COMP250-OOD Interface

disclaimer: we’ll talk about interfaces pre Java 8

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

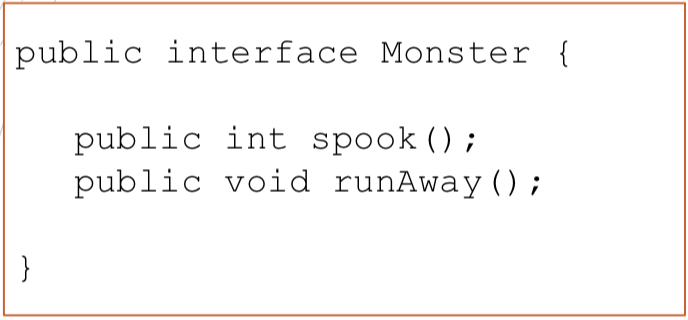
Similarly to classes, interfaces can have fields and methods BUT

**All methods are by default public and abstract.**

**All fields are by default public, static, and final.**

Interfaces cannot be instantiated.

We declare an interface using the *interface* keyword.



An interface is implicitly abstract and the methods inside it are all implicitly abstract.

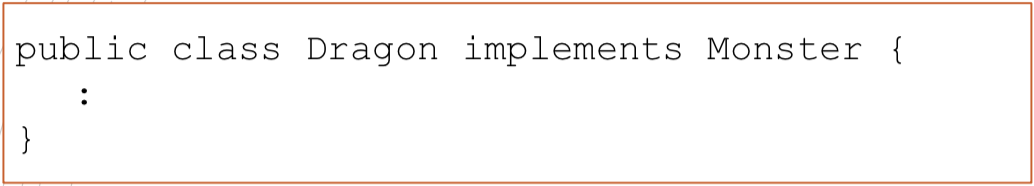
Do NOT need *abstract* keyword.

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]

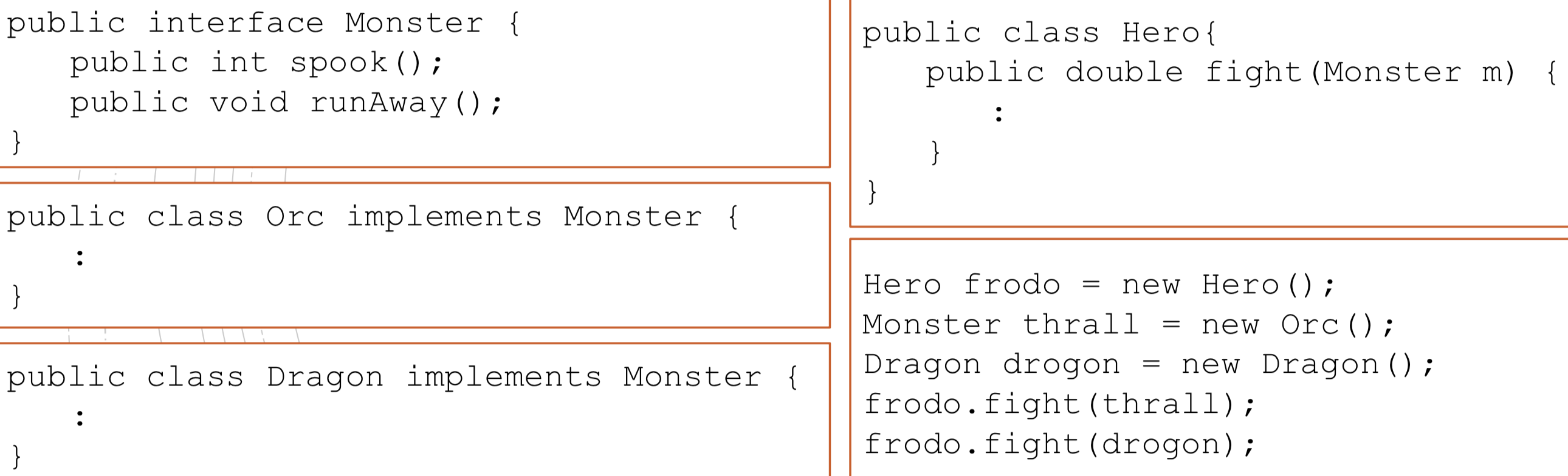


Inside the class Dragon, the methods spook() and runAway() must be implemented!

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.

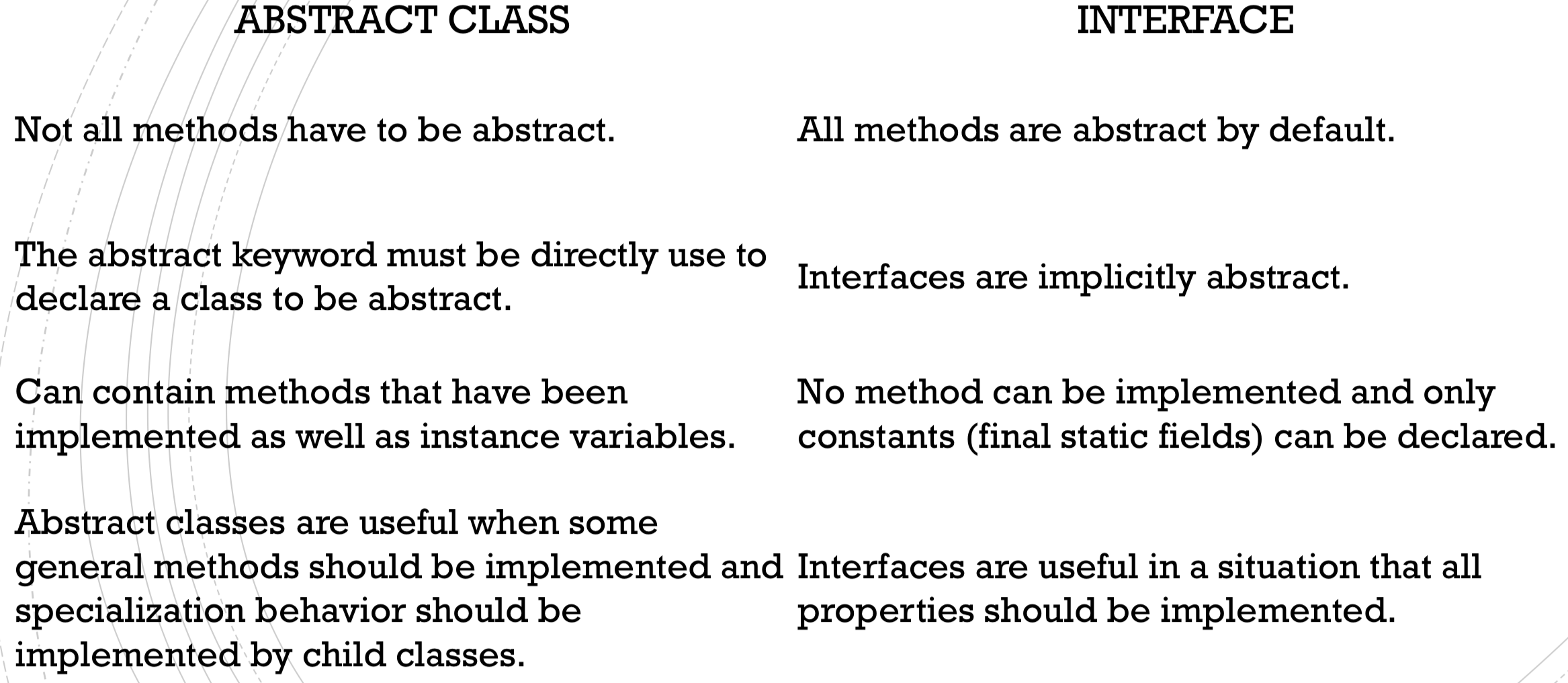
[Interface Instances]

