Recap: Toward Object-Oriented Design

- In good, old days… programs had no structures.
  - One dimensional code cannot deal with complex requirements.
- The notion of structures (or modularity) was introduced to manage complexity and improve reusability, maintainability and extensibility.
  - Modules in OOD: classes and interfaces
- How can/should you use classes and interfaces to gain reusability, maintainability and extensibility?
  - Reusability:
    - How easy (effortless) to use existing modules as they are (as a black box).
  - Maintainability:
    - How easy (cost effective) to revise existing code.
  - Extensibility:
    - How easy (pluggable) to introduce new features.

How to Gain Reusability, Maintainability and Extensibility?

- Design patterns can answer this question to some extent.
  - You can learn how to organize your code (i.e. how to use classes and interfaces, how to separate a class/interface from other classes/interfaces) to gain these properties.
  - *Strategy*
    - Making algorithms extensible and maintainable.
    - Making other data structures reusable.
  - *Visitor*
    - Making visitors extensible and maintainable.
    - Making other data structures reusable.
  - *Iterator*
    - Making access mechanisms (or drivers) extensible and maintainable.
    - Making other data structures reusable.
  - *State*
    - Making state-dependent behaviors extensible and maintainable.
    - Making other data structures reusable.

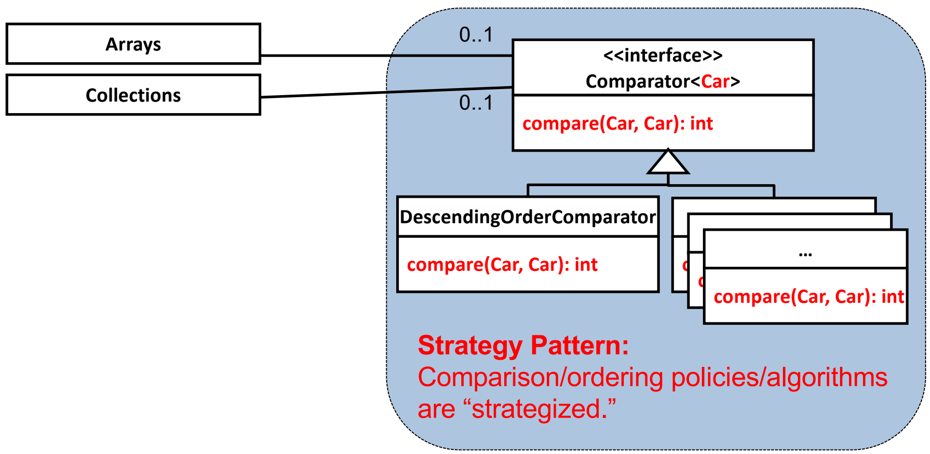
Unfortunately…

- You can learn about code organization for reusability, maintainability and extensibility only through writing and running your own code.
  - Only through DOING
  - Not talking.
  - Not listening to someone.
  - Not reading something.
  - Not drawing mental pictures.

Recap: Looking Ahead - AOP, Functional Programming, etc.

- OOD does a pretty good job in terms of modularity, but it is not perfect.
- OOD still has some modularity issues
  - Aspect Oriented Programming (AOP)
  - Dependency injection
    - Handles cross-cutting concerns well.
      - e.g. logging, security, DB access, transactional access to a DB
- Highly modular code sometimes look ***redundant***.
  - Functional programming
    - Makes code less redundant.
  - Lambda expressions in Java
    - Intended to make modular (OOD-backed) code less redundant.

An Example Redundancy



Functional Programming with Java

Notable Enhancements in Java 8

- Lambda expressions
  - Allow you to do *functional programming* in Java
- Static and default methods in interfaces

Lambda Expressions in Java

- Lambda expression
  - A block of code (or a function) that you can pass to a method.
- Before Java 8, methods could receive primitive type values and objects only.
  - `public void example(int i, String s, ArrayList<String> list)`
  - Methods could receive nothing else.
    - You couldn't do like this:
      ```
      – foo.example( [ if(Math.random()>0.5){
                          // Do something }
                        else{
                          // Do something else } ] )
      ```

How to Define a Lambda Expression?

