

## Interfaces in Java 8

- **Functional interface**: a special type of interface that has a **single abstract (or empty) method**.
  - Non-static and non-default 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**.
  - Calls traditional abstract/empty methods as *abstract methods*.
- `Comparator<T>` in Java 8 has...
  - one abstract method (`compare()`)
  - many *static* and *default* methods.

1

## 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)  
}
```

2

## 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.

3

## 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.

4

- ```

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.

```

5

- ```

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.

```
- ```

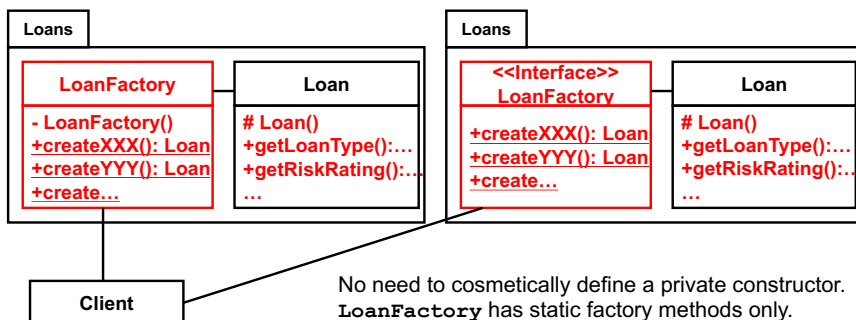
public class C1 implements I1, I2{
    public int getValue(){
        return I1.super.getValue(); } } // Returns 123.

```

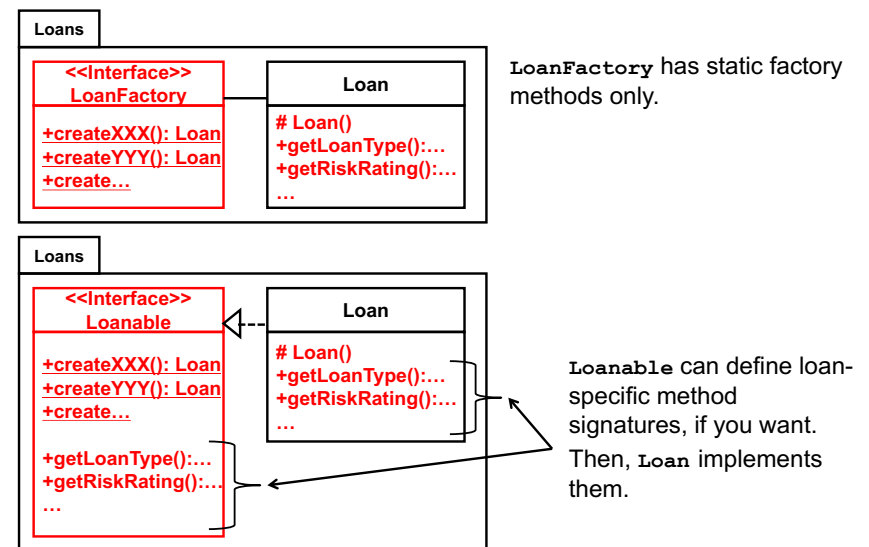
6

## Example Static Interface Methods

- Static factory methods** to create an object that implements an interface.
- They can be implemented as static interface methods.



7

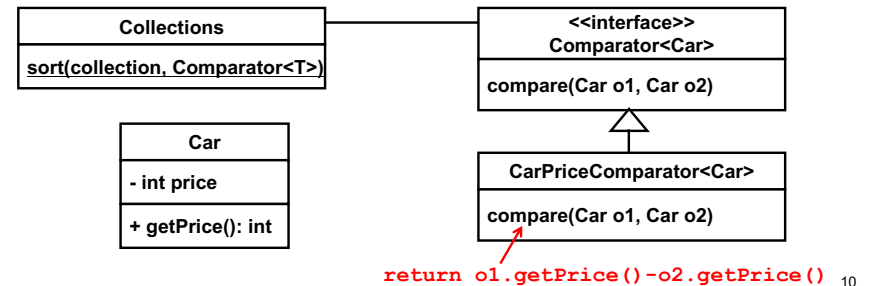


8

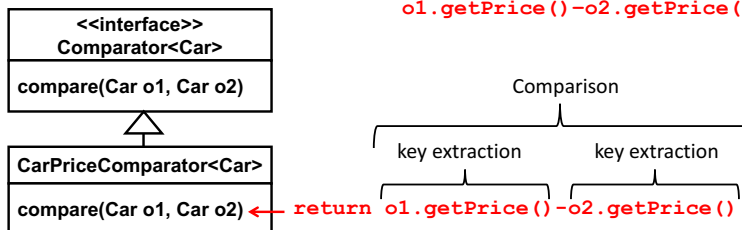
# Static Methods in Comparator

- `java.util.Comparator<T>` has...
  - one abstract method (`compare()`) and
  - many *static* and default methods.
- `static Comparator<T> comparing(Function<T, R> keyExtractor)`
  - Accepts a LE that extracts a **Comparable sort key** from `T`
    - Sort key (R): data/value to be used in ordering/sorting
    - `Function<T, R>`
      - » Represents a LE (lambda expression) that accepts a parameter (T) and returns a result (R).
      - » Its abstract method is: `R apply(T t)`.
  - Returns a `Comparator<T>`
- `class Car{ private int getPrice(); }`  
`Collections.sort(carList, Comparator.comparing(`  
`(Car car)-> car.getPrice() );`  
`//comparing() returns a Comparator<Car>` 9

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



- `Comparator.comparing()` uses/follows **ascending ordering** (natural ordering) by default.

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

- What if you want **descending ordering** with `comparing()`?

- `Collections.sort(carList,`  
`Comparator.comparing((Car car)-> car.getPrice(),`  
`Comparator.reverseOrder() );`
- `Collections.sort(carList,`  