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



Monster don’t have instructor!

[Interfaces VS Abstract Classes]



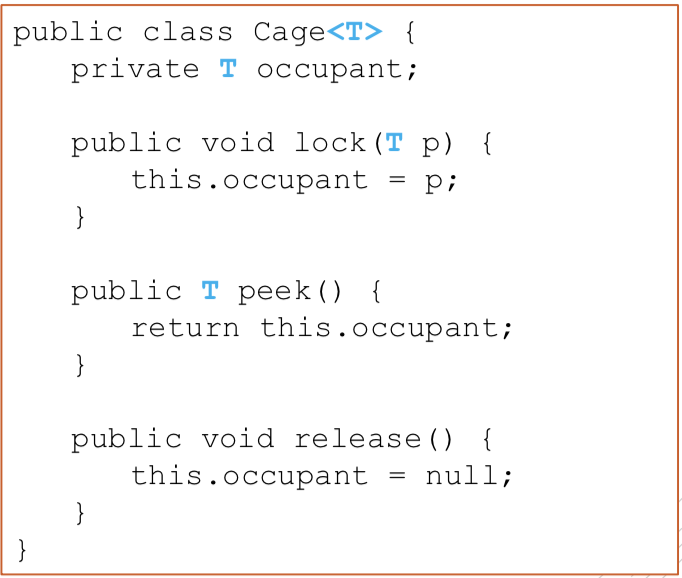
[Post JAVA8]

From Java 8 onwards, interfaces can contain following also

* Default methods
* Static methods
* Private methods
* Private Static methods

Generics

A generic type is a class or interface that is parameterized over types.

We use angle brackets (< >) to specify the type parameter.

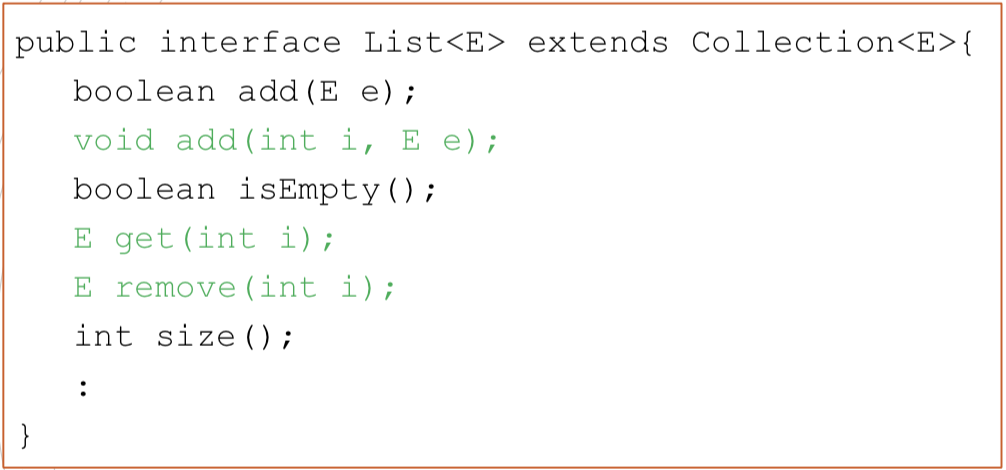
[Genetic Type Naming Conventions]

