Lecture 9 (Inheritance 2)

Subtype Polymorphism, Comparators, Comparable

CS61B, Fall 2024 @ UC Berkeley

Slides credit: Josh Hug

Bonus Content: DMS and Type Checking Puzzle

Online Video Only

Lecture 9, CS61B, Fall 2024

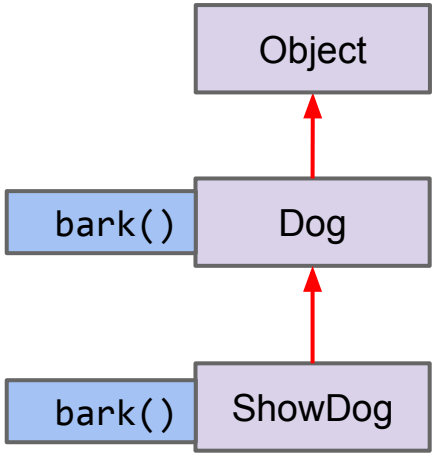
A Typing Puzzle

Suppose we have two classes:

- Dog: Implements bark() method.
- ShowDog: Extends Dog, overrides bark method.

Summarizing is-a relationships, we have:

- Every ShowDog is-a Dog
- Every Dog is-an Object.
  - All types in Java are a subtype of Object.



A Typing Puzzle

For each assignment, decide if it causes a compile error.

For each call to bark, decide whether: 1. Dog.bark() is called, 2. ShowDog.bark() is called, or 3. A syntax error results.

```
Object o2 = new ShowDog("Mortimer", "Corgi", 25, 512.2);

ShowDog sdx = ((ShowDog) o2);
sdx.bark();

Dog dx = ((Dog) o2);
dx.bark();

((Dog) o2).bark();

Object o3 = (Dog) o2;
o3.bark();
```

The rules:

- Compiler allows memory box to hold any subtype.
- Compiler allows calls based on static type.
- **Overridden non-static methods are selected at run time based on dynamic type**.
  - **Everything else is based on static type**, including overloaded methods. Note: No overloaded methods for problem at left.

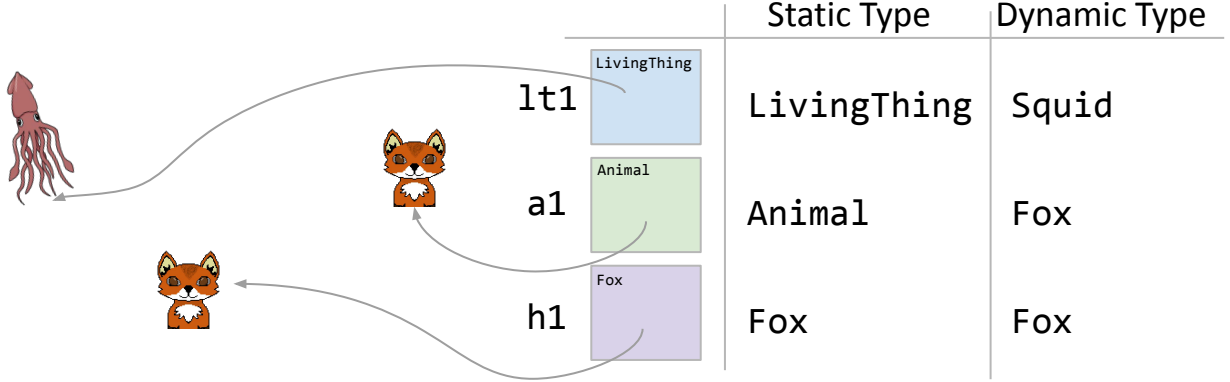
Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using `new`).
- Equal to the type of the object being pointed at.



Static Methods, Variables, and Inheritance

You may find questions on old 61B exams, worksheets, etc. that consider:

- What if a subclass has variables with the same name as a superclass?
- What if subclass has a static method with the same signature as a superclass method?
  - For static methods, we do not use the term overriding for this.
- What if a subclass has methods that overload superclass methods?

These practices are generally not a good idea.

