Object Oriented Programming in Java

10: Inner and nested classes. Anonymous classes. Lambda expressions.

Licence

Iterator and *Iterable* (1/2)

- If a class implements *Iterable* interface, then it can be used inside for-loop
  - *e.g. Collection* extends *Iterable*
- What about custom classes?
- Suppose that we have a custom class for storing a number and we want to iterate through the number's digits
  - We could extract digits, store it to a list (preferably unmodifiable), and return it as a result of a method call

```
MyNumber num = new MyNumber(12345);
for(Integer digit : num.digits())
       System.out.println(digit); // prints 1 2 3 4 5 respectively
```

Iterator and *Iterable* (2/2)

- What if we would like to write the following code:

```
MyNumber num = new MyNumber(12345);
for(Integer digit : num)
       System.out.println(digit); // prints 1 2 3 4 5 respectively
```

- In that case MyNumber should implement Iterable<Integer>
  - Interface Iterable<T> defines only one method to be implemented
    public Iterator<T> iterator()
  - Interface Iterator<T> defines that two methods need to be implemented
    public boolean hasNext() and *public T next()*
- First, a wrong implementation is given and discussed. Then the correct one would be shown, followed by improved version using nested and inner classes are given

Beginner’s attempt: Let’s store data in something that is iterable…

- Why not? Because you did not learn how to implement iterator
  - Prepares unnecessary data in advance (e.g. break iterating on odd digit) and consumes memory (suppose that instead digits it was a large array of elements)

```
public class MyNumber implements Iterable<Integer> {
    private int num;
    public MyNumber(int num) { this.num = num; }
    @Override
    public Iterator<Integer> iterator() {
        List<Integer> list = new LinkedList<>();
        int temp = num;
        while(temp > 0) { //assumption: num was positive
                list.add(0, temp % 10);
                temp /= 10;
        }
        return list.iterator();
  ...
```

10_InnerNestedLambda/swu/oopj/iterable/wrong/MyNumber.java

Iterator in a separate class

- Each time someone wants to iterate through the number's digits, new iterator instance is returned
  - Each iterator keeps it position

```
package swu.oopj.iterable;
import java.util.Iterator;
public class MyNumber implements Iterable<Integer> {
	private int num;
	public MyNumber(int num) {
		this.num = num;
	}

	@Override
	public Iterator<Integer> iterator() {
		return new DigitIterator(num);
	}
}
```

10_InnerNestedLambda/swu/oopj/iterable/MyNumber.java

How to get digits in desired order (1)

- Many ways… one is to calculate number of digits: $\lfloor \log n + 1 \rfloor$ and use division and module with proper exponent of 10.
- Next digits exist until number is positive
  - In each step one digit is removed from the number until it becomes zero

```
public class DigitIterator implements Iterator<Integer> {
  private int expOf10;
  private int num;
  public DigitIterator(int num) {
        this.num = num;
        expOf10 = (int) Math.pow(10, (int) Math.log10(num));
  }
  @Override
  public boolean hasNext() {
        return num > 0;
  }
  ...
```

10_InnerNestedLambda/swu/oopj/iterable/DigitIterator.java

How to get digits in desired order (2)

- If next digit exists divide number with the current exponent of 10, and prepare exponent and number for the next step

```
public class DigitIterator implements Iterator<Integer> {
  ...
  @Override
  public Integer next() {
        if (hasNext()) {
               int digit = num / expOf10;
               num %= expOf10;
               expOf10 /= 10;
               return digit;
        }
        else
               throw new NoSuchElementException("No more digits");
        }
}
```

10_InnerNestedLambda/swu/oopj/iterable/DigitIterator.java

Do we need to know for *DigitIterator*?

- The only thing we must know is that *MyNumber* is Iterable and returns an instance of iterator.
  - There is no need to know that it is exactly *DigitIterator*
    - In most cases this iterator does not have its purpose outside the context of *MyNumber*

```java
public class Main {
    public static void main(String[] args) {
        MyNumber number = new MyNumber(12345);
        for(Integer digit : number)
                System.out.println(digit); // prints 1 2 3 4 5
        System.out.println();

        for(Integer digit1 : number)
                for(Integer digit2 : number)
                        System.out.printf("%d - %d %n", digit1 , digit2);
    }
}
```

10_InnerNestedLambda/swu/oopj/iterable/Main.java

Nested classes

- Java allows that a class could be defined within another class. Such a class is called a nested class
- Way of logically grouping classes that are only used in one place.
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
  - Could leads to more readable and maintainable code (code of small nested classes is close to place of its only usage).
- Increases encapsulation:
  - Nested classes can access private variables of the outer class
  - In addition, nested class can be (but does not have to be) hidden be from the outside world.

Types of nested classes

- Nested classes are divided into two categories: static and non-static.

```
 public class OuterClass {
   static class StaticNestedClass {       …     }
   class InnerClass {        …      }
}
```

- Nested classes that are declared static are called static nested classes.
  - In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience, e.g.