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

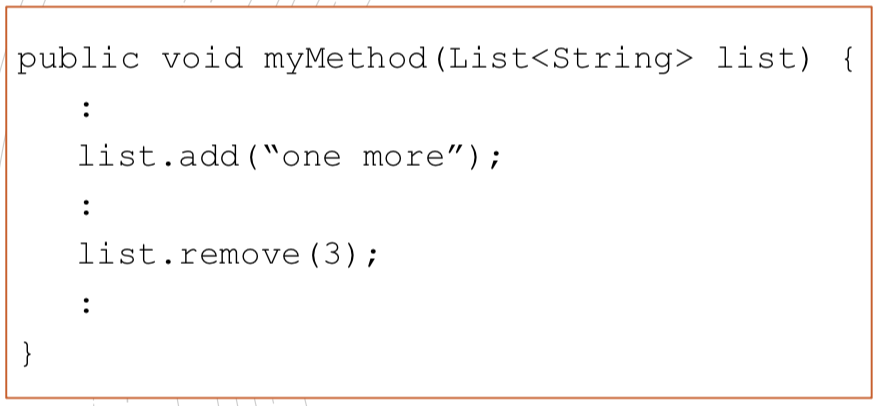
Example: List Interface



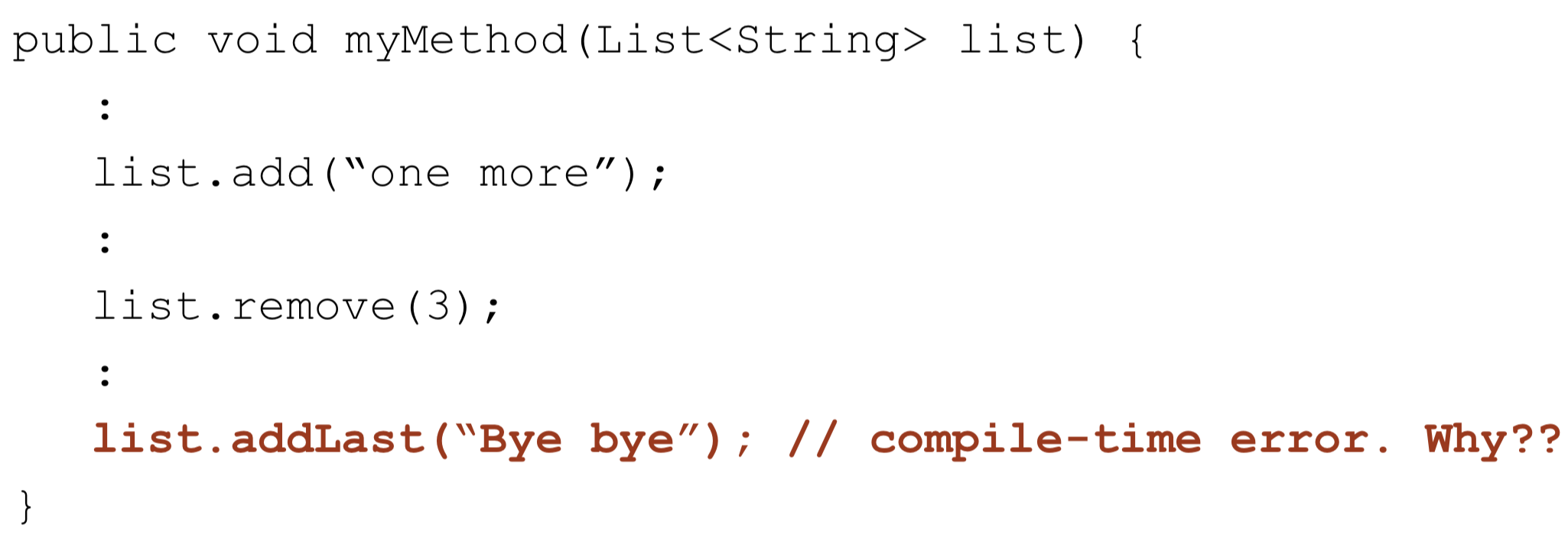
Some of the methods are inherited from the interface Collection, while others are declared inside List.

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!

**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.



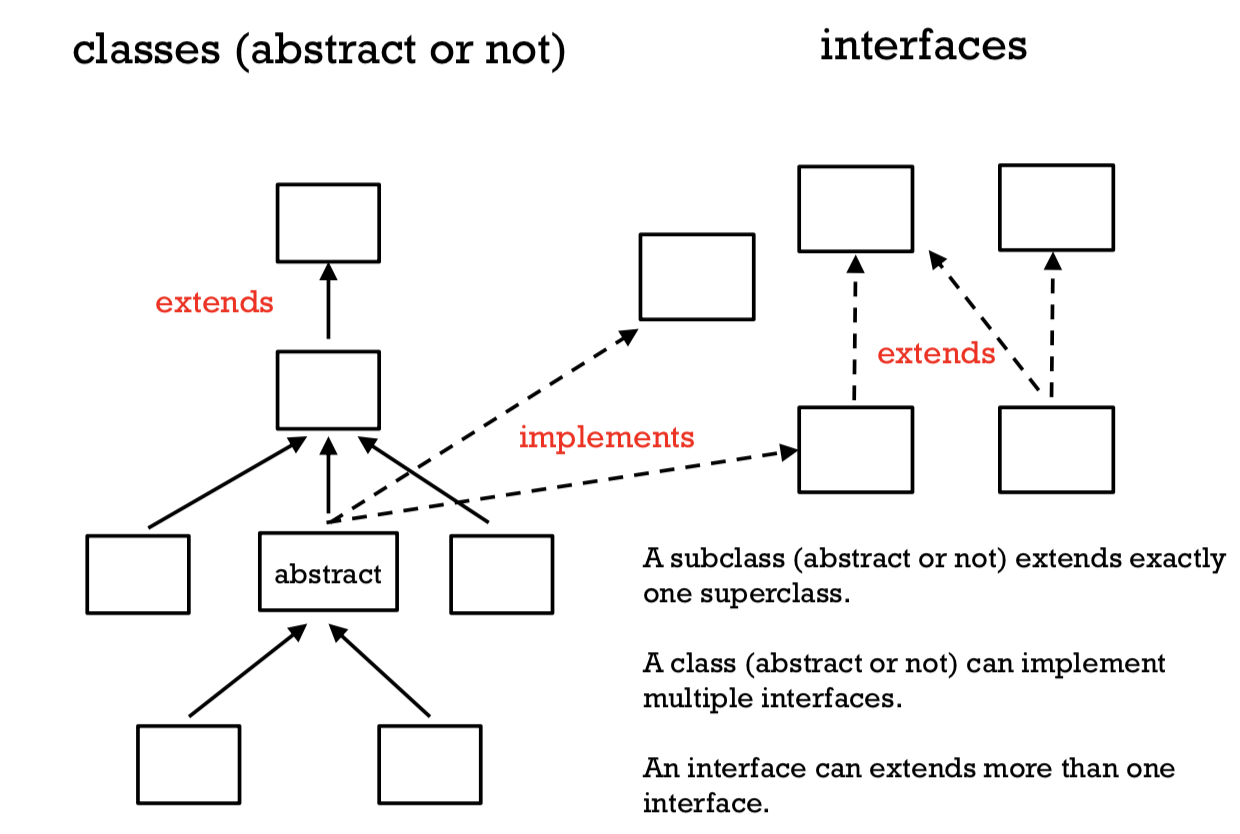
could NOT be linked list.

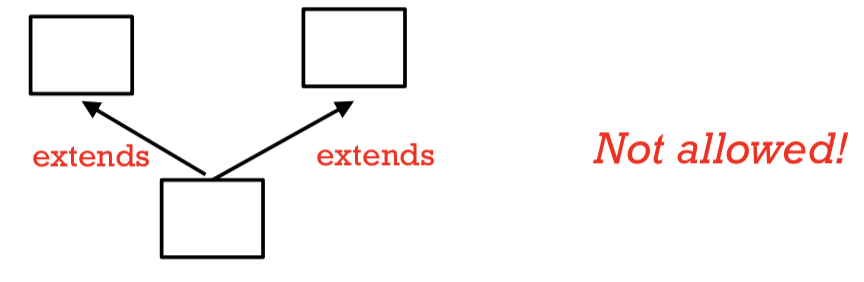
eg ArrayList

[Inheritance]

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

The problem could occur if two superclasses have implemented methods with the same signature. Which would be inherited by the subclass?



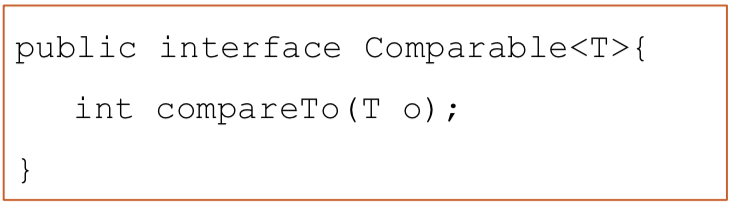


NOT ALLOWED!