- It is bad style.
- There is almost no good reason to ever do this.
- The rules for resolving the conflict are a bit confusing to learn.
- I've pushed 61B away from learning these rules.
- But if you want to learn them, see https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Subtype Polymorphism vs. Explicit Higher Order Functions

Lecture 9, CS61B, Fall 2024

Subtype Polymorphism vs. Explicit Higher Order Functions

Building a General Max Function

- The Naive Approach
- OurComparable
- Compilation Error Puzzle
- Comparable

Comparators

Some Miscellaneous Java Syntax

Subtype Polymorphism

The biggest idea of the last couple of lectures: ***Subtype Polymorphism***

- Polymorphism: “providing a single interface to entities of different types”

a.k.a. compile-time type

Consider a variable `deque` of static type `Deque`:

- When you call `deque.addFirst()`, the actual behavior is based on the dynamic type. ← a.k.a. run-time type
- Java automatically selects the right behavior using what is sometimes called “dynamic method selection”.

Curious about alternatives to subtype polymorphism? See wiki or CS164.

http://www.stroustrup.com/glossary.html#Gpolymorphism

Subtype Polymorphism vs. Explicit Higher Order Functions

Suppose we want to write a program that prints a string representation of the larger of two objects.

Sometimes called a “callback”.

Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

Subtype Polymorphism Approach

```
def print_larger(x, y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

Not to be confused with the fascinating Dr. Ernest Kaulbach, who taught my Old English class.

The Naive Approach

Lecture 9, CS61B, Fall 2024

Subtype Polymorphism vs. Explicit Higher Order Functions

Building a General Max Function

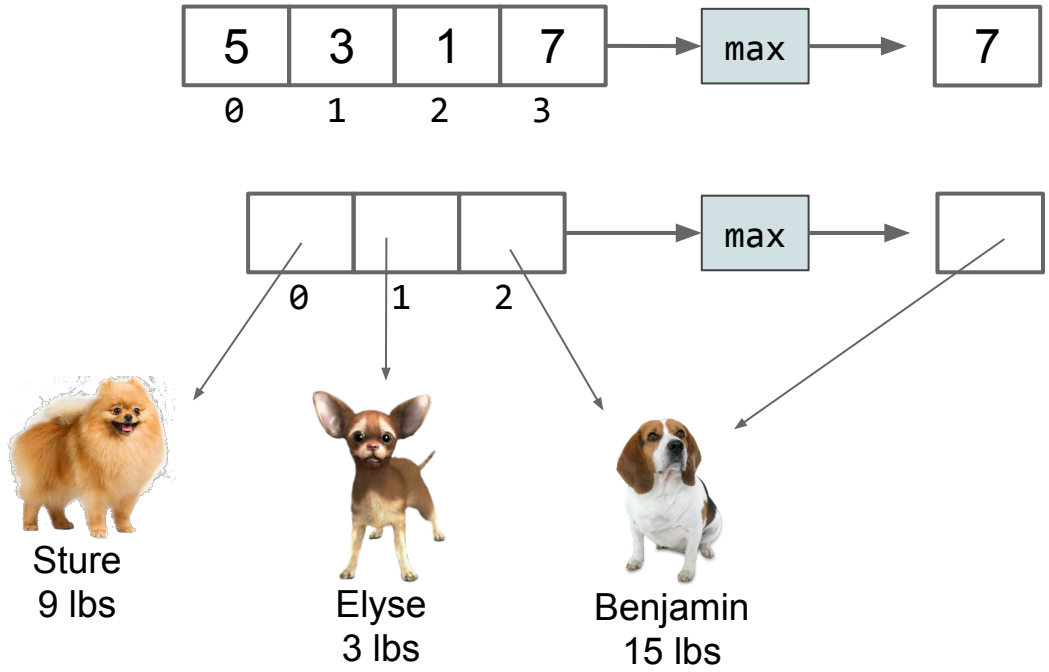
- **The Naive Approach**
- OurComparable
- Compilation Error Puzzle
- Comparable

Comparators

Some Miscellaneous Java Syntax

Goal: The One True Max Function

Suppose we want to write a function `max()` that returns the max of any array, regardless of type.



Compilation Error Challenge

Suppose we want to write a function `max()` that returns the max of any array, regardless of type. How many compilation errors are there in the code shown?

A. 0
B. 1
C. 2
D. 3

Maximizer.java

