Programming Paradigms

Data Abstraction

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Data Abstraction

- **Goal: Describe class of memory objects and their associated behavior**
- **Abstract data type**
  - Set of values and set of operations
- **Example: Stack**
  - Values: Data on stack
  - Operations: push, pop, etc.

Classes and Objects

- **Classes: Form of data abstraction**
  - Encapsulation and information hiding
- **Objects**
  - Instances of classes (in class-based PLs, e.g., Java, C++)
  - Primary entities (in prototype-based PLs, e.g., Smalltalk, JavaScript)

Overview

- Inheritance ←
- Initialization and Finalization
- Dynamic Method Binding

Inheritance

- **Code reuse by defining a new abstraction as extension or refinement of an existing abstraction**
- **Subclass inherits members of superclass**
  - Can add members
  - Can modify members

Subclasses vs. Subtypes

Are subclasses a subtype of the superclass?

- In principle, no
  - Subclassing is about reusing code inside a class
  - Subtyping enables code reuse in clients of a class
    - Client written for supertype works with any subtype
- In practice, most PLs merge both concepts

Liskov's Substitutability Principle

- **Each subtype should behave like the supertype when being used through the supertype**
- **Let `B` be a subtype of `A`**
  - Any object of type `A` may be replaced by an object of type `B`
  - Clients programming against `A` will also work with objects of type `B`

"A behavioral notion of subtyping" by B. Liskov and J. Wing, ACM T Progr Lang Sys, 1994

Demo

Liskov.java

Modifying Inherited Members

- **Can a subclass modify inherited members?**
- **Answer depends on the PL**
  - Java: Any method can be overridden
  - C++: Only methods declared as `virtual` by the base class can be overridden

Demo

Virtual.cpp

Modifying Inherited Members (2)

- **Can a subclass hide inherited members?**
  - Again, answer depends on the PL
- **Java and C#: Subclass can neither increase nor decrease the visibility of members**
- **Eiffel: Subclass can both restrict and increase visibility**

Modifying Inherited Members (3)

- **Public/protected/private inheritance in C++**
  - Makes all inherited members at most public/protected/private
  - E.g., all members (incl. public members) that are privately inherited are private in the subclass
  - Private inheritance does not imply a subtype relationship

Modifying Inherited Members (4)

Accessibility in derived class:

Inheritance	Private members	Protected members	Public members
Public	Yes	Yes	Yes
Protected	No	Yes	Yes
Private	No	Yes	Yes

Demo

Inheritance.cpp

Modifying Inherited Members (4)

- **More C++ rules**
  - Subclass can decrease visibility of superclass members, but never increase it
  - Subclass can hide superclass methods by deleting them

Quiz: Inheritance

Where is the compilation error (and why)?

```
class A {
    public:
    void foo() {}

    protected:
    void bar() {}
};
class B : private A {
};
class C : public B {
    public:
    void baz() {
        this->bar();
    }
};
int main() {
    C c;
    c.baz();
}
```

Quiz: Inheritance

Where is the compilation error (and why)?

```
class A {
    public:
    void foo() {}

    protected:
    void bar() {}
};
class B : private A {
};
class C : public B {
    public:
    void baz() {
        this->bar();
    }
};
int main() {
    C c;
    c.baz();
}
```

Error: `bar` is not visible

- `B` inherits `A` as `private` class, hence, all members are `private`
- `C` cannot access `private` members of `B`

Overview

- Inheritance
- Initialization and Finalization ⟵
- Dynamic Method Binding

Initialization

- **Each class: Zero, one, or more constructors**
- **Distinguished by**
  - Number and type of arguments (C++, Java, C#)
  - Name of the constructor (Eiffel)

Example: Eiffel Constructors

```
class COMPLEX
creation
  new_cartesian, new_polar
feature {ANY}
  x, y: REAL

  new_cartesian(x_val, y_val : REAL) is
    -- (...) constructor implementation

  new_polar(rho, theta : REAL) is
    -- (...) constructor implementation

  -- (...) other members
end
```

Implicit vs. Explicit Initialization

- **Some PLs (e.g., Java): Constructor must always be called explicitly**
- **Other PLs (e.g., C++): Constructor sometimes called implicitly**
  - Value model of variables: Object must be initialized
  - Declarating a variable implicitly calls zero-argument constructor

Implicit vs. Explicit Initialization (2)