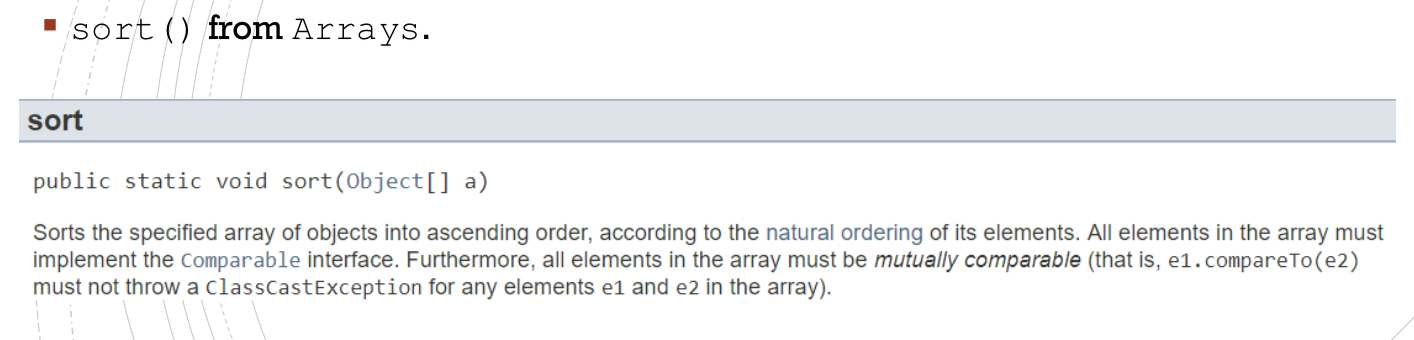
Comparable

The Java *Comparable* interface is used to define an ordering on objects of user-defined class.

Comparable is part of java.lang package and contains only one method named *compareTo(Object)*.



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.



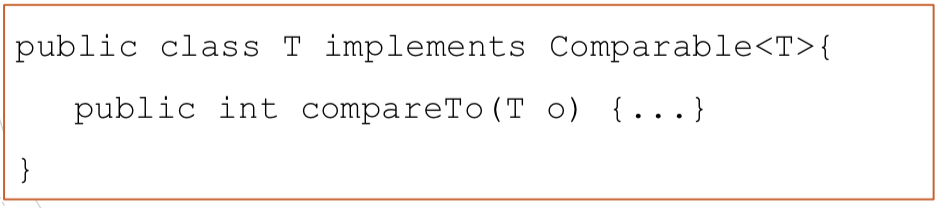
[Classes that implement Comparable]

*Character, Integer, Float, Double, BigInteger*, etc. all implement Comparable<T>.

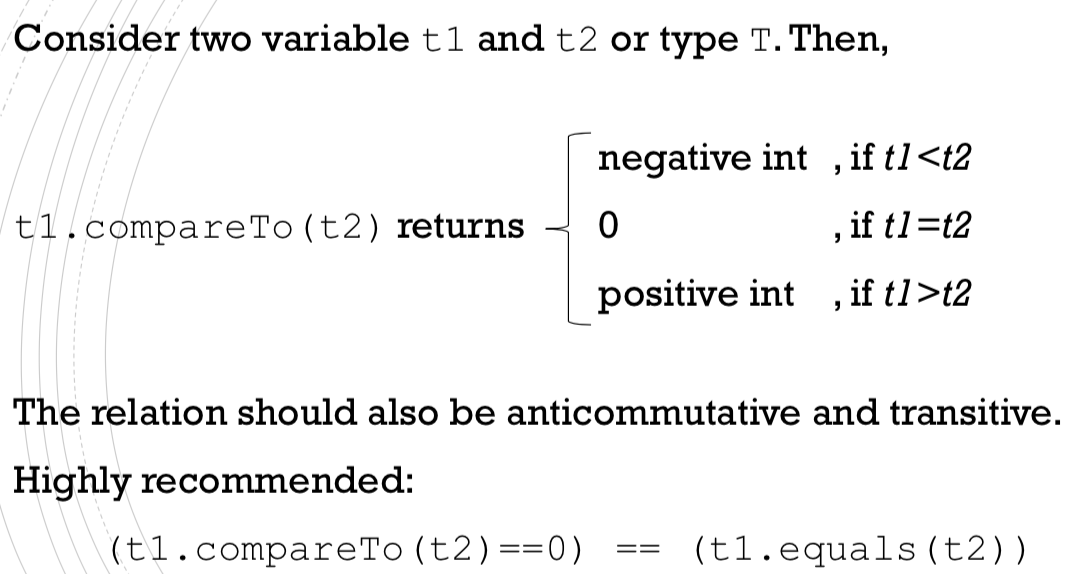
You cannot compare objects of these classes using the “<“ operator. Instead use compareTo().

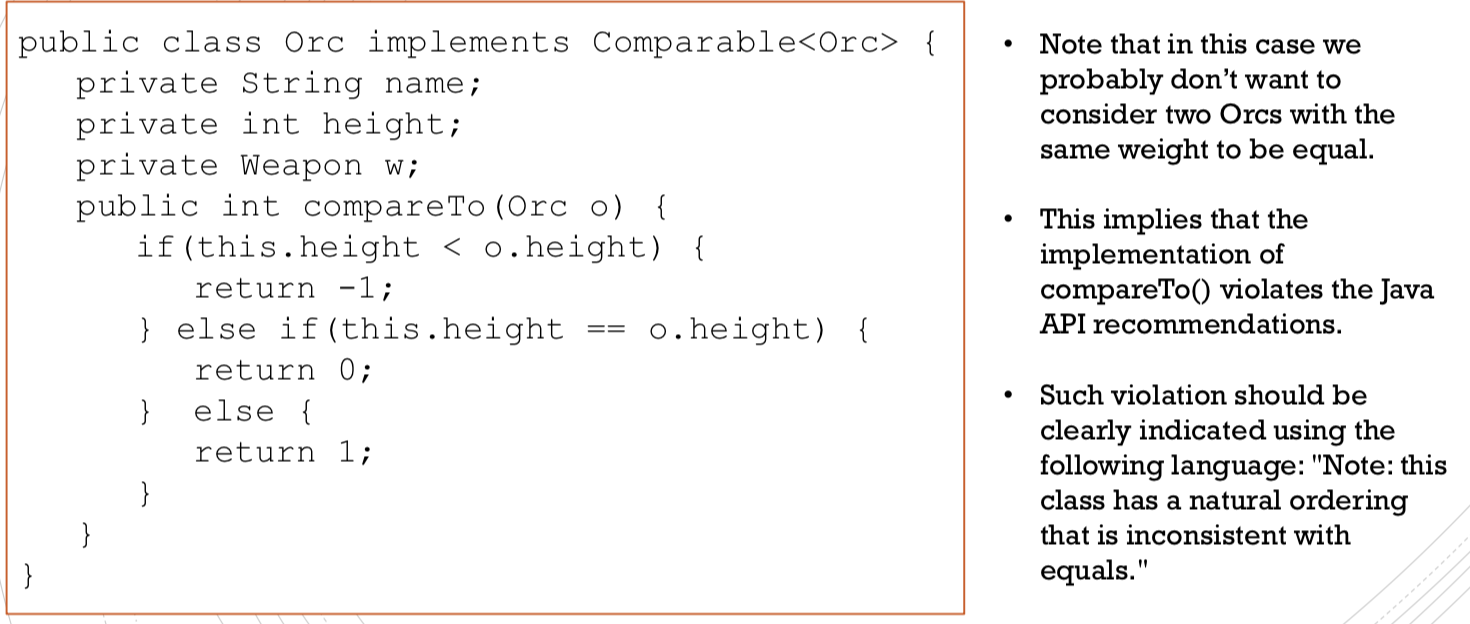
[How to implement Comparable]

