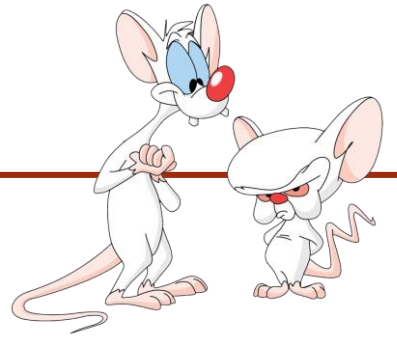
COMP 250

INTRODUCTION TO COMPUTER SCIENCE

Week 3-1 : OOP: Interfaces and Generics

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Interfaces (disclaimer: we'll talk about interfaces pre Java 8)
- Generics

INTERFACES

- `interface` is a reserved keyword in Java.

- Like classes, interfaces can be declared to be public or package-private.

- Similarly to classes, interfaces can have fields and methods but the following restrictions apply:

  - All methods are by default public and abstract.
  - All fields are by default public, static, and final.

- Interfaces cannot be instantiated.

SYNTAX

- We declare an interface using the `interface` keyword.


```
public interface myInterface {
    :
}
```


- An interface is implicitly abstract. You do not need to use the `abstract` keyword while declaring an interface.

EXAMPLE

```
public interface MonsterLike {

    public int spook();
    public void runAway();

}
```


- The methods are all implicitly abstract.

INHERITANCE

- To use an interface you first need a class that *implements* it. Interfaces specify what a class must do and not how. It is the blueprint of the class.

- A class can *implement* one or more interfaces using the keyword `implements`. Interfaces are used to achieve subtyping!

- If a class implements an interface and does not implement all methods specified by the interface then that class must be declared `abstract`.

- It is possible for a Java interface to *extend* another Java interface, just like classes *extend* other classes. You specify inheritance using the `extends` keyword.

IMPLEMENTS

```
public class Dragon implements MonsterLike {
   :
}
```


- **Inside the class** `Dragon`**, the methods** `spook()` **and** `runAway()` **must be implemented!**
- Note: if the interfaces are not located in the same packages as the implementing class, you will also need to import the interfaces. Java interfaces are imported using the import statements just like Java classes.

INTERFACE INSTANCES

- Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface


```
public interface MonsterLike {
    public int spook();
    public void runAway();
}
```

```
public class Orc implements MonsterLike {
    :
}
```

```
public class Dragon implements MonsterLike {
    :
}
```

```
public class Hero {
    public double fight(MonsterLike m) {
        :
    }
}
```

```
Hero frodo = new Hero();
MonsterLike thrall = new Orc();
Dragon drogon = new Dragon();
frodo.fight(thrall);
frodo.fight(drogon);
```

EXTENDS + IMPLEMENTS

Classes can extend at most one class, but they can implement multiple interfaces.

Example:


```
public class Dragon extends Enemy implements MonsterLike, FireBreather {
    :
}
```

`Dragon` **is a subtype of (at least)** `Enemy`, `MonsterLike`, **and** `FireBreather`**. An instance of** `Dragon` **can be used whenever an object of those types is required.**

INTERFACES VS ABSTRACT CLASSES

ABSTRACT CLASS	INTERFACE
Not all methods have to be abstract.	All methods are abstract by default.
The abstract keyword must be directly use to declare a class to be abstract.	Interfaces are implicitly abstract.
Can contain methods that have been implemented as well as instance variables.	No method can be implemented and only constants (final static fields) can be declared.
Abstract classes are useful when some general methods should be implemented and specialization behavior should be implemented by child classes.	Interfaces are useful in a situation that all properties should be implemented.

POST JAVA 8

From Java 8/9 onwards, interfaces can also contain the following

- Default methods
- Static methods
- Private methods
- Private Static methods

WORKING TOWARD GENERICS

- Suppose I'd like to create a class that defines a new type `Cage`. I would like to use this in a class called `Kennel` where I have a bunch of objects of type `Dog`.
- What if later on I also happen to need cages for objects of type `Bird`?
- Can I use the same class? Should I create a new class with the same features but where instead of `Dog` I use `Bird`? Is there a better solution?


```
public class Cage {
    private Dog occupant;

    public void lock(Dog p) {
        this.occupant = p;
    }

    public Dog peek() {
        return this.occupant;
    }

    public void release() {
        this.occupant = null;
    }
}
```

GENERICS IN JAVA

- A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter.
- Example →


```
public class Cage<T> {
    private T occupant;

    public void lock(T p) {
        this.occupant = p;
    }

    public T peek() {
        return this.occupant;
    }

    public void release() {
        this.occupant = null;
    }
}
```