    OuterClass.StaticNestedClass var = new OuterClass.StaticNestedClass()
- Non-static nested classes are called inner classes.
  - Instances of an inner class exist only within an instance of the outer class, e.g.

Iterator as a nested static class

- *DigitIterator* is implemented as a nested static class and marked as private
  - *Note: DigitIterator* would be able to access private members of *MyNumber* in case it has reference to object of type *MyNumber*

10_InnerNestedLambda/swu/oopj/iterable/nested//MyNumber.java

```
public class MyNumber implements Iterable<Integer> {
   private int num;
   public MyNumber(int num) {
       this.num = num;
   }

   @Override
   public Iterator<Integer> iterator() {
       return new DigitIterator(num);
   }
   private static class DigitIterator implements Iterator<Integer> {
               ...
```

Iterator as an inner class

- Inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
  - Thus it cannot define any static members itself.
- If they use same name for variables, reference to outer class variable is obtained by *OuterClassName.this.variable*

```
public class MyNumber implements Iterable<Integer> {
   private int num;
   ...
   public Iterator<Integer> iterator() {
       return new DigitIterator();
   }
   private class DigitIterator implements Iterator<Integer> {
      private int num;
      public DigitIterator() {
           this.num = MyNumber.this.num;
           ...
```

10_InnerNestedLambda/swu/oopj/iterable/inner/MyNumber.java

Variables of inner class type

- If inner class is not marked as private (it could be private, public, protected, or package private while outer classes can only be declared public or package private) then it is possible to create an instance of it, only as a part of some outer class

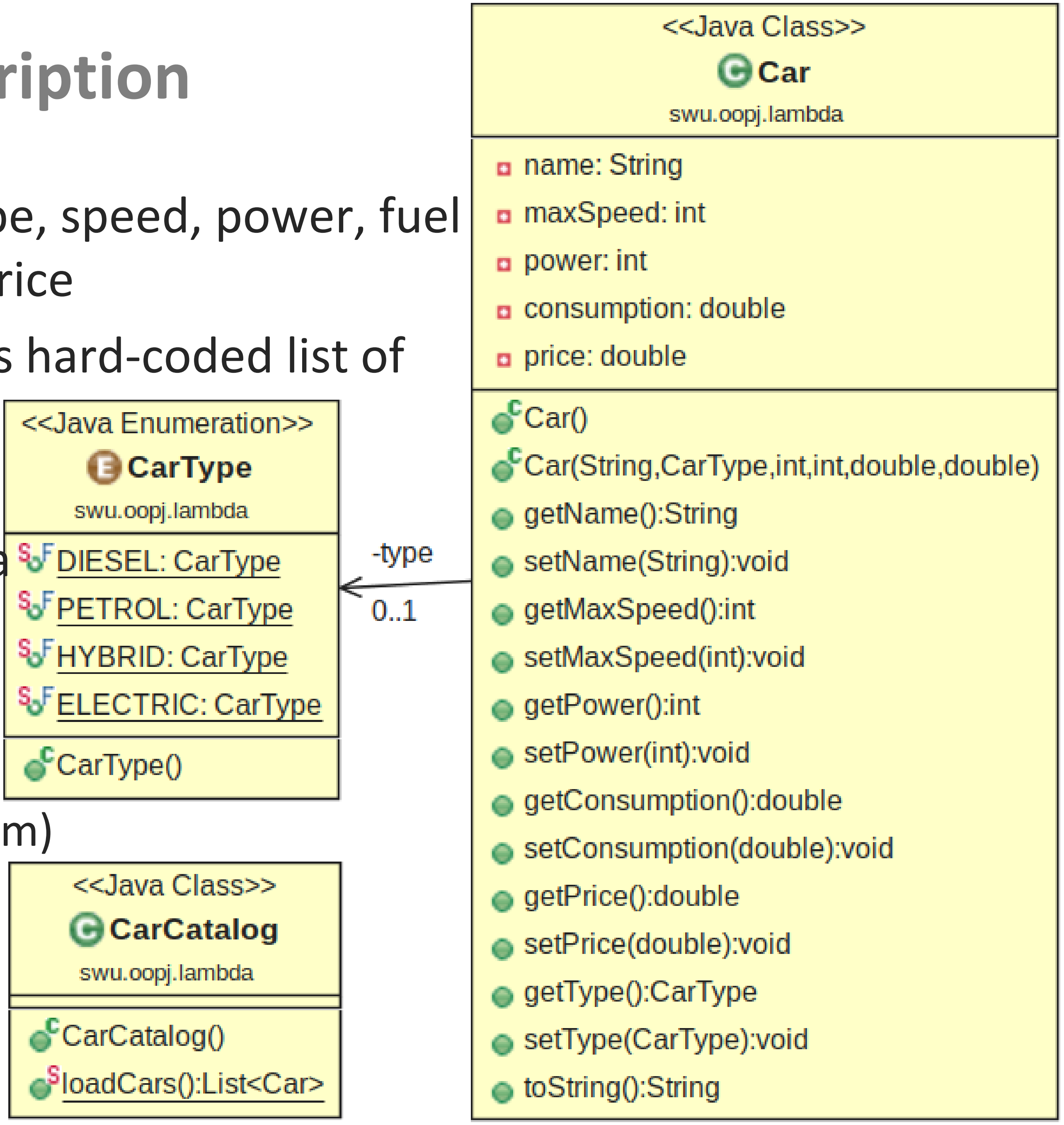
```
OuterClass outer = new OuterClass();
InnerClass inner = outer.new InnerClass();
```

```
StaticNestedClass nested = new OuterClass.StaticNestedClass();
```

