Inheritance Variations

Technion - Israel Institute of Technology

Slides by: Oren Afek, December 2016
Last Update: Oren Afek, June 2017
In This Tutorial

• Code Reusability
• Mixins
• Traits
• Interfaces Implementation
The DRY Principle:

Don’t Repeat Yourself

enforces us to define abstraction levels and use them to prevent code duplication.

- Abstract Building blocks: functions, classes
- Abstract extension: Inheritance
- We now present different ways to make an extension abstraction
Extension Directions

• **Upward** – “The standard way”, *Inheritance*.

• **Downward** – Adding capabilities to existing types, *Mixins*.

• **Sideward** – Dividing extension between several subparts, *Traits*. 
A **mixin** is a class designed to provide functionality for another class.

```cpp
template <typename T>
class LoggingClass : public T {
public:
    void execute() {
        cout << "LOG: starting task " << endl;
        T::execute();
        cout << "LOG: finishing task " << endl;
    }
};
```

```cpp
LoggingClass<MyClass1>() m1;
MyClass2 m2 = *new LoggingClass<MyClass2>();
m1.execute();
static_cast<LoggingClass<MyClass2>> (m2).execute();
```

As long as a class supports an “execute” method, LoggingClass can be used to add logging capabilities to the execution.
Mixins can be used to add any mixture of functionalities to a class:

```cpp
template <typename T>
class Flying: public T {
    public:
        void fly(){ //do something }
};

template <typename T>
class Swimming: public T {
    public:
        void swim(){ //do something }
};

template <typename T>
class Walking: public T {
    public:
        void walk(){ //do something }
};

class Animal{
    ...
}

Swimming<Animal> fish;
Swimming<Walking<Animal>> penguin;
Swimming<Walking<Flying<Animal>>> duck;
Walking<Flying<Animal>> butterfly;
```
Traits: Flattening over Linearization

- Trait: A composable unit of behaviour
- Serves as a type
- No fields
- Provides some methods (with behaviour)
- Requires other methods (abstract)
- **Can access each other’s methods**
- When composing traits, if a method has more than one implementation it becomes abstract
“Java with Traits”

• A research version of Java

trait T1 {
    abstract void add(int v);
    void inc() { add(1); }
}

trait T2 {
    abstract int getValue();
    abstract void setValue(int v);
    void add(int v) { setValue(getValue() + v); }
}

class Int uses T1, T2 {
    int n;
    int getValue() { return n; }
    void setValue(int v) { n = v; }
}
T1 t1 = new Int(); // A trait is also type
t1.add(3);
• A real example

```
Trait named: #T1 uses: {} category: 'TraitExample'
inc
    self add: 1

Trait named: #T2 uses: {} category: 'TraitExample'
add: val
    self setValue: ((self getValue) + val)

Object subclass: #Int uses: T1 - {#add:} + T2
instanceVariableNames: 'n'
    getValue
        ^n
    setValue: val
        n := val
```
• Java’s alternative to multiple inheritance: interfaces
  – No state, no implementation (until Java 8 default methods)
• All Java objects have to support some features:
  – Reflection – e.g., getClass()
  – Garbage collection
  – Synchronization
• Those features need be available regardless of static type
  – Even if referenced via interface type

How many levels does the Java object model have? And C++’s?
• Remember: the compiler knows the index of the vfunc
  – Given a pointer to the vtable, we get dynamic binding
• Have a vptr for each interface, move pointer on upcast

• Pro: fast dynamic dispatch
• Cons:
  – Per object space overhead
  – How do we get to the header?
• Have a vptr for each interface, never move pointer

• Pro: header always available

• Cons:
  – Per object space overhead
  – How do we get to the interface vtable?
• Move interface vtables to the class object

• Pros:
  – Header always available
  – No per object overhead

• Cons:
  – How do we get to the interface vtable?
    • Need a lookup, possibly using caching
  – Multiple memory accesses
The Price of Interfaces

- Interfaces are an important abstraction mechanism
- Interfaces avoid semantic problems typical to multiple inheritance
  - Until you get default methods...
- Implementing language support is not easier
  - Every JVM does it differently
  - Heavy use of JIT optimizations to mitigate lookups
- **DO NOT AVOID THE USE OF INTERFACES FOR PERFORMANCE REASONS!**
  - Unless you can prove that this really is the bottleneck
Software Design - 236700

- The natural sequence of OOP
- As we have taught you the mechanisms and the idea behind the implementations and the “Spirit of the languages”

- In *Software Design* you’ll learn how to use them in your code rightfully.

- Furthermore, you’ll learn on how to write code **THE RIGHT WAY**
  – How to test it, and make it reusable and steady.

- Cool stuff like design patterns and the best practices of Java.
The Exam

• 3 Hours

• 5 Questions
  – One from previous semesters (7 years back, only solved exams from the Web Course)
  – One HW-based question

• Material: Lectures, Tutorials and HW
• No auxiliary material