

CS61B



Lecture 10: Subtype Polymorphism vs. HoFs

- Dynamic Method Selection Puzzle
- Subtype Polymorphism vs. Explicit HoFs
- Application 1: Comparables
- Application 2: Comparators

Dynamic Method Selection Puzzle (Online Only)

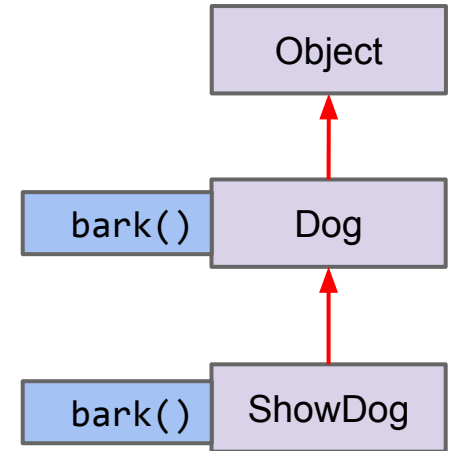
A Typing Puzzle

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 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.

These two practices above are called “hiding”.

- It is bad style.
- There is no good reason to ever do this.
- The rules for resolving the conflict are a bit confusing to learn.
- I decided last year to stop teaching it in 61B.
- But if you want to learn it, see

<https://docs.oracle.com/javase/tutorial/java/landl/override.html>

Subtype Polymorphism

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](http://www.stroustrup.com/glossary.html#Gpolymorphism) or CS164.

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.

Explicit
HoF
Approach

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

Sometimes called a “callback”.

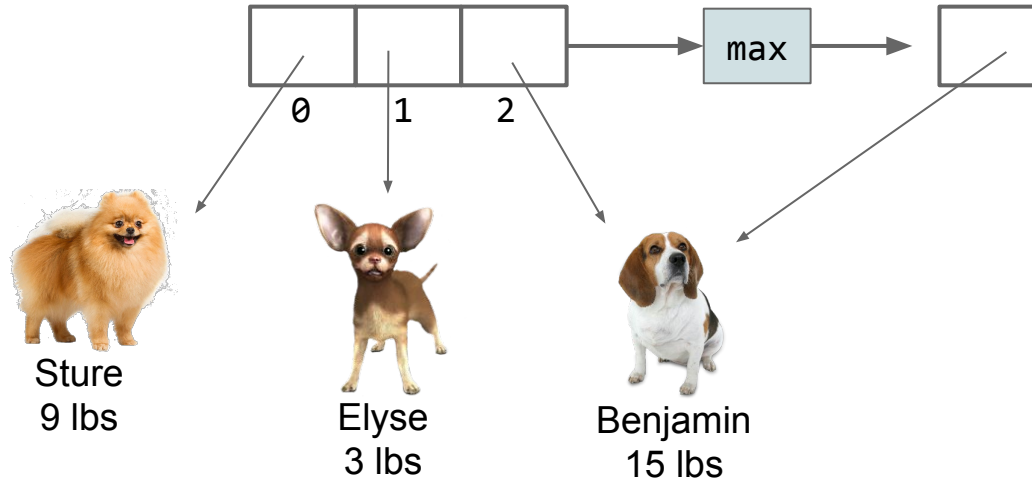
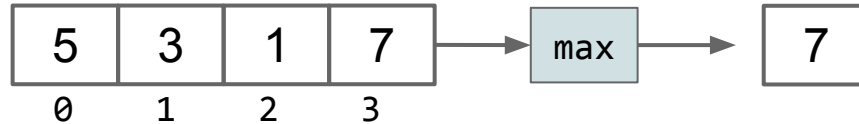
Subtype
Polymorphism
Approach

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

Not to be confused
with the amazing [Dr.
Ernest Kaulbach](#),
who taught my Old
English class.

DIY Comparison

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



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

```
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];  
}
```

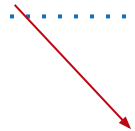
Maximizer.java

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

DogLauncher.java

Writing a General Max Function

and give up on our dream
of a one true max function



Objects cannot be compared to other objects with >

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

```
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];  
}
```

Maximizer.java

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

DogLauncher.java

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?

```
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];  
}
```

Maximizer.java

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

DogLauncher.java

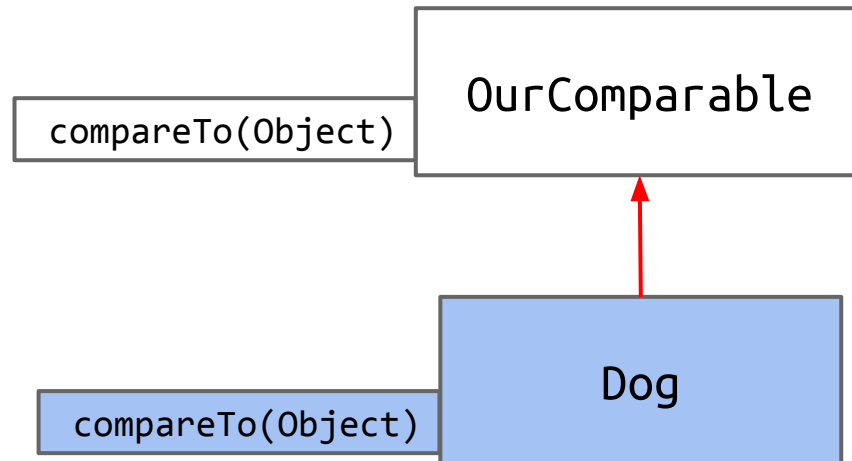
Solution

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

Create an interface that guarantees a comparison method.