- A lambda expression consists of
  - A code block
  - A set of parameters to be passed to the code block

```
– (String str) -> str.toUpperCase()

– (StringBuilder first, StringBuilder second)
               -> first.append(second)

– (int first, int second) -> second - first
```


- No need to specify the name of a function.
  - Lambda expression: *anonymous* function/method that is not bound to a class/interface

```
– (int first, int second)-> second - first

– public int subtract(int first, int second){
      return second - first; }
```


- No need to state the return type.
  - Your Java compiler automatically infers that.
- Single-expression code block does not require the **`return`** keyword.

```
– (int first, int second)-> second - first

– public int subtract(int first, int second){
      return second - first; }
```


- Multi-expression code block
  - Surround expressions with curly brackets (**{** and **}**). Use ; in the end of each expression.

```
– (double threshold)-> {
      if(Math.random() > threshold) return true;
      else return false; }

– () -> {
      if(Math.random) > 0.5) return true;
      else return false; }
```

- Multi-expression code block
  - Needs a **return** statement in each control flow.
    - Every conditional branch must return a value.

```
• () -> {
      if(Math.random() > 0.5) return true;
      else return false; }

• () -> {
      if(Math.random() > 0.5) return true;
  //    else return false; ← A compilation error occurs
                             here if this line is
                             commented out.
  }
```


How to Pass a Lambda Expression?

- A method can receive a lambda expression *as a method parameter*.

```
• foo.example( (int first, int second) -> second-first )
```

  - What is the type of that parameter?
    - ***Functional interface*!**

Functional Interface

- A special type of interface
  - An interface that has a single abstract (or empty) method.

Functional Interface

- A special type of interface
  - An interface that has a single abstract (or empty) method.
- Example: `java.util.Comparator`
  - Has `compare()`, which is the only abstract method.
    - A new annotation introduced in Java 8:

```
– @FunctionalInterface
  public interface Comparator<T>
```

      - All functional interfaces in Java API have this annotation.
        » The API documentation says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"

Using `Comparator`

- Example functional interface: `java.util.Comparator`
  - Has `compare()` as the only abstract method.

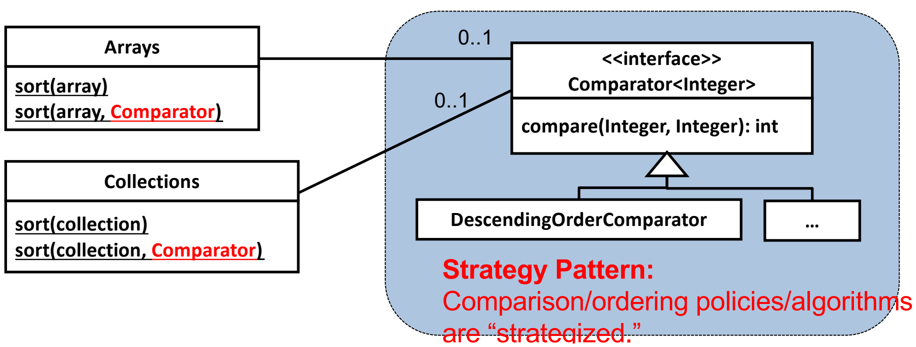
- `Collections.sort(List, Comparator<T>)`
  - The second parameter can accept a lambda expression (LE).
  - ```
    Collections.sort( aList,
                      (Integer first, Integer second)->
                          second.intValue()-first.intValue() );
    ```


Recap: `Collections.sort()`

- Sorting collection elements:
  - ```
    ArrayList<Integer> years2 = new ArrayList<Integer>();
    years2.add( Integer.valueOf(2010) );
    years2.add( Integer.valueOf(2000) );
    years2.add( Integer.valueOf(1997) );
    years2.add( Integer.valueOf(2006) );
    Collections.sort(years2);
    for(Integer y: years2)
        System.out.println(y);
    ```
  - `java.util.Collections`: a utility class (i.e., a set of static methods) to process collections and collection elements
  - `sort()` orders collection elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010

