Programming Paradigms
Data Abstraction and Object-Orientation (Part 3)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Summer 2020
Overview

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

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

Please vote via Ilias.
Quiz

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

Implicitly creates object of class C

Please vote via Ilias.
Quiz

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

Class with two superclasses

Please vote via Ilia.
Quiz

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

Please vote via Ilias.
Quiz

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

Please vote via Ilias.
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
// 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 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) {
        ...
    }
}

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)
// 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
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;
}
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