Object-Orientation

COMP 524: Programming Language Concepts
Björn B. Brandenburg
The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.
What is OO?

Conceptual model.
- Objects: opaque entities that have an identity, state, and behavior.
- Objects communicate by sending messages to each other.

Metaphors.
- Orchestra model.
  - Lot’s of experts that can do one task well.
  - One conductor that coordinates overall problem solution.
- Service provider model.
  - An object provides (exactly) one service.
  - May rely on sub-contractors.
What is OO?

Conceptual model.
- Objects: opaque entities that have an identity, state, and behavior.
- Objects communicate by sending messages to each other.

Metaphors.
- Orchestra model.
  ‣ Lot’s of experts that can do one task well.

OO is a natural fit for problem decomposition: humans tend to think in terms of “objects” that “do” “things”. OO recognizes this and supports this way of thinking.
Benefits of OO

Key features.

- **Encapsulation**, information hiding.
  ‣ Reduces **complexity**, conceptual load, likelihood of **errors**.

- **Inheritance**.
  ‣ Increases **productivity** and code **reuse**.

- **Abstraction**, clean interfaces.
  ‣ Improves code reuse, separation of concerns.
  ‣ Enables large teams to **develop in parallel**.

- **Sub-type polymorphism**.
  ‣ Code reuse.

- **Decoupling**.
  ‣ Code reuse.
Benefits of OO

Key features.

- **Encapsulation**, information hiding.
  - Reduces **complexity**, conceptual load, likelihood of **errors**.

- **Inheritance**.
  - Increases **productivity** and code **reuse**.

- **Abstraction**, clean interfaces.
  - Improves code reuse, separation of concerns.
  - Enables large teams to **develop in parallel**.

- **Sub-type polymorphism**.
  - Code reuse.

- **Decoupling**.
  - Code reuse.

---

**OO has succeeded in practice** because it makes **individual developers** and **teams as a whole** more productive (compared to procedural languages).
Two Flavors of OO

Focus on OO Concepts.
 ➣ Pioneered by Smalltalk.
   ‣ Adopted by Ruby, Python, Javascript, etc.

 ➣ Very dynamic.
   ‣ Late binding.
   ‣ Dynamic type checking.
   ‣ Objects of the same class can differ in structure.

Focus on Implementation.
 ➣ Pioneered by Simula 67.
   ‣ Adopted by C++, Java, C#, Eiffel, etc.

 ➣ Composite types.
 ➣ Some components are functions.
 ➣ All objects of one class must have same structure (memory layout).
 ➣ Optional early-binding.
Two Flavors of OO

Focus on OO Concepts.
- Pioneered by Smalltalk.
  ‣ Adopted by Ruby, Python, Javascript, etc.
- Very dynamic.
  ‣ Late binding.
  ‣ Dynamic type checking.
  ‣ Objects of the same class can differ in structure.

Focus on Implementation.
- Pioneered by Simula 67.
  ‣ Adopted by C++, Java, C#, Eiffel, etc.
- Composite types.
- Some components are functions.
- All objects of one class must have same structure (memory layout).
- Optional early-binding.

Pure object orientation: everything is an object (even numbers, functions, etc).
Model and Implementation

Upon receipt of a message (= method call),

an object may change state (= update its attributes),

collaborate with other objects
(= call methods of other objects),

and finally reply (= return value).
Multiple Inheritance

class Person {
    void haveFun() {...};
    void work() {...};
}

class Teacher extends Person {
    void study() { ... }; // newly define study()
    void work() { study(); ... }; // override work()
}

class Researcher extends Person {
    void study() { ... }; // newly define study()
    void work() { study(); ... }; // override work()
}

class Professor extends Teacher, Researcher {
    void haveFun() { work(); }
}

(new Professor()).haveFun();
Multiple Inheritance

```java
class Person {
    void haveFun() {
    }
    void work() {
    }
}
class Teacher extends Person {
    void study() {
    }
    void work() {
        study();
        ...
    }
}
class Researcher extends Person {
    void study() {
    }
    void work() {
        study();
        ...
    }
}
class Professor extends Teacher, Researcher {
    void haveFun() {
        work();
    }
}
(new Professor()).haveFun();
```
Mix-in Inheritance

Restricted alternative to multiple inheritance.

- Linear “true” inheritance: only single base class.
- Can mix-in traits with a class.
  - e.g., Java interfaces.

Interfaces + delegation.