```
public static Object max(Object[] items) {
   int maxDex = 0;
   for (int i = 0; i < items.length; i += 1) {
       if (items[i] > items[maxDex]) {
           maxDex = i;                }}
   return items[maxDex];
}
```

DogLauncher.java

```
public static void main(String[] args) {
   Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                 new Dog("Benjamin", 15)};
   Dog maxDog = (Dog) Maximizer.max(dogs);
   maxDog.bark();
}
```

Writing a General Max Function

Objects cannot be compared to other objects with >

- One (bad) way to fix this: Write a max method in the Dog class.

Maximizer.java

```
public static Object max(Object[] items) {
   int maxDex = 0;
   for (int i = 0; i < items.length; i += 1) {
       if (items[i] > items[maxDex]) {
           maxDex = i;                }}
   return items[maxDex];
}
```

DogLauncher.java

```
public static void main(String[] args) {
   Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                 new Dog("Benjamin", 15)};
   Dog maxDog = (Dog) Maximizer.max(dogs);
   maxDog.bark();
}
```

Dog.maxDog

One approach to maximizing a Dog array: Leave it to the Dog class.

- What is the disadvantage of this?

```
/** Returns maximum of dogs. */
public static Dog maxDog(Dog[] dogs) {
    if (dogs == null || dogs.length == 0) {
        return null; }
    Dog maxDog = dogs[0];
    for (Dog d : dogs) {
        if (d.size > maxDog.size) {
            maxDog = d;           }}
    return maxDog;
}
```

```
Dog[] dogs = new Dog[]{d1, d2, d3};
Dog largest = Dog.maxDog(dogs);
```

The Fundamental Problem

Objects cannot be compared to other objects with >

- How could we fix our Maximizer class using inheritance / HoFs?

Maximizer.java

```
public static Object max(Object[] items) {
   int maxDex = 0;
   for (int i = 0; i < items.length; i += 1) {
       if (items[i] > items[maxDex]) {
           maxDex = i;                }}
   return items[maxDex];
}
```

DogLauncher.java

```
public static void main(String[] args) {
   Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                 new Dog("Benjamin", 15)};
   Dog maxDog = (Dog) Maximizer.max(dogs);
   maxDog.bark();
}
```

OurComparable

Lecture 9, CS61B, Fall 2024

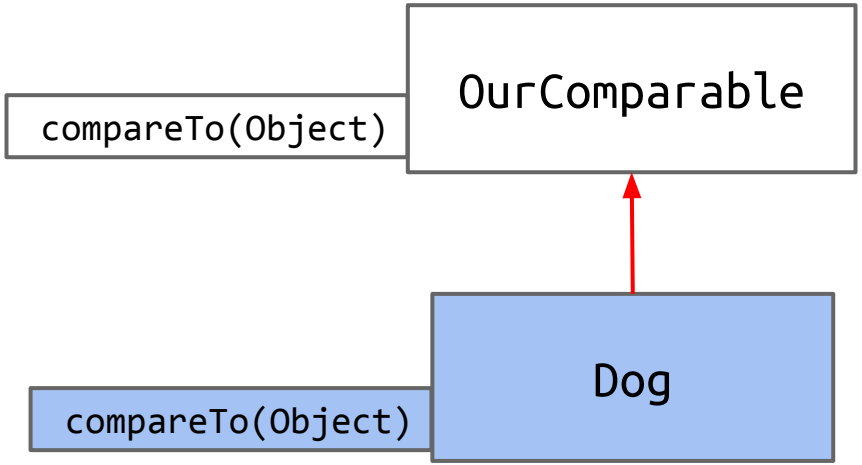
Solution

Create an interface that guarantees a comparison method.

- Have Dog implement this interface.
- Write Maximizer class in terms of this interface.

Interface inheritance says **what** a class can do, in this case compare.

```
public static OurComparable max(OurComparable[] items) { ...
```



Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static Object max(Object[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {
         if (items[i] > items[maxDex]) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }

   public static void main(String[] args) {
      Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                    new Dog("Benjamin", 15)};
      Dog maxDog = (Dog) Maximizer.max(dogs);
      maxDog.bark();
   }
}
```

This doesn't compile because you can't compare objects with the > operator.

