some method for containing and potentially hiding parts of definitions, similar to those present in module systems and used with ADTs;
- ▶ **Inheritance:** It should be possible for one class to inherit functionality from one or more other classes, allowing for reuse of existing functionality;
- ▶ **Polymorphism:** Different objects should be able to respond to the same request in different ways; essentially, we should be able to ask objects to “do something”, allowing them to figure out how to do it.

Different languages approach these three points in different ways.

Encapsulation

Encapsulation is part of what makes objects objects, instead of just ADTs (caveat: some ADTs, such as in Euclid, have similar functionality, with ADTs as types):

- ▶ Each object contains data associated to it;
- ▶ each object contains functionality that knows how to operate over that data;
- ▶ objects ask other objects to do things for them by calling methods or sending messages.

Basic Encapsulation

At its most basic, encapsulation just allows data members (also properties or fields) and function members (methods) to be declared as part of a class.

- ▶ Note that this does not give us any data hiding;
- ▶ this also doesn't give us any "internal" functionality – any object could call a method on any other object

Encapsulation Options

A range of choices about how to handle encapsulation is available in current OO languages:

- ▶ are fields visible outside a class?
- ▶ can objects of a class see inside other objects of the same class?
- ▶ is it possible to set different levels of visibility on fields and methods?
- ▶ do the visibility mechanisms interact with the inheritance mechanism?
- ▶ do the visibility mechanisms interact with other naming/grouping mechanisms?

Inheritance

In OO languages, **inheritance** allows a class to reuse functionality from (usually) another class, without having to rewrite or copy/paste the code.

- ▶ The **parent**, **base**, or **super** class is the class being inherited from.
- ▶ The **child**, **derived**, or **sub** class is the class doing the inheriting.
- ▶ The child class can reuse the functionality from the parent class, including functionality the parent got from its parent, etc.

Types of Inheritance

- ▶ Some languages only allow a class to inherit from one other class – this is **single inheritance**.
- ▶ Some languages allow a class to inherit from multiple classes, which is **multiple inheritance**.
- ▶ An alternate model of inheritance allows functionality to be pulled out into distinct units which can then be used to form new classes. These units of reuse are called **mixins**.
- ▶ Mixins arguably introduce some complications, which **traits** are intended to solve.

Single Inheritance

Single-inheritance languages only allow a class to inherit from one other class.

- ▶ Examples include Java, C#, and Smalltalk
- ▶ Generally includes a designated base class from which all other classes inherit (Object in Java)
- ▶ Semantically cleaner
- ▶ Easier to predict behavior
- ▶ Easier to translate/compile
- ▶ Cannot model some situations

Multiple Inheritance

Multiple inheritance allows a single class to have multiple base classes.

- ▶ Examples include C++ and Eiffel
- ▶ Classes do not necessarily inherit from another class (in C++, classes need not have a parent class)
- ▶ Can model more situations
- ▶ Semantically more confusing
- ▶ Harder to correctly compile

Problems with Multiple Inheritance

Multiple inheritance is not without its problems. The biggest have to do with *duplication* of functionality and data (so-called *repeated inheritance*):

- ▶ **Functionality duplication:** how do you figure out which method to call when two base classes have methods with the same name? or which field to access when two fields of the same name exist?
- ▶ **Data duplication:** what happens when we inherit the same data member from two different base classes?

Problem: Functionality Duplication

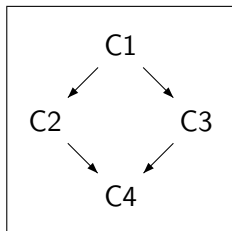
```
1 class CardGame {
2     ...
3     draw() { ... } // draw a card
4     ...
5 }
6
7 class GraphicsObject {
8     ...
9     draw() { ... } // draw the object
10    ...
11 }
12
13 class GraphicalCardGame
14     inherits CardGame, GraphicsObject {
15     ...
16 }
```

Possible Solutions

- ▶ Pick one randomly (not a good idea!)
- ▶ Require explicit disambiguation (C++)
- ▶ Use explicit import and renaming on inheritance (Eiffel)
- ▶ Use declaration order (CLOS)
- ▶ Use an ordered search of the inheritance graph (Python)

Problem: Data Duplication

- ▶ If we have a field defined in C1, how many copies are in C4?
- ▶ May want one copy, may want more than one
- ▶ Also know as the “diamond problem” (or the more dramatic “diamond of death”!)



