

Inheritance

Extending Classes



SoftUni Team
Technical Trainers
Software University
<http://softuni.bg>

*Java OOP
Basics*

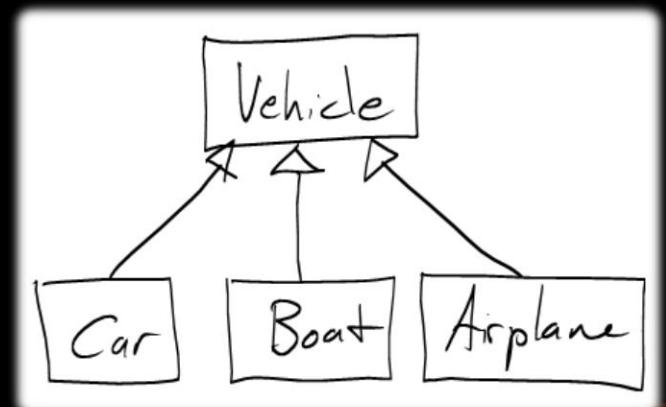


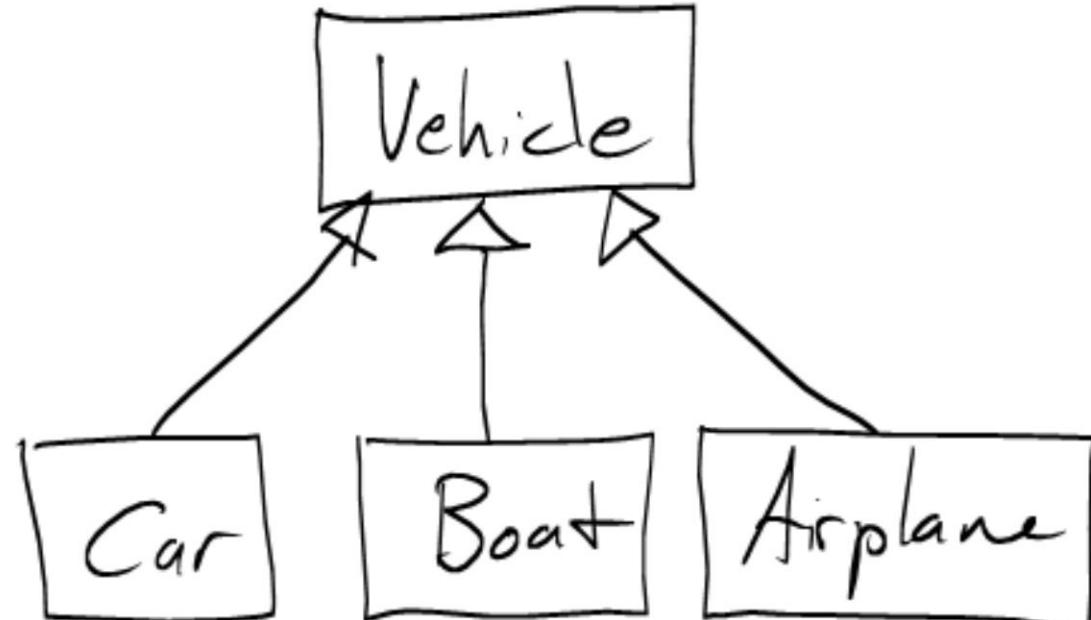
Table of Contents

1. Inheritance
2. Class Hierarchies
3. Inheritance in Java
4. Accessing Members of the Base Class
5. When to Use Inheritance
6. Composition