Anonymous classes and lambda expression

Example description

- A Car has name, type, speed, power, fuel consumption and price
- *CarCatalog* contains hard-coded list of cars
- Print cars that satisfy some criteria
  - all petrol cars
  - all cars with
  - consumption less that 6 (e.g. l/100km)
  - All diesel chars cheaper than 100 000
  - …



Example #1 – print all diesel cars from the list

- Method argument does not have to be of List type
  - It just have to be *Iterable*

```
public static void main(String[] args) {
       List<Car> cars = CarCatalog.loadCars();
       printDieselCars(cars);
}
private static void printDieselCars(Iterable<Car> cars) {
       for(Car car : cars){
              if (car.getType() == CarType.DIESEL){
                     System.out.println(car);
              }
       }
}
```

10_InnerNestedLambda/swu/oopj/lambda/example1/Main.java

- What if we want to print all petrol cars?
  - Write a new method or change the method name to *printCars* and add parameter *carType*?

Behavior parametrization

- What if we want to extend previous method in order to print chars cheaper than 100 000?
  - Write another method?
  - Add additional criteria and additional argument saying what criteria should be used?
- How to cover so many combinations?
- Such methods would be almost the same
  - Iterating and printing is always the same
  - Only that changes is the part that checks selected criteria
- How to parametrize the changing part?
  - A similar example exists in C for quick sort: function *qsort* accepts generic (void) pointer, number of elements in array, size of single element, and pointer to function that compares two elements. The algorithm work same for any type and only needs those parameters.

Predicate and functional interfaces

- Predicate (in general): boolean value function, i.e. statement that may be true or false depending on the values of its variables.
- Java defines interface Predicate<T> with method

  boolean test(T t)
- Method *test* is the only method that should be written in a class in order to implement *Predicate*
- Interfaces with only one abstract method (only one that has to be implemented) are called ***functional interfaces***.
  - Annotated with *@FunctionalInterface* as a hint to compiler (helps identifying accidental violations of design intent)
  - Provide target types for lambda expressions and method references
    - Described few slides later
  - Note: Interface can contain default methods, but only this one is abstract

Example #2 – examples of various predicates

- Initially predicates described as separate classes

10_InnerNestedLambda/swu/oopj/lambda/example2/*.java

```
public class CheapCarPredicate implements Predicate<Car> {
       @Override
       public boolean test(Car car) {
              return car.getPrice() < 100000;
       }
}
```

```
public class DieselCarPredicate implements Predicate<Car> {
       @Override
       public boolean test(Car car) {
              return car.getType() == CarType.DIESEL;
       }
}
```

Example #2 – usage of various predicates

- The method for printing cars changed to accept a predicate on which it depends whether the car would be printed or not

10_InnerNestedLambda/swu/oopj/lambda/example2/Main.java

```
public static void main(String[] args) {
	List<Car> cars = CarCatalog.loadCars();
	System.out.println("Cheap cars:");
	printCars(cars, new CheapCarPredicate());
	System.out.println("Diesel cars:");
	printCars(cars, new DieselCarPredicate());
}
private static void printCars(Iterable<Car> cars,
                             Predicate<Car> predicate) {
	for(Car car : cars){
		if (predicate.test(car))
			System.out.println(car);
	}
}
```

Benefits of the previous solution

- Reduced code redundancy
  - There are no many methods with the (almost) same code
- Changing or adding new criteria (predicates) does not change *printCars* method
- Allows method reuse
  - Our “clever” method can be exported, and a user must only prepare a predicate
    - This concept would be demonstrated later with few built-in default methods in interfaces from Java Collection Framework

Example #3 – anonymous classes

- Predicates from example #2 were in separated classes, and used only use (and probably would be never used later)
  - For such cases Java anonymous classes could be used
- Declare and instantiate a class at the same time
  - The class does not have a name (anonymous)
  - The class extends or implements another class or interface

```
public static void main(String[] args) {
        List<Car> cars = CarCatalog.loadCars();
        printCars(cars, new Predicate<Car>(){
                @Override
                public boolean test(Car car) {
                        return car.getType() == CarType.DIESEL;
                }
        });
}
```

