Static Variables & Methods

Brandon Krakowsky

Penn Engineering

A Look Ahead – *Optional* Final Project

A Look Ahead – *Optional* Final Project

For the final assignment, you'll have the option to work as a group (no more than 2 students) and **define your own project**!

- This will be an alternative to the last assigned HW (details to be provided)
- You'll need to first outline your project and submit a proposal
  - If approved, you'll be permitted to work on your project instead of the last assigned HW
  - If not approved, you'll be required to complete the last assigned HW

A Look Ahead – *Optional* Final Project

For the final assignment, you'll have the option to work as a group (no more than 2 students) and **define your own project**!

- This will be an alternative to the last assigned HW (details to be provided)
- You'll need to first outline your project and submit a proposal
  - If approved, you'll be permitted to work on your project instead of the last assigned HW
  - If not approved, you'll be required to complete the last assigned HW

The main requirement for this project is the use of **Advanced Java**

- This *optional* project is intended for those students who:
  - Are chomping at the bit to explore more **Advanced Java** (maybe something we have not, or will not, cover in this class)
  - Want to work on something specific that they can add to their resume/portfolio
  - Are working on a research project and see a place where programming can be of use

A Look Ahead – *Optional* Final Project

For the final assignment, you'll have the option to work as a group (no more than 2 students) and **define your own project**!

- This will be an alternative to the last assigned HW (details to be provided)
- You'll need to first outline your project and submit a proposal
  - If approved, you'll be permitted to work on your project instead of the last assigned HW
  - If not approved, you'll be required to complete the last assigned HW

The main requirement for this project is the use of **Advanced Java**

- This *optional* project is intended for those students who:
  - Are chomping at the bit to explore more **Advanced Java** (maybe something we have not, or will not, cover in this class)
  - Want to work on something specific that they can add to their resume/portfolio
  - Are working on a research project and see a place where programming can be of use

More details to be provided soon!

Static Variables

Instance Variables vs. Static Variables

- Java classes can have *instance variables* and *static variables*

Instance Variables vs. Static Variables

- Java classes can have *instance variables* and *static variables*
- *instance variables* can be different for every instance of a class
  - You define them as variables inside the class

```
public class Employee {
    //instance variable: different for every instance of Employee
    String name;

    public Employee(String name) {
        this.name = name;
    }
}
```

Instance Variables vs. Static Variables

- You reference an *instance variable* using the instance of a class

Instance Variables vs. Static Variables

- You reference an *instance variable* using the instance of a class
- If you create multiple instances of Employee, every employee can have a different value for name

```
Employee employee1 = new Employee(“Brad”);
Employee employee2 = new Employee(“Sue”);
System.out.println(employee1.name); //prints “Brad”
System.out.println(employee2.name); //prints “Sue”
```

  - You need an instance of Employee to access name

Static Variables

- *static variables* are the same for every instance of the class
  - For reference, these are equivalent to class variables in a Python class
  - You define them as variables inside the class, using the keyword *static*
  - You typically use all uppercase characters when defining static variables, separating syllables with underscores
  - They often refer to properties that are common to all instances of the class

```
public class Employee {
    //static variable: same for all instances of Employee
    static String DEPARTMENT = ”Accounting”;

    //instance variable: different for every instance of Employee
    String name;

    public Employee(String name) {
        this.name = name;
    }
}
```

Static Variables

- You typically reference a *static variable* with a class name, not an instance of a class

Static Variables

- You typically reference a *static variable* with a class name, not an instance of a class
- Even if you create multiple instances of Employee, every employee will have the same value for DEPARTMENT

```
Employee employee1 = new Employee("Brad");
Employee employee2 = new Employee("Sue");
System.out.println(Employee.DEPARTMENT); //prints "Accounting"
```

  - You actually don't need an instance of Employee to access DEPARTMENT

Static Variables

- You typically reference a *static variable* with a class name, not an instance of a class
- Even if you create multiple instances of Employee, every employee will have the same value for DEPARTMENT

```
Employee employee1 = new Employee("Brad");
Employee employee2 = new Employee("Sue");
System.out.println(Employee.DEPARTMENT); //prints "Accounting"
```

  - You actually don't need an instance of Employee to access DEPARTMENT

- Even IF you use instances of Employee to access DEPARTMENT, they will all have the same value!

```
System.out.println(employee1.DEPARTMENT); //also prints "Accounting"
System.out.println(employee2.DEPARTMENT); //also prints "Accounting"
```