Coding Demo: OurComparable

OurComparable.java

```
public interface OurComparable {



}
```

Coding Demo: OurComparable

OurComparable.java

```
public interface OurComparable {



   public int compareTo(Object o);
}
```

Coding Demo: OurComparable

OurComparable.java

```
public interface OurComparable {
   /** Return -1 if this < o.
    *  Return 0 if this equals o.
    *  Return 1 if this > o.
    */
   public int compareTo(Object o);
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog {
   private String name;
   private int size;









}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;









}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;


   public int compareTo(Object o) {









   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {









   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {


       if (this.size < o.size) {
           return -1;
       }









   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {


       if (this.size < o.size) {
           return -1;
       } else if (this.size == o.size) {
           return 0;
       }




   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {


       if (this.size < o.size) {
           return -1;
       } else if (this.size == o.size) {
           return 0;
       }
       return 1;




   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {
       Dog uddaDog = (Dog) o;

       if (this.size < uddaDog.size) {
           return -1;
       } else if (this.size == uddaDog.size) {
           return 0;
       }
       return 1;




   }
}
```

Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static Object max(Object[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {

         if (items[i] > items[maxDex]) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }

   public static void main(String[] args) {
      Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                    new Dog("Benjamin", 15)};
      Dog maxDog = (Dog) Maximizer.max(dogs);
      maxDog.bark();
   }
}
```

Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static OurComparable max(OurComparable[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {

         if (items[i] > items[maxDex]) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }

   public static void main(String[] args) {
      Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                    new Dog("Benjamin", 15)};
      Dog maxDog = (Dog) Maximizer.max(dogs);
      maxDog.bark();
   }
}
```

Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static OurComparable max(OurComparable[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {
         int cmp = items[i].compareTo(items[maxDex]);
         if (items[i] > items[maxDex]) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }

   public static void main(String[] args) {
      Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                    new Dog("Benjamin", 15)};
      Dog maxDog = (Dog) Maximizer.max(dogs);
      maxDog.bark();
   }
}
```

Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static OurComparable max(OurComparable[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {
         int cmp = items[i].compareTo(items[maxDex]);
         if (cmp > 0) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }

   public static void main(String[] args) {
      Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9),
                    new Dog("Benjamin", 15)};
      Dog maxDog = (Dog) Maximizer.max(dogs);
      maxDog.bark();
   }
}
```

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns -1 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {
       Dog uddaDog = (Dog) o;

       if (this.size < uddaDog.size) {
           return -1;
       } else if (this.size == uddaDog.size) {
           return 0;
       }
       return 1;
   }
}
```

This code is kind of long. We can simplify it with the following trick.

Coding Demo: OurComparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {
       Dog uddaDog = (Dog) o;

       return this.size - uddaDog.size;









   }
}
```

This code is kind of long. We can simplify it with the following trick.

Coding Demo: OurComparable

OurComparable.java

```
public interface OurComparable {
   /** Return -1 if this < o.
    *  Return 0 if this equals o.
    *  Return 1 if this > o.
    */
   public int compareTo(Object o);
}
```

We need to modify our interface specification accordingly.

Coding Demo: OurComparable

OurComparable.java

```
public interface OurComparable {
   /** Return negative number if this < o.
    *  Return 0 if this equals o.
    *  Return positive number if this > o.
    */
   public int compareTo(Object o);
}
```

We need to modify our interface specification accordingly.

The OurComparable Interface

```
public interface OurComparable {
    int compareTo(Object o);
}
```

Could have also been OurComparable. No meaningful difference.

Specification, returns:

- Negative number if ***this*** is less than obj.
- 0 if ***this*** is equal to object.
- Positive number if ***this*** is greater than obj.