Comparison/Ordering Policies

- What if you want a custom (non-default) comparator?
  - `Collections.sort()` implement ascending ordering only.
    - They do not implement any other policies.
- Define a custom comparator by implementing `java.util.Comparator`



Sorting Collection Elements with a Custom Comparator

```
– ArrayList<Integer> years = new ArrayList<Integer>();
  years.add(new Integer(2010)); years.add(new Integer(2000));
  years.add(new Integer(1997)); years.add(new Integer(2006));

  Collections.sort(years);
  for(Integer y: years)
      System.out.println(y);

  Collections.sort(years, new DescendingOrderComparator());
  for(Integer y: years)
      System.out.println(y);
```

- 1997 -> 2000 -> 2006 -> 2010
- 2010 -> 2006 -> 2000 -> 1997

Okay, so What's the Point?

- Now, you have 2 different ways to do the same thing.
  - Without a lambda expression (LE)
    - ```
      public class DescendingOrderComparator<Integer>{
        implements Comparator<Integer>{
          public int compare(Integer o1, Integer o2){
            return o2.intValue()-o1.intValue(); } }

      Collections.sort(years, new DescendingOrderComparator());
      ```
  - With a lambda expression (LE)
    - ```
      Collections.sort(years,(Integer o1, Integer o2)->
                             o2.intValue()-o1.intValue());
      ```


- Without a LE
  - ```
    public class DescendingOrderComparator{
      implements Comparator<Integer>{
        public int compare(Integer o1, Integer o2){
          return o2.intValue()-o1.intValue(); } }

    Collections.sort(years, new DescendingOrderComparator());
    ```
- With a LE
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                           o2.intValue()-o1.intValue());
    ```
- Code gets more concise (less redundant/repetitive).
  - The LE defines `DescendingOrderComparator`'s `compare()` in a concise way.
- The LE version is a *syntactic sugar* for the non-LE version.
  - Your compiler does program transformation at compilation time.

FYI: Anonymous Class

- The most expressive (default) version
  - ```
    public class DescendingOrderComparator<Integer>{
      implements Comparator<Integer>{
        public int compare(Integer o1, Integer o2){
          return o2.intValue()-o1.intValue();
        }
    }
    Collections.sort(years, new DescendingOrderComparator());
    ```
- With an anonymous class
  - ```
    Collections.sort(years,
                     new Comparator<Integer>(){
                       @Override
                       public int compare(Integer o1, Integer o2){
                         return o2.intValue()-o1.intValue();
                       }
                     } );
    ```
- With a LE (more concise and less ugly)
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                           o2.intValue()-o1.intValue());
    ```


How Do You Know Where You can Use a Lambda Expression?

- `Collections.sort(List, Comparator<T>)`
- Check out `Comparator` in the API doc.
- Notice that `Comparator` is a functional interface.
  - ```
    @FunctionalInterface
    public interface Comparator<T>
    ```
    - The API doc says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"
  - This means you can pass a LE to `sort()`.

- Find out the abstract (or empty) method in `Comparator`.
  - `public int compare(T o1, T o2)`

- Define a LE that represents the body of `compare()` and pass it to `sort()`.

```
ArrayList<Integer> aList = new ArrayList<Integer>();
Collections.sort(aList,
                 (Integer first, Integer second)->
                    second.intValue()-first.intValue());
```


Assignment of a LE to a Functional Interface

- A LE can be assigned to a variable that is typed with a functional interface.

```
Comparator<Integer> comparator =
      (Integer o1, Integer o2)-> o2.intValue()-o1.intValue();
Collections.sort(years, comparator);
```

- Parameter types can be omitted through type inference.

```
Comparator<Integer> comparator =
      (o1, o2)-> o2.intValue()-o1.intValue()
```

```
ArrayList<Integer> aList = new ArrayList<Integer>();
Collections.sort(aList,
                 (first, second)->
                    second.intValue()-first.intValue());
```