Static Variables for Hard-Coded Values

- *static* variables are extremely useful for “hard-coded values”
  - These are values that are the same for all instances of a class
  - For example, if a class utilizes a standard sales tax rate (SALES_TAX)
  - It will be the same for every instance of that class, so you can declare it as static

```
public class BankAccount {
    //static variable: same for all instances of BankAccount
    static double SALES_TAX = .06;

    //instance variable: different for every instance of BankAccount
    double balance;

    public void purchase(double amount) {
        //reference the static variable using the full class name
        this.balance -= ((BankAccount.SALES_TAX * amount) + amount);
    }
}
```

Static Variables for Hard-Coded Values

- *static* variables are extremely useful for "hard-coded values"
  - These are values that are the same for all instances of a class
  - For example, if a class utilizes a standard sales tax rate (SALES_TAX)
  - It will be the same for every instance of that class, so you can declare it as static

```
public class BankAccount {
    //static variable: same for all instances of BankAccount
    static final double SALES_TAX = .06;

    //instance variable: different for every instance of BankAccount
    double balance;

    public void purchase(double amount) {
        //reference the static variable using the full class name
        this.balance -= ((BankAccount.SALES_TAX * amount) + amount);
    }
}
```

- If a *static* variable is never going to change, you can add the *final* keyword after *static*

Static Methods

Static Methods

- Java classes can also have *static* methods
  - Just like *static* variables, you do not need to create an instance of a class to call a *static* method

Static Methods

- Java classes can also have *static* methods
  - Just like *static* variables, you do not need to create an instance of a class to call a *static* method
- For example, the Math class has a static method *sqrt*

```
int retVal = Math.sqrt(9);
```

  - You do not create an instance of the Math class to call *sqrt*
  - Instead, you use the class name to call the method

Static Methods

- Java classes can also have *static* methods
  - Just like *static* variables, you do not need to create an instance of a class to call a *static* method
- For example, the Math class has a static method *sqrt*

```
int retVal = Math.sqrt(9);
```

  - You do not create an instance of the Math class to call *sqrt*
  - Instead, you use the class name to call the method
- Often times, Java “helper” methods are static
  - Helper methods are utility methods that assist a program in doing some basic error checking or processing of a given input

Static Helper Methods

- Here we have a class HelperClass with various “helpful” static methods for checking the validity of a number

```
class HelperClass {
    //Returns true if x is valid
    public static boolean isValid(int x) {
        return HelperClass.isGreaterThanZero(x) && HelperClass.isEven(x);
    }
    //Returns true if x is greater than 0
    public static boolean isGreaterThanZero(int x) {
        return (x > 0);
    }
    //Returns true if x is even
    public static boolean isEven(int x) {
        return (x % 2 == 0);
    }
}
```

Static Helper Methods

- Here we have a class HelperClass with various “helpful” static methods for checking the validity of a number

```
class HelperClass {
    //Returns true if x is valid
    public static boolean isValid(int x) {
        return HelperClass.isGreaterThanZero(x) && HelperClass.isEven(x);
    }
    //Returns true if x is greater than 0
    public static boolean isGreaterThanZero(int x) {
        return (x > 0);
    }
    //Returns true if x is even
    public static boolean isEven(int x) {
        return (x % 2 == 0);
    }
}
```

- You do not create an instance of HelperClass to call its static methods

```
boolean numIsValid = HelperClass.isValid(0); //call static method with class name
```

About Java's *main* Method

public static void main

- The very first method we saw in Java was the static *main* method
- public static void main is the first method Java looks for when running a program

public static void main

- The very first method we saw in Java was the static *main* method
- public static void main is the first method Java looks for when running a program
- How does Java run the following program?

HelloWorld.java

```
1
2  public class HelloWorld {
3
4      public static void main(String[] args) {
5          System.out.println("Hello World!");
6      }
7
8  }
9
```

- It looks for a static *main* method in HelloWorld and runs it **without creating an instance HelloWorld**

public static void main (String[ ] args)

- Incidentally, the main method accepts a single argument of type String array
  - This is also called the *command line arguments*
  - It's an array of String values passed to your *main* method, when running your program

public static void main (String[ ] args)

