Programming Language Concepts
Object Oriented Languages

Janyl Jumadinova

5–7 April, 2023

Three Properties of Object-Oriented Languages:

- Encapsulation
- Inheritance
- Dynamic method binding (polymorphism)

Encapsulation

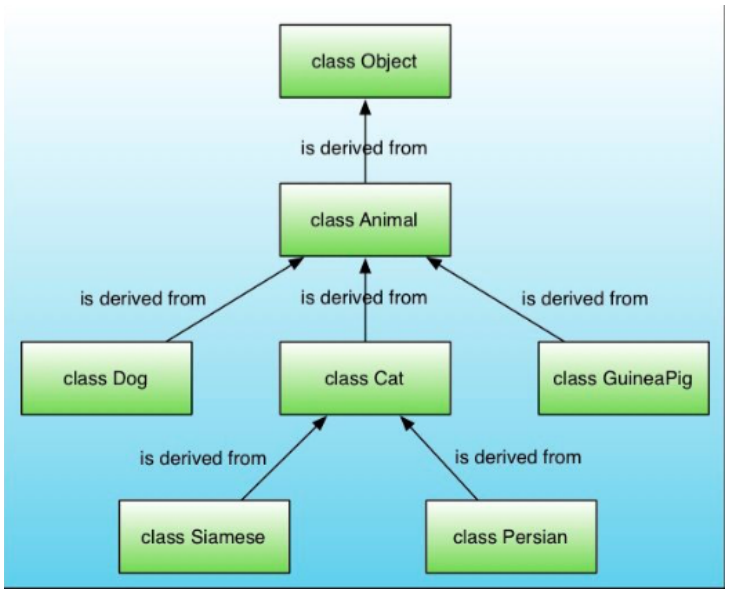
- Data and functions bound together into a single object.
- "Data hiding" – hide implementation details from user.
- More accurately, control access to data using public and private variables and methods.

Inheritance

- Hierarchy of classes and objects.
- Shared behaviors and data – code re-use.
- **Static polymorphism**: one interface for many kinds of object.

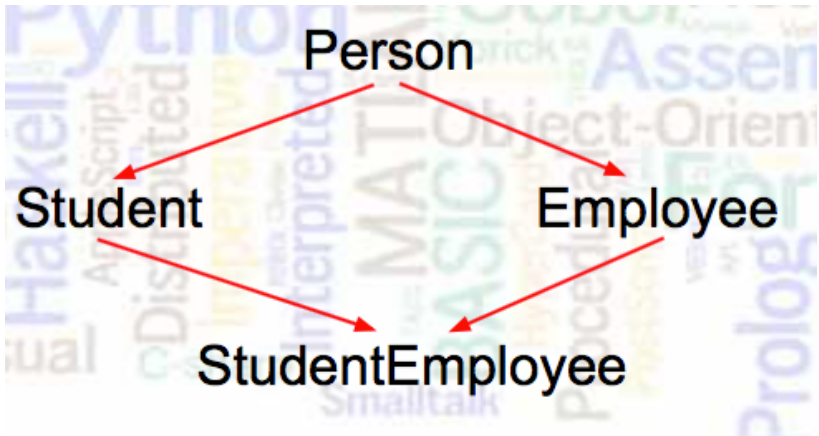
Multiple Inheritance

- In Java, classes and subclasses form a tree – a class may have many subclasses, but each subclass extends exactly one parent class. This is called the **class hierarchy**.
- Java does not permit "multiple inheritance."



