Lecture 20 — Anonymous Classes, Lambdas, and Javadoc

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 14 and 16, and Appendix B of the textbook
- Anonymous classes
- Functional interfaces and lambdas
- Javadoc

Anonymous Classes

- Whenever we want to create a new type of object we have so far created a new class first
- This can be annoying, sometimes we want to create a one-of-a-kind object
- *Anonymous classes* allow us to create a new custom object without declaring a class
- To see an example of when we would want to do this, let's first look at the `Comparator<T>` interface

The Comparator<T> Interface

- The `Arrays.sort(array, comparator)` method sorts any array
- It takes the array and an instance of the `Comparator<T>` interface
- The `Comparator<T>` interface contains a single method we need to override: `compare(a, b)`
- Returns a negative number if `a` is less than `b`
- Returns a positive number if `a` is greater than `b`
- Returns zero if `a` is equal to `b`
- `Arrays.sort(array, comparator)` uses the comparator object to sort the array
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html`
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html`

MyStringComparator Example

```
class MyStringComparator implements Comparator<String> {
    private static int countOccurences(String s, char c) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == c) count++;
        }
        return count;
    }

    @Override
    public int compare(String s1, String s2) {
        return countOccurences(s1, 'e') - countOccurences(s2, 'e');
    }
}
```

- Compares strings based on the number of ‘e’ characters

Sorting Example

```
public class Sorting {
    public static void main(String[] args) {
        String[] strings = {"elevated", "banana", "elephant", "early"};
        Arrays.sort(strings, new MyStringComparator());
        // Arrays.toString() is a useful method for formatting arrays as
            strings
        System.out.println(Arrays.toString(strings));
    }
}
```

- Uses the comparator to sort the array of strings
- Notice that we had to make a new class to implement the `Comparator<String>` interface?
- Let's see an *anonymous class*

AnonymousSorting Example

```
public class AnonymousSorting {
    private static int countOccurences(String s, char c) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == c) count++;
        }
        return count;
    }

    public static void main(String[] args) {
        String[] strings = {"elevated", "banana", "elephant", "early"};
        Arrays.sort(strings, new Comparator<String>() {
            public int compare(String s1, String s2) {
                return countOccurences(s1, 'e') - countOccurences(s2, 'e');
            }
        });
        // Arrays.toString() is a useful method for formatting arrays as strings
        System.out.println(Arrays.toString(strings));
    }
}
```

- This is more concise; we didn't need to define a new class!

Anonymous Class Syntax

```
new ExistingClassOrInterface(arg1, arg2) {
    // override methods here
    // you can add fields too
}
```

- This is the general form of an anonymous class
- They save us writing an entire class just to create a single object
- They're usually used to create a unique object that implements some interface
- They can also extend a class

Functional Interfaces

- Java 8 introduced the concept of *functional interfaces*
- A functional interface is an interface with exactly one method to override
- Examples: `Comparator`, `Runnable` (recall multithreading), etc.
- They can be used in conjunction with *lambda expressions*, which we will now see

Lambda Expressions

- Java 8 also introduced lambda expressions
- They're like anonymous classes, but for a single method
- Essentially an anonymous method (or function)
- They replace most usages of anonymous classes as they are often even more convenient
- The syntax is: `(parameter) -> expression`
- If there are multiple parameters, use parentheses: `(param1, param2) -> expression`

Lambda Expression Example

```
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

- This is a lambda that compares two strings by their length
- We are allowed to leave out the types of parameters, since they are already defined in the functional interface
- More concise than creating an anonymous class
- Now let's use them to implement our previous sorting example that sorts by the number of 'e' characters

LambdaSorting Example

```
public class LambdaSorting {
    private static int countOccurences(String s, char c) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == c) count++;
        }
        return count;
    }

    public static void main(String[] args) {
        String[] strings = {"elevated", "banana", "elephant", "early"};
        Arrays.sort(strings, (s1, s2) -> countOccurences(s1, 'e') -
            countOccurences(s2, 'e'));
        System.out.println(Arrays.toString(strings));
    }
}
```

LambdaSorting Example

```
public class LambdaRunnable {
    public static void main(String[] args) {
        Runnable r = () -> {
            System.out.println("Hello, world!");
            System.out.println("This is a lambda expression!");
        };
        r.run();
    }
}
```

- Lambda expressions can also accept no parameters `() -> expression`
- They can also execute multiple statements (as above)
- The syntax is `(params) -> { statements; }`
- If you use a statement block, you may need to use `return` if a value is expected

Introduction to Javadoc

- Javadoc is an automatic documentation generator
- It generates API documentation in HTML format from Java source code
- Javadoc comments are placed directly before class, field, and method declarations
- They begin with `/**` (instead of `/*`) and end with `*/`
- By convention, every line between starts with a `*`

Basic Javadoc Comment Structure

```
/**
 * This is a Javadoc comment.
 * It describes the class, method, or field that it precedes.
 */
```

- The first sentence of each Javadoc comment should be a summary sentence, containing a concise but complete description
- This sentence ends with a period

Javadoc Tags

```java
/**
 * This method does something.
 *
 * @param param Description of the parameter
 * @return Description of the return value
 * @throws ExceptionType When the exception would be thrown
 */
public ReturnType methodName(Type param) throws ExceptionType {
    // method body
}
```

- Javadoc comments can contain tags
- These tags provide more specific information about code elements
- There are many kinds of tags. A few examples can be found above

Example Class

```java
/**
 * This is an example of a Javadoc comment on a class.
 */
public class Javadoc {
    /**
     * This method returns true if the given integer is even, and false otherwise.
     * @param x the integer to check
     * @return true if x is even, false otherwise
     */
    public boolean isEven(int x) {
        return x % 2 == 0;
    }
}
```

- Here is an example class
- Let's generate the documentation

Generating Javadoc HTML

- Run the command: `javadoc Javadoc.java -d mydocs`
- You can give it some .java files, or an entire package folder
- Usually, IDEs will be able to generate this automatically
- The `mydocs` folder now contains the HTML documentation for the class
- Notice how this looks like the Java API?
- The Java API documentation is created using Javadoc!
- `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

End-lecture Break