What does `Collections.sort()` do?

```
class Collections
    static ... sort(List<T> list, Comparator<T> c){
        for each pair (o1 and o2) of elements in list{
            int result = c.compare(o1, o2);
            if(result < 0){
                ...
            }else if(result > 0){
                ...
            }else if(result==0){
                ...
      } } }
```

- c.f. Run this two-line code.

```
Comparator<Integer> comparator =
                       (o1, o2)-> o2.intValue()-o1.intValue();
comparator.compare(1, 10);
// compare() returns 9 (10-1)
```


Some Notes

- A LE can be assigned to a functional interface.

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

```
Comparator<Integer> comparator =
      (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()

Collections.sort(years, comparator);
```

- It CANNOT be assigned to `Object`.

```
Object comparator =
      (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
```

- Without a LE
  - ```
    public class DescendingOrderComparator<Integer>{
      implements Comparator<Integer>{
        public int compare(Integer o1, Integer o2){
          return o2.intValue()-o1.intValue();
        }
    }
    Collections.sort(years, new DescendingOrderComparator());
    ```
- With a LE
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                                o2.intValue()-o1.intValue());
    ```
- A type mismatch results in a compilation error.
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                              o2.floatValue()-o1.floatValue());
    ```
  - The return value type must be int, not float.

- A LE cannot throw an exception
  - if its corresponding functional interface does not specify that for its abstract method.
- Not good (Compilation fails.)
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2)
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                          if(...) throw new XYZException;
                          else return ... );
    ```
- Good
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2) throws ZYZException
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                          if(...) throw new XYZException;
                          else return ... );
    ```


LEs make Your Code Concise, but…

- You still need to clearly understand
  - the *Strategy* design pattern
    - `Comparator` and its implementation classes
    - What `compare()` is expected to do
- Using or not using LEs just impact how to *express* your code.
  - This does not impact how to *design* your code.

A Benefit of Using LEs

- Your code gets more concise (less redundant/repetitive).
  - This may or may not mean “easier to understand” depending on how much you are familiar with LEs.
    ```

Interfaces in Java 8

- *Functional interface*: a special type of interface that has a single ***abstract*** (or empty) method.
- Before Java 8, all methods defined in an interface were abstract.
  - `public interface Foo{`
    `public void boo() }`
  - `public interface Comparator<T>{`
    `public int compare(T o1, T o2) }`
  - No methods could have their bodies (ipmls) in an interface.
- Java 8
  - Introduces ***2 extra types*** of methods to interfaces: ***static*** *methods* and ***default*** *methods*.
- `Comparator<T>` in Java 8 has…
  - one abstract method (`compare()`)
  - many *static* and *default* methods.

Abstract Interface Methods

- Java 8 introduces the keyword `abstract`.
  - ```
    public interface Foo{
        public abstract void Boo()
    }
    ```
  - `abstract` can be omitted.
    - ```
      public interface Comparator<T>{
          public int compare(T o1, T o2)
      }
      ```
    - ```
      public interface Comparator<T>{
          public abstract int compare(T o1, T o2)
      }
      ```


Static Interface Methods

- ```
  public interface I1{
      public static int getValue(){ return 123; } }
  I1.getValue();      // Returns 123.
  ```
- ```
  public interface I2 extends I1{}
  I2.getValue();      // I2 does not inherit getValue(). Compilation error.
  ```
- ```
  public interface I2 extends I1{}
      public static int getValue(){ return 987; } }
  I2.getValue();      // I2 can override getValue(). Returns 987.
  ```
- ```
  public class C1 implements I1{}
  C1.getValue();      // Results in a compilation error.
  ```
- Can call a static method of an interface without a class that implements the interface.
  - Classes never implement/have static interface methods.