General Maximization Function Through Inheritance

the origin of uddaDog

```
public interface OurComparable {
   int compareTo(Object o);
}
```

```
public class Dog implements OurComparable {
   public int compareTo(Object obj) {
       /** Warning, cast can cause runtime error! */
       Dog uddaDog = (Dog) obj;
       return this.size - uddaDog.size;
   } ...
```

```
public class Maximizer {
   public static OurComparable max(OurComparable[] a) {
   ...
   }
```

```
Dog[] dogs = new Dog[]{d1, d2, d3};
Dog largest = (Dog) Maximizer.max(dogs);
```

General Maximization Function Through Inheritance

Benefits of this approach:

- No need for array maximization code in every custom type (i.e. no `Dog.maxDog(Dog[])` function required).
- Code that operates on multiple types (mostly) gracefully, e.g.

```
OurComparable[] objs = getItems("somefile.txt");
return Maximizer.max(objs);
```

Compilation Error Puzzle

Lecture 9, CS61B, Fall 2024

Interfaces Quiz #1

```
public class DogLauncher {
  public static void main(String[] args) {
    ...
    Dog[] dogs = new Dog[]{d1, d2, d3};
    System.out.println(Maximizer.max(dogs));
   }
}
```

```
public class Dog
implements OurComparable {
  ...
  public int compareTo(Object o) {
      Dog uddaDog = (Dog) o;
      return this.size
             - uddaDog.size;
  } ...
```

Q: If we omit `compareTo()`, which file will fail to **compile**?

A. DogLauncher.java
B. Dog.java
C. Maximizer.java
D. OurComparable.java

```
public class Maximizer {
  public static OurComparable max(
          OurComparable[] items) {
    ...
    int cmp = items[i].
      compareTo(items[maxDex]);
    ...
  } ...
```

Interfaces Quiz #2: yellkey.com

```
public class DogLauncher {
  public static void main(String[] args) {
    ...
    Dog[] dogs = new Dog[]{d1, d2, d3};
    System.out.println(Maximizer.max(dogs));
   }
}
```

```
public class Dog
implements OurComparable {
  ...
  public int compareTo(Object o) {
     Dog uddaDog = (Dog) o;
     return this.size
             - uddaDog.size;
  } ...
```

Q: If we omit `implements OurComparable`, which file will fail to **compile**?

A. DogLauncher.java
B. Dog.java
C. Maximizer.java
D. OurComparable.java

```
public class Maximizer {
  public static OurComparable max(
          OurComparable[] items) {
    ...
    int cmp = items[i].
      compareTo(items[maxDex]);
    ...
  } ...
```

Answers to Quiz

Problem 1: Dog will fail to compile because it does not implement all abstract methods required by OurComparable interface. (And I suppose DogLauncher will fail as well since Dog.class doesn't exist)

Problem 2: DogLauncher will fail, because it tries to pass things that are not OurComparables, and Maximizer expects OurComparables.

Comparable

Lecture 9, CS61B, Fall 2024

Subtype Polymorphism vs. Explicit Higher Order Functions

Building a General Max Function

- The Naive Approach
- OurComparable
- Compilation Error Puzzle
- **Comparable**

Comparators

Some Miscellaneous Java Syntax

The Issues With OurComparable

Two issues:

- Awkward casting to/from Objects.
- We made it up.
  - No existing classes implement OurComparable (e.g. String, etc).
  - No existing classes use OurComparable (e.g. no built-in max function that uses OurComparable)

```
public class Dog implements OurComparable {
    public int compareTo(Object obj) {
        /** Warning, cast can cause runtime error! */
        Dog uddaDog = (Dog) obj;
        return this.size - uddaDog.size;
    } ...
```

```
Dog[] dogs = new Dog[]{d1, d2, d3};
Dog largest = (Dog) Maximizer.max(dogs);
```

The Issues With OurComparable

Two issues:

- Awkward casting to/from Objects.
- We made it up.
  - No existing classes implement OurComparable (e.g. String, etc).
  - No existing classes use OurComparable (e.g. no built-in max function that uses OurComparable)

The industrial strength approach: Use the built-in Comparable interface.

- Already defined and used by tons of libraries. Uses generics.

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

```
public interface OurComparable {
   public int compareTo(Object obj);
}
```

Coding Demo: Comparable

Dog.java

```
public class Dog implements OurComparable {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {
       Dog uddaDog = (Dog) o;

       return this.size - uddaDog.size;
   }
}
```

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

Replacing OurComparable with the built-in Comparable interface.

Coding Demo: Comparable

Dog.java

```
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Object o) {
       Dog uddaDog = (Dog) o;

       return this.size - uddaDog.size;
   }
}
```

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

Replacing OurComparable with the built-in Comparable interface.