sli.do

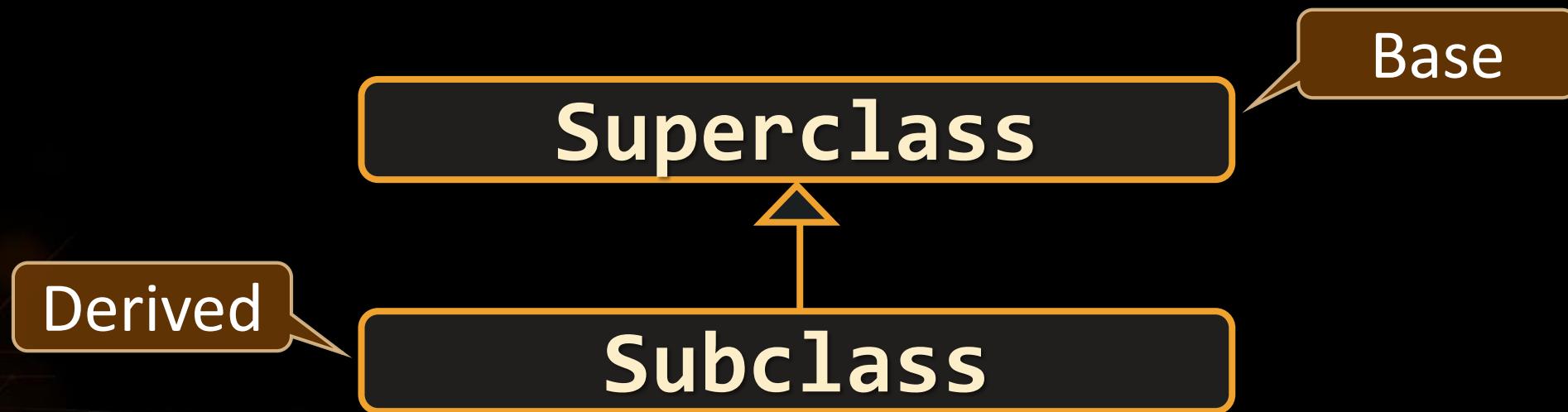
#JavaFundamentals



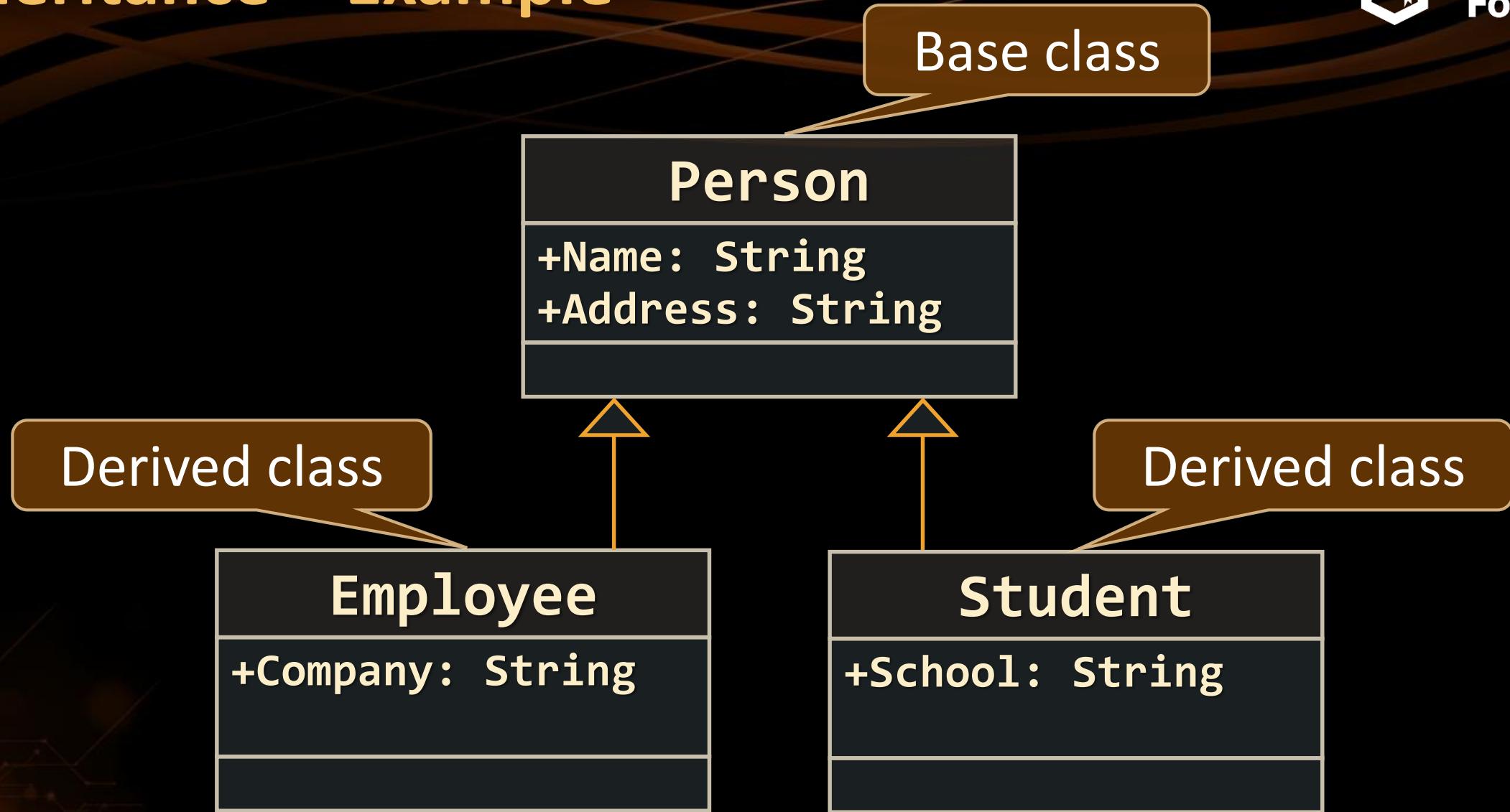
Inheritance

Extending Classes

- **Superclass** - Parent class, Base Class
 - The class giving its members to its child class
- **Subclass** - Child class, Derived Class
 - The class taking members from its base class