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



* compareTo()





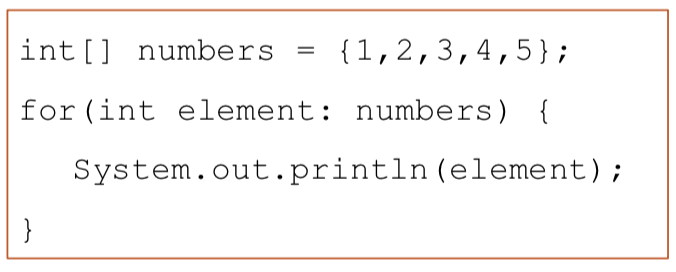


RECAP

Comparable defines a natural ordering.

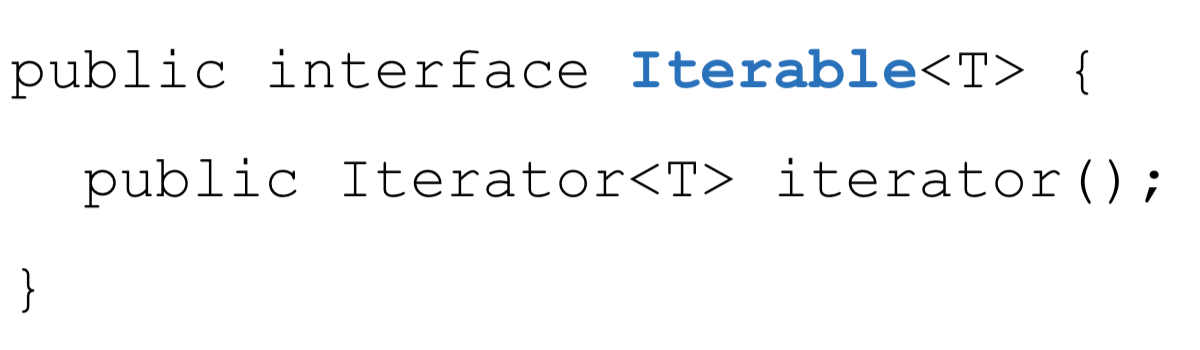
If you define a new data type for which sorting makes sense to you, then you should implement comparable to define a natural ordering on objects of such type.

Iterable & Iterator

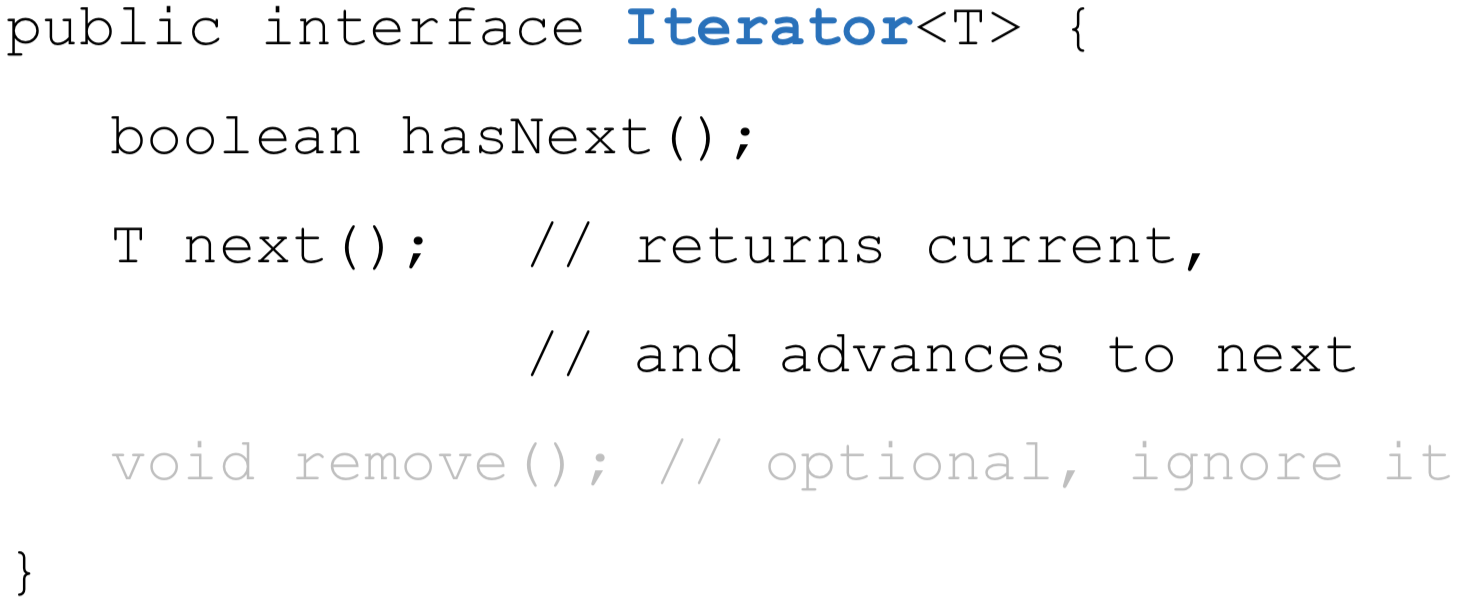


The use of a **for-each loop** is made possible by the use of two interfaces: *Iterator* and *Iterable*.

* Objects of type *Iterable* are representations of a series of elements that can be iterated over. (e.g. a specific ArrayList)
* Objects of type *Iterator* allows you to iterate through objects that represent a collection (a series of elements).

A class that implements *Iterable* needs to implement the *iterator()* method.

