

# COMP 250

## INTRODUCTION TO COMPUTER SCIENCE

Lecture 15 - Interfaces

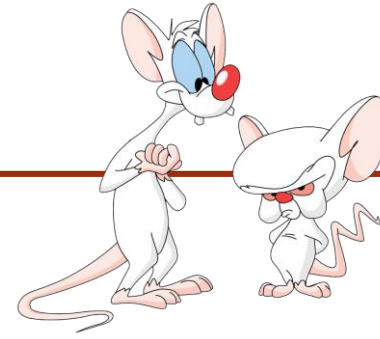
Giulia Alberini, Fall 2018

## FROM LAST WEEKS

---

- **Linear data structures and some sorting algorithms.**

# WHAT ARE WE GOING TO DO TODAY?



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

# 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 Monster {  
  
    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 multiple inheritance!
- 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 Monster {  
    :  
}
```

- **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` instruction 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 Monster {  
    public int spook();  
    public void runAway();  
}
```

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

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

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

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

# INTERFACES VS ABSTRACT CLASSES

## ABSTRACT CLASS

Not all methods have to be abstract.

The abstract keyword must be directly use to declare a class to be abstract.

Can contain methods that have been implemented as well as instance variables.

Abstract classes are useful when some general methods should be implemented and specialization behavior should be implemented by child classes.

## INTERFACE

All methods are abstract by default.

Interfaces are implicitly abstract.

No method can be implemented and only constants (final static fields) can be declared.

Interfaces are useful in a situation that all properties should be implemented.

## POST JAVA 8

---

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

- 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;  
    }  
}
```

# GENERIC 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 snoop = new Dog();  
crate.lock(snoop);  
  
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();
```

# GENERIC TYPE NAMING CONVENTIONS

- Java Generic Type Naming convention helps us understanding 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

# 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))
```

## 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

## 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 from 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 from 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.

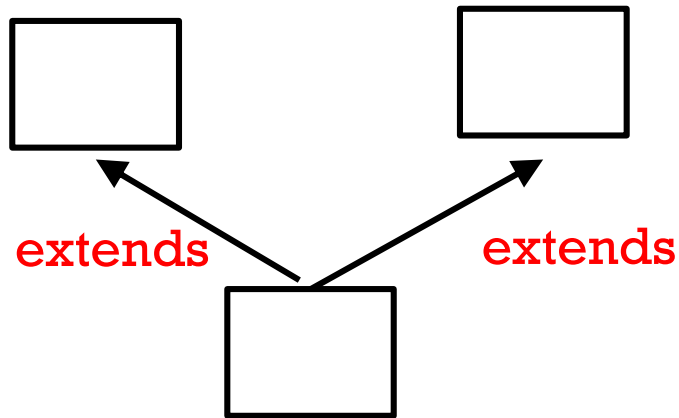
## HOW ARE INTERFACES USED?

```
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).

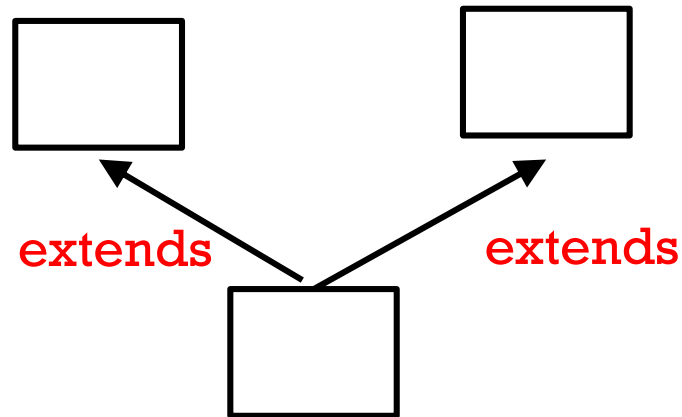


*Not allowed!*

- Why not?