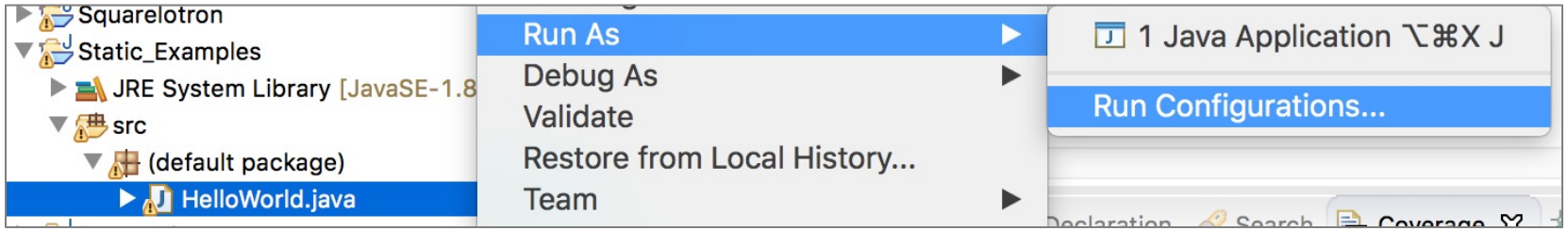
- Incidentally, the main method accepts a single argument of type String array
  - This is also called the *command line arguments*
  - It's an array of String values passed to your *main* method, when running your program
- Here's an example of how to access (and print) the command line arguments

HelloWorld.java

```
1
2  public class HelloWorld {
3
4      public static void main(String[] args) {
5          for (String arg : args) {
6              System.out.println(arg);
7          }
8      }
9  }
10
```

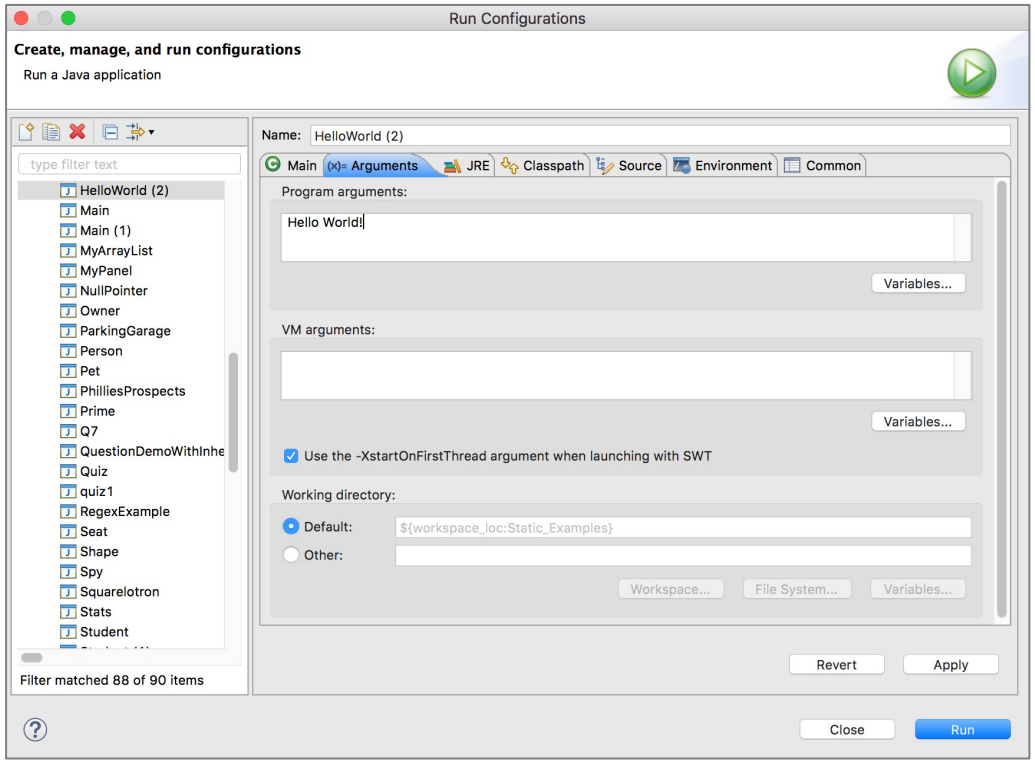
public static void main (String[ ] args)

- Here's how to pass command line arguments when executing a program in Eclipse
  - Select your program (Java file) in the Package Explorer
  - Go to "Run As" → "Run Configurations …"



