Programming Paradigms

Lecture 15:
Data Abstraction and Object-Orientation (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Wake-up Exercise

What does the following Java code print?

class WakeUp {
    {
        System.out.println("a");
    }
    WakeUp() {
        System.out.println("b");
    }
    static {
        System.out.println("c");
    }
}
// ...
WakeUp w = new WakeUp();

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following Java code print?

class WakeUp {
    {
        System.out.println("a");
    }
    WakeUp() {
        System.out.println("b");
    }
    static {
        System.out.println("c");
    }
}
// ...
WakeUp w = new WakeUp();

Result: cab
Wake-up Exercise

What does the following Java code print?

class WakeUp {
    {
        System.out.println("a");
    }
    WakeUp() {
        System.out.println("b");
    }
    static {
        System.out.println("c");
    }
}
// ...
WakeUp w = new WakeUp();

Static initializer: 
Executed when class loaded

Result: cab
Wake-up Exercise

What does the following Java code print?

class WakeUp {
    {
        System.out.println("a");
    }
    WakeUp() {
        System.out.println("b");
    }
    static {
        System.out.println("c");
    }
}

// ...
WakeUp w = new WakeUp();

Result: cab

Instance initializer:
Executed when class instantiated, just before the constructor

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following Java code print?

class WakeUp {
    {
        System.out.println("a");
    }
    WakeUp() {
        System.out.println("b");
    }
    static {
        System.out.println("c");
    }
}

// ...
WakeUp w = new WakeUp();

Result: cab
Overview

- Introduction
- Encapsulation and Information Hiding
- Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Mix-in Inheritance
- Multiple Inheritance
Static vs. Dynamic Method Binding

- Given: Subclass that defines a method already defined in the superclass
- How to decide which method gets called?
  - Based on type of variable
  - Based on type of object the variable refers to
Example

class person { ... }
class student : public person { ... }
class professor : public person { ... }

void person::print_mailing_label() { ... }
void student::print_mailing_label() { ... }
void professor::print_mailing_label() { ... }

student s;
professor p;

person *x = &s;
person *y = &p;

s.print_mailing_label();
p.print_mailing_label();

x->print_mailing_label();
y->print_mailing_label();
Example

class person { ... }
class student : public person { ... }
class professor : public person { ... }

void person::print_mailing_label() { ... }
void student::print_mailing_label() { ... }
void professor::print_mailing_label() { ... }

student s;
professor p;

person *x = &s;
person *y = &p;

s.print_mailing_label();
p.print_mailing_label();

x->print_mailing_label();
y->print_mailing_label();

Subclasses also define method print_mailing_label
Example

class person { ... }
class student : public person { ... }
class professor : public person { ... }

void person :: print_mailing_label() { ... }
void student :: print_mailing_label() { ... }
void professor :: print_mailing_label() { ... }

student s;
professor p;

person *x = &s;
person *y = &p;

s.print_mailing_label();
p.print_mailing_label();
x->print_mailing_label();
y->print_mailing_label();
Example

class person { ... }
class student : public person { ... }
class professor : public person { ... }

void person :: print_mailing_label() { ... }
void student :: print_mailing_label() { ... }
void professor :: print_mailing_label() { ... }

student s;
professor p;

person *x = &s;
person *y = &p;

s.print_mailing_label();
p.print_mailing_label();

x->print_mailing_label();
y->print_mailing_label();

Methods of subclasses called
Example

class person { ... }
class student : public person { ... }
class professor : public person { ... }

void person::print_mailing_label() { ... }
void student::print_mailing_label() { ... }
void professor::print_mailing_label() { ... }

student s;
professor p;

person *x = &s;
person *y = &p;

s.print_mailing_label();
p.print_mailing_label();

x->print_mailing_label();
y->print_mailing_label();

Which methods to call here?
Answer 1: **Bind methods based on type of variable**

- Can be *statically resolved* (i.e., at compile time)
- Will call `print_mailing_label` of `person` because `x` and `y` are pointers to `person`
Dynamic Method Binding

- **Answer 2:** Bind methods based on type of object the variable refers to
  - In general, cannot be resolved compile time, but only at runtime
  - Will call `print_mailing_label` of student for `x` because `x` points to a student project (and likewise for `y` and professor)
Pros and Cons