10_InnerNestedLambda/swu/oopj/lambda/example3/Main.java

How to define anonymous class

- Notice that *Predicate* is interface, so *new Predicate<Car>()* if used standalone would cause syntax error, but in this case, it does not mean we are (only) creating new instance of Predicate
  - It follows with curly brackets and overriding (implementing) methods from Predicate
- We are creating on object of anonymous class type that implements interface *Predicate<Car>* with implementation inside curly braces

```
printCars(cars, new Predicate<Car>(){
        @Override
        public boolean test(Car car) {
                return car.getType() == CarType.DIESEL;
        }
});
```

Properties of anonymous classes

- Anonymous classes can be defined by extending a class (does not have to an abstract class), or by implementing an interface
  - In case of abstract class or interface, all abstract method must be implemented (because we are creating an object at the same time)
  - It may have additional methods and fields
- Anonymous classes do not have names, thus they do not may have constructors, but may have initialization blocks

Anonymous classes and *this*

- An anonymous class defines new scope
  - Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.
  - <u>*this* in an anonymous class refers to the object of the anonymous class</u>
    - Object of the enclosing scope can be accessed with OuterClass.this

- An anonymous class has access to the members of its enclosing class and to final or effectively final local variables in its enclosing scope
  - A variable or parameter whose value is never changed after it is initialized is effectively final.

Example #4 – lambda expressions

- Anonymous classes adds many formal constructs and makes the code less readable
- In case that anonymous implementation of a functional interfaces is needed, lambda expression could be used instead

```
public static void main(String[] args) {
        List<Car> cars = CarCatalog.loadCars();
        printCars(cars, (car) -> {
                        return car.getPrice() < 100000;
                }
        );
        printCars(cars, (car) -> car.getType() == CarType.DIESEL);
}
```

10_InnerNestedLambda/swu/oopj/lambda/example4/Main.java

Lambda expressions

- As the second argument for *printCars* the following code are used

  (~~Car~~ car) -> car.getType() == CarType.DIESEL

  (~~Car~~ car) -> { return car.getPrice() < 100000; }

- How to read/interpret such code?
  - Compiler reads printCars's signature and find that object of type *Predicate<Car>* is expected.
  - Predicate is functional interface with one abstract (functional) method called *test* that accepts *Car* and returns *boolean*
  - On the left side of the arrow are parameters of the functional method (type can be omitted as it is deducted by its usage)
  - Right side contains return value (in this case *boolean*)
- If there are more than one statement on the right side, curly braces must be used, and *return* must be used if the functional methods is not of void type

Lambda expressions and *this*

- A lambda expression does not define new scope
- Thus, *this* in lambda expression has the same meaning as outside of lambda expression
  - different from anonymous classes

Example #5 – Functional interface *Consumer<T>*

- In the previous examples cars that satisfy a predicate were printed to system output. Let’s change the code to be more abstract general and leave the action of printing (or whatever else) to the one who would use the method.
- Instead of printing, some action is done (i.e. object is consumed)
  - Defined using interface *Consumer<T>* with functional method *void accept(T t)*

```
private static void printCars(Iterable<Car> cars,
                Predicate<Car> predicate, Consumer<Car> action) {
        for(Car car : cars){
                if (predicate.test(car)){
                        action.accept(car);
                }
        }
}
```

10_InnerNestedLambda/swu/oopj/lambda/example5/Main.java

Example #5 – Functional interface *Consumer<T>*

- Object of type Consumer<Car> can be defined in various ways
- In this example it is done using lambda expression.

10_InnerNestedLambda/swu/oopj/lambda/example5/Main.java

```
public static void main(String[] args) {
       List<Car> cars = CarCatalog.loadCars();
       printCars(cars,
              (car) -> car.getPrice() < 100000,
              (car) -> System.out.println("Cheap car: " + car)
       );
       printCars(cars,
              car -> car.getType() == CarType.DIESEL,
              car -> System.out.println("Diesel car: " + car)
       );
}
private static void printCars(Iterable<Car> cars,
              Predicate<Car> predicate, Consumer<Car> action) {
       ...
```

Example #6 – *BiFunction* and *BiConsumer* *(1/2)*

- Write a method to find two most similar cars and do some action with them
- Criteria for the car similarity is not know in advance
  - print cars, remove them from list, reduce price, … whatever
- Similarity function is (mathematically) $f: Car\ x\ Car\ \rightarrow\ \mathbb{Z}$
  - Function of two variables that return an integer
  - Java contains functional interface for this case
    public interface *BiFunction<T, U, R>* with method *R apply(T t, U u);*
    - accepts t (of type T) and u (of type U) and return the result of type R
- *Consumer<T>* consumes (do an action on) an object of type T
- *BiConsumer<T, U>* consumes (do an action on) two object of types T and U respectively

Example #6 – *BiFunction and BiConsumer* (2/2)

- As in previous example required function interfaces are implemented using lambda expressions