Multiple Inheritance

- On the other hand, the C++ language allows classes to inherit from several different parent classes: multiple inheritance.
- For example, consider the following set of classes:



Multiple Inheritance in C++

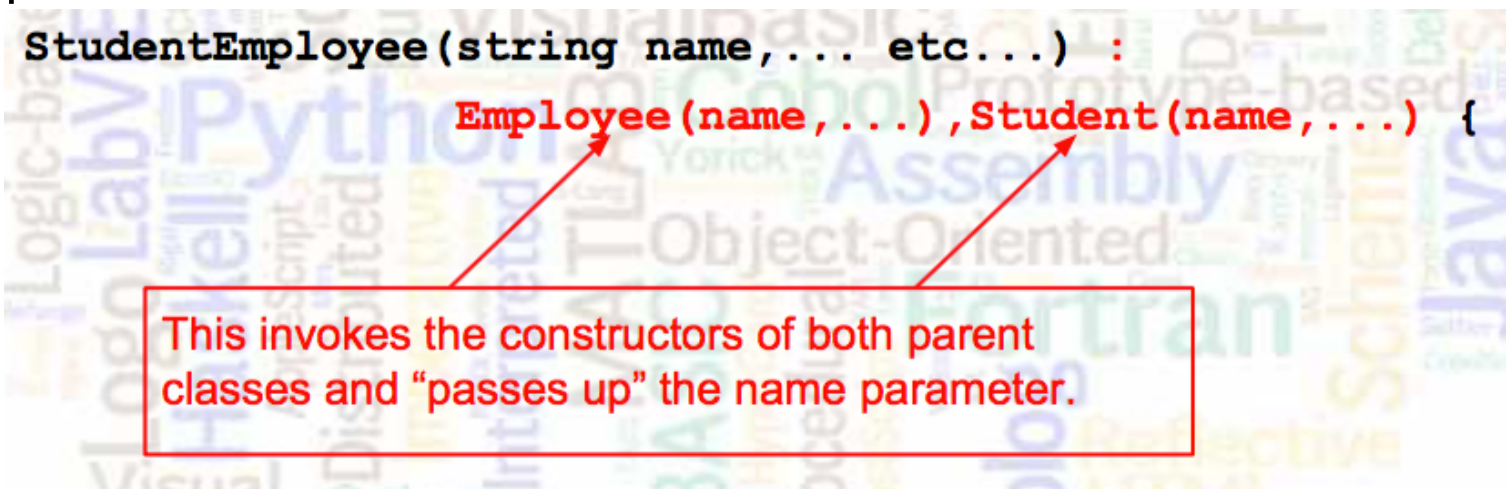
```
class Person { ... };
class Student : public Person {  ... };
class Employee : public Person {  ... };
class StudentEmployee : public Student, public Employee
{...};
```

In C++, constructors for subclasses can invoke the constructors of their parent classes, e.g.,

`Student(string name, int year, double gpa):` `Person(name)...`

Multiple Inheritance in C++

In our example, StudentEmployee can invoke the constructors of both parents:



Multiple Inheritance in C++

- In our example, assume `name` is an instance variable in the class `Person`, with accessor method `getName()`.
- Then both `Student` and `Employee` will inherit this variable as well as the `getName` method.

Multiple Inheritance in C++

- In our example, assume `name` is an instance variable in the class `Person`, with accessor method `getName()`.
- Then both `Student` and `Employee` will inherit this variable as well as the `getName` method.
- Now consider the class `StudentEmployee`. It inherits `name` and `getName` from both `Student` and `Employee`.

Multiple Inheritance in C++

- In our example, assume `name` is an instance variable in the class `Person`, with accessor method `getName()`.
- Then both `Student` and `Employee` will inherit this variable as well as the `getName` method.
- Now consider the class `StudentEmployee`. It inherits `name` and `getName` from both `Student` and `Employee`.

```
StudentEmployee joe("Joe Jones", ... etc. ...);
cout << joe.getName() << " graduates in " << ... etc. ...
```

```
error: request for member 'getName' is ambiguous
error: candidates are: std::string Person::getName()
error:                 std::string Person::getName()
```

Multiple Inheritance

- This is called the "diamond problem" (a.k.a. "Diamond of Death").
- In C++, one way to avoid the error on the previous page is to simply choose one of the "getName()" methods and ignore the other one.