Static method binding

- No performance penalty because resolved at compile-time
- But: Subclass cannot control its own state

Dynamic method binding

- Subclass can control its state
- But: Performance penalty of runtime method dispatch
Example (C++)

class text_file {
    char *name;
    // file pointer
    long position;
    public:
    void seek(long offset) {
        // (...) 
    }
};

class read_ahead_text_file : public text_file {
    char *upcoming_chars;
    public:
    void seek(long offset) {
        // redefinition
    }
};
Example (C++)

```cpp
class text_file {
    char *name;
    // file pointer
    long position;
public:
    void seek(long offset) {
        // (...)
    }
};

class read_ahead_text_file : public text_file {
    char *upcoming_chars;
public:
    void seek(long offset) {
        // redefinition
    }
}
```

- Subclass needs to change `upcoming_chars` in `seek`
- But with static method binding, cannot guarantee that it gets called
Support in Popular PLs

Static method binding

Dynamic method binding
Support in Popular PLs

Static method binding

Dynamic binding

Dynamic binding for all methods:
Smalltalk, Python, Ruby
Support in Popular PLs

Dynamic binding by default, but method or class can be marked as **not overridable**: Java, Eiffel
Support in Popular PLs

Static method binding by default, but programmer can specify dynamic binding.
Java, Eiffel: Final/frozen Methods

- Mark individual **methods** (or classes) as **non-overridable**
  - Java: `final` keyword for methods and classes
  - Eiffel: `frozen` keyword for individual methods
C++, C#: Overriding vs. Redefining

Override method: Dynamic binding

Redefine methods with same name: Static binding

- C++: Superclass must mark method as `virtual` to allow overriding
- C#: Subclass must mark method with `override` to override the superclass method
Demo

Virtual.cpp
procedure print_mailing_label(r : person) is ...  
procedure print_mailing_label(s : student) is ...  
procedure print_mailing_label(p : professor) is ...

procedure print_label(r : person'Class) is 
begin
  print_mailing_label(r);
  -- calls appropriate overloaded version, 
  -- depending on type of r at runtime 
end print_label
procedure print_mailing_label (r : person) is ...
procedure print_mailing_label (s : student) is ...
procedure print_mailing_label (p : professor) is ...

procedure print_label (r : person'Class) is
begin
    print_mailing_label (r);
    -- calls appropriate overloaded version,
    -- depending on type of r at runtime
end print_label

"Class-wide type" of person,
i.e., person itself and all its subtypes
Abstract Methods and Classes

- **Abstract method**
  - Implementation omitted
  - Subclass must provide it

- **Abstract class**: Class with at least one abstract method
Demo

Abstract.java

Abstract.cpp
Quiz: Method Binding

# Pseudo code

class A:
    void foo():
        ...
    void bar():
        ...

class B extends A:
    void bar():
        ...

A x = new B()
B y = x
x.bar() # call 1
y.bar() # call 2

What is called when
a) PL always uses dynamic method binding
b) PL always uses static method binding
c) PL always uses static method binding unless overriding method marked with override
Quiz: Method Binding

```python
# Pseudo code

class A:
    void foo():
        ...
    void bar():
        ...

class B extends A:
    void bar():
        ...

A x = new B()
B y = x
x.bar() # call 1
y.bar() # call 2
```

What is called when

a) PL always uses dynamic method binding

b) PL always uses static method binding

c) PL always uses static method binding unless overriding method marked with override

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Method Binding

# Pseudo code
```python
class A:
    void foo():
        ...
    void bar():
        ...

class B extends A:
    void bar():
        ...
```

A x = new B()
B y = x
x.bar() # call 1
y.bar() # call 2

What is called when
a) PL always uses dynamic method binding
b) PL always uses static method binding
c) PL always uses static method binding unless overriding method marked with `override`
Quiz: Method Binding

What is called when
a) PL always uses dynamic method binding
b) PL always uses static method binding
c) PL always uses static method binding unless overriding method marked with override

# Pseudo code
class A:
    void foo():
        ...
    void bar():
        ...
class B extends A:
    void bar():
        ...
A x = new B()
B y = x
x.bar() # call 1
y.bar() # call 2
Method Lookup

With dynamic method binding, how does the program **find the right method to call?**

- Most common implementation:
  - Virtual method table ("vtable")