EXAMPLE – CAGE<>

We can now create cages containing different type of objects, depending on the need:

```
Cage<Dog> crate = new Cage<Dog>();
// now inside crate we can lock only Dogs!
Dog snoopy = new Dog();
crate.lock(snoopy);

Cage<Bird> birdcage = new Cage<Bird>();
// if we call lock on birdcage we must provide a Bird as input.
Bird tweety = new Bird();
birdcage.lock(tweety);

// peek() called on crate returns a Dog,
// peek() called on birdcage returns a Bird!
Dog d = crate.peek();
Bird b = birdcage.peek();
```

GENERICS TYPE NAMING CONVENTIONS

- Java Generic Type Naming convention helps us understand code easily.

- Usually type parameter names are single uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:
  - E – Element
  - K – Key (Used in Map)
  - N – Number
  - T – Type
  - V – Value (Used in Map)
  - S,U,V etc. – 2nd, 3rd, 4th types

- More about generic type: https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html https://docs.oracle.com/javase/tutorial/java/generics/bounded.html

BOUNDED TYPE

- Sometimes we might want to restrict the types that can be used to create a `Cage` (or a general parameterized type)
- We can do that using the keyword `extends`. In this context, `extends` is used to mean either "extends" (as in classes) or "implements" (as in interfaces)
- Not only this will limit the types we can use to instantiate a generic type, but it will also allow us to use methods defined in the bounds.

BOUNDED TYPE

- **Example:**

```
public class Cage<T extends MonsterLike> {
    private T occupant;

    public T peek() {
        this.occupant.spook();
        return this.occupant;
    }

    public void release() {
        this.occupant = null;
        this.occupant.ranAway();
    }
}
```

EXAMPLE – LIST INTERFACE

java.util

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

EXAMPLE – LIST INTERFACE

```
public interface List<E> extends Collection<E>{
    boolean add(E e);
    void add(int i, E e);
    boolean isEmpty();
    E get(int i);
    E remove(int i);
    int size();
    :
}
```


Some of the methods are inherited from the interface Collection, while others are declared inside List.

EXAMPLE – LIST INTERFACE

The documentation explains exactly how each of these method should behave. For example:

add

```
boolean add(E e)
```

Appends the specified element to the end of this list (optional operation).

Lists that support this operation may place limitations on what elements may be added to this list. In particular, some lists will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. List classes should clearly specify in their documentation any restrictions on what elements may be added.

Specified by:

`add` in interface `Collection<E>`

Parameters:

`e` - element to be appended to this list

Returns:

`true` (as specified by `Collection.add(E)`)

https://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-

EXAMPLE – LIST INTERFACE

The documentation explains exactly how each of these method should behave. For example:

add

```
boolean add(E e)
```

Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add `null` elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning `false`). This preserves the invariant that a collection always contains the specified element after this call returns.

Parameters:

`e - element whose presence in this collection is to be ensured`

Returns:

`true if this collection changed as a result of the call`

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#add-E-

EXAMPLE – ARRAYLIST

```
public class ArrayList<E> implements List<E>{
    boolean add(E e) {...}
    void add(int i, E e) {...}
    boolean isEmpty() {...}
    E get(int i) {...}
    E remove(int i) {...}
    int size() {...}
    void ensureCapacity(int i) {...}
    void trimToSize() {...}
}
```


All of the methods inherited from `List` are implemented. In addition, others are declared and implemented in `ArrayList`.

EXAMPLE – LINKEDLIST

```
public class LinkedList<E> implements List<E>{
    boolean add(E e) {...}
    void add(int i, E e) {...}
    boolean isEmpty() {...}
    E get(int i) {...}
    E remove(int i) {...}
    int size() {...}
    void addFirst(E e) {...}
    void addLast(E e) {...}
}
```


All of the methods inherited from `List` are implemented. In addition, others are declared and implemented in `LinkedList`.

HOW ARE INTERFACES USED?

```
List<String> greetings;

greetings = new ArrayList<String>();
greetings.add("Hello");
   :
greetings = new LinkedList<String>();
greetings.add("Good day!");
```


Interfaces define new data types.

We can create variables of those type and assign to them any value referencing to instances of classes that implement the specified interface!

HOW ARE INTERFACES USED?

```
public void myMethod(List<String> list) {
    :
    list.add("one more");
    :
    list.remove(3);
    :
}
```


Whenever an object of type `List` is required, any instance of any of the classes that implement `List` can be used.