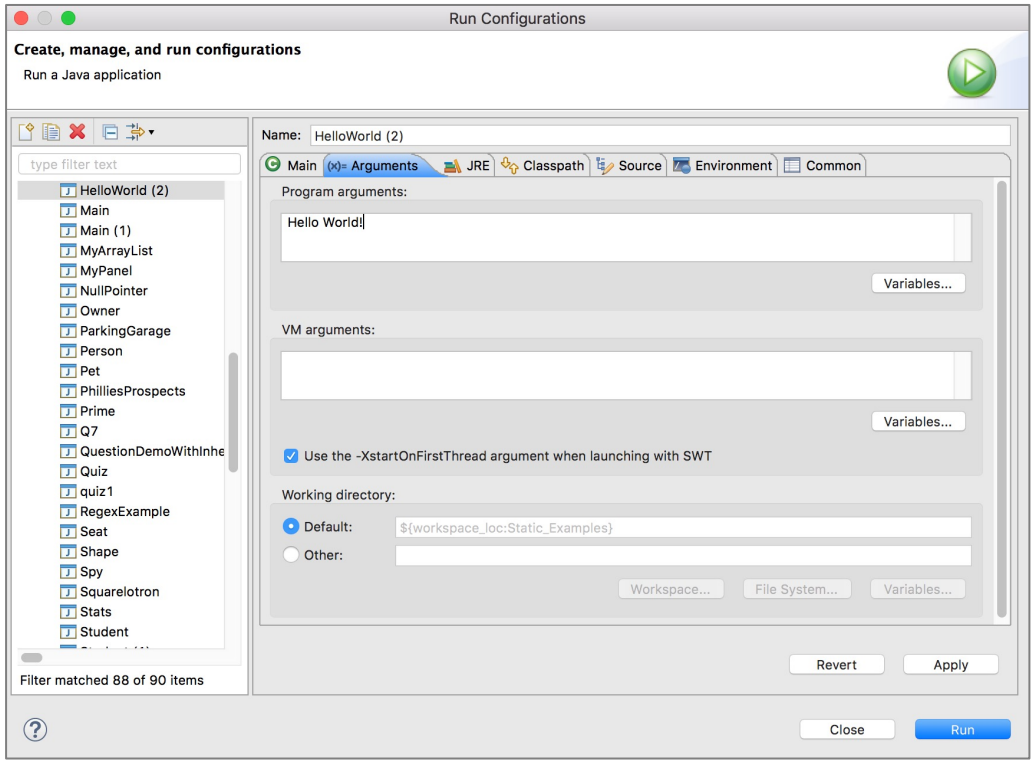
public static void main (String[ ] args)

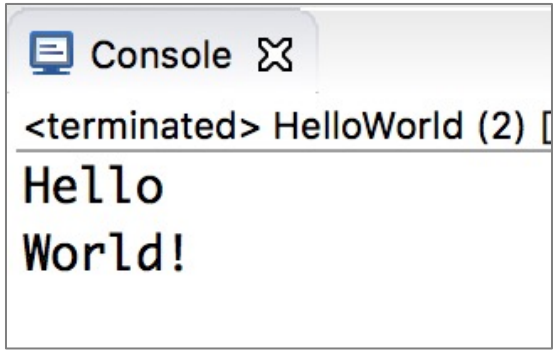
- Go to the “Arguments” tab and specify values, separated by spaces



public static void main (String[ ] args)

- Go to the “Arguments” tab and specify values, separated by spaces





- Click “Run” and your main method will have access to (and print) the String args

More About Static Methods

When Should You Use a Static Method?

- When there is no need for the method to belong to an instance of the object
  - It could be as simple as, the method doesn't need to access, manipulate, or store any data in an *instance* variable

When Should You Use a Static Method?

- When there is no need for the method to belong to an instance of the object
  - It could be as simple as, the method doesn't need to access, manipulate, or store any data in an *instance* variable
- As another example, in a class Fraction, you might have a static method *gcd*

```
public class Fraction {
    //instance variables
    int numerator;
    int denominator;

    //static method returning greatest common divisor
    public static int gcd(int a, int b) {
        //returns gcd of a and b
        //has nothing to do with the numerator or denominator in Fraction
    }
}
```

When Should You Use a Static Method?