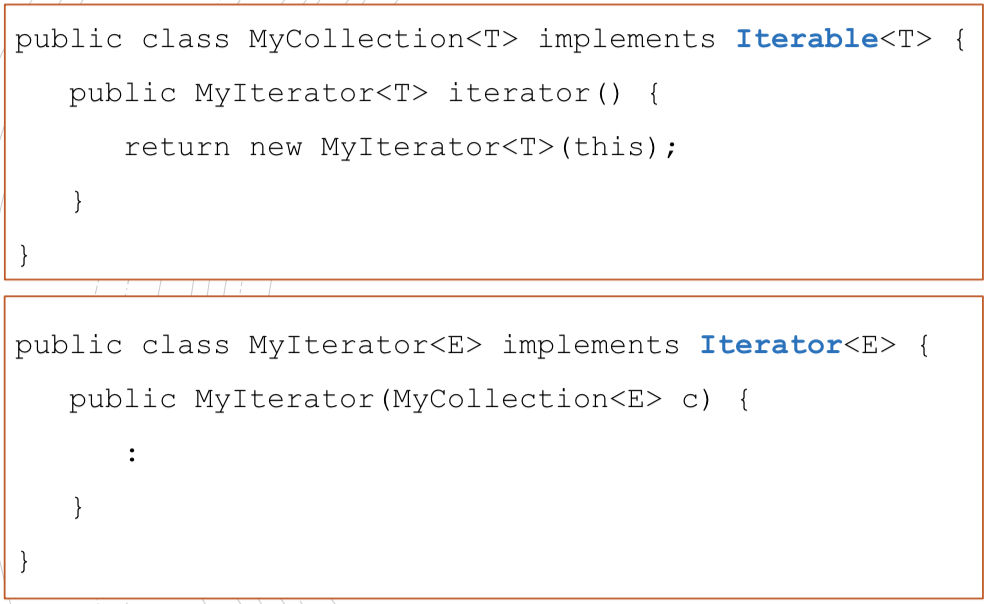
The iterator() method returns an object of type Iterator to the start of the collection that can then be used to iterate through the elements of the object to that class.

A class that implements *Iterator* needs to implement the methods *hasNext()* and *next().*

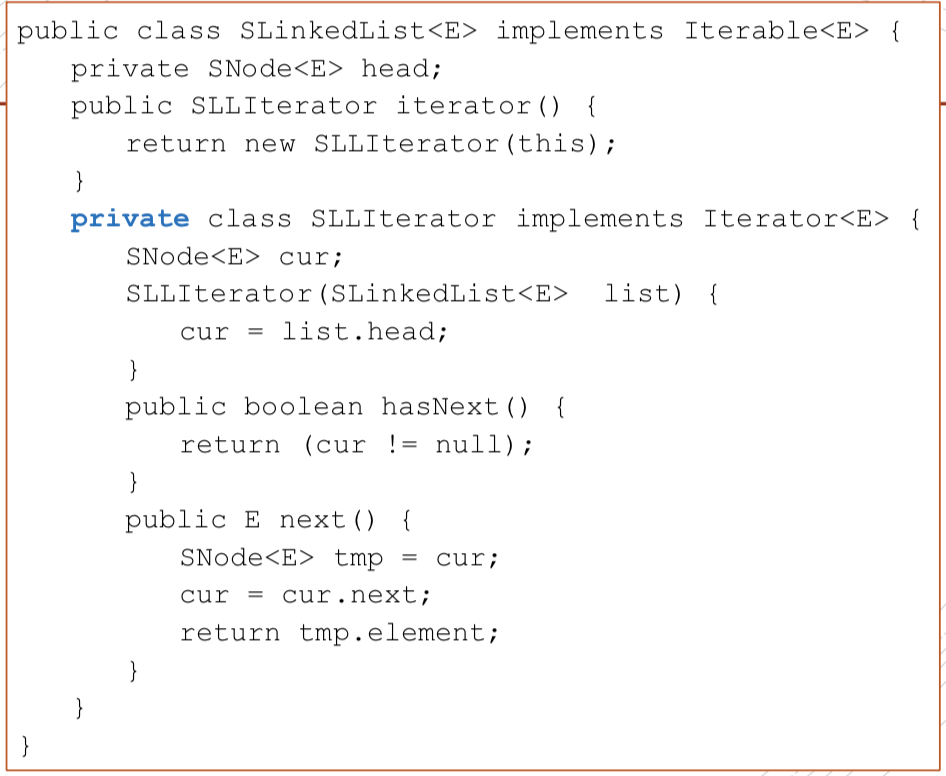
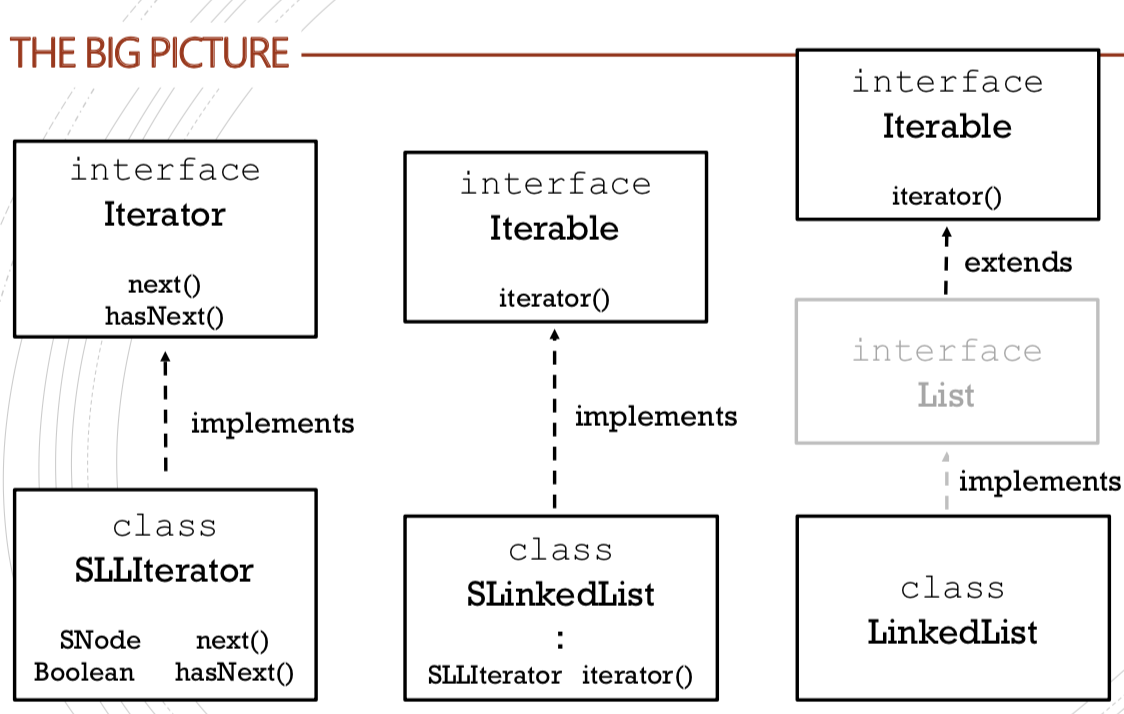
The *Iterator()* method returns an iterator to the start of the collection. Using *hasNext()* and *next()* you can move forward in the collection. If you want to traverse the collection again, you’ll need a new Iterator.

