Introduction to Object-Oriented Programming

Lambda Expressions

Christopher Simpkins

chris.simpkins@gatech.edu
Recall from SuperTroopers.java the MustacheComparator class:

```java
static class MustacheComparator implements Comparator<Trooper> {
    public int compare(Trooper a, Trooper b) {
        if (a.hasMustache() && !b.hasMustache()) {
            return 1;
        } else if (b.hasMustache() && !a.hasMustache()) {
            return -1;
        } else {
            return a.getName().compareTo(b.getName());
        }
    }
}
```

which we can use just like any other named class:

```java
Collections.sort(troopers, new MustacheComparator());
```
Anonymous Inner Classes

We can subclass `Comparator` and make an instance of the subclass at the same time using an *anonymous inner class*. Here’s a mustache comparator as an inner class:

```java
Collections.sort(troopers, new Comparator<Trooper>() {
    public int compare(Trooper a, Trooper b) {
        if (a.hasMustache() && !b.hasMustache()) {
            return 1;
        } else if (b.hasMustache() && !a.hasMustache()) {
            return -1;
        } else {
            return a.getName().compareTo(b.getName());
        }
    }
});
```

The general syntax for defining an anonymous inner class is

```
new SuperType < TypeArgument > () { class_body }
```
Functional Interfaces

Any interface with a single abstract method is a functional interface. For example, `Comparator` is a functional interface:

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

As in the previous examples, we only need to implement the single abstract method `compare` to make an instantiable class that implements `Comparator`.

Note that there’s an optional `@FunctionalInterface` annotation that is similar to the `@Override` annotation. Tagging an interface as a `@FunctionalInterface` prompts the compiler to check that the interface indeed contains a single abstract method and includes a statement in the interface’s Javadoc that the interface is a functional interface.
A lambda expression is a syntactic shortcut for defining the single abstract method of a functional interface and instantiating an anonymous class that implements the interface. The general syntax is

\[(T_1 \ p_1, ..., T_n \ p_n) \rightarrow \{method\_body\}\]

Where
- \(T_1, ..., T_n\) are types and
- \(p_1, ..., p_n\) are parameter names just like in method definitions.

If \(method\_body\) is a single expression, the curly braces can be omitted.
Here's our mustache comparator from `SuperTroopers.java` as a lambda expression:

```java
Collections.sort(troopers, (Trooper a, Trooper b) -> {
    if (a.hasMustache() && !b.hasMustache()) {
        return 1;
    } else if (b.hasMustache() && !a.hasMustache()) {
        return -1;
    } else {
        return a.getName().compareTo(b.getName());
    }
});
```

- **Because** `Collections.sort(List<T> l, Comparator<T> c)` **takes a** `Comparator<T>`, **we way that** `Comparator<T>` **is the target type** of the lambda expression passed to the `sort` method.
- The lambda expression creates an instance of an anonymous class that implements `Comparator<Trooper>` and passes this instance to `sort`
Given the `Bar` interface, the call:

```java
foo((Trooper a, Trooper b) -> {
    if (a.hasMustache() && !b.hasMustache()) {
        return 1;
    } else if (b.hasMustache() && !a.hasMustache()) {
        return -1;
    } else {
        return a.getName().compareTo(b.getName());
    }
});
```

creates an instance of the `Bar` interface using the same lambda expression.

- The type of object instantiated by a lambda expression is determined by the `target type` of the call in which the lambda expression appears.
Remember the rank comparator we defined for WordCount:

```java
public class WordCount {

    private Map<String, Integer> wordCounts;

    public Set<String> getWordsRanked() {
        Comparator<String> rankComparator = new Comparator<String>() {
            public int compare(String k1, String k2) {
                return wordCounts.get(k2) - wordCounts.get(k1);
            }
        };
        TreeSet<String> rankedWords = new TreeSet<>(rankComparator);
        rankedWords.addAll(wordCounts.keySet());
        return rankedWords;
    }
}
```
We can replace the anonymous inner class definition with a lambda expression:

```java
public Set<String> getWordsRanked() {
    Comparator<String> rankComparator =
        (String k1, String k2) -> wordCounts.get(k2) - wordCounts.get(k1);
    TreeSet<String> rankedWords = new TreeSet<>(rankComparator);
    rankedWords.addAll(wordCounts.keySet());
    return rankedWords;
}
```

Notice that since the body of the lambda expression is a single expression, we leave off the curly braces and return keyword.
Free and Bound Variables

```java
public class WordCount {
    private Map<String, Integer> wordCounts;

    public Set<String> getWordsRanked() {
        Comparator<String> rankComparator =
            (String k1, String k2) -> wordCounts.get(k2) - wordCounts.get(k1);
        TreeSet<String> rankedWords = new TreeSet<>(rankComparator);
        rankedWords.addAll(wordCounts.keySet());
        return rankedWords;
    }
}
```

In `rankComparator`:

- `k1` and `k2` are **bound variables**. They are defined in the parameter list or body of the lambda expression.
- `wordCounts` is a **free variable**. It is defined outside the lambda expression. Free variables must be **effectively final**.

We say that the lambda expression **captures** the `wordCount` variable. Such lambda expressions are called **closures**.
A lambda expression is a compact notation for specifying the implementation of the abstract method in a functional interface.

A method reference is a compact notation for a lambda expression that supplies the implementation of the abstract method in a functional interface from a compatible named method that has already been defined.

If a method already exists that fits the specification for a parameter that could take a lambda expression as an argument, you can use a method reference instead of a lambda expression.
Method References Example

Say we have a functional interface whose abstract method takes a single `Object` and returns `void`:

```java
public interface Foo {
    void bar(Object o);
}
```

and a method that takes an instance of an object implementing this functional interface as a parameter:

```java
void doo(Foo f) {
    f.bar("baz");
}
```

We can supply a method reference to any method that is *lambda equivalent* to the `bar` method above (same parameter list and return type):

```java
doo(System.out::println);
```

which is equivalent to:

```java
doo(x -> System.out.println(x));
```
Method References

Three kinds of method references:

- **object::instanceMethod** - like \( x \rightarrow \) 
  \( \text{object.instanceMethod}(x) \)

- **Class::staticMethod** - like \( x \rightarrow \) 
  \( \text{Class.staticMethod}(x) \)

  ```java
  someList.forEach(System.out::println);
  ```

- **Class::instanceMethod** - like \( (x, y) \rightarrow \) 
  \( x.instanceMethod(y) \)

  ```java
  Comparator<Trooper> byName = 
  Comparator.comparing(Trooper::getName);
  ```

Let’s see some examples …
Functional(ish) Composition

Remember how our mustache comparator ordered by mustache, then by name?
With lambdas we can make that even more concise and clear:

```
Comparator<Trooper> byMustacheThenName = 
    Comparator.comparing(Trooper::hasMustache)
    .thenComparing(Trooper::getName);
Collections.sort(troopers, byMustacheThenName);
```

Look at the Comparator API for details on these methods.