- When there is no need for the method to belong to an instance of the object
  - It could be as simple as, the method doesn't need to access, manipulate, or store any data in an *instance* variable
- As another example, in a class Fraction, you might have a static method *gcd*

```
public class Fraction {
    //instance variables
    int numerator;
    int denominator;

    //static method returning greatest common divisor
    public static int gcd(int a, int b) {
        //returns gcd of a and b
        //has nothing to do with the numerator or denominator in Fraction
    }
}
```

- Do you need an instance of Fraction in order to compute the gcd of 2 numbers? No, so it can be static

More Examples of Static Variables

We know that *static* variables can be used for constant values

- These are values that are the same for all instances of a class

```
public class Circle {
    //static variable
    static final double PI = 3.1415;

    public double calculateArea(double radius) {
        //reference the static variable using the full class name
        return (Circle.PI * (radius * radius));
    }
}
```

More Examples of Static Variables

We know that *static* variables can be used for constant values

- These are values that are the same for all instances of a class

```
public class Circle {
    //static variable
    static final double PI = 3.1415;

    public double calculateArea(double radius) {
        //reference the static variable using the full class name
        return (Circle.PI * (radius * radius));
    }
}
```

- Here, the instance method *calculateArea* is accessing the static variable *PI*
  - This is legal!
  - Rule: Instance *methods* can access static *variables*

Another Use Case for Static

- Another common usage is to use static variables to share data across instances of an object, e.g. to keep track of the object instances created

```
public class Car {
    //list of all created cars
    static ArrayList<Car> CAR_LIST = new ArrayList<Car>();

    public Car() {
        Car.CAR_LIST.add(this); //create car and add to list
    }

    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car();
        for (Car car : Car.CAR_LIST) {
            System.out.println(car);
        }
    }
}
```

Other Rules For Static

- A static *method* can access only static *variables*
  - It cannot access instance variables

Other Rules For Static

- A static *method* can access only static *variables*
  - It cannot access instance variables
- A static *method* can call another static *method*

Other Rules For Static

- A static *method* can access only static *variables*
  - It cannot access instance variables
- A static *method* can call another static *method*
- An instance *method* can call a static *method* or access a static *variable*

Other Rules For Static

- A static *method* can access only static *variables*
  - It cannot access instance variables
- A static *method* can call another static *method*
- An instance *method* can call a static *method* or access a static *variable*
- The keyword “this” does not make any sense inside a static *method*

Customer Tracking Project

Customer Class

Customer.java

```
1  import java.util.ArrayList;
2
3  /**
4   * Represents a customer with name, ID, and geography.
5   * @author lbrandon
6   *
7   */
8  public class Customer {
9
10     //static variables
11     //shared across all instances of Customer
12
13     /**
14      * The company for all customers.
15      */
16     static final String COMPANY = "CVS";
17
18     /**
19      * List of all customers.
20      */
21     static ArrayList<Customer> CUSTOMERS = new ArrayList<Customer>();
22
23     /**
24      * To generate and keep track of customer IDs.
25      */
26     static Counter COUNTER;
27
```

Customer Class

```
27
28      //instance variables
29
30      /**
31       * Name for customer.
32       */
33      String name;
34
35      /**
36       * Geography for customer.
37       */
38      String geography;
39
40      /**
41       * ID for customer.
42       */
43      int ID;
44
```

Counter Class

Customer.java | Counter.java

```java
/**
 * Represents a counter with methods for manipulating a count.
 * @author lbrandon
 *
 */
public class Counter {

    /**
     * Internal count for counter.
     */
    int count;

    /**
     * Creates a Counter with initial count.
     * @param initialCount for counter
     */
    public Counter(int initialCount) {
        this.setCount(initialCount);
    }
```

Counter Class

```
21
22      /**
23       * Increments internal count.
24       */
25      public void increment() {
26          this.count++;
27      }
28
29      /**
30       * Returns current count.
31       * @return current count
32       */
33      public int getCount() {
34          return this.count;
35      }
36
37      /**
38       * Sets count starting at given count.
39       * @param count to start counter
40       */
41      public void setCount(int count) {
42          this.count = count;
43      }
44
```

Customer Class

```
44
45      //constructor
46
47      /**
48       * Creates a customer with given name and geography.
49       * Adds customer to list and increments the counter.
50       * @param name for customer
51       * @param geography for customer
52       */
53      public Customer(String name, String geography) {
54          this.name = name;
55          this.geography = geography;
56
57          //get ID from counter
58          this.ID = Customer.COUNTER.getCount();
59
60          //increment counter
61          Customer.COUNTER.increment();
62
63          //add customer to list
64          Customer.CUSTOMERS.add(this);
65      }
66
```