Implementing the Iterable interface allows an object to make use of the for-each loop. It does that by internally calling the iterator() method on the object!

* As always when implementing interfaces, a class that implements an interface must implement every method from such interface.
* Generally, when we write a class that implements the interface Iterable we also write a class that implements the interface Iterator. Often, such class is defined as an inner class of the first class.
* Why? To implement Iterable, we need to implement the method iterator(). Such method need to return an object of type Iterator that can iterate through the elements of a specific object of the outer class. We need a class that can create such object.

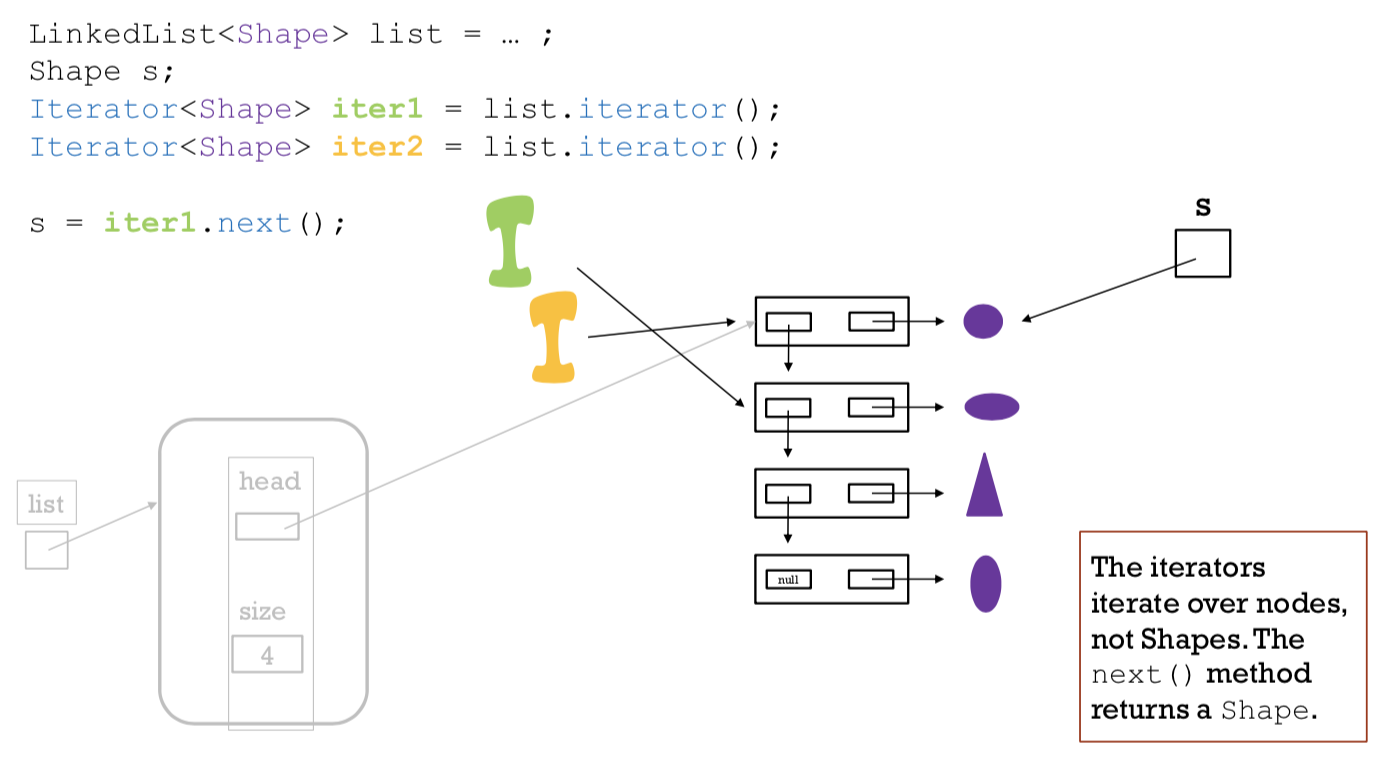


In general, if the class MyIterator is used only by the class MyCollection, good practice is to make that class a private inner class of MyCollection.



next() returns the element of the list that the Iterator is currently referencing, and then moves to the next node.

iterator() returns an object of type Iterator that points to the head of the provided list.



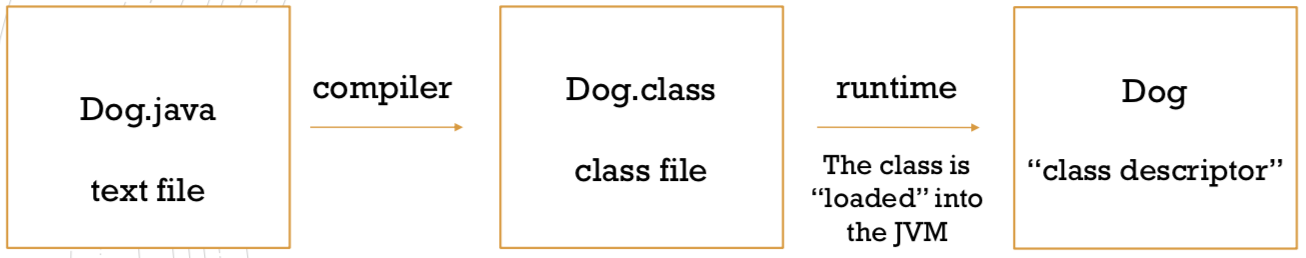
Suppose we have a SLinkedList of Shapes:

SLinkedList<Shape> list = ...

Then by calling iterator() we create an object of type SLLIterator that points to the head of the list.

SLLIterato iter =list.iterator();