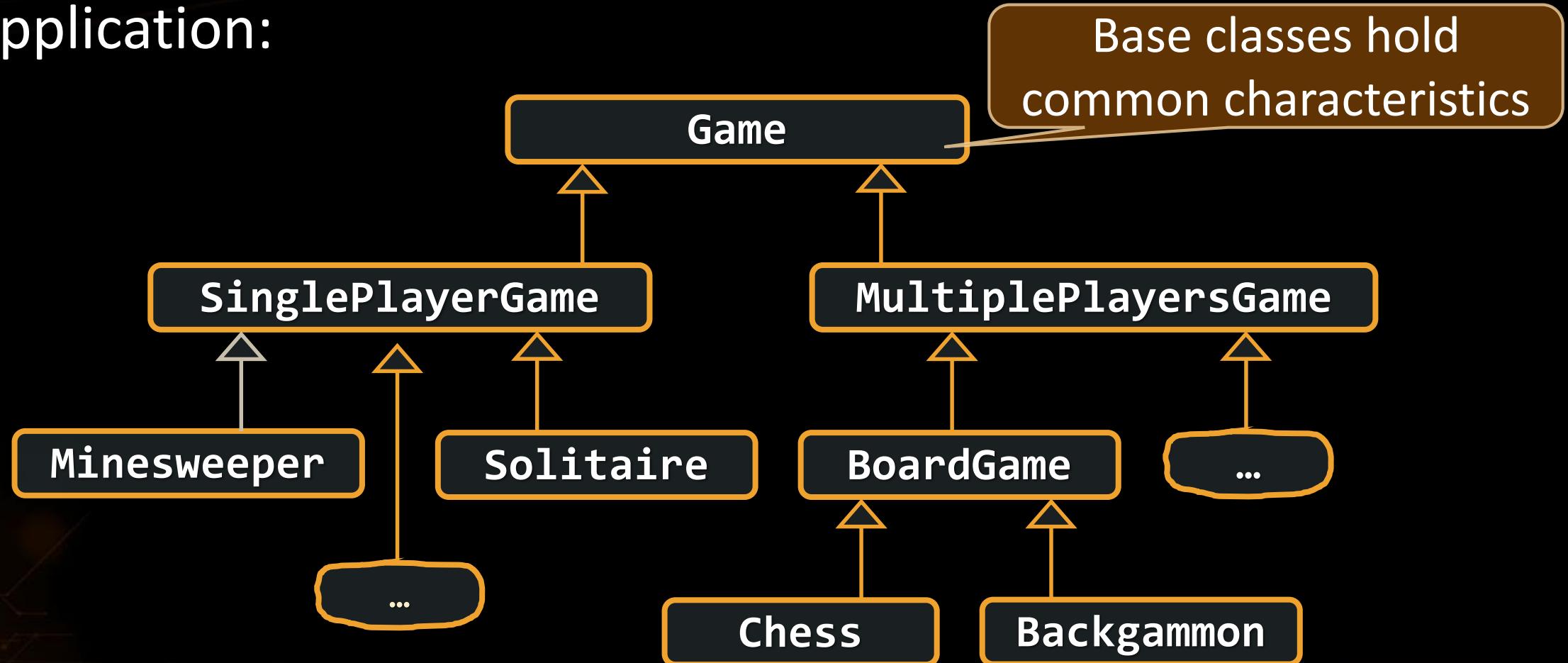


Inheritance – Example

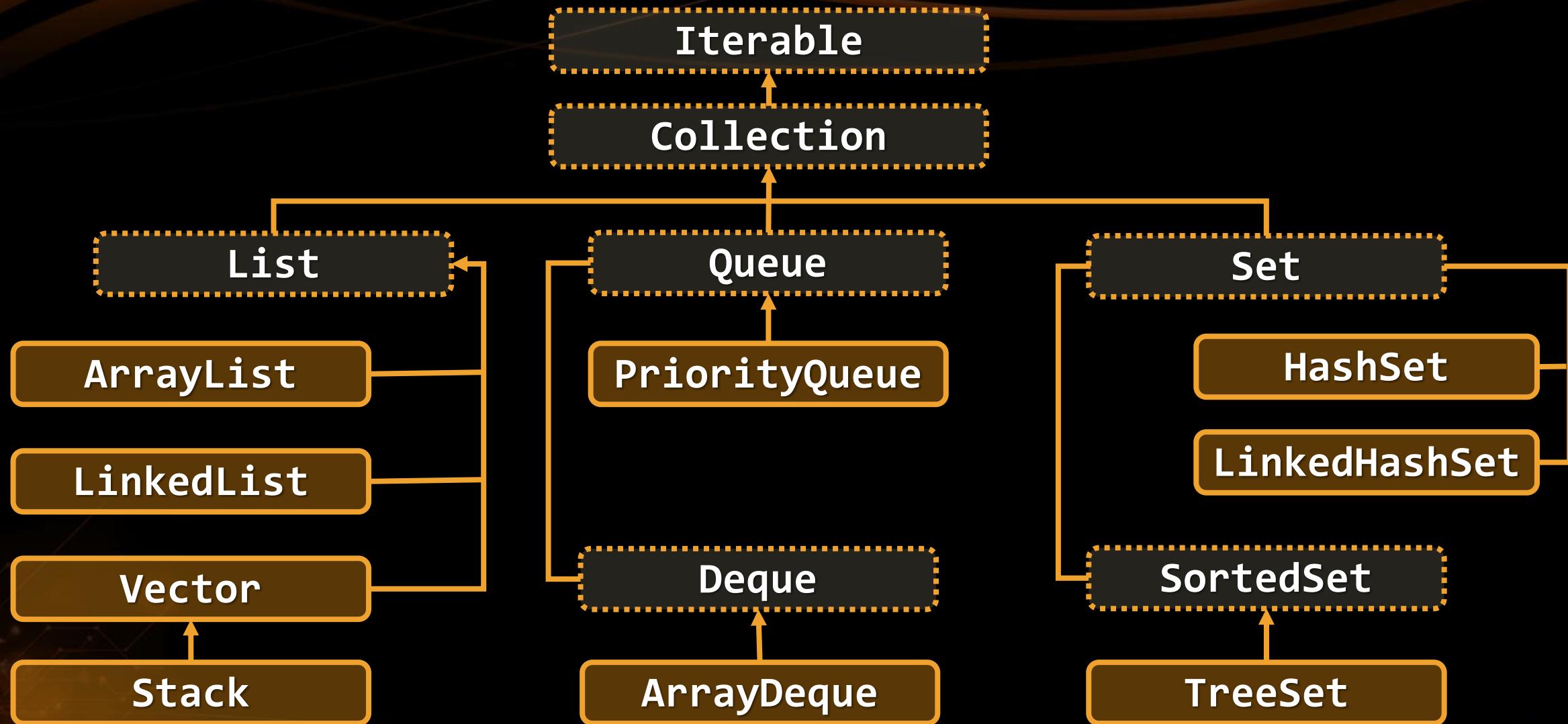


Class Hierarchies

- Inheritance leads to **hierarchies** of classes and/or interfaces in an application:

