Anonymous Objects, Anonymous Classes, Lambda Expressions

Anonymous Objects

- An object that isn't bound to a variable or has a particular name.
- Useful if we only care about the object and not necessarily referring to it specifically after constructing it.

Example

```
public class SomeContainer {

	public static final int SIZE = 100;

	private String[] stringArray;
	private int count;


	public SomeContainer() {
		stringArray = new String[SIZE];
		count = 0;
	}

	public void add(String s) throws ArrayIndexOutOfBoundsException {
		if (count < SIZE) {
			stringArray[count] = s;
			count++;
		} else {
			throw new ArrayIndexOutOfBoundsException();
			// also an anonymous object
		}
	}

	public int getCount() {
		return count;
	}

	public String[] getStringArray() {
		return stringArray;
	}
}

// SomeContainerTester
import static org.junit.Assert.*;
import org.junit.Test;

public class SomeContainerTester {
	@Test (expected=ArrayIndexOutOfBoundsException.class)
	public void testAddMax() {
		SomeContainer container = new SomeContainer();

		//container.add("1");
		//container.add("2");
		// ... ☹
```

```
            // populate stringArray with anonymous String objects
            for (int i = 0; i < SomeContainer.SIZE; i++) {
                  container.add(new String(Integer.toString(i)));
                  System.out.println(i);
            }

            assertEquals(container.getCount(), 100);
            container.add("someString"); // should throw exception
      }
}
```

Anonymous Classes

- We can create classes "on-the-fly" the same way.
- A class definition can be assigned "in-line" when declaring the reference.
- Can be used to define classes extending abstract classes or implementing Interfaces.
- Good for code organization
    - If we only want to use the class once and define specific behavior
        - We could implement a class file…
        - Or just define the behavior and move on.
            - If a specific behavior is used more than once, then providing its own implementation is better for readability / redundancy reasons.

Example

```
// Known as a functional interface: only has a single method.
@FunctionalInterface // optional annotation
public interface Animal {
      public String getSound();
}
////////////
public static void main(String[] args) {

      //Animal a = new Animal(); // ERROR!
      // Anonymous class representing a dog
      Animal dog = new Animal() {
            public String getSound() {
                  return "BARK!";
            }
      };
      // Anonymous class representing a cat
      Animal cat = new Animal() {
            public String getSound() {
                  return "MEOW!";
            }
      };
      // Anonymous class representing a cow
      Animal cow = new Animal() {
            public String getSound() {
                  return "MOO!";
            }
      };
```

```
	ArrayList<Animal> animalList = new ArrayList<Animal>();
	animalList.add(dog);
	animalList.add(cat);
	animalList.add(cow);

	for (Animal a : animalList) {
		System.out.println(a.getSound());
	}
}
```

- Can even use anonymous objects and classes together:

```
animalList.add(new Animal() {
	public String getSound() {
		return "BARK!";
	}
});
```

- This is a feature to simplify code, but use it wisely
  - Code scattered with Anonymous Objects / Classes can be hard to follow.

Lambda Expressions

- If you decide to go for the anonymous class route, consider using lambda expressions for Functional Interfaces.
- A shorter replacement for anonymous classes.
- Specifically used for Functional Interfaces (Interfaces with only one abstract method).
- There are already a lot of functional interfaces in Java
  - Comparator - .compare()
  - Callable - .call() returns a result and may throw an exception
  - Runnable - .run() like callable, but doesn't return a result
  - ActionListener - .actionPerformed(ActionEvent e) invoked when action occurs. Used for GUI components like clicking a button
  - Any single abstract method custom Interface…

```
// represented with Lambda Expressions
// Compiler knows this corresponds to the unimplemented method
// getSound() for the functional interface
Animal dog = () -> { return "BARK!"; };
Animal cat = () -> { return "MEOW!"; };
Animal cow = () -> { return "MOO!"; };
```

Lambda Expressions with Parameter Example

```
// change getSound() to getSound(int weight)

Animal dog = (int weight) -> {
	if (weight > 50) {
		return "BARK!!!!!";
	} else {
		return "BARK!";
	}
};
```

```
Animal cat = (int weight) -> {
      if (weight > 50) {
            return "MEOW!!!!!";
      } else {
            return "MEOW!";
      }
};

Animal cow = (int weight) -> {
      if (weight > 50) {
            return "MOO!!!!!";
      } else {
            return "MOO!";
      }
};

ArrayList<Animal> animalList = new ArrayList<Animal>();
animalList.add(dog);
animalList.add(cat);
animalList.add(cow);

for (Animal a : animalList) {
      System.out.println(a.getSound(60)); // change to 40 and see…
}
```

Implementing Comparator Interface for Sorting

- Collections (java.util.Collections) has a method .sort() method to automatically sort Lists.
- .sort takes a List and a Comparator object with compare(Object o1, Object o2) defined.
    - We can define this anonymously!

Example (using Anonymous Classes)

```
ArrayList<String> a = new ArrayList<String>();
a.add("RICHERT");
a.add("MIKE");
a.add("TANYA");
a.add("ZORRA");

// old way
Collections.sort(a, new Comparator<Object>() {
      public int compare(Object o1, Object o2) {
            String x = (String) o1;
            String y = (String) o2;
            return x.compareTo(y);
      }
});

for (String s : a) {
      System.out.println(s);
}
```

Example (using Lambda Expressions)

```
Collections.sort(a, (String s1, String s2) -> {
      return s1.compareTo(s2);
});
```