Examples in Java API

- `List.of()`
  - `static List<T> of (T... elements)`
    - ```
      List<Double> p1, p2;
      p1 = List.of(2.0,3.0);
      p2 = List.of(5.0,7.0);
      Distance.get(p1, p2);        // returns 5
      ```
- `Set.of()`
- `Map.of()`
- `Map.ofEntries()`

Default Interface Methods

- ```
  public interface I1{
      public default int getValue(){ return 123; } }
  I1.getValue();  // Cannot call it like a static method. Compilation error.
  ```
- ```
  public class C1 implements I1{}
  C1 c = new C1();
  c.getValue();   // Returns 123.
  ```
- ```
  public interface I2 extends I1{}
  public class C2 implements I2{}
  C2 c = new C2();
  c.getValue();   // I2 inherits getValue(). Returns 123.
  ```
- ```
  public interface I2 extends I1{
      public default int getValue(){ return 987; } }
  public class C2 implements I2{}
  C2 c = new C2();
  c.getValue();   // I2 can override getValue(). Returns 987.
  ```
- ```
  public class C1 implements I1{
      public int getValue(){ return 987; } }
  ```
- ```
  C1 c = new C1();
  c.getValue();   // C1 can override getValue(). Returns 987.
  ```



```
public interface I1{
    public default int getValue(){ return 123; } }
public class C1{
    public int getValue(){ return 987; } }

public class C2 extends C1 implements I1{}
C2 c = new C2();
c.getValue();    // Returns 987.
```
- **Precedence rule:** The super class's method precedes an interface's default method.
- You can call an interface's default method, if you want.

```
– public class C2 extends C1 implements I1{
      public int getValue(){
          return I1.super.getValue(); } }
– C2 c = new C2();
  c.getValue(); // Returns 123.
```



- ```
  public interface I1{
      public default int getValue(){ return 123; } }
  public interface I2 {
      public default int getValue(){ return 987; } }

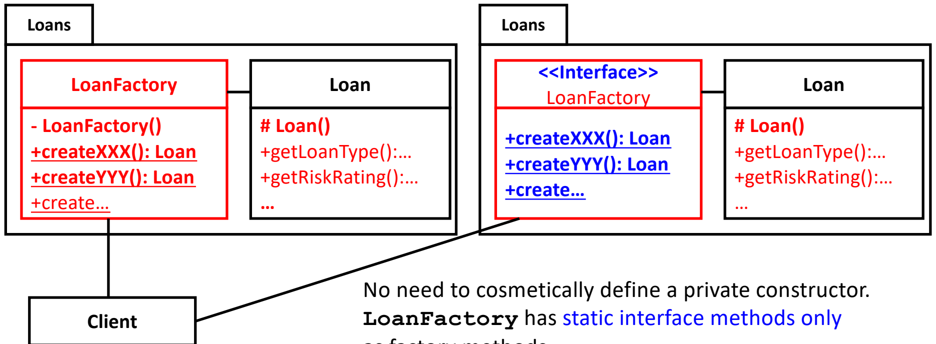
  public class C1 implements I1, I2{}      // Compilation error.
  ```
  - Default methods from different interfaces conflict.
- ```
  public class C1 implements I1, I2{
      public int getValue(){
          return I1.super.getValue(); } }    // Returns 123.
  ```


Examples in Java API

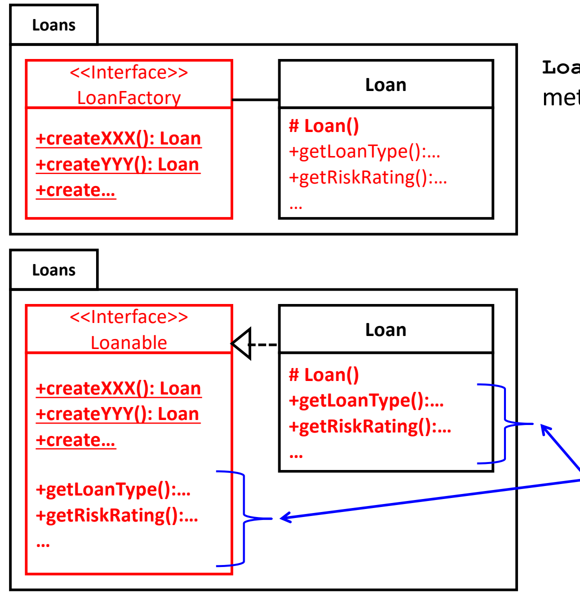
- `Map.getOrDefault()`
- `Map.putIfAbsent()`
- `Map.remove()`
- `Map.replace()`