Possible Solutions

- ▶ Always just include one copy
- ▶ Always include all the copies
- ▶ Allow programmer to choose (C++)

Note the first two solutions will exclude some valid programs, since we may want just one, or more than one, copy in various situations.

Compromise: Interface Inheritance

Single-inheritance languages like Java and C# allow multiple *interfaces* to be inherited. Interfaces are just named groups of method declarations, generally without bodies and without state (partially finished classes, with some methods left undefined, are usually called *abstract* classes).

- ▶ Advantage: specifies that a class supports specific functionality, maybe multiple interfaces; supports polymorphism (below)
- ▶ Disadvantage: doesn't allow for reuse, since interfaces don't include method bodies; leads to copy/paste reuse

Mixins

An alternate model of reuse is provided by *mixins*.

- ▶ mixins provide specific pieces of state and functionality
- ▶ new classes are created by adding mixins to existing classes
- ▶ order is important – if two mixins provide the same method name, the last one wins
- ▶ can lead to brittle schemes where no proper ordering of the mixins is possible

Traits

Traits were introduced in Smalltalk to improve on the mixin model and allow some benefits of multiple inheritance without the cost.

- ▶ Traits only provide functionality – no state
- ▶ Traits access needed state through methods
- ▶ Each trait lists which methods it needs and which it provides
- ▶ Trait composition then needs to resolve all the required methods
- ▶ Traits allow hiding of methods and renaming to resolve conflicts

Overloading and Overriding

Inheritance allows methods in derived classes to **override** methods in base classes:

- ▶ New method should have same signature as original method;
- ▶ new method hides original method in some cases, based on language rules

A method with the same name but a different signature is said to **overload** that name, giving it multiple meanings:

- ▶ Overload rules differ between languages, including what can be a valid overload
- ▶ Overload interaction with inheritance also differs between languages – what happens if you overload an inherited name?

Using Inherited Functionality

- ▶ Non-overridden methods and fields can be directly referenced (based on visibility rules)
- ▶ Base-class methods can usually be called using keywords (`super.method` in Java) or scope resolution (`MyBase::method` in C++)
- ▶ These are based on the visibility rules from encapsulation – in Java, `private` methods from a parent aren't suddenly `public` in a child, for instance, so the child cannot see them.

Alternate Model – Beta

The Beta language uses an alternate model. Instead of calling the method and using `super` to call the parent definition, the definition at the root of the inheritance hierarchy is called first and `inner` is used to call back towards the child.

- ▶ Note that this makes calling the child optional!
- ▶ Helps to support *behavioral subtyping* – the child cannot ignore the parent, unlike in languages with `super` but not `inner`, so the parent can enforce behavior

Polymorphism

Polymorphism is the third key feature of OO languages. At its core, polymorphism provides the mechanism for letting objects of different classes figure out how to do things themselves – a `Circle` object would draw itself differently than a `Triangle` object, for instance, but one should be able to use them both as `Graphics` objects.

Methods of Enabling Polymorphism

Polymorphism can be enabled in two major ways:

- ▶ **Structurally**: especially in dynamically typed or untyped languages, if I have a reference to an object I can send it a message, and if it understands it then it can take action
- ▶ **Subtype Polymorphism** – especially in statically typed languages, if class A inherits from class B then an object of class A should be usable wherever an object of class B can be used, but I should be able to “customize” class A to respond differently to the same requests

Virtual Methods

The key to getting subtype polymorphism to work is that the method to invoke should be choosable *at runtime*. This way the behavior can vary, based on the type of the object. Often methods that allow this are called **virtual** methods.

- ▶ **Virtual** comes from virtual function table, the structure often used to resolve the correct function to invoke;
- ▶ in some languages (like Java) all methods are virtual;
- ▶ in other languages (like C++) methods can be static or virtual; the former has better performance;
- ▶ this way of selecting methods is also called **dynamic dispatch**.

Method Selection

In the presence of dynamic dispatch, the way to choose the proper method to invoke varies from language to language:

- ▶ The proper virtual method is chosen at runtime based on the dynamic type of the object;
- ▶ the proper static method is chosen at compile time based on the static type of the variable holding the object (pointer/reference);
- ▶ some languages have more involved models – *multimethods* in CLOS or *predicate dispatch* (a current research topic).

