

Object-Oriented Programming

©Andrey Kruglyak, 2010

Friday, May 7, 2010

1

Intuition

- ◀ World around us consists of objects
- ◀ Each object has its properties, some are constant, some are variable
- ◀ We know how to use each object (look from the outside)
- ◀ We know how each object operates (look from the inside)
- ◀ Let's structure our programs using objects as well!
 - ◀ should decrease complexity, make parts of the code re-usable...

Friday, May 7, 2010

2

Some history

- ◀ OO has its roots in 1960s
 - ◀ Smalltalk (by Alan Kay and others) in 1970s
- ◀ It became mainstream in 1990s (C++, Delphi, Objective C, Java, ...)
- ◀ Originally OO was a way to decrease system complexity of imperative programs by fragmenting system state, but has now found its way to languages representing other programming paradigms
- ◀ People often speak about OO paradigm, but note that it actually represents another dimension than imperative/functional/logic/etc. programming

Friday, May 7, 2010

3

OO, attempt 1

- ◀ Let us have procedures (blocks of sequential code) that create objects. Each object will have state variables and methods that operate on these...
- ◀ Problem: MASSIVE duplication of code! Besides, it is difficult to verify that it is safe to use a certain object in a certain way...

```
obj = makeObject37
obj.someMethod - is this method defined in obj?
```

Friday, May 7, 2010

4

OO, attempt 2

- ◁ Let all objects created with a certain procedure share method code. Then all we need to keep in each object is a reference to an array of methods, its state variables and maybe some extra constant parameters (typically arguments to the procedure)
- ◁ We thus obtain classes of objects that operate in the same way, and we typically call such procedures class definitions

OO, attempt 2+

- ◁ Let us go one step further and describe how we can use a certain set of objects, i.e. describe their interface
- ◁ The interfaces are typically described using types (obj1 and obj2 are of the same type T = we can use them in the same way)
- ◁ Important: each class definition will produce objects with the same interface, but there may be multiple classes that produce objects conforming to a single interface: Interface <-> Class*

OO in short

- ◁ Objects have state variables and methods
- ◁ Objects can be created using classes (IMPLEMENTATION)
- ◁ Objects have interfaces (formally defined as types) (USE)

Principle ideas of OO

- ◁ State encapsulation
- ◁ Multiple representations
- ◁ Subtyping
- ◁ Inheritance
- ◁ Open recursion

State encapsulation

- An object is a collection of state variables and methods
- Only the object's methods can directly inspect or manipulate its state variables - important for maintaining state consistency and limit code modifications if the state variables are changed
 - Smalltalk, Timber - state variables can only be named within the object
 - Java - strictly speaking, no state encapsulation at all (there is no way to make a state variable inaccessible from another object of the same class); what Java has is rules for lexical scoping, the same for both state variables and methods

Friday, May 7, 2010

9

Multiple representations

- Another cornerstone of OO is the separation between an interface ("what we can do with an object") and implementation ("how it actually works"), paving the way for multiple implementations, each of which can be used together with the rest of the code
 - BTW, violation of encapsulation also undermines this property
- Each object thus needs to carry with it the implementation (directly, or in the form of a method table, where a method is looked up at run-time)

Friday, May 7, 2010

10

Subtyping

- A type of the object is its interface (i.e. which methods are defined)
- Subtyping is thus a relation on interfaces:
"A < B if any object of type A can be used in any context where an object of type B is expected"
- Subtyping can be
 - structural (ML, OCaml)
 - nominative/declarative (Java, Timber)
- Subtyping is most useful together with implicit coercions

Friday, May 7, 2010

11

Inheritance (subclassing)

- If objects of two classes share some of the behavior, and one is a subset of the other both in terms of state variables and of methods, we may want to re-use the first class definition
- This is called subclassing - an automatic way to derive new classes from existing classes by adding / overriding methods
- Two implementation techniques:
 - embedding (incorporate definition directly)
 - delegation* (an object of a super class is created, with its state variables and methods)
- Inheritance always implies subtyping!
- *not delegation as in Objective-C and some other languages, where it is used to denote the Listener pattern

Friday, May 7, 2010

12

Implementation of subtyping

- ◁ Unified method tables for types
- ◁ If we allow overriding/shadowing of methods - which method should be called?
 - ◁ static dispatch (method lookup at compile time)
=> method tables for types work fine
 - ◁ dynamic dispatch (method lookup at run time)
=> we need method tables for objects (or more precisely, a reference in the object to the method table of its actual type)

Friday, May 7, 2010

13

Implementation of subtyping

- ◁ If we allow multiple subtyping, we will need to perform method lookup by name - or by type - at run time
- ◁ Using multiple method tables allows to implement type coercion as a simple indirection and it also saves some memory (often used when we have subtyping and subclassing at the same time)
- ◁ Finally, typeclasses are implemented using "witnesses" (these are normally determined statically, but may be implemented dynamically if we keep a witness table for each type and perform lookup by name - or by type - at run time)

Friday, May 7, 2010

14

Open recursion and late binding

- ◁ If a method of an object invokes another method of the same object (a "recursive" call), in a language with inheritance this is done via the special "self" (or "this") variable, which is normally late-bound (i.e. its type depends on how the object was created)
- ◁ This gives rise to interesting effects - methods defined in superclasses become in fact polymorphic if they invoke other methods that may be overridden

Friday, May 7, 2010

15

A note on terminology

- ◁ invoke a method = call a method = post/send a message
- ◁ caller = message sender
- ◁ callee = message recipient
- ◁ The "message" naming convention helps when we decouple method invocation from method execution

Friday, May 7, 2010

16

A note on terminology

- ◀ A method belongs to an object and operates on its state
- ◀ A function is a computation that operates on its arguments and returns a value; an impure function may also modify some global variables and/or perform output
- ◀ A procedure is a sequence of statements/commands

Java

- ◀ State encapsulation - no, scoping rules instead
- ◀ Multiple representations - yes, for interfaces
- ◀ Subtyping - "extends" for interfaces, "extends" and "implements" for classes (multiple subtyping)
- ◀ Inheritance - "extends" for classes (single inheritance, implemented by delegation, dynamic dispatch)
- ◀ Open recursion - yes

Timber

- ◀ State encapsulation - yes! this becomes especially important due to object-level concurrency
- ◀ Multiple representations - yes! types are completely separate from classes (note that we mean something different by "interfaces")
- ◀ Subtyping - yes, declarative subtyping for struct types and supertyping for data types, automatic subtyping for function types, multiple sub-/supertyping
- ◀ Inheritance - no, cannot even implement delegation "by hand" without introducing a different behavior due to object-level concurrency
- ◀ Open recursion - no, since there is no inheritance