10_InnerNestedLambda/swu/oopj/lambda/example6/Main.java

```
public static void main(String[] args) {
       List<Car> cars = CarCatalog.loadCars();
       theMostSimilarCar(cars,
          (a, b) -> (int) Math.abs(a.getPrice() - b.getPrice()),
          (a, b) -> System.out.format(
                      "The most similar are: %n\t%s%n\t%s%n", a, b)
       );
}

public static void theMostSimilarCar(Iterable<Car> cars,
                    BiFunction<Car, Car, Integer> distanceFunction,
                    BiConsumer<Car, Car> action){
       ...
```

Example #6 – Local classes

- Pair of cars stored in a custom class that is not needed outside our method
  - Implemented as a local class

```
public static void theMostSimilarCar(Iterable<Car> cars,
       BiFunction<Car, Car, Integer> distanceFunction,
       BiConsumer<Car, Car> action){

              class CarPair{
                     public Car first, second;
                     public CarPair(Car first, Car second){
                            this.first = first;
                            this.second = second;
                     }
              }

              CarPair pair = null;
              ...
```

10_InnerNestedLambda/swu/oopj/lambda/example6/Main.java

Example #6 – The rest of the code

```
public static void theMostSimilarCar(Iterable<Car> cars,
        BiFunction<Car, Car, Integer> distanceFunction,
        BiConsumer<Car, Car> action){
        ...
  int min = Integer.MAX_VALUE; //although it could be anything
  for(Car first : cars){
        for(Car second : cars){
               if (first == second) continue;
               int distance = distanceFunction.apply(first, second);
               if (pair == null || distance < min){
                      pair = new CarPair(first, second);
                      min = distance;
               }
        }
  }
  if (pair != null)
        action.accept(pair.first,  pair.second);
}
```

10_InnerNestedLambda/swu/oopj/lambda/example6/Main.java

Example #7 – *Reference to a method*

- If there is an appropriate method, a reference to the method can be passed using *Class::methodName* or *object::methodName*
  - depends whether the method is static or not

```
public static void main(String[] args) {
       List<Car> cars = CarCatalog.loadCars();
       theMostSimilarCar(cars,
              Main::distance,
              (a, b) -> System.out.format(
                     "The most similar are: %n\t%s%n\t%s%n", a, b)
       );
}
private static int distance(Car a, Car b) {
       return (int) Math.abs(a.getPrice() - b.getPrice());
}
public static void theMostSimilarCar(Iterable<Car> cars,
                     BiFunction<Car, Car, Integer> distanceFunction,
                     BiConsumer<Car, Car> action){ ...
```

10_InnerNestedLambda/swu/oopj/lambda/example7/Main.java

When to use which type of class

https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html

- Lambda expression:
  - When encapsulating a single unit of behavior (simple instance of a functional interface) that need to be passed to other code
    - no need for a constructor, a named type, fields, or additional methods
- Anonymous class
  - When fields or additional methods must be declared or if lambda expression cannot be used
    - e.g. if implementing something that is not function interface
- Local class:
  - If not needed outside a method in which is defined, but we need its name for some reason, e.g. constructor, definition of variable(s)
- Static Nested class:
  - When the type must be more widely available (comparing to local class)
- Inner class:
  - When access to an enclosing instance's non-public fields and methods is needed