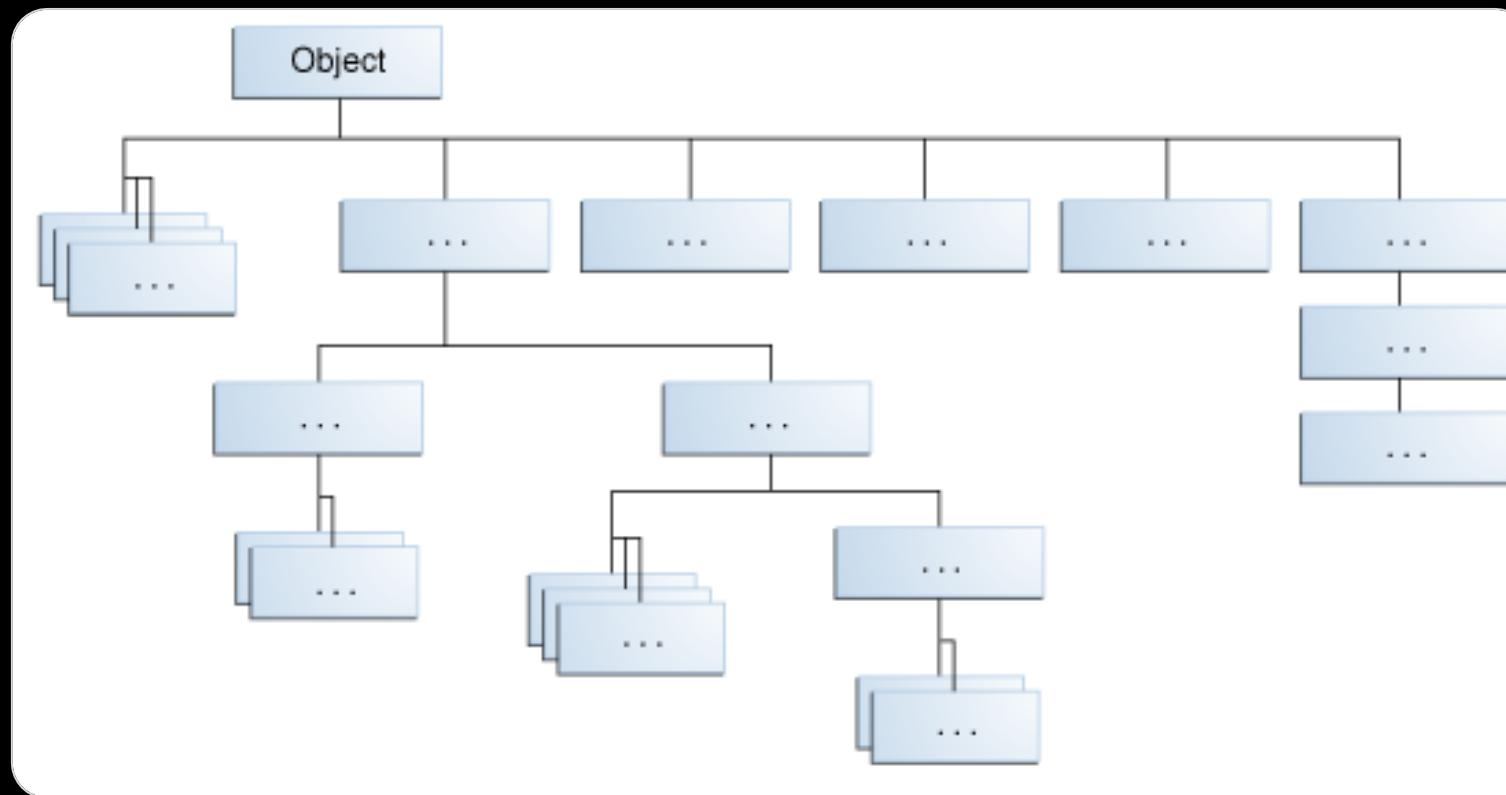


Class Hierarchies – Java Collection



Java Platform Class Hierarchy

- **Object** is at the root of Java Class Hierarchy



Inheritance in Java

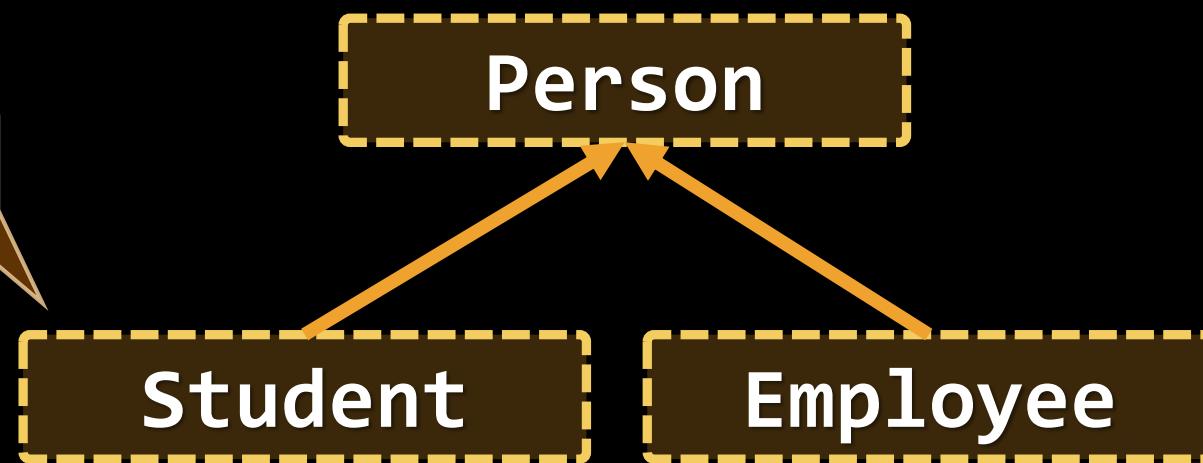
- Java supports inheritance through **extends** keyword

```
class Person { ... }
```

```
class Student extends Person { ... }
```

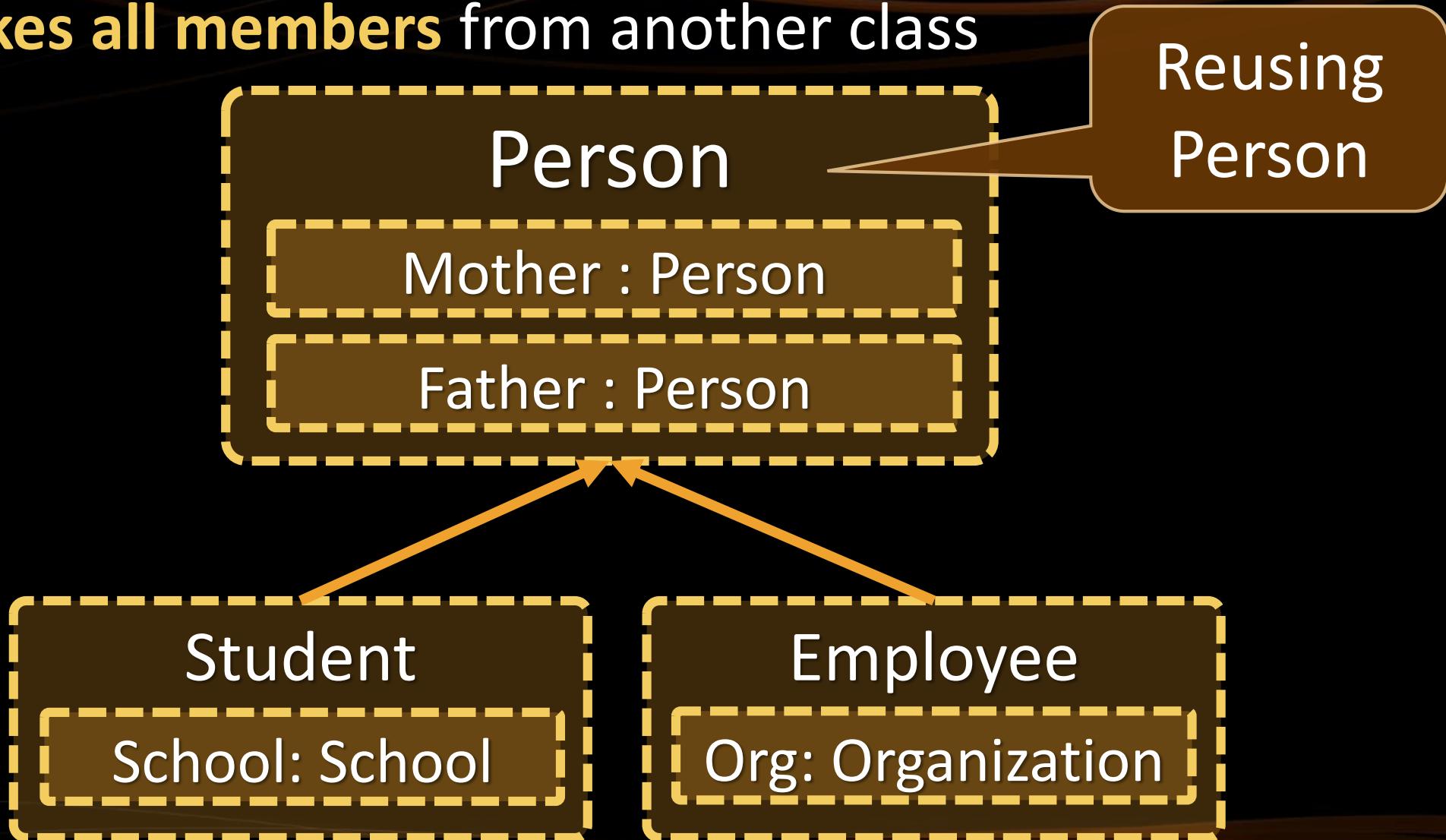
```
class Employee extends Person { ... }
```

Student
extends person



Inheritance - Derived Class

- Class **takes all members** from another class



Using Inherited Members

- You can access inherited members as usual

```
class Person { public void sleep() { ... } }
class Student extends Person { ... }
class Employee extends Person { ... }
```

```
Student student = new Student();
student.sleep();
Employee employee = new Employee();
employee.sleep();
```

Reusing Constructors

- Constructors are **not inherited**
- Constructors **can be reused** by the child classes