Coding Demo: Comparable

Dog.java

```
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {


       return this.size - uddaDog.size;
   }
}
```

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

Replacing OurComparable with the built-in Comparable interface.

Coding Demo: OurComparable

Maximizer.java

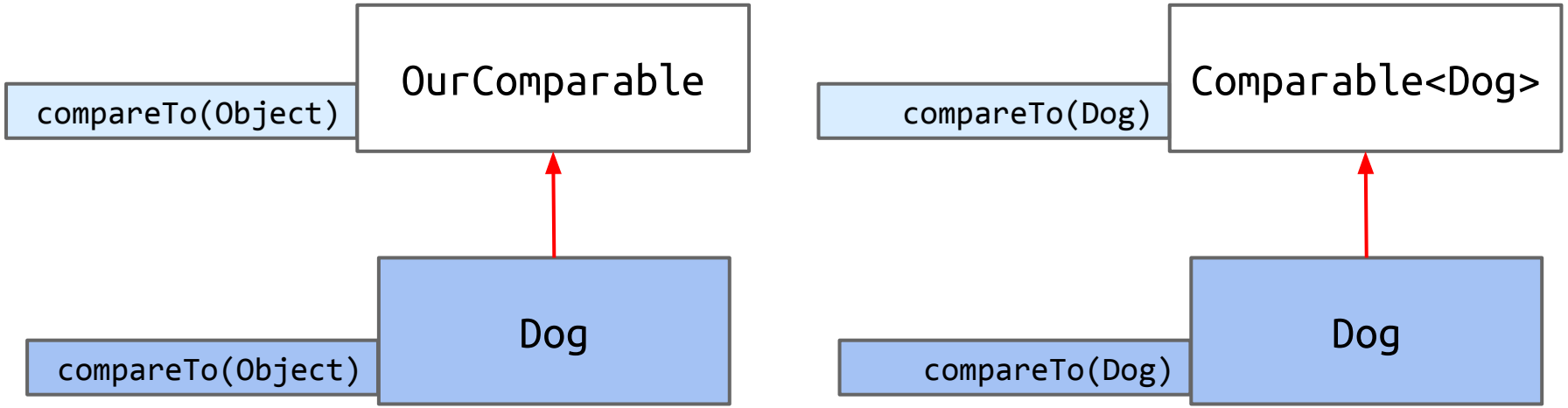
```java
public class Maximizer {
   public static OurComparable max(OurComparable[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {
         int cmp = items[i].compareTo(items[maxDex]);
         if (cmp > 0) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }
}
```

Coding Demo: OurComparable

Maximizer.java

```
public class Maximizer {
   public static Comparable max(Comparable[] items) {
      int maxDex = 0;
      for (int i = 0; i < items.length; i += 1) {
         int cmp = items[i].compareTo(items[maxDex]);
         if (cmp > 0) {
            maxDex = i;
         }
      }
      return items[maxDex];
   }
}
```

Comparable vs. OurComparable



Comparable Advantages

- Lots of built in classes implement Comparable (e.g. String).
- Lots of libraries use the Comparable interface (e.g. Arrays.sort)
- Avoids need for casts.

```
public class Dog implements Comparable<Dog> {
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }
```

← Much better!

```
public class Dog implements OurComparable {
    public int compareTo(Object obj) {
        Dog uddaDog = (Dog) obj;
        return this.size - uddaDog.size;
    } ...
```

Implementing Comparable allows library functions to compare custom types (e.g. finding max).

```
Dog[] dogs = new Dog[]{d1, d2, d3};
Dog largest = Collections.max(Arrays.asList(dogs));
```

Comparators

Lecture 9, CS61B, Fall 2024

Natural Order

The term “Natural Order” is sometimes used to refer to the ordering implied by a `Comparable`’s `compareTo` method.

- Example: Dog objects (as we’ve defined them) have a natural order given by their size.

“Doge”, size: 5

“Grigometh”, size: 200

“Clifford”, size: 9000

Natural Order

May wish to order objects in a different way.

- Example: By Name.