Using functional interfaces for some *List* and *Map* default methods

A “classic” approach to print a map content

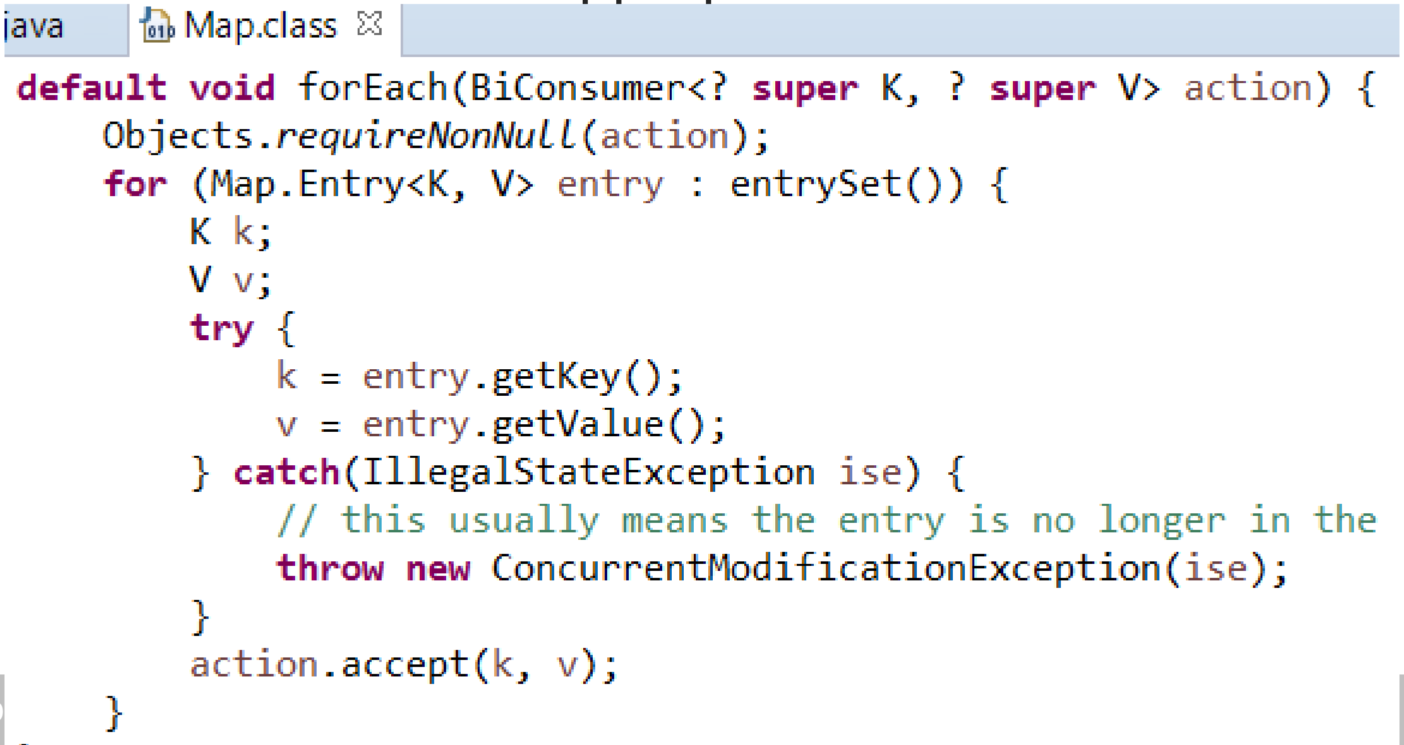
- Suppose there is a map *Map<CarType, Integer> carTypesCount* that count occurrence of each car type
- An iterative way to print map content would be something like

```
for(Map.Entry<CarType, Integer> entry : carTypesCount.entrySet()) {
  System.out.format("%s occured %d times. %n",
              entry.getKey(), entry.getValue());
}
```

- Since Java 8 there are several useful default methods in Map, List and other interfaces from Java Collection Framework

forEach method in *Map*

- *Map* contains *forEach* that does exactly what we wanted to achieve in previous iterative code
  - Instead of printing *forEach* do some action (consumes) each pair as defined with parameter of type BiConsumer <K, V>
    - forEach calls *void accept(K k, V v)* sending key and value to the method
- A user of this method writes an appropriate *BiConsumer*



```
default void forEach(BiConsumer<? super K, ? super V> action) {
    Objects.requireNonNull(action);
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch(IllegalStateException ise) {
            // this usually means the entry is no longer in the
            throw new ConcurrentModificationException(ise);
        }
        action.accept(k, v);
    }
```

forEach method in *Map* – an example of use

- Using anonymous classes

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
...
carTypesCount.forEach(new BiConsumer<CarType, Integer>() {
       @Override
       public void accept(CarType t, Integer u) {
              System.out.println(t + " occured " + u + " times");
       }
});
```

10_InnerNestedLambda/swu/oopj/defmethods/ExampleMapCompute.java

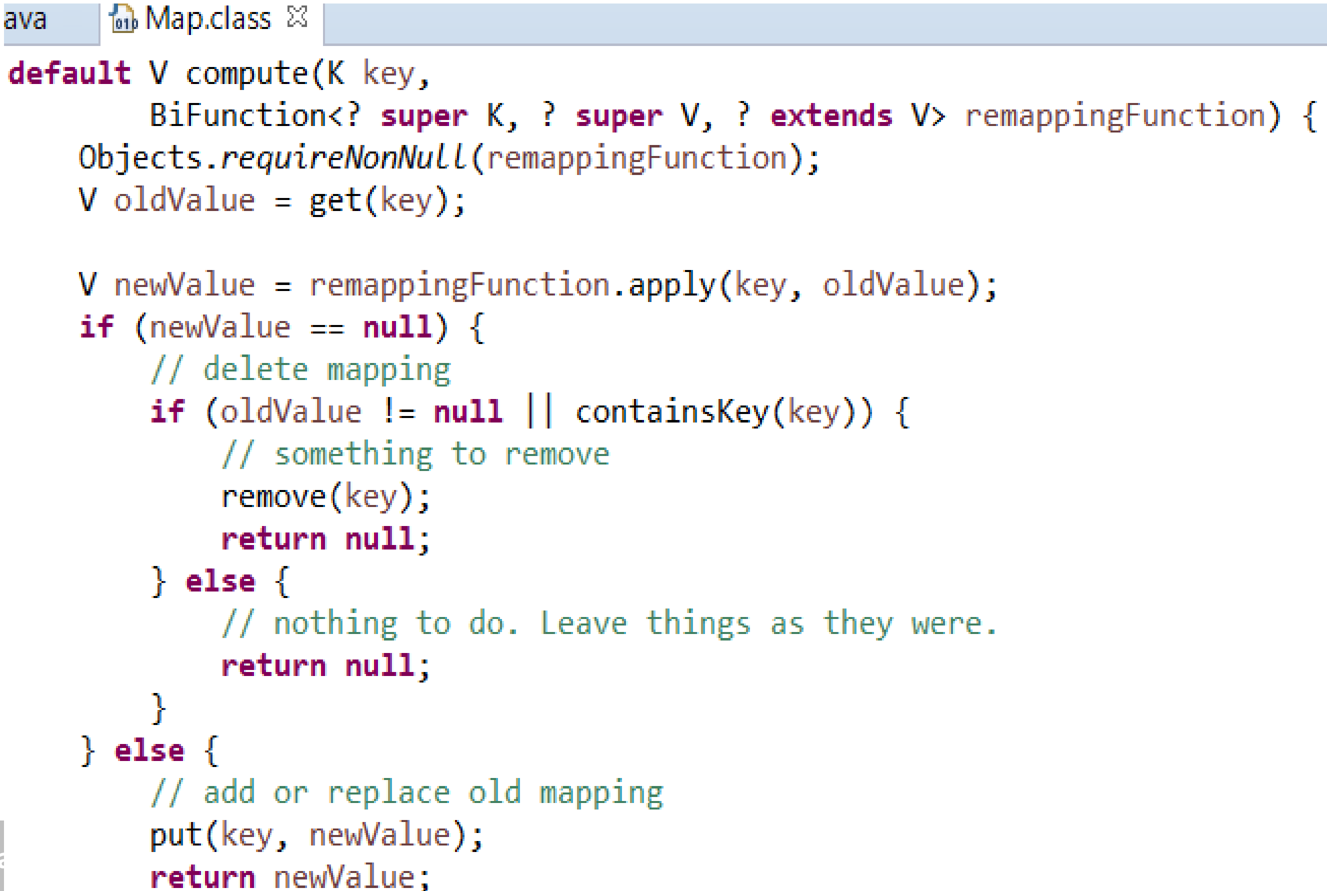
- Using lambda expression

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
...
carTypesCount.forEach(
   (type, num) ->
       System.out.println(type + " occured " + num + " times"));
```