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

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



The OurComparable Interface

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

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

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

```
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);
```

Interfaces Quiz #1: yellkey.com/special

```
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 Comparable {  
    ...  
    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 Comparable max(  
        Comparable[] items) {  
        ...  
        int cmp = items[i].  
            compareTo(items[maxDex]);  
        ...  
    } ...
```

Interfaces Quiz #2: yellkey.com/wish

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

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

- A. DogLauncher.java
- B. Dog.java
- C. Maximizer.java
- D. Comparable.java

```
public class Maximizer {  
    public static Comparable max(  
        Comparable[] 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.

Comparables

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)

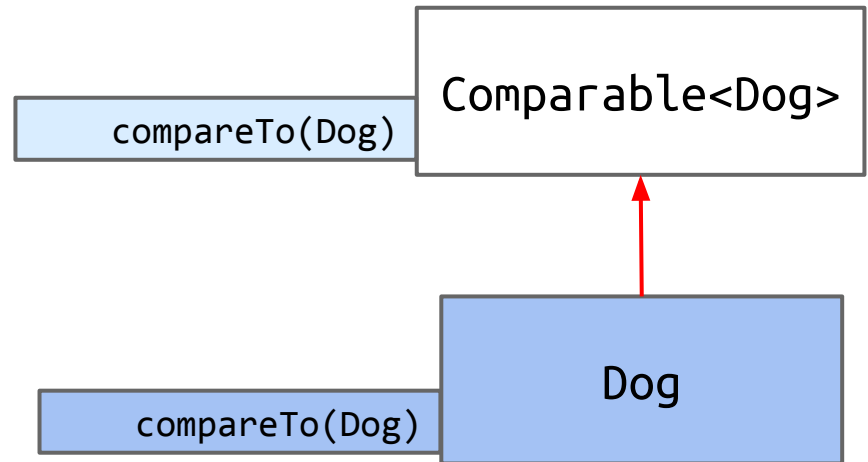
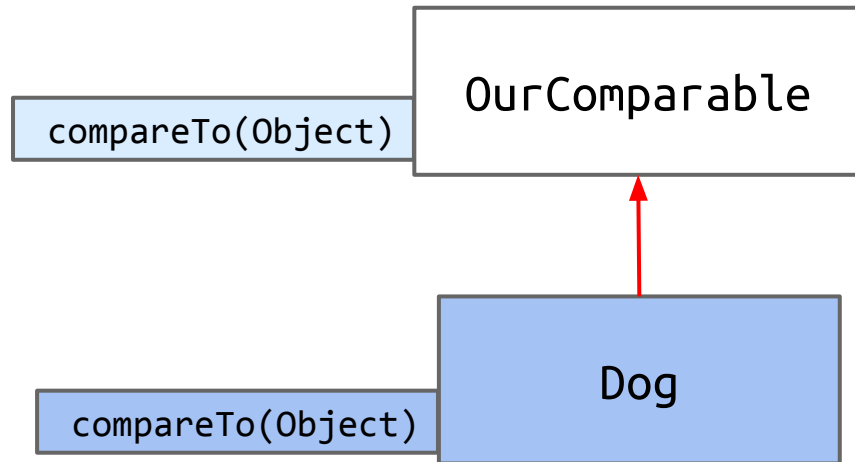
The industrial strength approach: Use the built-in Comparable interface.

- 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);  
}
```


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

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)
```



Subtype
Polymorphism
Approach??

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

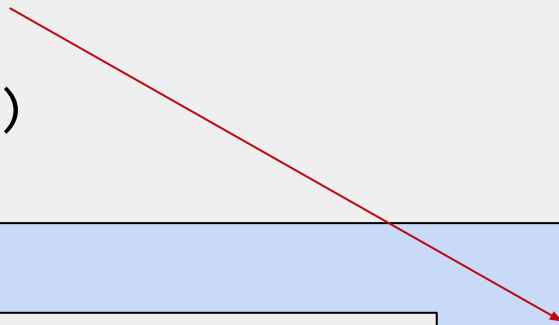
Can simply pass a
different compare
function.

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.

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 1B).

```
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.

Dog

compare(T, T)

Comparator<T>

compare(Dog,
Dog)

NameComparator

compare(Dog,
Dog)

SizeComparator

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.

Comparable and Comparator Summary

Interfaces provide us with the ability to make *callbacks*:

- Sometimes a function needs the help of another function that might not have been written yet.
 - Example: max needs compareTo
 - The helping function is sometimes called a “callback”.
- Some languages handle this using explicit function passing.
- In Java, we do this by wrapping up the needed function in an interface (e.g. `Arrays.sort` needs `compare` which lives inside the `comparator` interface)
- `Arrays.sort` “calls back” whenever it needs a comparison.
 - Similar to giving your number to someone if they need information.
 - See Project 1B to explore how to write code that uses comparators.

Citations

Title screenshot from [Neuromancer](#)

How to draw a doge: <http://i.imgur.com/iePIABL.png>

Demon Dog:

http://orig02.deviantart.net/e8fd/f/2011/154/5/e/pen_sketchbook_demon_dog_by_synnabar-d3hxrms.jpg