Example Use Case of Static Interface Methods

- Static factory methods to create class instances that implement an interface.
- They can be implemented as static interface methods.



No need to cosmetically define a private constructor. `LoanFactory` has static interface methods only as factory methods.



`LoanFactory` has static interface methods only as factory methods.

`Loanable` can define loan-specific methods as abstract methods, if you want.

Then, `Loan` implements them.

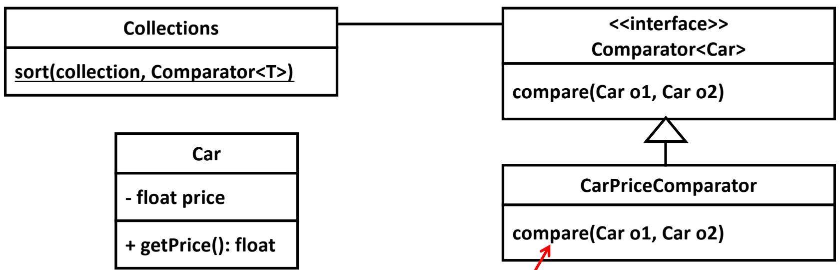
Static Methods in `Comparator`

- `java.util.Comparator<T>` has...
  - one abstract method (`compare()`) and
  - many *static* and *default* methods.
    - `static Comparator<T> comparing(Function<T, R> keyExtractor)`

- `java.util.Comparator<T>` has...
  - `static Comparator<T> comparing(Function<T, R> keyExtractor)`
    - Accepts a LE that extracts a `Comparable` sort key from `T`
      - Sort key (`R`): data/value to be used in ordering
      - `Function<T, R>`
        - » Represents a function (lambda expression) that accepts a parameter (`T`) and returns a result (R).
    - Returns a `Comparator<T>`

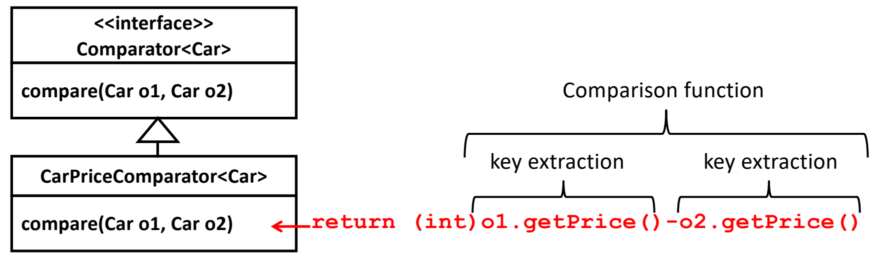
```
class Car{ private float getPrice(); }
ArrayList<Car> carList = new ...
...
Collections.sort(carList, Comparator.comparing(
                              (Car car)-> car.getPrice());
                    //comparing() returns a Comparator<Car>
```

- `class Car{ private float getPrice(); }`
- ```
  Collections.sort(carList,
                   Comparator.comparing(
                          (Car car)-> car.getPrice() );
  ```
- ```
  Collections.sort(carList,
                   (Car o1, Car o2)->
                          (int)o1.getPrice()-o2.getPrice())
  ```



`return (int)o1.getPrice()-o2.getPrice();`

- What `Comparator.comparing()` does is to
  - Transform a *key extraction function* to a *comparison function*
- *Higher-order function*
  - Accepts a function as a parameter and produces/returns another function as a result
    - `class Car{ public int getPrice();}`
    - ```
      Collections.sort(carList,
                       Comparator.comparing(
                          (Car car)-> car.getPrice() );
      ```
    - ```
      Collections.sort(carList,
                       (Car o1, Car o2)->
                          (int)o1.getPrice()-o2.getPrice());
      ```