10_InnerNestedLambda/swu/oopj/defmethods/ExampleMapComputeLambda.java

compute method in *Map*

- *compute* changes assigned value, or add or remove entry from a map based on the old and the new value
  - Needs a BiFunction<K, V, V> to calculate new
    - *compute* calls *V apply(K k, V v)* sending key and old value and uses return value to decide what to do
- Hides complexity of the algorithm and a user only needs to write appropriate *BiFunction*



```
default V compute(K key,
        BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    V oldValue = get(key);

    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue == null) {
        // delete mapping
        if (oldValue != null || containsKey(key)) {
            // something to remove
            remove(key);
            return null;
        } else {
            // nothing to do. Leave things as they were.
            return null;
        }
    } else {
        // add or replace old mapping
        put(key, newValue);
        return newValue;
```

compute method in *Map* – an example of use (1)

10_InnerNestedLambda/swu/oopj/defmethods/ExampleMapCompute.java

- Counting how many cars of each type has occurred
  - Using anonymous classes

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
List<Car> cars = CarCatalog.loadCars();
cars.forEach(new Consumer<Car>() {
   @Override
   public void accept(Car t) {
       Integer newVal = carTypesCount.compute(t.getType(),
              new BiFunction<CarType, Integer, Integer>() {
                  @Override
                  public Integer apply(CarType key, Integer value){
                      return value == null ? 1 : value + 1;
                  }
               });
        System.out.printf("%s raises number of %s cars to %d %n",
              t.getName(), t.getType(), newVal);
   }
});
```

compute and super and extends (1)

- In the example we have exact types