“Clifford”, size: 9000

“Doge”, size: 5

“Grigometh”, size: 200

Subtype Polymorphism vs. Explicit Higher Order Functions

Suppose we want to write a program that prints a string representation of the larger of two objects according to some specific comparison function.

Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

Can simply pass a different compare function.

Subtype Polymorphism Approach??

```
def print_larger(T x, T y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

Subtype Polymorphism vs. Explicit Higher Order Functions

Suppose we want to write a program that prints a string representation of the larger of two objects according to some specific comparison function.

Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

Can simply pass a different compare function.

Some possible designs (not the best):

- Add more functions `compareTo2`, `compareTo3`, `compareTo4`, etc.
- Add an extra argument to specify which comparison you want:
  `public int compareTo(Dog uddaDog, String whichCompare)`

Subtype Polymorphism vs. Explicit Higher Order Functions

Suppose we want to write a program that prints a string representation of the larger of two objects according to some specific comparison function.

Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

Subtype Polymorphism Approach

```
def print_larger(T x, T y, comparator<T> c):
   if c.compare(x, y):
       return x.str()
   return y.str()
```

Can simply pass a different compare function.

Coding Demo: Comparator

Dog.java

```
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }




}
```

Coding Demo: Comparator

Dog.java

```
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public class NameComparator implements Comparator<Dog> {



   }
}
```

Coding Demo: Comparator

Dog.java

```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public class NameComparator implements Comparator<Dog> {



   }
}
```

Coding Demo: Comparator

Dog.java

```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {

       }
   }
}
```

Coding Demo: Comparator

Dog.java

```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }
}
```

Coding Demo: Comparator

Dog.java

```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }
}
```

Coding Demo: Comparator

DogLauncher.java

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};









   }
}
```

Coding Demo: Comparator

DogLauncher.java

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = new Dog.NameComparator();









   }
}
```

Coding Demo: Comparator

DogLauncher.java

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = new Dog.NameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet

       }




   }
}
```

Coding Demo: Comparator

DogLauncher.java

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = new Dog.NameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       }




   }
}
```

Coding Demo: Comparator

DogLauncher.java

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = new Dog.NameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }


   }
}
```

Coding Demo: Comparator

Dog.java

Slight change to reflect Java convention.

```
import java.util.Comparator;
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   public static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }




}
```

Coding Demo: Comparator

Slight change to reflect Java convention.

Dog.java

```
import java.util.Comparator;
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   private static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }




}
```

Coding Demo: Comparator

Slight change to reflect Java convention.

Dog.java

```
import java.util.Comparator;
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   private static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }

   public static Comparator<Dog> getNameComparator() {

   }
}
```

Coding Demo: Comparator

Dog.java

Slight change to reflect Java convention.

```
import java.util.Comparator;
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   private static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }

   public static Comparator<Dog> getNameComparator() {
       return new NameComparator();
   }
}
```

Coding Demo: Comparator

DogLauncher.java

Slight change to reflect Java convention.

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = new Dog.NameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }

   }
}
```

Coding Demo: Comparator

DogLauncher.java

Slight change to reflect Java convention.

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Dog.NameComparator nc = Dog.getNameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }

   }
}
```

Coding Demo: Comparator

DogLauncher.java

Slight change to reflect Java convention.

```
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};




       Comparator<Dog> nc = Dog.getNameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }

   }
}
```

Coding Demo: Comparator

DogLauncher.java

Slight change to reflect Java convention.

```
import java.util.Comparator;
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};



       Comparator<Dog> nc = Dog.getNameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }

   }
}
```

Additional Orders in Java

In some languages, we'd write two comparison functions and simply pass the one we want :

- sizeCompare()
- nameCompare()

The standard Java approach: Create `SizeComparator` and `NameComparator` classes that implement the `Comparator` interface.