`Comparator.comparing((Car car)-> car.getPrice(),`  
`Comparator.naturalOrder() );`

- `class Car{ public int getPrice();}`
- `Collections.sort(carList, Comparator.comparing( (Car car)-> car.getPrice() ));`
- `Collections.sort(carList, (Car o1, Car o2)-> o1.getPrice()-o2.getPrice());`
  - Ascending order (natural order) by default

### • What if you want *descending ordering* with `comparing()`?

- `Collections.sort(carList, Comparator.comparing( (Car car)-> car.getPrice(), Comparator.reverseOrder() ));`
- `Collections.sort(carList, Comparator.comparing( (Car car)-> car.getPrice(), Comparator.naturalOrder() ));`

13

## A Bit More about Comparator

- `class Car{ public int 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)`

15

## Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
- Can enjoy the power of functional programming
  - e.g., **higher-order functions**
- Can gain a new way to access collections
  - “Internal” iteration as opposed to traditional “external” iteration
    - Enables **Map-Reduce** data processing (a topic in CS681)
- Can simplify **concurrent programming (multi-threading)** in Java
  - A topic in CS681

14

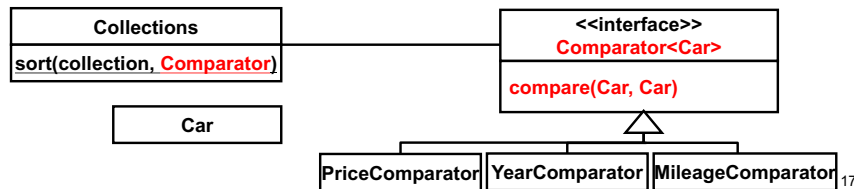
- `class Car{ public int getPrice();}`
- `Collections.sort(carList, Comparator.comparing( (Car car)-> car.getPrice() ));`
- `Collections.sort(carList, Comparator.comparing( Car::getPrice ));`
- `Collections.sort(carList, (Car o1, Car o2)-> o1.getPrice()-o2.getPrice());`
  - Ascending order (natural order) by default
- What if you want *descending ordering* with `comparing()`?
  - `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() ));`

16

## HW 13

- Work on HW 6 with lambda expressions, NOT with 3 classes that implement `Comparator`.
  - Implement the Car class

```
class Car{
    public int getPrice();
    public int getYear();
    public float getMileage(); }
```
  - Instead of defining 3 `Comparator` classes, define the body of each `compare()` method as a lambda expression and pass it to `Collections.sort()`.



- Pass 3 different lambda expressions to `Collections.sort()`
  - `Collections.sort(carList, (Car car1, Car car2)->{ ... } );`
  - `Collections.sort(carList, (Car car1, Car car2)->{ ... } );`
  - `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`.
  - [Optional] Implement Pareto comparison with a LE.