- Pure interfaces: lot’s of repeated code.
  - Java’s interfaces do not include default implementation.
- Better alternative: provide a default class; delegate to member object.
Delegation Example

```java
interface Bar {
    void bar();
}

class DefaultBar implements Bar {
    void bar() {
        ...
    }
}

class MyClass implements Bar {
    private DefaultBar barImpl = new DefaultBar();

    void bar() {
        barImpl.bar();
    }
}
```
Delegation Example

Default implementation to avoid repetition.

```java
interface Bar {
    void bar();
}

class DefaultBar implements Bar {
    void bar() { ... };
}

class MyClass implements Bar {
    private DefaultBar barImpl = new DefaultBar();

    void bar() { barImpl.bar(); }
}
```
Delegation Example

Delegate calls to default implementation.

```java
interface Bar {
    void bar();
}

class DefaultBar implements Bar {
    void bar() {
        ...
    }
}

class MyClass implements Bar {
    private DefaultBar barImpl = new DefaultBar();

    void bar() {
        barImpl.bar();
    }
}
```
Delegation Example

```csharp
interface Bar {
    void bar();
}

class DefaultBar implements Bar {
    void bar() {
    ...
    }
}

class MyClass implements Bar {
    private DefaultBar barImpl = new DefaultBar();

    void bar() {
        barImpl.bar();
    }
}
```

C# provides explicit delegate syntax
Delegation Example

interface Bar {
    void bar();
}

class DefaultBar implements Bar {
    void bar() {
        ...
    }
}
class MyClass implements Bar {
    private DefaultBar barImpl = new DefaultBar();
    void bar() {
        barImpl.bar();
    }
}

Scala’s traits allow default implementations as part of the interface definition:

trait Similarity {
    def isSimilar(x: Any): Boolean
    def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}

From: http://www.scala-lang.org/node/126
Early vs. Late Binding

**Early Binding.**
- **Static name resolution.**
- Compiler determines at compile time which code will be called.
- As **efficient** as a regular procedure call.

**Late Binding.**
- Name is resolved at **runtime**.
- Requires **dynamic method dispatch**.
- Incurs (small) overhead.
Binding Time Example

class A {
    void aFun() { ... };
}

class B extend A {
    void aFun() { ... };
}

A obj = new B();
obj.aFun();
Binding Time Example

Super-class reference type.

class A {
    void aFun() {...};
}

class B extend A {
    void aFun() {...};
}

A obj = new B();
obj.aFun();
class A {
    void aFun() {...};
}

class B extend A {
    void aFun() {...};
}

A obj = new B();
obj.aFun();

Late binding: B.aFun() is called.
Binding Time Example

```java
class A {
    void aFun() {
        ...
    }
}

class B extend A {
    void aFun() {
        ...
    }
}

A obj = new B();
obj.aFun();
```

Early binding: `A.aFun()` is called.
Binding Time Example

**Late binding:** type of the **object** determines the method.

**Early binding:** type of the **reference** determines the method.

```java
class A {
    void aFun() {...};
}

class B extends A {
    void aFun() {...};
}

A obj = new B();
obj.aFun();
```
Fragile Base Classes

apparently correct changes to a base class that break subclasses

Version 1

class Base { 
    void f() { ... };  
    void g() { ... }; 
}  

Version 2

class Base { 
    void f() { ... };  
    void g() { ...; f(); ... }; 
}  

Client

class Child extends Base { 
    void f() { ....; g(); .... }; 
}
Fragile Base Classes

apparently correct changes to a base class that break subclasses

Version 1

```java
class Base {
    void f() { ... };
    void g() { ... };
}
```

Version 2

```java
class Base {
    void f() { ... };
    void g() { ...; f(); ... };
}
```

Client

```java
class Child extends Base {
    void f() { ....; g(); .... };
}
```

After upgrade:
infinite recursion.
Fragile Base Classes

*apparently correct changes to a base class that break subclasses*

Version 1

```java
class Base {
    void f() { ... };
    void g() { ... };
}
```

Version 2

```java
class Base {
    void f() { ... };
    void g() { ...; f(); ... };
}
```

Client

```java
class Child extends Base {
    void f() { ....; g(); .... };
}
```

After upgrade: infinite recursion.
Fragile Base Classes

Large problem in practice.

➡ Many systems ship with large class libraries.
   ‣ E.g., Java, C#/.NET, Objective-C.

➡ Developers can subclass system classes.
➡ Every upgrade can break previously-working code!

Avoidance.

➡ Requires careful class design.
➡ Later implementation changes should make very little assumptions.
Fragile Base Classes

Large problem in practice.

- Many systems ship with large class libraries.
  - E.g., Java, C#/.NET, Objective-C.
- Developers can subclass system classes.
- Every upgrade can break previously-working code!

Avoidance.

- Requires careful class design.
- Later implementation changes should make very little assumptions.

Related problem: binary compatibility vs. separate compilation. Recompilation necessary if base class changes.
Class Modification at Runtime
aka “monkey patching”

Pure OO: Everything is an object.
- **Even classes**.
- Objects can change state.
- In many dynamic languages this can be used to **modify classes at runtime**.
  ‣ E.g., Python, Ruby,…

Inheritance vs. modification.
- Inheritance leaves the superclass unchanged.
- Direct modification **affects all modules** using the class.
- Imagine amending the built-in string class…
Example: Runtime Patches

```python
class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()

def a_function(self, msg):
    print "a_function was called", msg

# Modify class at runtime!
Base.any_name = a_function

# Added method works on previously-created instances..
obj.any_name("as a method of Base!

# Replace existing method at runtime!
Base.a_method = dangerous

obj.a_method()
```