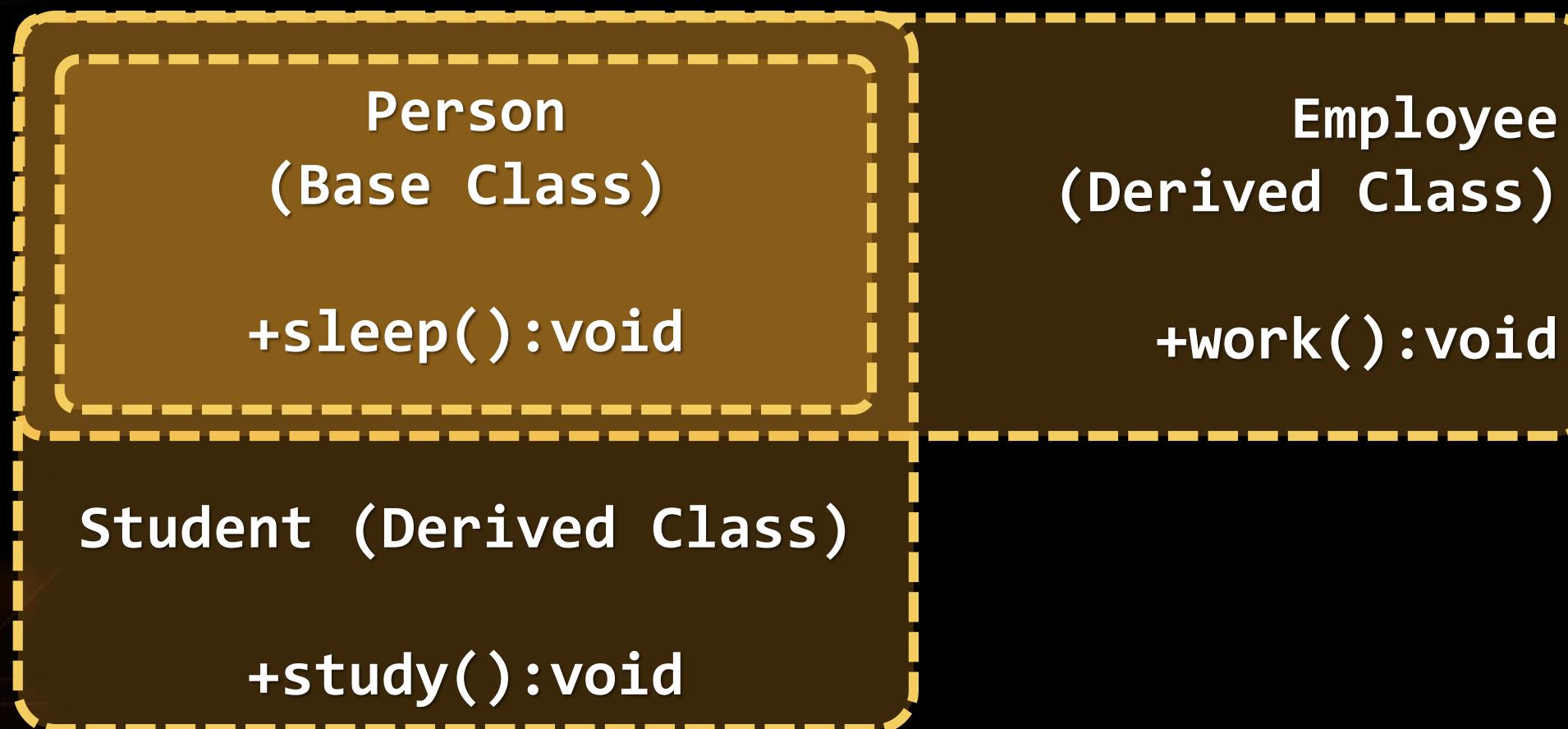
```
class Student extends Person {  
    private School school;  
    public Student(String name, School school) {  
        super(name);  
        this.school = school;  
    }  
}
```



Constructor call
should be first

Thinking About Inheritance - Extends

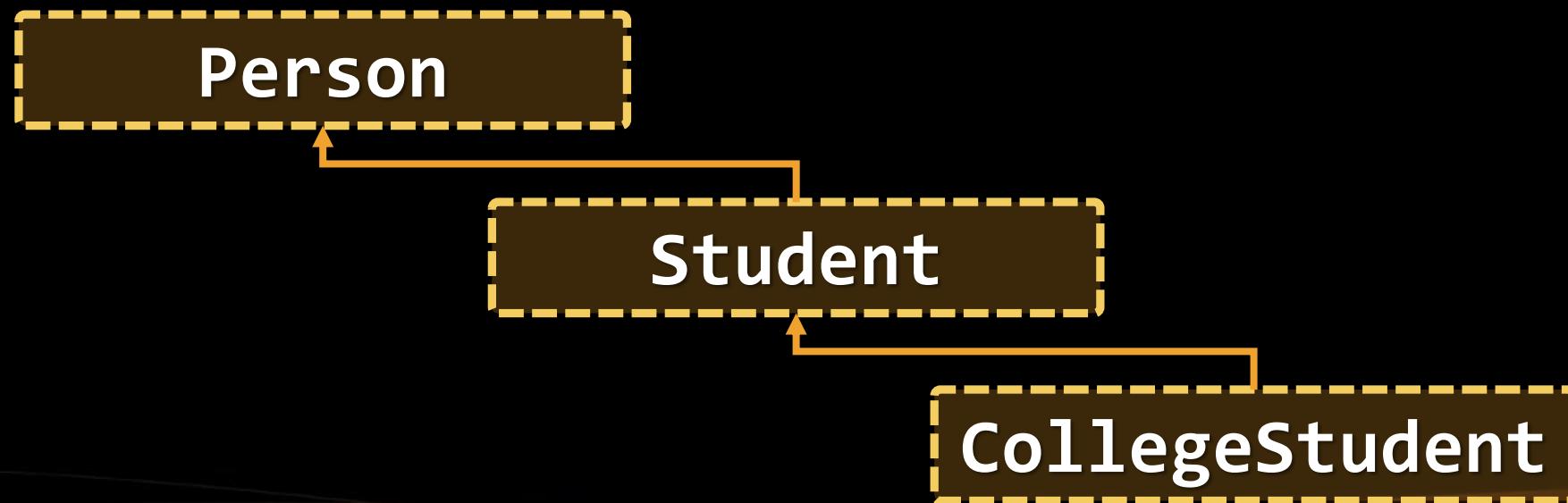
- Derived class instance **contains** instance of its base class



Inheritance

- Inheritance has a **transitive relation**

```
class Person { ... }  
class Student extends Person { ... }  
class CollegeStudent extends Student { ... }
```



Multiple Inheritance

- In Java there is no **multiple** inheritance
- Only **multiple interfaces can be implemented**



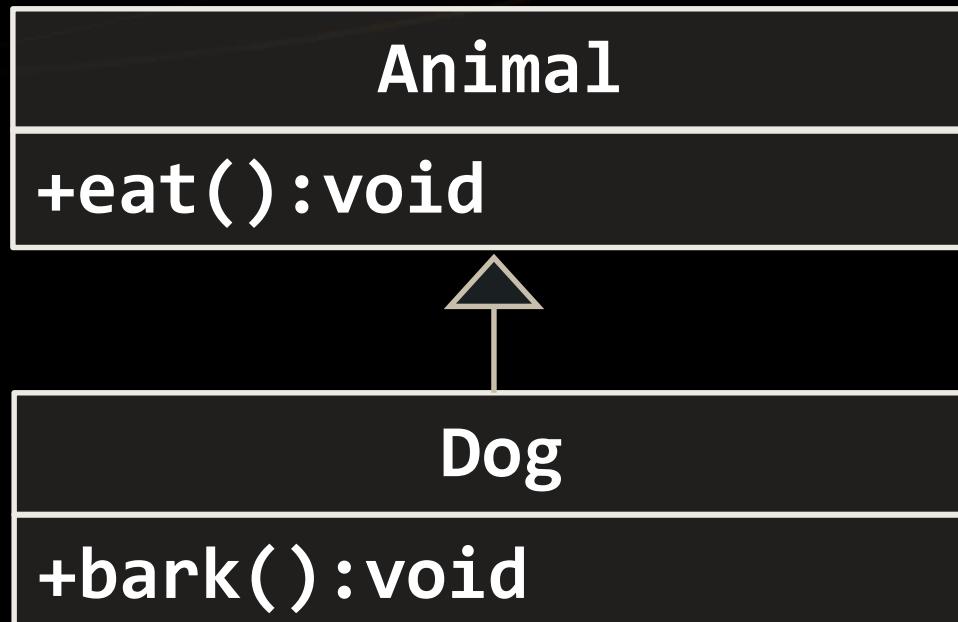
Access to Base Class Members

- Use the **super** keyword

```
class Person { ... }
```

```
class Employee extends Person {
    void fire(String reasons) {
        System.out.println(
            super.name +
            " got fired because " + reasons);
    }
}
```