- Every object points to table with its methods

- Table is shared among all instances of a class
class foo {
    int a;
    double d;
    char c;
    public:
        virtual void k() { ... }
        virtual int k() { ... }
        virtual void m() { ... }
        virtual double n() { ... }
} F;

```c
r1 := F
r2 := * r1   // vtable address
r2 := * (r2 + (3 - 1) * 4)   // assuming sizeof(address) == 4 bytes
call * r2
```

compiler-generated code for dynamic method binding (for foo::m)
Implementation of Inheritance

- Representation of **subclass instance**, including its vtable: Fully compatible with superclass

  □ Can use subclass instance like a superclass instance without additional code
class bar: public foo {
    int w;
    public:
    void m() { .. }
    virtual double s() { .. }
    virtual char *t() { .. }
} B;

bar's vtable

foo::k
foo::l
bar::m
foo::n
bar::s
bar::t

code pointers
Implementation of Inheritance (2)

```cpp
class foo { ... }
class bar : public foo { ... }

foo F;
bar B;

foo *q;
bar *s;

q = &B;
s = &F;
```
Implementation of Inheritance (2)

class foo { ... }
class bar : public foo { ... }

foo F;
bar B;

foo *q;
bar *s;

q = &B;
s = &F;

Okay: References through q will use prefixes of B’s data space and vtable
Implementation of Inheritance (2)

class foo { ... }
class bar : public foo { ... }

foo F;
bar B;

foo *q;
bar *s;

q = &B;
s = &F;

Static semantic error: F lacks the additional data and vtable entries of a bar object
Overview

- Introduction
- Encapsulation and Information Hiding
- Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Mix-in Inheritance
- Multiple Inheritance
Motivation

- Designing an inheritance tree with exactly one parent class: Difficult in practice
- Examples
  - Cat may be both Animal and Pet
  - Widget in database maybe Sortable, Graphable, and Storable
Mix-in Inheritance

- **Mix-in**: Class-like abstraction that provides methods to be used in other classes without inheriting from the mix-in
- A class can mix in multiple mix-ins
- Example:
  - Combine mix-ins `Animal` and `Pet` into `Cat`
Support in Popular PLs

Many variants of the mix-in idea

- Java: Interfaces are a lightweight version of mix-ins
  - Since Java 8: Default implementations of interface methods
- Scala: “Traits”
- Ruby: include modules into a class
Overview

- Introduction
- Encapsulation and Information Hiding
- Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Mix-in Inheritance
- Multiple Inheritance
True Multiple Inheritance

- Sometimes, want to **expand multiple classes**
- Allowed in some PLs, e.g., C++, Python, OCaml
- Example (C++):

  ```
class student : public person, public system_user {
  ...
}
```
True Multiple Inheritance

- Sometimes, want to **expand multiple classes**
- Allowed in some PLs, e.g., C++, Python, OCaml
- Example (C++):

  ```
  class student : public person, public system_user {
    ...
  }
  ```

Gets all fields and methods of both superclasses
. Single inheritance: Class hierarchy is a tree

. Multiple inheritance: No longer a tree, but a directed, acyclic graph (DAG)

  "diamond problem": multiple paths to same superclass
  (here: \( Y \rightarrow X \))
Semantic Issues

- What if two parent classes provide a method with the same name?
- What if two parent classes are both derived from a common “grandparent”? Does the “grandchild” have one or two copies of the grandparent’s fields?
- How to represent objects in memory? Can each parent be a prefix of the child?
Semantic Issues

- What if two parent classes provide a method with the same name?
- What if two parent classes are both derived from a common “grandparent”? Does the “grandchild” have one or two copies of the grandparent’s fields?
- How to represent objects in memory? Can each parent be a prefix of the child?

Answers depend on the PL and go beyond this lecture.
Quiz: Data Abstraction

Which of the following is true?

- Static and dynamic method binding are the same in statically typed PLs.
- vtables store the fields of an object.
- A class can build upon multiple mix-ins.
- In Java, a class can directly extend multiple classes.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Data Abstraction

Which of the following is true?

- Static and dynamic method binding are the same in statically typed PLs.
- Vtables store the fields of an object.
- A class can build upon multiple mix-ins.
- In Java, a class can directly extend multiple classes.
Overview

- Introduction
- Encapsulation and Information Hiding
- Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Mix-in Inheritance
- Multiple Inheritance