Output:

```
Thu", April 15, 2010
```
Example: Runtime Patches

```python
class Base(object):
    def a_method(self):
        print "a_method was called"
```

```python
obj = Base()
obj.a_method()
```

```python
def a_function(self, msg):
    print "a_function was called", msg
```

# Modify class at runtime!
Base.any_name = a_function

# Added method works on previously-created instances..
obj.any_name("as a method of Base!
```

```python
def dangerous(self):
    print "Replacing methods can cause tricky bugs!"
```

# Replace existing method at runtime!
Base.a_method = dangerous

```python
obj.a_method()
```

Output:

Class definition with one method.
Example: Runtime Patches

class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()
Example: Runtime Patches

class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()

def a_function(self, msg):
    print "a_function was called", msg

# Modify class at runtime!
Base.any_name = a_function

Output:
a_method was called

Define top-level function...

...and add it to the class at runtime.
Example: Runtime Patches

class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()

def a_function(self, msg):
    print "a_function was called", msg

# Modify class at runtime!
Base.any_name = a_function

# Added method works on previously-created instances..
obj.any_name("as a method of Base!")

Output:
a_method was called
a_function was called as a method of Base!

New “method” is immediately available in all instances, as if declared in the class itself.
Example: Runtime Patches

```python
class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()

def a_function(self, msg):
    print "a_function was called", msg

# Added method works on previously-created instances..
obj.any_name("as a method of Base!")

def dangerous(self):
    print "Replacing methods can cause tricky bugs!"

# Replace existing method at runtime!
Base.a_method = dangerous

obj.a_method()
```

Output:
```
a_method was called
a_function was called as a method of Base!
Replacing methods can cause tricky bugs!
```

Can also replace (or remove) previously-declared methods.
In **Python**, some built-in classes that are implemented in C cannot be modified. In **Ruby**, virtually every class can be modified.

```python
class Base(object):
    def a_method(self):
        print "a_method was called"

obj = Base()
obj.a_method()

def a_function(self, msg):
    print "a_function was called", msg

# Modify class at runtime!
Base.any_name = a_function

# Added method works on previously-created instances..
obj.any_name("as a method of Base!")

def dangerous(self):
    print "Replacing methods can cause tricky bugs!"

# Replace existing method at runtime!
Base.a_method = dangerous

obj.a_method()
```

Output:
```
a_method was called
a_function was called as a method of Base!
Replacing methods can cause tricky bugs!
```
Runtime Patches: Discussion

Uses.

- **Add functionality**, e.g., logging, caching, invariant checking,…
- **Fix bugs** in third-party module.
- **Add convenience methods**.
  ‣ E.g., add a “*make a file with this name*” method to the string class (this is actually done in the Ruby-based `brew` package manager).

Dangers.

- **Two patches** for the same class.
  ‣ Unpredictable application: “last one wins.”
  ‣ Incompatible changes.
- **Corresponding source hard to find** (maintenance problem).
  ‣ Eg., if you notice a **bug in a class in module A**, the corresponding **code could reside in modules B, C, D, …**
- **Fragile** updates: changes to the class being patched can render runtime patches in any number of modules incorrect.
Objects without Classes

prototype-based languages

Some languages avoid classes completely.

- Pioneered by the language Self.
- Gaining in popularity (JavaScript is prototype-based.)

Concept.

- Everything is an object.
- Objects have a prototype (reference to another object):
  - Messages (i.e., method calls, member references) not handled by an object are redirected to the prototype.

- Objects are created by cloning an existing object, which becomes the prototype.
- Exact details vary between languages.
Prototype Example

(JavaScript)

function Bar() {
    this.credits = "created by Bar"
}

function Foo() {
    this.credits = "created by Foo"
}

Bar.prototype.get_proto_name = function () {
    return "I'm a Bar."
}

Foo.prototype.get_proto_name = function () {
    return "I'm a Foo."
}

obj1 = new Bar()
obj2 = new Foo()

document.write("<br><br>--Before--<br>")
document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
document.write("<br>")
document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())

obj1.__proto__ = Foo.prototype;
obj2.__proto__ = Bar.prototype;

document.write("<br><br>--After--<br>")
document.write("obj1 was " + obj1.credits + ": " + obj1.get_proto_name())
document.write("<br>")
document.write("obj2 was " + obj2.credits + ": " + obj2.get_proto_name())
function Bar() {
    this.credits = "created by Bar"
}

function Foo() {
    this.credits = "created by Foo"
}

Bar.prototype.get_proto_name = function () {
    return "I'm a Bar."
}

Foo.prototype.get_proto_name = function () {
    return "I'm a Foo."
}

obj1 = new Bar()
obj2 = new Foo()

Output:
--Before--
obj1 was created by Bar: I'm a Bar.
obj2 was created by Foo: I'm a Foo.

--After--
obj1 was created by Bar: I'm a Foo.
obj2 was created by Foo: I'm a Bar.

Can change prototype at runtime. Equivalent to changing the “class.”