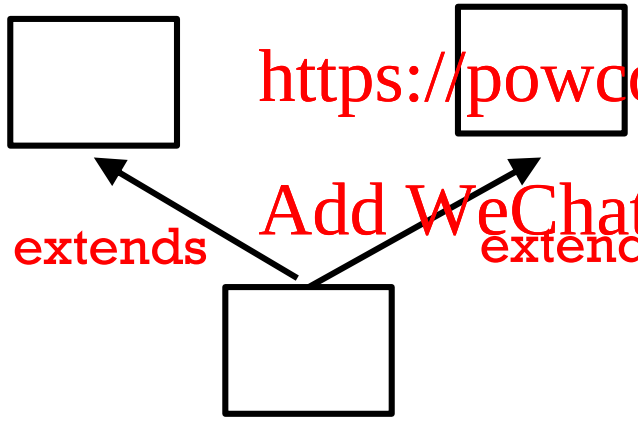
So, in this case, `myMethod()` can be called both with an `ArrayList` or a `LinkedList` as a parameter.

HOW ARE INTERFACES USED?

```
public void myMethod(List<String> list) {
    :
    list.add("one more");
    :
    list.remove(3);
    :
    list.addLast("Bye bye"); // compile-time error. Why??
}
```

INHERITANCE

Remember that a class (abstract or not) cannot extend more than one class (abstract or not).

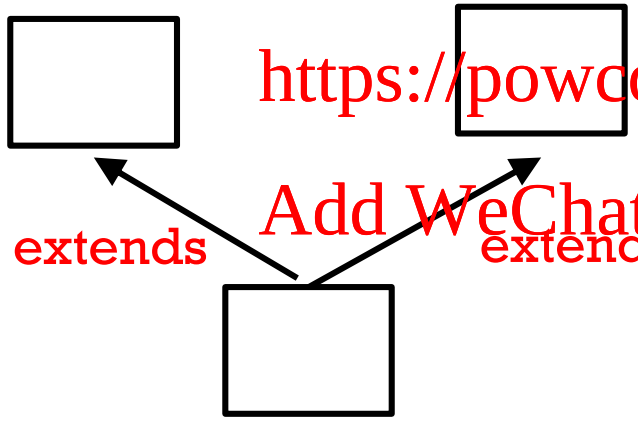


Not allowed!

- Why not?

INHERITANCE

Remember that a class (abstract or not) cannot extend more than one class (abstract or not).