```
Integer newVal = carTypesCount.compute(t.getType(),
       new BiFunction<CarType, Integer, Integer>() {
             @Override
             public Integer apply(CarType key, Integer value) {
                      return value == null ? 1 : value + 1;
             }
        });
```

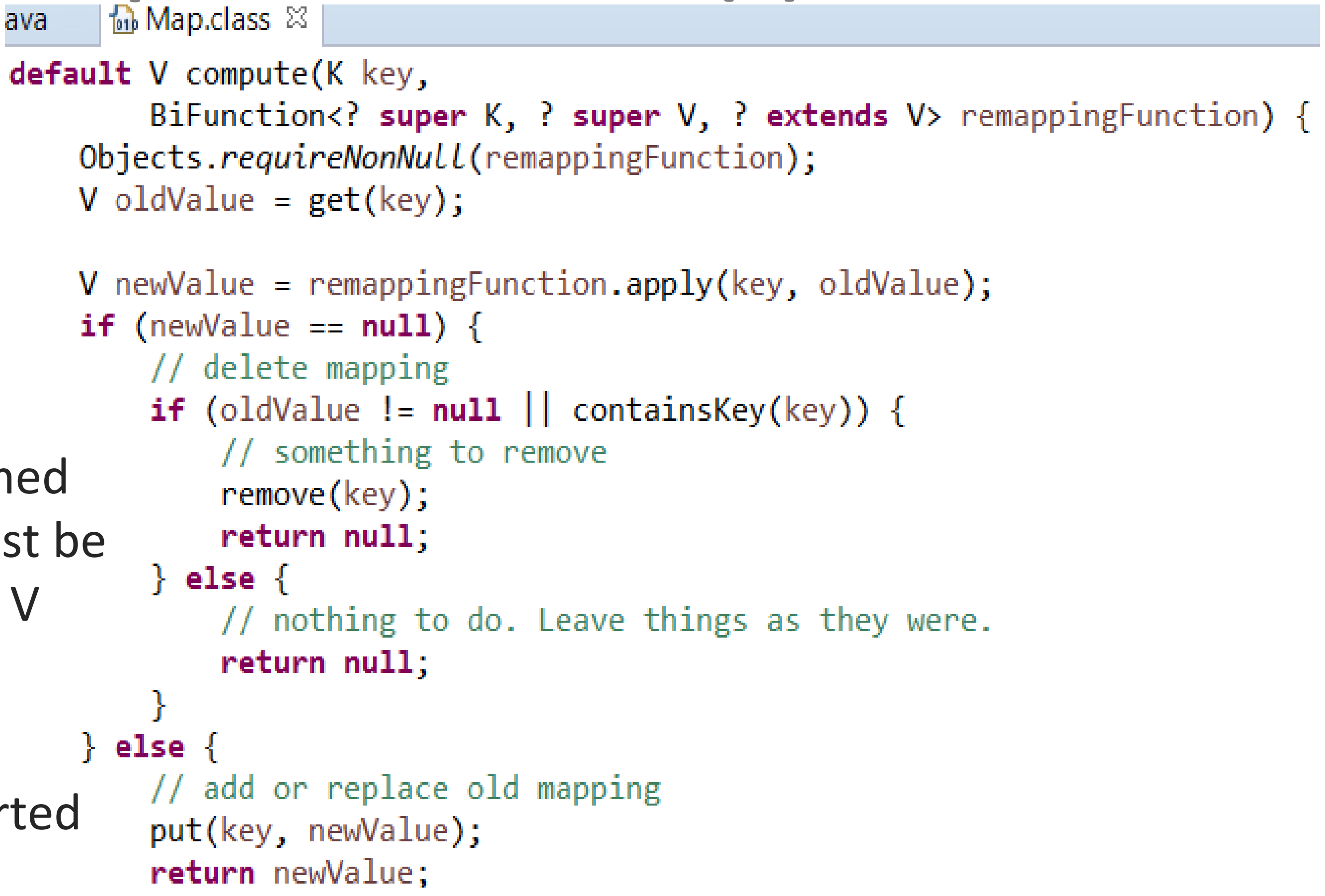
- Suppose that Integer could be extended (in reality it is final), then we could have a hierarchy Object – Number – Integer – MyInt
  - Then the following code would be valid

```
Integer newVal = carTypesCount.compute(t.getType(),
       new BiFunction<CarType, Number, MyInt>() {
             @Override
             public MyInt apply(CarType key, Number value) {
                      ...
```

compute and super and extends (2)

- PECS

  "Producer extends Consumer super"
- compute sends key and value of types K and V to be consumed by a method that must be able to receive K and V
  - upcast is allowed
- Returned (produced) value must be converted to V
  - It would be upcasted to V (thus it may be some subclass of V)



```
default V compute(K key,
        BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    V oldValue = get(key);

    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue == null) {
        // delete mapping
        if (oldValue != null || containsKey(key)) {
            // something to remove
            remove(key);
            return null;
        } else {
            // nothing to do. Leave things as they were.
            return null;
        }
    } else {
        // add or replace old mapping
        put(key, newValue);
        return newValue;
```

```
Integer newVal = carTypesCount.compute(t.getType(),
       new BiFunction<CarType, Number, MyInt>() {
            @Override
            public MyInt apply(CarType key, Number value) {
```

compute method in *Map* – an example of use (2)

- Counting how many cars of each type has occurred
  - Using lambda expressions

10_InnerNestedLambda/swu/oopj/defmethods/ExampleMapComputeLambda.java

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
List<Car> cars = CarCatalog.loadCars();
cars.forEach(car -> {
   Integer newVal = carTypesCount.compute(car.getType(),
               (key, value) -> value == null ? 1 : value + 1);
   System.out.printf("%s raises number of %s cars to %d %n",
                      car.getName(), car.getType(), newVal);
});
```

merge method in *Map*

- Merge is similar to compute
  - If the specified key (first parameter) is not already associated with a value or is associated with null, associates it with the given non-null value (second argument)
  - Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null

10_InnerNestedLambda/swu/oopj/defmethods/ExampleMapMergeLambda.java

```
Map<CarType, Integer> carTypesCount = new HashMap<>();
List<Car> cars = CarCatalog.loadCars();
cars.forEach(car -> carTypesCount.merge(car.getType(), 1,
                            (oldValue, value) -> oldValue + value));
```