Example: Java

```
class Foo { ... }

Foo f;
```

- Uninitialized reference to a `Foo` object
- Has value `null`

Example: C++

```
class Foo { ... }

Foo f;
```

- Implicitly initialized with `Foo`'s default constructor
- Variable contains the object

Superclass Constructors

- During initialization of subclass, also **initialize inherited superclass fields**

```java
// Java example
class A { ... }

class B extends A {
  B(int k) {
    super(k);
  }
}
```

```cpp
// C++ example
class A { ... }

class B : public A {
  public:
  B(int k) : A(k) {
    ..
  }
}
```

Superclass Constructors

- During initialization of subclass, also **initialize inherited superclass fields**

```java
// Java example
class A { ... }

class B extends A {
  B(int k) {
    super(k);
  }
}
```

```cpp
// C++ example
class A { ... }

class B : public A {
  public:
  B(int k) : A(k) {
    ..
  }
}
```

Call to super constructor

Execution Order of Constructors

- **Constructor(s) of base class(es) execute before constructors of subclass**
  - C++: Implicit in PL
  - Java: Enforced by not allowing any statement before `super()`

Destructors

- **In some PLs (e.g., C++), each class can define a destructor**
- **Called when**
  - Object goes out of scope
  - `delete` operator called on object
- **Optional, but highly recommended if class dynamically allocates memory**
  - Must free memory in destructor (otherwise: memory leak)

Destructors: Example

```
// C++ example
cout << string("Hi there").length(); // prints 8
```

Destructors: Example

```
// C++ example
cout << string("Hi there").length(); // prints 8
```

- First, calls `string(const char*)` constructor
- Afterwards, calls `~string()` destructor because object goes out of scope

Execution Order of Destructors

- **Destructor of subclass called before destructor(s) of superclass(es)**
  - Reverse order of constructors
  - Intuition: First clean up added state, then inherited state

Finalization

- **Java and C#: No destructors but finalizers**
- **Called immediately before object gets garbage-collected**
  - Use to clean up resources, e.g., file handles
  - Note: May never be called, e.g., in short-running programs
    - `finalize` has been deprecated in Java 9

Demo

Immortal.java

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
    public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
    public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Result: `ABC~C~B~A`

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
      public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Implicitly creates object of class `C`

Result: `ABC~C~B~A`

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
    public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Class with two superclasses

Result: `ABC~C~B~A`

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
    public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Constructor and destructor

Result: ABC~C~B~A

Quiz: Initialization & Finalization

What does the following C++ code print?

```
class A {
  public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class B {
  public:
    B() { cout << "B"; }
    ~B() { cout << "~B"; }
};
```

```
class C :
      public A, private B {
  public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
};

int main() {
  C c;
}
```

Result: `ABC~C~B~A`

Execution order of constructors and destructors

Overview

- Inheritance
- Initialization and Finalization
- Dynamic Method Binding ⟵

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();
```

Variables of subtypes

Variables of supertype

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?**

Static Method Binding

- **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 at 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;
  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++)

```
class text_file {
  char *name;
  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 method binding**

Dynamic binding for all methods: **Smalltalk, Python, Ruby**

Support in Popular PLs

Static method binding ⟷ Dynamic method binding

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

Support in Popular PLs

Static method binding ⟷ **Dynamic method binding**

Static binding by default, but programmer can specify dynamic binding: C++, C#

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

Quiz: Method Binding

```
# Pseudo code
class A:
  void foo():
    ...
  void bar():
    print("a")

class B extends A:
  void bar():
    print("b")

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

What is printed when

a) PL uses dynamic method binding

b) PL uses static method binding

Quiz: Method Binding

```
# Pseudo code
class A:
  void foo():
    ...
  void bar():
    print("a")

class B extends A:
  void bar():
    print("b")