# INHERITANCE

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

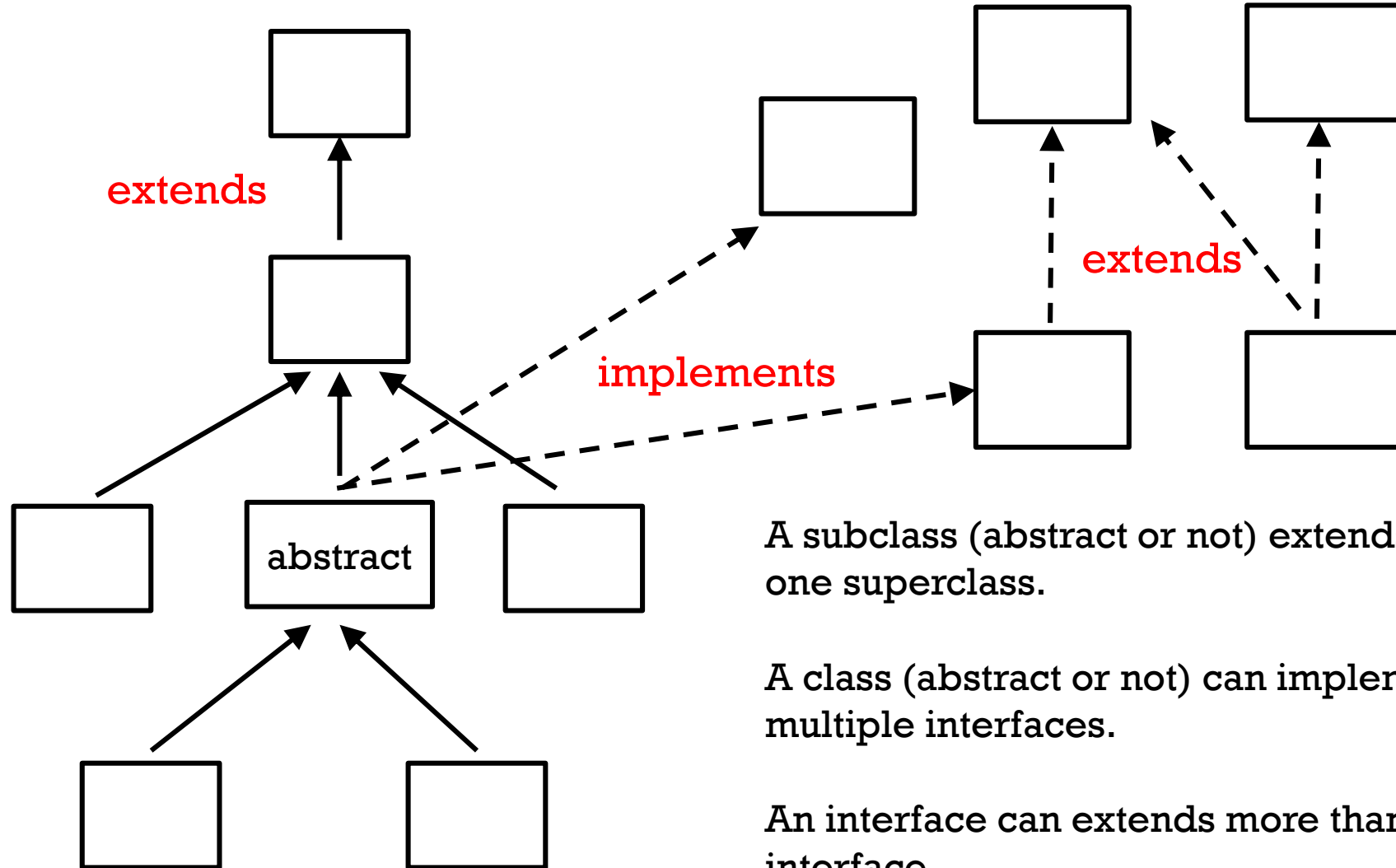


*Not allowed!*

- 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 extend more than one interface.

The background features a series of concentric circles in a light gray color, some of which are dashed. A solid dark red rectangle is positioned in the center of the image. The word "COMPARABLE" is written in white, uppercase, sans-serif font within this red rectangle.

COMPARABLE

# JAVA Comparable INTERFACE

- The Java Comparable interface is used to define an ordering on objects of user-defined class.
- Why would you want that? Well, if you have a list of objects from a given class you might want to be able to sort it.
- Comparable is part of `java.lang` package and contains only one method named `compareTo(Object)`.

# JAVA Comparable INTERFACE

```
public interface Comparable<T>{  
    int compareTo(T o);  
}
```

<https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>

# JAVA Comparable INTERFACE

Some of the methods from certain Java classes use `compareTo()` in their implementation. To function correctly, they assume to be working with Comparable generic types. Examples:

- `sort()` from Arrays.

## sort

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

# JAVA Comparable INTERFACE

Some of the methods from certain Java classes use `compareTo()` in their implementation. To function correctly, they assume to be working with Comparable generic types. Examples:

- `sort()` from `Collections`.

## sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).



# String IMPLEMENTS Comparable

## compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let  $k$  be the smallest such index; then the string whose character at position  $k$  has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position  $k$  in the two strings -- that is, the value:

```
this.charAt(k) - anotherString.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length() - anotherString.length()
```

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

## CLASSES THAT IMPLEMENT `Comparable`

- `Character`, `Integer`, `Float`, `Double`, `BigInteger`, **etc.** **all implement** `Comparable<T>`.
- You cannot compare objects of these classes using the “<” operator. Instead use `compareTo()`.

## HOW TO IMPLEMENT Comparable

- **Add** `implements Comparable` **in the definition of the class.**
- **Implement** `compareTo()` **inside your class.**

```
public class T implements Comparable<T>{  
    public int compareTo(T o) {...}  
}
```

## REQUIREMENT FOR IMPLEMENTING `compareTo()`

Consider two variable `t1` and `t2` or type `T`. Then,

`t1.compareTo(t2)` returns  $\begin{cases} \text{negative int} & , \text{if } t1 < t2 \\ 0 & , \text{if } t1 = t2 \\ \text{positive int} & , \text{if } t1 > t2 \end{cases}$

The relation should also be anticommutative and transitive.

Highly recommended:

`(t1.compareTo(t2) == 0) == (t1.equals(t2))`

## EXAMPLE - CIRCLE

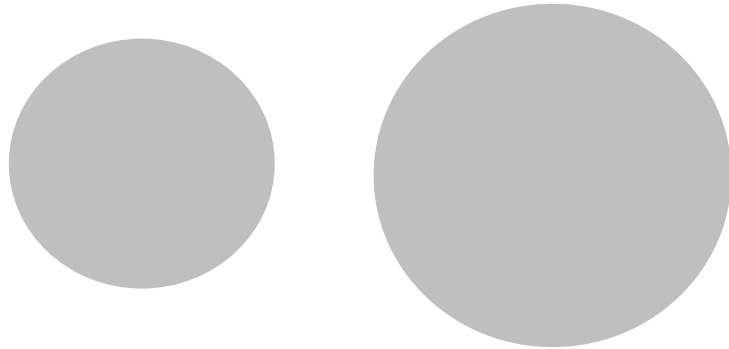
- Sometimes deciding how to compare elements of a given type can be straightforward.
- Let's think about the data type `Circle`.

```
public class Circle {  
    private double radius;  
    :  
}
```

- How should we implement `compareTo()` and `equals()` in order to establish a *natural ordering* between elements of type `Circle`?

## EXAMPLE - CIRCLE

- How should we implement `compareTo()` and `equals()` in order to establish a *natural ordering* between elements of type `Circle`?



We could simply compare their radius (or their area).

## EXAMPLE – ORC

- Other times, is not so straightforward. Suppose we have created a new data type `Orc`.
- How should we compare and sort elements of this type?



Base on their name? On their height? On their weapon? On who is scarier?

## ORC – compareTo () TAKE 1

```
public class Orc implements Comparable<Orc> {  
    private String name;  
    private int height;  
    private Weapon w;  
    public int compareTo(Orc o) {  
        if(this.height < o.height) {  
            return -1;  
        } else if(this.height == o.height) {  
            return 0;  
        } else {  
            return 1;  
        }  
    }  
}
```

- Note that in this case we probably don't want to consider two Orcs with the same weight to be equal.
- This implies that the implementation of compareTo() violates the Java API recommendations.
- Such violation should be clearly indicated using the following language: "Note: this class has a natural ordering that is inconsistent with equals."



## ORC – compareTo () TAKE 1

```
public class Orc implements Comparable<Orc> {
    private String name;
    private int height;
    private Weapon w;
    public int compareTo(Orc o) {
        int result = this.w.compareTo(o.w);
        if(result==0) {
            result = Integer.compare(this.weight, o.weight);
        }
        if(result == 0) {
            result = this.name.compareTo(o.name);
        }
        return result;
    }
}
```

- When we also use `compareTo()` to compare multiple characteristics.
- Generally, it is better to reuse existing code than to write our own. Thus, in this case, can use the `compareTo()` methods from other classes to.

## TO RECAP

---

- Comparable defines a natural ordering.
- If you define a new data type for which sorting makes sense to you, then you should implement comparable to define a natural ordering on objects of such type.



# Coming Soon

- Iterators
- Memory allocation
- The class Class
- Garbage collection (maybe)