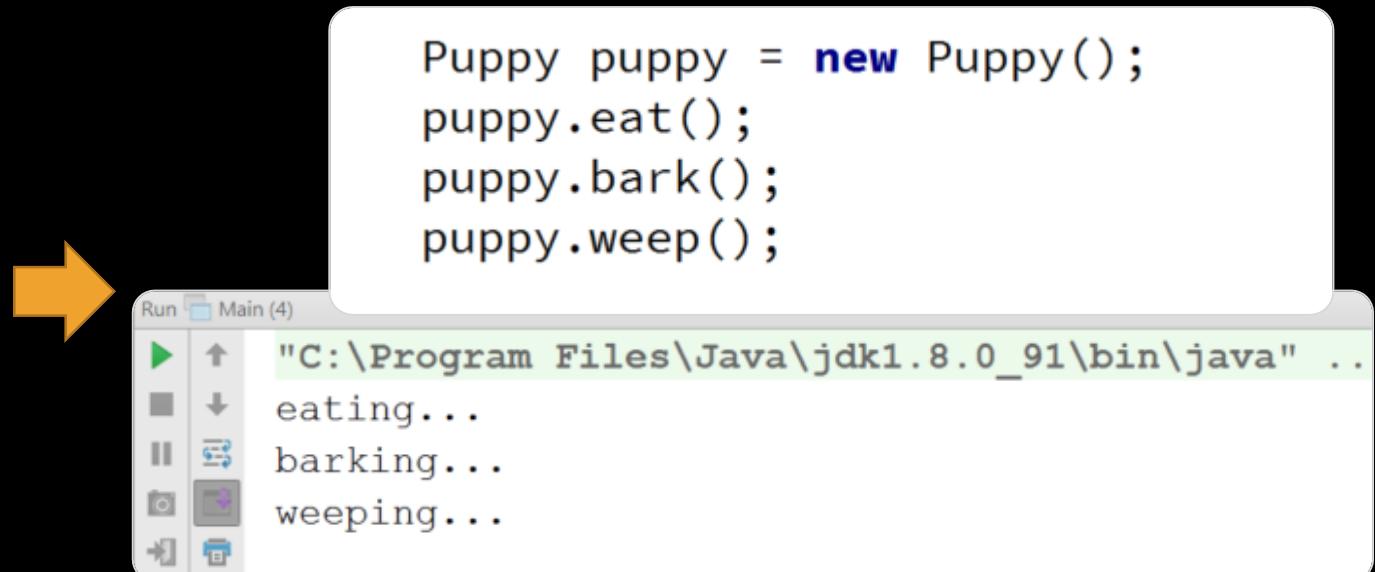
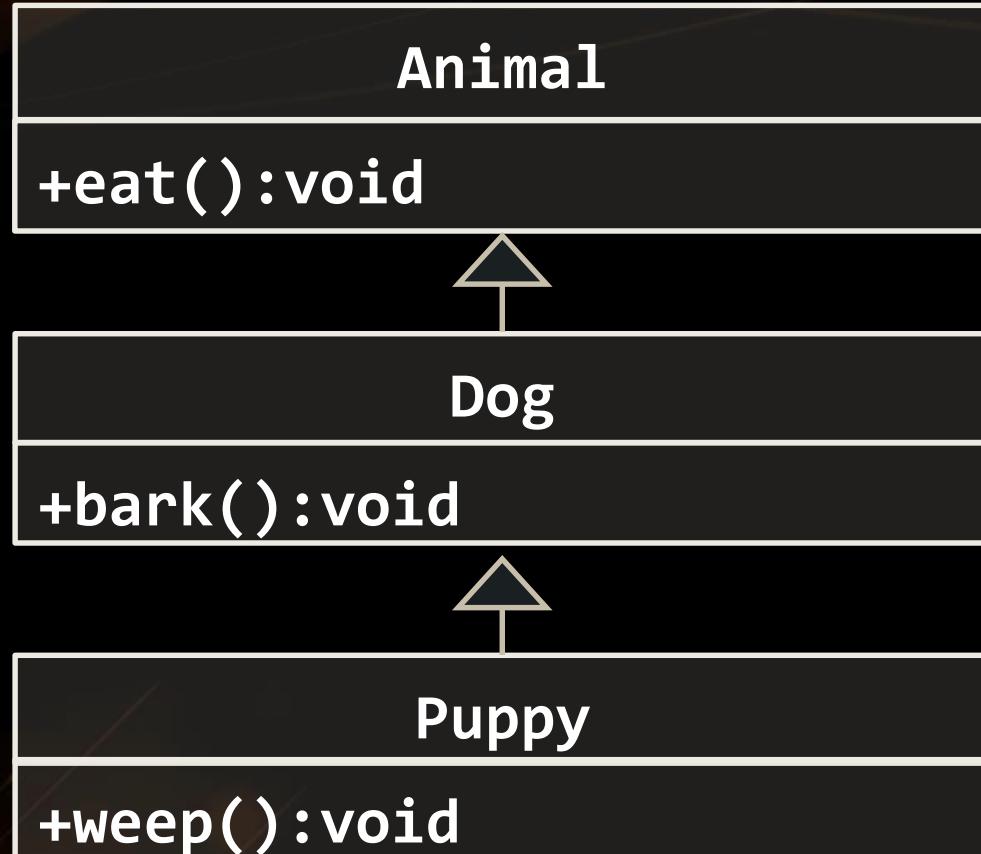
Problem: Single Inheritance



```
public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat();
    dog.bark();
}
```

```
Run Main (2)
C:\Program Files\Java\jdk1.8.0_91\bin\java ...
↑
↑ eating...
↓
barking...
```