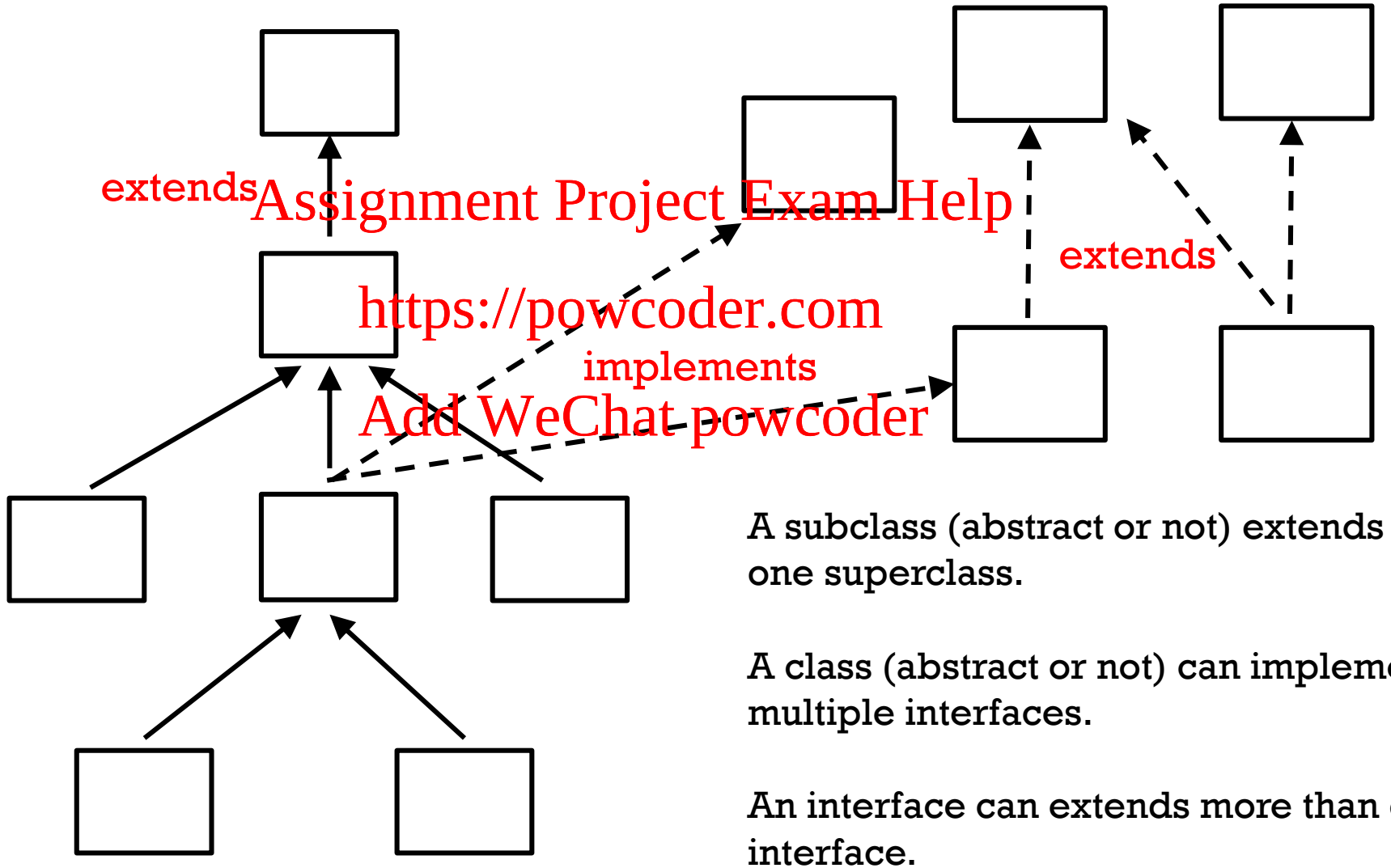


Not allowed!

- Why not? *The problem could occur if two superclasses have implemented methods with the same signature. Which would be inherited by the subclass?*

classes (abstract or not)

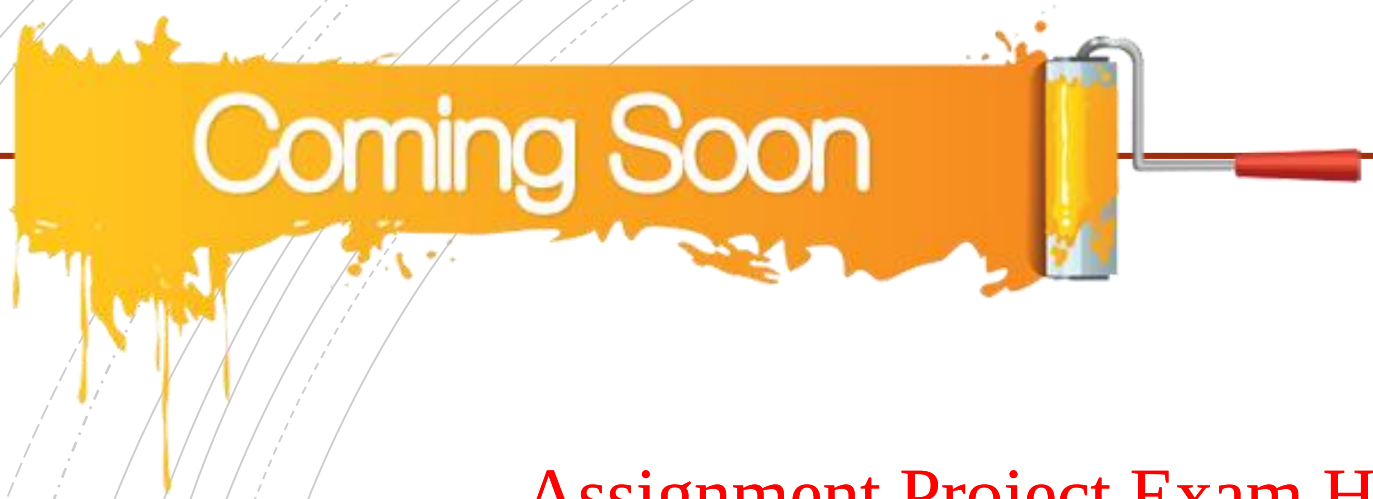
interfaces



A subclass (abstract or not) extends exactly one superclass.

A class (abstract or not) can implement multiple interfaces.

An interface can extends more than one interface.



In the next video:

- Comparable