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

What is printed when

a) PL uses dynamic method binding

b) PL uses static method binding

Quiz: Method Binding

```
# Pseudo code
class A:
  void foo():
    ...
  void bar():
    print("a")

class B extends A:
  void bar():
    print("b")

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

What is printed when

a) PL uses dynamic method binding

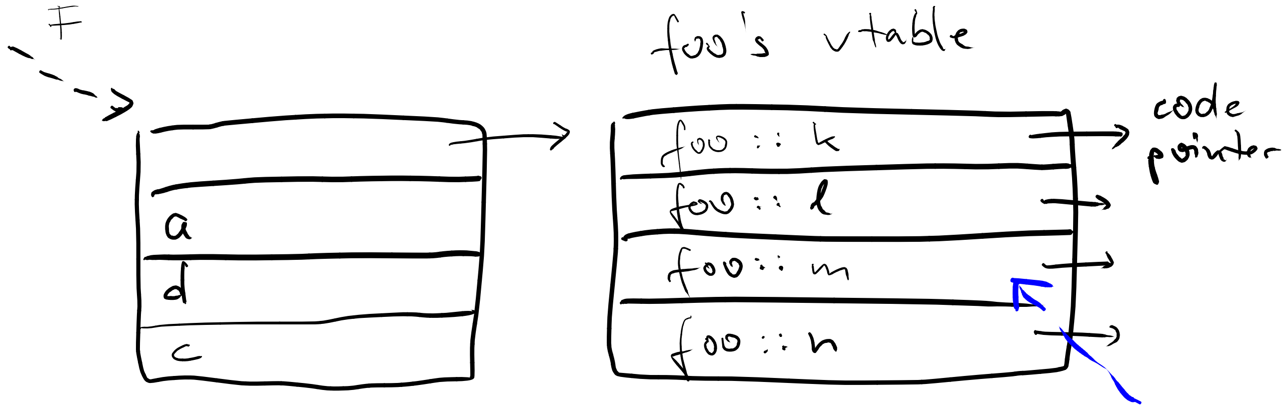
b) PL uses static method binding

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 l() {...}
    virtual void m() {..}
    virtual double n() {..}
} F;
```



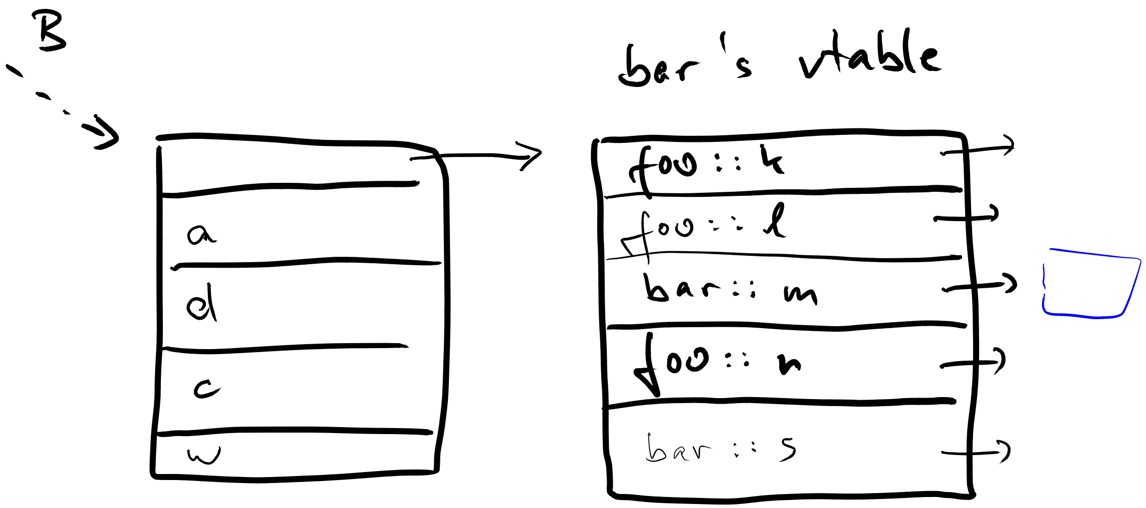
Compiler-generated code for dynamic method binding for F.m():

```
r1 := F
r2 := *r1
r2 := *(r2 + 4·(3-1))
call *r2
```

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() {...}
} B;
```



Compiler-generated code for calls to m() is same for foo and bar.

Quiz: Data Abstraction

Which of the following is true?

- Java enforces Liskov's substitutability principle.
- Static and dynamic method binding matter only in PLs that support inheritance.
- Subtyping is about code reuse in clients, subclassing is about code reuse in classes.
- In C++, destructors implicitly free all memory allocated in the constructor.

Quiz: Data Abstraction

Which of the following is true?

- ~~Java enforces Liskov's substitutability principle.~~
- Static and dynamic method binding matter only in PLs that support inheritance.
- Subtyping is about code reuse in clients, subclassing is about code reuse in classes.
- ~~In C++, destructors implicitly free all memory allocated in the constructor.~~