Lecture 7

Functional Programming

Passing code with behavior parameterization

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Concordia University

Coping with changing requirements

René Witte
Concordia University

Requirement 1: Filter green apples

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
  List<Apple> result = new ArrayList<>();
  for(Apple apple: inventory){
    if( "green".equals(apple.getColor() ) {
      result.add(apple);
    }
  }
  return result;
}
```

Coping with changing requirements

René Witte
Concordia University

Requirement 2: Filter by arbitrary color

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
                                              String color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
      if ( apple.getColor().equals(color) ) {
        result.add(apple);
      }
    }
    return result;
}
```

Calling

```
List<Apple> greenApples = filterApplesByColor(inventory, "green");
List<Apple> redApples = filterApplesByColor(inventory, "red");
...
```

Coping with changing requirements

René Witte

Requirement 3: Filter by weight

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                               int weight) {
  List<Apple> result = new ArrayList<>();
  for (Apple apple: inventory){
    if ( apple.getWeight() > weight ){
      result.add(apple);
    }
  }
  return result;
}
```

Coping with changing requirements

Requirement 4: Filter by weight *or* color

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       String color,
                                       int weight,
                                       boolean flag) {
  List<Apple> result = new ArrayList<>();
  for (Apple apple: inventory){
    if ( (flag && apple.getColor().equals(color)) ||
         (!flag && apple.getWeight() > weight) ){
      result.add(apple);
    }
  }
  return result;
}
```

Calling

```
List<Apple> greenApples = filterApples(inventory, "green", 0, true);
List<Apple> heavyApples = filterApples(inventory, "", 150, false);
...
```

René Witte

Outline

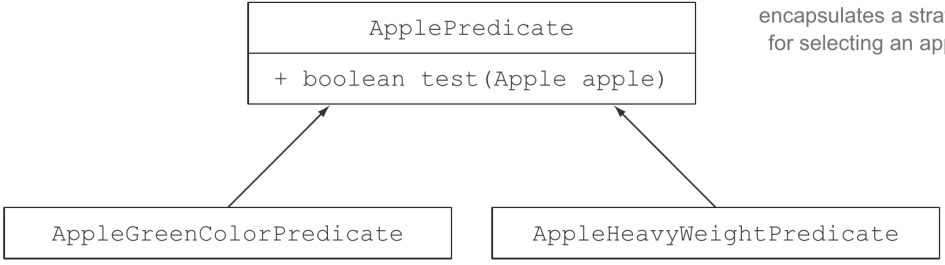
Behavior parameterization

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

René Witte

Apple selection strategies

René Witte



ApplePredicate encapsulates a strategy for selecting an apple.

`ApplePredicate` Implementations

René Witte

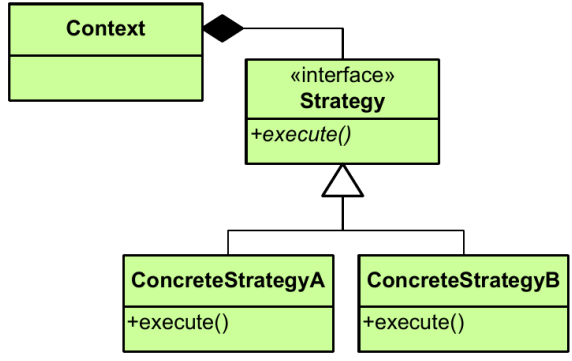

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
  public boolean test(Apple apple){
    return apple.getWeight() > 150;
  }
}

public class AppleGreenColorPredicate implements ApplePredicate {
  public boolean test(Apple apple){
    return "green".equals(apple.getColor());
  }
}
```

René Witte
Concordia University

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Filtering by Predicate

René Witte


```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p){
  List<Apple> result = new ArrayList<>();
  for(Apple apple: inventory){
    if(p.test(apple)){
      result.add(apple);
    }
  }
  return result;
}
```

Passing Code/Behavior

René Witte
Concordia University

Find red apples heavier than 150g?