Problem: Multilevel Inheritance



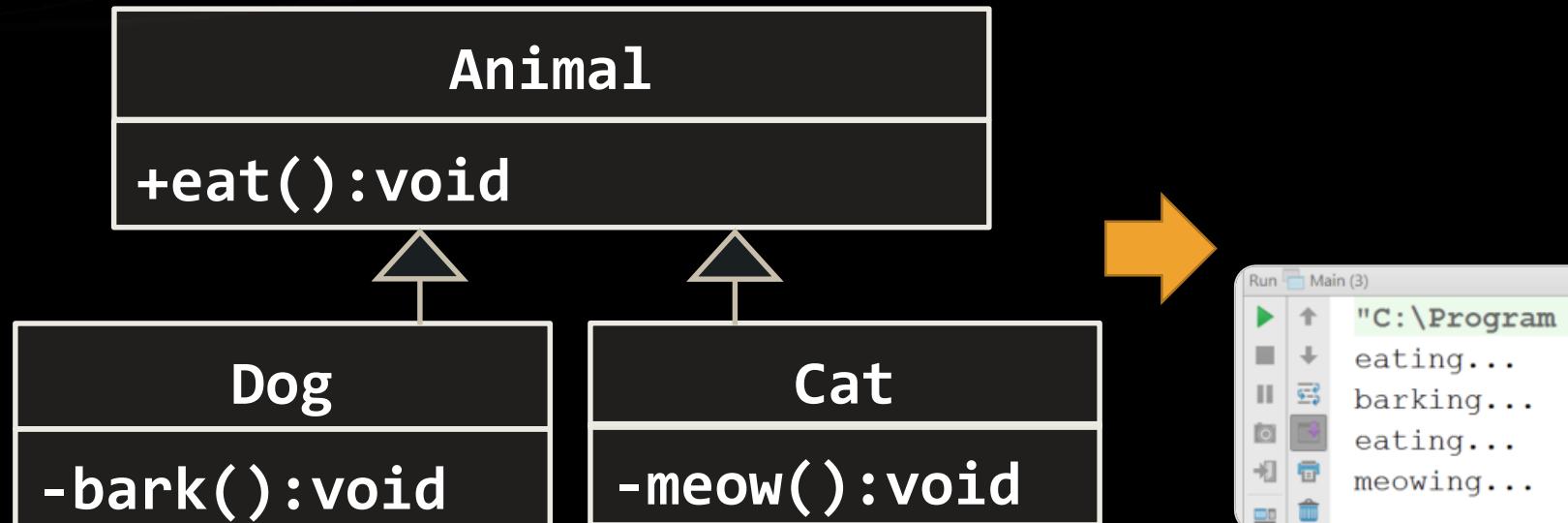
```
Puppy puppy = new Puppy();
puppy.eat();
puppy.bark();
puppy.weep();
```

Run Main (4)

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
↑
↓
|| 
|| 
|| 
|| 
|| 
||
```

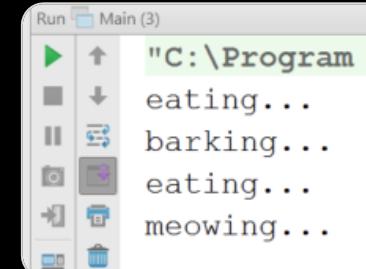
eating...
barking...
weeping...

Problem: Hierarchical Inheritance



```
Dog dog = new Dog();
dog.eat();
dog.bark();

Cat cat = new Cat();
cat.eat();
cat.meow();
```



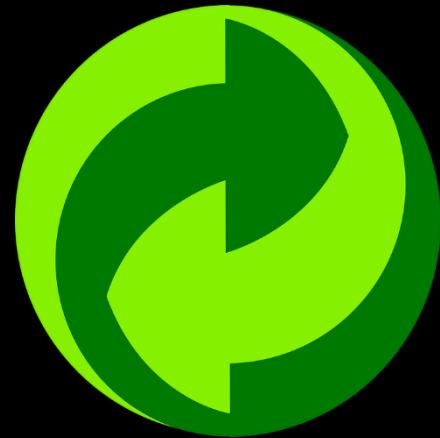
A screenshot of a Java IDE's run log window. It shows the output of the code above. The log entries are:

- "C:\Program ..."
- eating...
- barking...
- eating...
- meowing...



Inheritance

Live Exercises in Class (Lab)



Reusing Classes

Reusing Code at Class Level

- Derived classes **can access all public and protected members**
- Derived classes can access **default** members **if in same package**
- **Private** fields **aren't inherited** in subclasses (can't be accessed)

```
class Person {  
    private String id;  
    String name;  
    protected String address;  
    public void sleep();  
}
```

can be accessed through
other methods

Shadowing Variables

- Derived classes **can hide** superclass variables

```
class Person { protected int weight; }
```

```
class Patient extends Person {  
    protected float weight;  
    public void method() {  
        double weight = 0.5d;  
    }  
}
```

hides **int weight**

hides both

Shadowing Variables - Access

- Use **super** and **this** to specify member access

```
class Person { protected int weight; }
```

```
class Patient extends Person {  
    protected float weight;  
    public void method() {  
        double weight = 0.5d;  
        this.weight = 0.6f;           Local variable  
        super.weight = 1;            Instance member  
    }                            Base class member  
}
```

Overriding Derived Methods

- A **child class** can redefine existing methods

```
public class Person {  
    public void sleep()  
    { sout("Person sleeping"); }  
}
```

Method in base class
must not be **final**

```
public class Student extends Person {  
    @Override public void sleep()  
    { sout("Student sleeping"); }  
}
```

Signature and return
type should match

Final Methods

- **final** – defines a method that **can't be overridden**

```
public class Animal {  
    public final void eat() { ... }  
}
```

```
public class Dog extends Animal {  
  
    @Override  
    public void eat() {} // Error...  
}
```

Final Classes

- Inheriting from a final classes is forbidden

```
public final class Animal {  
    ...  
}
```

```
public class Dog extends Animal { } // Error...  
public class MyString extends String { } // Error...  
public class MyMath extends Math { } // Error...
```

Problem: Fragile Base Class

- Classes: Animal, Predator, Food
- When Predator feeds, gains +1 health

Empty class

protected

@Override
maybe?



Predator

Solution: Fragile Base Class (Fragile)

```
public class Animal {  
    private List<Food> foodEaten;  
  
    public eat(Food food)  
    { foodEaten.add(food); }  
  
    public eatAll(Food[] food)  
    { for (Food f : food) { eat(f); } }  
}
```

In case of change, can
break subclasses

Solution: Fragile Base Class

```
public class Animal {  
    protected List<Food> foodEaten;  
    Safe to make changes  
    public final eat(Food food)  
    { foodEaten.add(food); }  
  
    public final eatAll(Food[] food)  
    { for (Food f : food) { eat(f); } }  
}
```

Inheritance Benefits - Abstraction

- One approach for providing abstraction

Focus on common properties