OO Type Systems

Type systems for object-oriented languages are generally more complex than for imperative languages because of two factors:

- ▶ **Recursive Types:** All types are recursive, with methods requiring implicit or explicit `self` or `this` pointers.
- ▶ **Subtyping:** Most languages with subclassing use this mechanism to establish subtypes as well; the subtypes are constrained in certain ways to ensure that the language remains type safe.

Recursive types have been more of a problem in the *theory* behind typing OO languages; subtyping has led to more problems in actual *implementations* of type checkers.

Subtyping

There is a fair amount of theory behind subtyping. For our purposes, we need to be concerned with two main features of the subtyping relation, as it applies here:

- ▶ For class A to be a valid subtype of class B, the fields of class A must have the **same** types as those of class B
- ▶ For class A to be a valid subtype of class B, each method of class A which overrides a method of class B must be a subtype of the overridden method
- ▶ For method F to be a subtype of method G each parameter of F must be the same or a supertype of the same parameter in G and the return type of F must be the same or a subtype of the return type of G

Terminology

This rule for method subtyping is often stated as

- ▶ the parameter types are **contravariant**,
- ▶ while the return type is **covariant**

But, this isn't what we want...

One problem with this is that it doesn't get us what we would like.

- ▶ In Java, the `Object` class include an `equals` method
- ▶ This method has a parameter of type `Object`
- ▶ This parameter type *cannot* be overridden in subclasses, even though we want to

While this is known now, it wasn't when some OO languages were created. Eiffel allowed covariant parameter types, which introduced holes into the type system. Those are now caught with runtime checks.

Remember the rules

- ▶ Types of instance variables cannot change (caveat: in Java you can “change” them, since if a subclass creates a new instance variable of the same name it’s a brand new instance variable – it shadows the old one)
- ▶ If a method is defined in a superclass, an override in a subclass can either leave the parameters with the same type or can change them, but can only change the type to be a **superclass** of the current type
- ▶ If a method is defined in a superclass, an override in a subclass can either leave the return type the same or can change it to be a **subclass** of the current type

Example, in Eiffel

```
1 class LINKABLE[G]
2   item: G;
3   right: like Current;
4
5   putRight (other: like Current) is
6   do
7     right := other
8   end;
9 end
```

Example in Eiffel, 2

```
1 class BILINKABLE[G] inherit LINKABLE[G]
2   left: like Current;
3   putRight (other: like Current) is
4     do
5       right := other;
6       if (other /= Void) then
7         other.putLeft(Current)
8       end;
9   putLeft (other: like Current) is
10    do
11      left := other
12    end;
13 end
```

Example in Eiffel, 3

Now, with `biNode` of class `BILINKABLE` and `node` of class `LINKABLE`, what happens?

```
1 trouble(p,q: LINKABLE[RATIONAL]) is
2 do
3   p.putRight(q);
4   ...
5 end
6
7 trouble(biNode,node);
```

This will compile fine, but will crash at runtime!

Dynamic Types

Some OO languages use dynamic types instead.

- ▶ Generally, if the receiver understands the message, there is no type error;
- ▶ not based on any inheritance hierarchy, just on receiver functionality;
- ▶ also called **duck typing** (“if it walks like a duck and quacks like a duck, it must be a duck”)

Why do I need classes?

Some languages forego using classes altogether. These languages are often called just **object** languages, or **prototype** languages.

- ▶ New objects are created by cloning existing objects;
- ▶ objects contain slots that hold either data members or methods;
- ▶ operations can be performed on an object to change the values in slots, add or remove slots, etc;
- ▶ various languages have different ways to access to parent to permit **delegation**, a way to request functionality from another object

Method Calls in Prototype Languages

One major difference with languages such as SELF and standard OO languages is that methods are executed in the context of the *caller* – i.e. not in the context of the actual owner of the method. The method is found and returned, and then has access to the state of the object that called it.

Further Reading

This is an area of current research, and these languages are not widely used. A good high-level introduction is:

- ▶ The Interpretation of Object-Oriented Programming Languages, by Iain Craig.