```
public class AppleRedAndHeavyPredicate implements ApplePredicate{
  public boolean test(Apple apple){
    return "red".equals(apple.getColor()) && apple.getWeight() > 150;
  }
}
```

Calling

```
List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

Behavior Parameterization

`ApplePredicate` object

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
   public boolean test(Apple apple){
      return "red".equals(apple.getColor())
             && apple.getWeight() > 150;
   }
}
```

Pass as argument

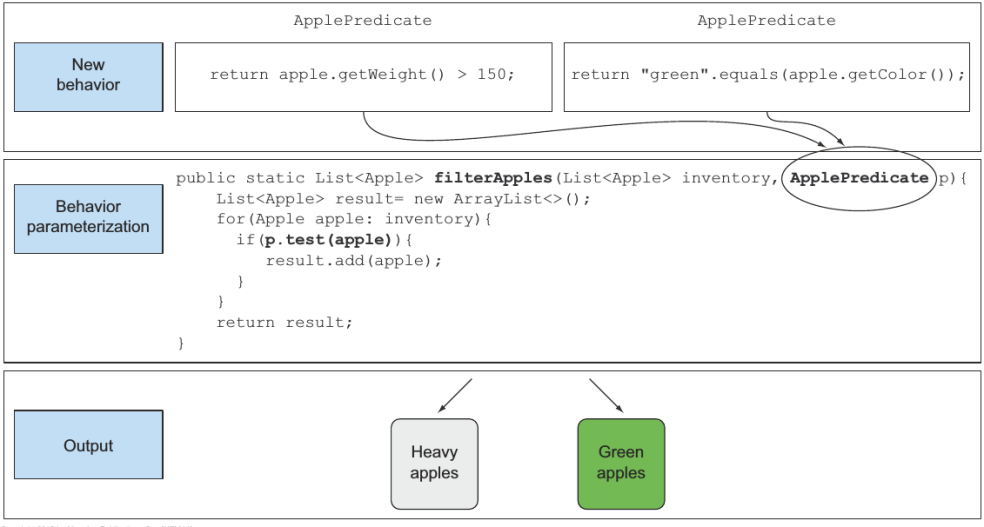
```
filterApples(inventory,          );
```

Pass a strategy to the filter method: filter the apples by using the boolean expression encapsulated within the ApplePredicate object. To encapsulate this piece of code, it is wrapped with a lot of boilerplate code (in bold).

René Witte

Multiple behaviors, one parameter

René Witte



Outline

Tackling verbosity

René Witte
Concordia University


```
public interface ApplePredicate{
  boolean test (Apple apple);
}

public class AppleHeavyWeightPredicate implements ApplePredicate{
  public boolean test(Apple apple){
    return apple.getWeight() > 150;
  }
}

public class AppleGreenColorPredicate implements ApplePredicate{
  public boolean test(Apple apple){
    return "green".equals(apple.getColor());
  }
}
```

Tackling verbosity (II)

```
public class FilteringApples{
  public static void main(String...args){
    List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                          new Apple(155, "green"),
                                          new Apple(120, "red"));
    List<Apple> heavyApples =
        filterApples(inventory, new AppleHeavyWeightPredicate());
    List<Apple> greenApples =
        filterApples(inventory, new AppleGreenColorPredicate());
  }

