Programming Paradigms

Lecture 14:

Data Abstraction and Object-Orientation (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

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;
}
```

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

Wake-up Exercise

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`

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

Wake-up Exercise

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`

Wake-up Exercise

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`

Wake-up Exercise

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`

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

Wake-up Exercise

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

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

Overview

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

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

- **Most common form of data abstraction in today's PLs**
- **Two kinds of members**
  - Data members a.k.a. fields
  - Subroutine members a.k.a. methods
- **Class hides its implementation from clients**
- **Code reuse via inheritance**

Object-Oriented Programming

- **Objects**
  - Instances of classes (in class-based PLs, e.g., Java, C++)
  - Primary entities (in prototype-based PLs, e.g., Smalltalk, JavaScript)
- **(Most) data stored in fields of objects**
- **Objects call other objects' methods**

A Bit of History

Simula

- Developed in 1960s in Norway by Dahl and Nygaard
- First OO language
- Classes, objects, inheritance

Smalltalk

- Developed at Xerox PARC by Alan Kay and others
- Message-based programming, dynamic typing

C++, Eiffel, Ada95, Java, C#

Overview

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

Encapsulation

- **Bundle related data with operations on the data**
  - Class members: Fields and methods
- **Instance-level vs. class-level**
  - Instance-level members: Specific to each individual object
  - Class-level members: Exist once for all objects of a class

Example

account.cpp

Information Hiding

- **Classes hide irrelevant details from their clients**
  - How the data is stored
  - How the behavior is implemented
- **Allows changing internals of a class without adapting the clients**

Getters and Setters

- **Hide details of how data is stored in fields**
- **Instead, access fields via**
  - Getter method: Returns the current value
  - Setter method: Set a new value
- **Clients read/write fields via getter/setter**

Properties/Accessors

- Special accessor methods for a field
- Transparent to clients: Looks like direct field access

```
// Example: C#
class Time {
  private double seconds;
  public double Hours {
    get { return seconds / 3600; }
    set {
      if (value < 0 || value > 24)
        // handle illegal argument
      seconds = value * 3600;
}}}
```

Properties/Accessors

- Special accessor methods for a field
- Transparent to clients: Looks like direct field access

```
// Example: C#
class Time {
  private double seconds;
  public double Hours {
    get { return seconds / 3600; }
    set {
      if (value < 0 || value > 24)
        // handle illegal argument
      seconds = value * 3600;
}}}
```

```
Time t = new Time();
t.Hours = 5;
double h = t.Hours;
```

Visibilities

- **Most class-based PLs provide visibilities for class members**
  - `private`: Visible only to the class itself
  - `public`: Visible to every client of the class
- **Expose members via `public` only when necessary**
  - Maximizes adaptability without affecting clients

Visibilities: PL Specifics

- **Java**
  - Default visibility: Visible in same package
  - `protected`: Visible in same package and all subclasses
- **C++**
  - `protected`: Visible in current class and subclasses
  - Friend classes: Can access private and protected members

Quiz: Encapsulation

Which of the following is true?

- Encapsulation bundles related data and operations, while hiding irrelevant details from clients.
- Clients of a class must adapt to how the class represents its internal data.
- `protected` means the same in Java and C++.
- Class-level members should always be public.

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

Quiz: Encapsulation

Which of the following is true?

- Encapsulation bundles related data and operations, while hiding irrelevant details from clients.
- ~~Clients of a class must adapt to how the class represents its internal data.~~
- ~~`protected` means the same in Java and C++~~.
- ~~Class level members should always be public.~~

Overview

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

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
  - Eiffel: Must explicitly rename inherited methods to make them available to clients

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

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

Alternatives to Inheritance

- **Inheritance: Is-a relation**
- **Instead, sometimes a Has-a relation is sufficient for code reuse**
  - Field with class to reuse
  - Forward calls to object stored in this field
  - E.g., reuse class `List` in class `Registrations`
    - Could inherit from `List` (store all registrations)
    - Instead: Field of type `List` in `Registrations`

Quiz: Inheritance

Where is the compilation error (and why)?

```
class A {
    protected:
    int f = 23;
    void foo() {}

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

Quiz: Inheritance

Where is the compilation error (and why)?

Error: `bar` is not visible

- B inherits A as `protected` class, hence, all members are at most `protected`
- Clients cannot call `protected` methods

```
class A {
    protected:
    int f = 23;
    void foo() {}

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

Overview

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

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
  - Declaring 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 example
class A { ... }

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

```
// 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

Example (Again)

```
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`

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

Overview

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