Multiple Inheritance

```
class StudentEmployee: public Student, public Employee {
    ...
  string getName() { return Student::getName(); }
    ...
};
```

We explicitly name one of the two conflicting "getName" methods in the StudentEmployee class.

Multiple Inheritance

Can we gain the benefits of multiple inheritance in Java?

- Sort of ... in Java we can create “interfaces”.
- They are similar to classes, but an interface has no instance variables and contains only abstract methods.
- A class can implement more than one interface.
- It is not quite the same as multiple inheritance, but yields many of the same benefits.

Dynamic Method Binding

An object's methods are determined at runtime rather than compilation time, since subclasses can override methods and can be used wherever the superclass is allowed.

Dynamic Binding

```
class Super {
  public String talk() {
    return "hello";}
  }
}
class Sub extends Super {
  public String talk() {
    return "goodbye";
  }
}
```

```
...
Super x = new Super();
Sub y = new Sub();
f(x); f(y);
...
public static void f(Super a) {
  System.out.println(a.talk());
}
```

What gets printed when we call f(y)?

Dynamic Binding

```
class Super {
  public String talk() {
    return "hello";}
  }
}
class Sub extends Super {
  public String talk() {
    return "goodbye";
  }
}
```

```
...
Super x = new Super();
Sub y = new Sub();
f(x); f(y);
...
public static void f(Super a) {
  System.out.println(a.talk());
}
```

What gets printed when we call f(y)?

`DynamicDemo.java` in the class activities repo.

Dynamic Binding

```
class Super {
  public String talk() {
    return "hello";}
  }
}
class Sub extends Super {
  public String talk() {
    return "goodbye";
  }
}
```

```
...
Super x = new Super();
Sub y = new Sub();
f(x); f(y);
...
public static void f(Super a) {
  System.out.println(a.talk());
}
```

What gets printed when we call f(y)?

`DynamicDemo.java` in the class activities repo.

At runtime, Java determined the correct class of the parameter and invoked y's "talk" method. This is dynamic method binding.

Static vs. Dynamic Binding

- See program `staticbind.cpp` in the class activities repo.
- There is a parent class `Super` and a child class `Sub`, each with a `get()` method.

```
Sub a(10,20);
    Super b = a;
    cout << a.get() << endl; // Which "get"?
    cout << b.get() << endl; // Which "get"?
```

Both will use Super's "get()" method!

Static vs. Dynamic Binding

- By default, C++ uses static binding.
- However, you can still obtain the same behavior as dynamic binding by using virtual methods and pointers as shown in program `dynamicbind.cpp`

Overloading

- When two methods in a class have the same name but different parameters, we say that the method name is “overloaded.”

Overloading

- When two methods in a class have the same name but different parameters, we say that the method name is "overloaded."
- This is familiar from Java (where, for instance, we have two different "substring" methods for the String class or multiple constructor methods).
- In C++ we can even overload symbolic operators like "+" and "*" (really, any operator).

Overloading an Operator in C++

Overload.cpp

```
class Pirate {
  public:
    Pirate(string name) { this->name = name;}
    Pirate operator +(Pirate p) {
      return Pirate(p.getName() + name);
}
  ... some code omitted ...
...
  Pirate x("Fred");
  Pirate y("Mary");
  Pirate a = x + y; // Creates a Pirate named "MaryFred"
```

Overloading an Operator in C++

- There are many aspects of operators that we must worry about: precedence, associativity, etc.
- C++ avoids these by forcing overloaded operators to have the same precedence and associativity that the original operators had.

Overloading an Operator in C++

- There are many aspects of operators that we must worry about: precedence, associativity, etc.
- C++ avoids these by forcing overloaded operators to have the same precedence and associativity that the original operators had.
- Inspect programs `overload1.cpp` and `overload2.cpp` in the class activities repo and write your observations as comments.