  public static List<Apple> filterApples(List<Apple> inventory,
                                         ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory){
      if (p.test(apple)){
        result.add(apple);
      }
    }
    return result;
  }
}
```

René Witte

Anonymous classes

René Witte

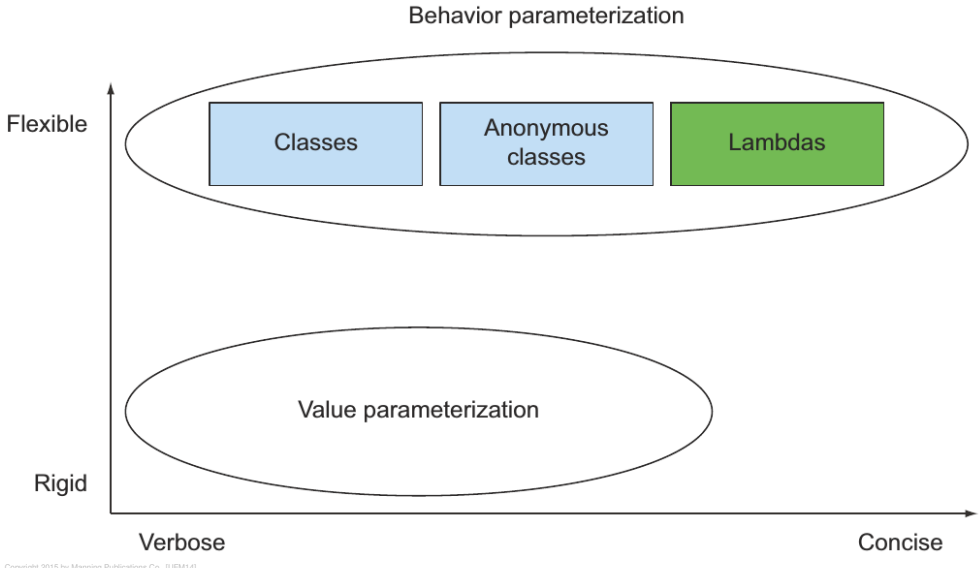

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
  public boolean test(Apple apple){
    return "red".equals(apple.getColor());
  }
});
```

Using a lambda expression

René Witte


```
List<Apple> result =
filterApples(inventory, (Apple apple)->"red".equals(apple.getColor()));
```

Behavior parameterization vs. value parameterization



René Witte

Abstracting over List type

Filter anything, not just `Apple`?

```
public interface Predicate<T>{
  boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p){
  List<T> result = new ArrayList<>();
  for(T e: list){
    if(p.test(e)){
      result.add(e);
    }
  }
  return result;
}
```

Calling

```
List<Apple> redApples =
  filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<Integer> evenNumbers = filter(numbers, (Integer i) -> i % 2 == 0);
```

René Witte

Sorting with a Comparator

`List.sort` in Java 8

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

Implementing a `Comparator` using an anonymous class

```
inventory.sort(new Comparator<Apple>() {
  public int compare(Apple a1, Apple a2){
    return a1.getWeight().compareTo(a2.getWeight());
  }
});
```

With a lambda expression

```
inventory.sort(
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

René Witte

Executing a block of code with Runnable

Threads

```
// java.lang.Runnable
public interface Runnable{
  public void run();
}
```

Creating a `Thread` using an anonymous class

```
Thread t = new Thread(new Runnable() {
 public void run(){
   System.out.println("Hello world");
 }
});
```

With a lambda expression

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

René Witte

GUI event handling

René Witte
Concordia University

JavaFX `EventHandler` using an anonymous class

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent event) {
    label.setText("Sent!!");
  }
});
```

With a lambda expression

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

Outline

Summary

René Witte
Concordia University

Behavior parameterization

- Ability for a method to take multiple different behaviors as parameters and use them internally to accomplish different behaviors
- Lets you make your code more adaptive to changing requirements and saves on engineering efforts in the future

Implementation

- Passing code is a way to give new behaviors as arguments to a method
- Very verbose prior to Java 8
- Anonymous classes were used to get rid of the verbosity associated with declaring multiple concrete classes for an interface that are needed only once
- The Java API contains many methods that can be parameterized with different behaviors, which include sorting, threads, and GUI handling
- Much more concise, readable, and flexible with Java 8 lambdas

Outline

Reading Material

René Witte

Required

- [UFM14, Chapter 2] (Passing Code with behavior parameterization)

Supplemental

- [GHJV95] (Strategy Design Pattern)

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.