```
Person person = new Person();  
Student student = new Student();
```

```
List<Person> people = new ArrayList();
```

```
people.add(person);  
people.add(student);
```

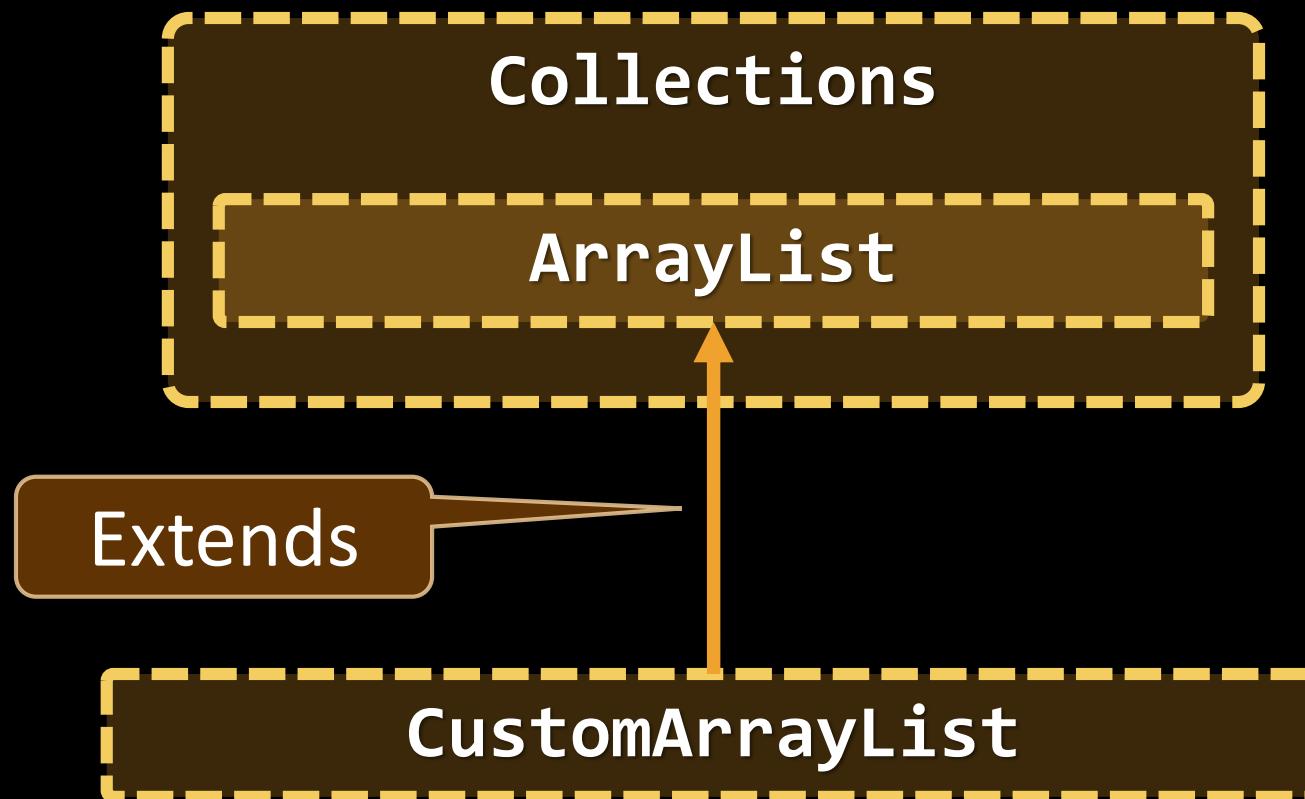
Polymorphism

Person (Base Class)

Student (Derived Class)

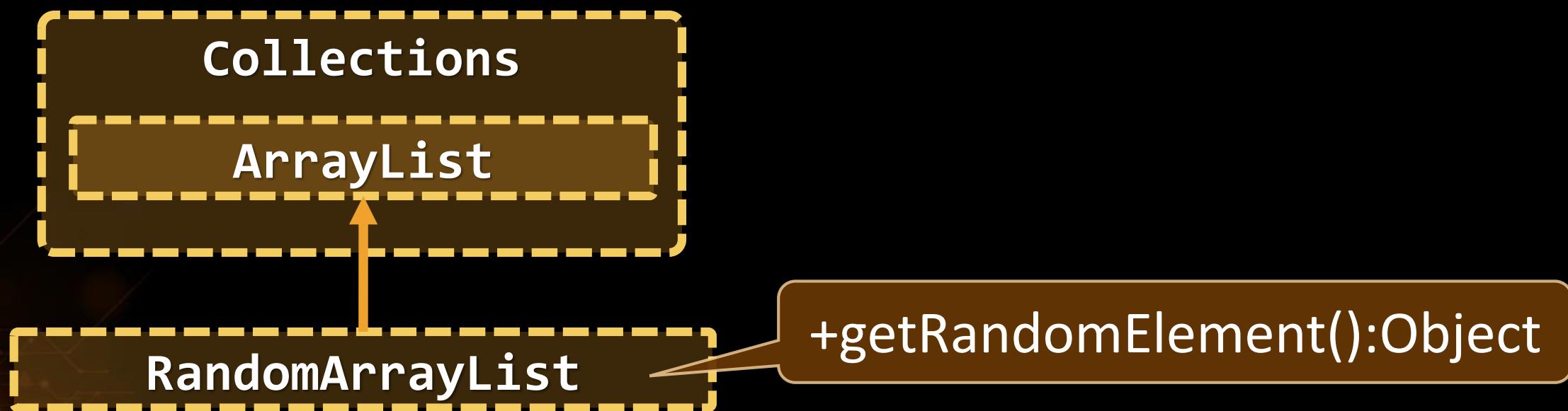
Inheritance Benefits – Extension

- We can **extend a class** that we **can't otherwise change**



Problem: Random Array List

- Create an array list that has
 - All functionality of an ArrayList
 - Function that returns and removes a random element



Solution: Random Array List



```
public class RandomList extends ArrayList {  
    private Random rnd; // Initialize this...  
  
    public Object getRandomElement() {  
        int index = rnd.nextInt(super.size());  
        swap(index, super.size() - 1);  
        return super.remove(super.size() - 1);  
    }  
}
```



Types of Class Reuse

Extension, Composition, Delegation

Extension

- **Duplicate code** is error prone
- **Reuse classes** through **extension**
- Sometimes the only way



Composition

- Using classes to define classes

```
class Laptop {  
    Monitor monitor;  
    Touchpad touchpad;  
    Keyboard keyboard;  
    ...  
}
```

Reusing
classes



Delegation

```
class Laptop {  
    Monitor monitor;  
    void incrBrightness() {  
        monitor.brighten();  
    }  
  
    void decrBrightness() {  
        monitor.dim();  
    }  
}
```

Laptop

Monitor

increaseBrightness()
decreaseBrightness()

Problem: Stack of Strings

- Create a simple Stack class which can store only strings

StackOfStrings

-data: List<String>

+push(String) :void

+pop(): String

+peek(): String

+isEmpty(): boolean

StackOfStrings

ArrayList

```
StackOfStrings sos = new StackOfStrings();
sos.push("one");
System.out.println(sos.pop());
System.out.println(sos.isEmpty());
System.out.println(sos.peek());
```

Solution: Stack of Strings

```
public class StackOfStrings {  
    private List<String> container;  
  
    public void push(String item)  
    { container.add(item); }  
  
    public String pop()  
    { container.remove(container.size() - 1); }  
}
```

TODO: Validate if
list is not empty

When to Use Inheritance

- Classes share **IS-A** relationship Too simplistic
- Derived class **IS-A-SUBSTITUTE** for the base class
- Share the **same role**
- Derived class is the **same as the base class** but adds a **little bit more functionality**



Reusing Classes

Live Exercises in Class (Lab)

Summary

- Inheritance is a powerful tool for **code reuse**
- **Subclass inherits** members from **Superclass**
- Subclass can **override** methods
- Look for classes with the **same role**
- Look for **IS-A** and **IS-A-SUBSTITUTE** for relationship
- Consider **Composition** and **Delegation** instead



Inheritance



Questions?



Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