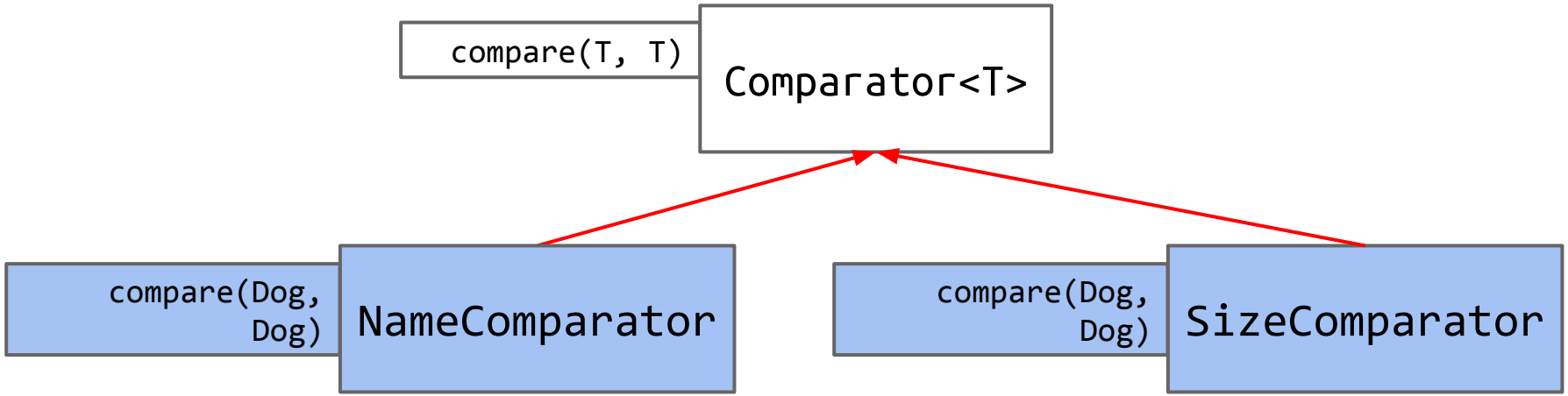
- Requires methods that also take Comparator arguments (see project 1C).

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Dogs and Comparators

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Dog not related by inheritance to any of the classes below.



Example: NameComparator

```
public class Dog implements Comparable<Dog> {
  private String name;
  private int size;

  public static class NameComparator implements Comparator<Dog> {
      public int compare(Dog d1, Dog d2) {
          return d1.name.compareTo(d2.name);
      }
  }
  ...
}
```

```
Comparator<Dog> cd = new Dog.NameComparator();
if (cd.compare(d1, d3) > 0) {
    d1.bark();
} else {
    d3.bark();
}
```

Result: If d1 has a name that comes later in the alphabet than d3, d1 barks.

Some Miscellaneous Java Syntax

Lecture 9, CS61B, Fall 2024

Java Access Modifiers

Java philosophy:

- When declaring variables, *minimize* the set of operations permitted on them.
- If you're only planning to use an SLList as a List61B, declare it as a List61B.
- This way, the compiler prevents you from using SLList-specific methods.

Why?

- Greater flexibility when you go back and change code.
  - Can swap SLList with any other List61B class easily.
- Enforces abstraction barriers.
  - Less for you to think about when building on top of that code.
- Communicates to future readers of your code your intended use for that variable.

```
SLList<Integer> L = new SLList<>();
```

Bad.

```
List61B<Integer> L = new SLList<>();
```

Good.

Final

If you declare a variable with `final`, you can only assign it a value *once*.

- Reassigning a `final` variable is not allowed.
- If you try, you'll get a compiler error.

```
final int x = 0;
x = 1;
```

Will not compile!
You can't reassign a final variable.

Protected

So far, we've seen:

- `public`: The instance variable is accessible by other classes.
- `private`: The instance variable is only accessible in the current class.
  - Not even accessible by subclasses!

```
public class SLList {
  private int size;
  ...
}
```

```
public class VengefulSLList extends SLList {
    public int removeLast() {
        size -= 1;
        ...
    }
    ...
}
```

Will not compile!
Subclass cannot access parent's private variables.

Protected

So far, we've seen:

- `public`: The instance variable is accessible by other classes.
- `private`: The instance variable is only accessible in the current class.
  - Not even accessible by subclasses!

Another access modifier you can use:

- `protected`: The instance variable is accessible in **the current class, and its subclasses**.

```
public class SLList {
  protected int size;
  ...
}
```

```
public class VengefulSLList extends SLList {
    public int removeLast() {
        size -= 1;
        ...
    }
    ...
}
```

Now, this will compile. Subclass can access parent's protected variables.