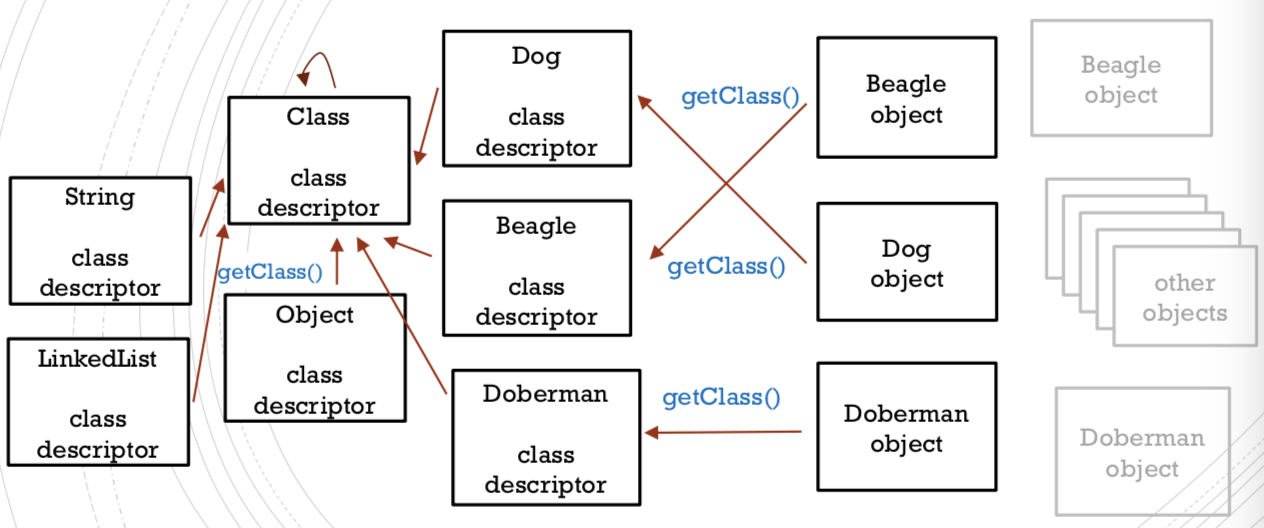
Class



getClass()

All classes inherit a method called getClass() from the Object class.

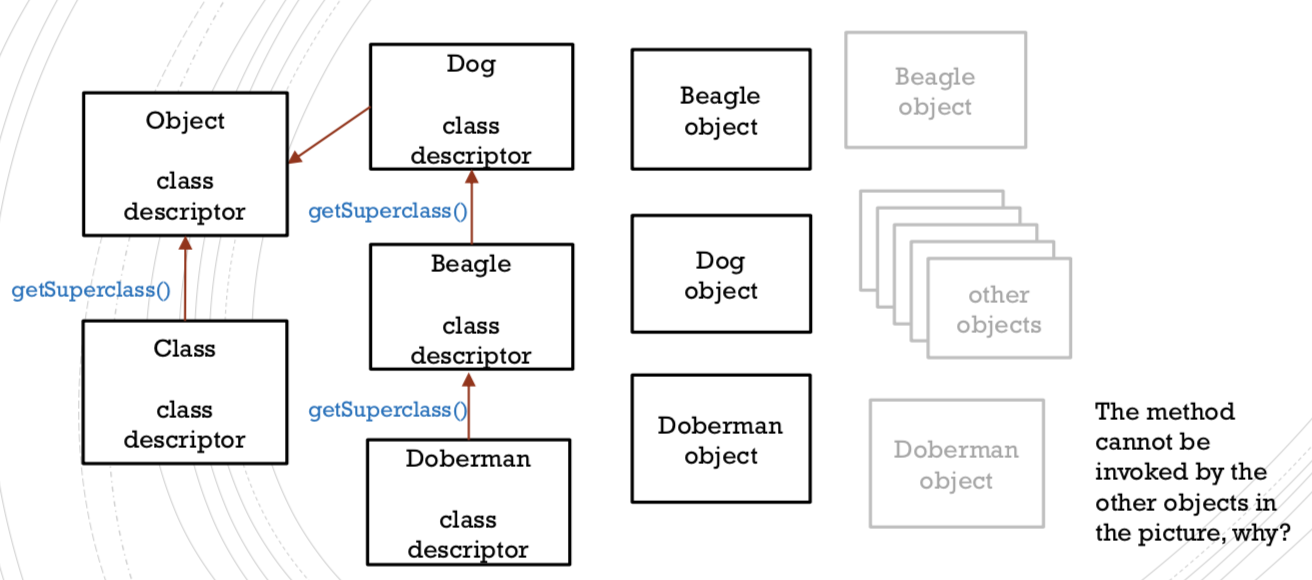
This method returns the run-time class of this object



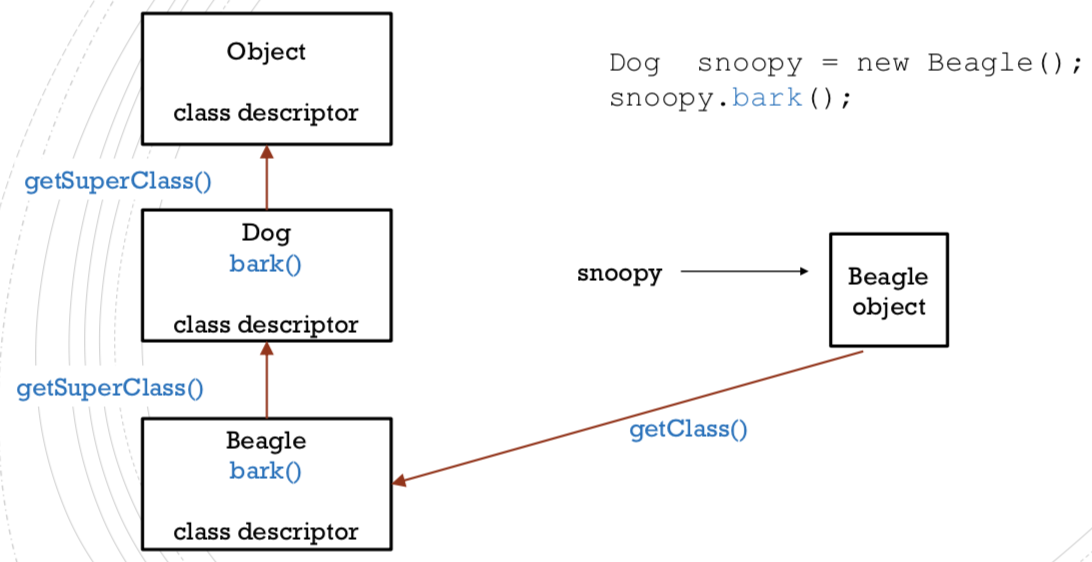
getSuperclass()

This is one of the methods from the Class class.

It returns the Class representing the superclass of the class represented by this Class.



There is no superclass



when running

[MEMORY ALLOCATION – HEAP VS STACK]

* The **Java Virtual Machine (JVM)** divides memory between Java Heap Space and Java Stack Memory
* Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.
* Java Stack memory is used for execution of a thread. They contain method specific values and references to other objects in the heap that are getting referred from the method.

Note: Stack is much smaller than heap

**[JAVA STACK]**

Java stack memory uses a **LIFO** data structure.

* Each time a method is invoked, it creates a new block in the stack for that particular method.
* Each method block has all the local values, as well as references to other objects that are being used by the method.
* When the method ends, its block will be erased and will be available for use by the next method.
* The values stored in each block are accessible only from that particular method.

**[JAVA HEAP SPACE]**

* Whenever we create any object, it’s always created in the Heap space.
* There is no specific order in reserving blocks in a heap.
* Any object created in the heap space has global access and can be referenced from anywhere of the application.
* Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference.