- `Comparator.comparing()` uses ascending ordering (natural ordering) by default.
  - `class Car{ public float getPrice();}`
  - ```
    Collections.sort(carList,
                     Comparator.comparing(
                          (Car car)-> car.getPrice() );
    ```
  - ```
    Collections.sort(carList,
                     (Car o1, Car o2)->
                          (int)o1.getPrice()-o2.getPrice());
    ```
- What if you want *descending ordering*?
  - ```
    Collections.sort(carList,
                     Comparator.comparing((Car car)-> car.getPrice(),
                                          Comparator.reverseOrder());
    ```
  - ```
    Collections.sort(carList,
                     Comparator.comparing((Car car)-> car.getPrice(),
                                          Comparator.naturalOrder());
    ```


Benefits of Using LEs

- Can make your code more concise (less repetitive)
- Can enjoy the power of functional programming
  - e.g., higher-order functions

A Bit More about Comparator

- `class Car{ public float getPrice(); }`
- ```
  Collections.sort(carList,
                   Comparator.comparing( (Car car)-> car.getPrice() ));
  ```
- ```
  Collections.sort(carList,
                   Comparator.comparing( Car::getPrice ) );
  ```

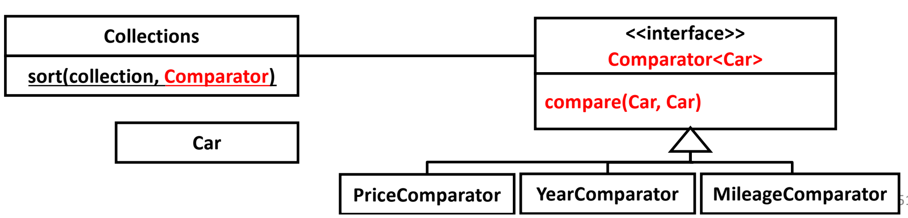
- *Method references* in lambda expressions
  - *object::method*
    - `System.out::println` (`System.out` contains an instance of `PrintStream`.)
    - `(int x) -> System.out.println(x)`
  - *Class::staticMethod*
    - `Math::max`
    - `(double x, double y) -> Math.max(x, y)`
  - *Class::method*
    - `Car::getPrice`
    - `(Car car)-> car.getPrice()`
    - `Car::setPrice`
    - `(Car car, int price)-> car.setPrice(price)`


- ```
  class Car{ public float getPrice();}
  Collections.sort(carList,
                   Comparator.comparing((Car car)-> car.getPrice() );
  ```
- ```
  Collections.sort(carList,
                   Comparator.comparing( Car::getPrice ) );
  ```
  - Ascending order (natural order) by default

- What if you want *descending ordering*?
  - ```
    Collections.sort(carList,
                     Comparator.comparing(Car::getPrice,
                                          Comparator.reverseOrder() );
    ```
  - ```
    Collections.sort(carList,
                     Comparator.comparing(Car::getPrice,
                                          Comparator.naturalOrder() );
    ```
  - ```
    Collections.sort(carList,
                     Comparator.comparing((Car::getPrice).reversed() );
    ```


HW 13

- Revise your HW 12 solution with LEs.
  - Instead of defining 4 classes that implement `Comparator<Car>`, define the body of each `compare()` method as a LE and pass it to `Collections.sort()`.



- Pass 4 different LEs to `Collections.sort()`
  - ```
    Collections.sort(carList,
                     (Car car1, Car car2)->{ ... } );
    ...
    ```
  - Use `Comparator.comparing()`, if you like. You will get an extra point.

- Create several `Car` instances and sort them with each lambda expression.
  - Minimum requirement: ascending ordering (natural ordering)
  - [Optional] Do descending ordering as well with `reverseOrder()` or `reserved()` of `Comparator`.