Customer Class

```
69      /**
70       * Prints all customers for company.
71       */
72      public static void printAllCustomers() {
73          System.out.println("All customers: ");
74
75          for (Customer c : Customer.CUSTOMERS) {
76              System.out.println(" " + c);
77          }
78
79          System.out.println("\n");
80      }
81
```

Customer Class

```
95      /**
96       * To represent a customer as a string.
97       * Returns ID, name of customer, and geography.
98       */
99      public String toString() {
100         return this.ID + ": " + this.name
101                 + ", Company: " + Customer.COMPANY
102                 + ", Location: " + this.geography;
103     }
104
```

Customer Class

```
76
77      /**
78       * To compare two customers for equality.
79       * If they have the same name and geography, they are considered equal.
80       */
81      public boolean equals(Object obj) {
82
83          //cast object to customer
84          Customer otherCustomer = (Customer) obj;
85
86          //compare name and geography for customers
87          if ((this.name.equals(otherCustomer.name)
88                  && (this.geography.equals(otherCustomer.geography)))) {
89              return true;
90          }
91
92          return false;
93      }
94
```

CustomerTest Class

Customer.java | Counter.java | CustomerTest.java

```
1  import static org.junit.jupiter.api.Assertions.*;
5
6  class CustomerTest {
7
8      //create customers for testing
9      Customer c1;
10     Customer c2;
11     Customer c3;
12
13     @BeforeEach
14     void setUp() throws Exception {
15
16         //initialize a counter for generating unique IDs for customers
17         Customer.COUNTER = new Counter(1);
18
19         //initialize customers for testing
20         this.c1 = new Customer("Brandon", "Los Angeles");
21         this.c2 = new Customer("Brandon", "NYC");
22         this.c3 = new Customer("Brandon", "NYC");
23     }
24
```

CustomerTest Class

```
25      @Test
26      void testEqualsObject() {
27
28          //they should not be considered equal
29          //(because they have different geographies)
30          assertNotEquals(c1, c2);
31
32          //their names are equal
33          assertEquals(c1.name, c2.name);
34          //their geographies are different
35          assertNotEquals(c1.geography, c2.geography);
36
```

CustomerTest Class

```
37          //they should be considered equal
38          //(because they have same names and geographies)
39          assertEquals(c2, c3);
40
41          //their names are equal
42          assertEquals(c2.name, c3.name);
43          //their geographies are equal
44          assertEquals(c2.geography, c3.geography);
45
46      }
47
48  }
49
```

Customer Class

```
81
82      /**
83       * Removes the given customer from list of customers.
84       * @param customer to remove
85       */
86      public static void removeCustomer(Customer customer) {
87
88          //find customer
89          int removeIndex = Customer.findCustomer(customer);
90
91          //if index is valid, remove customer
92          if (removeIndex >= 0) {
93              Customer.CUSTOMERS.remove(removeIndex);
94          }
95      }
96
97      /**
98       * Locates given customer in list of customers.
99       * @param customer to find
100      * @return index of customer if located, otherwise -1
101      */
102     public static int findCustomer(Customer customer) {
103
104         //set default index
105         int index = -1;
106
107         //iterate over customers list and find
108         for (int i = 0; i < Customer.CUSTOMERS.size(); i++) {
109             if (Customer.CUSTOMERS.get(i).equals(customer)) { //calls equals method in customer class
110                 index = i;
111                 break;
112             }
113         }
114
115         return index;
116     }
117
```

Customer Class

```
145
146     public static void main(String[] args) {
147
148         //set initial count to 1
149         int initialCount = 1;
150
151         //check for any String args to the main method
152         if (args.length > 0) {
153             //assumes the first String arg can be casted to an int
154             //parse the first String arg and cast to int
155             initialCount = Integer.parseInt(args[0]);
156         }
157
158         //create counter for customers
159         Customer.COUNTER = new Counter(initialCount);
160
```

Customer Class

```
160
161         //create customer
162         Customer c1 = new Customer("chenyun", "Los Angeles");
163
164         //print customers
165         Customer.printAllCustomers();
166
167         //create another customer
168         Customer c2 = new Customer("huize", "NYC");
169
170         //print customers again
171         Customer.printAllCustomers();
172
173         //create another customer
174         Customer c3 = new Customer("jeffrey", "Australia");
175
176         //print customers again
177         Customer.printAllCustomers();
178
179         //remove customer
180         Customer.removeCustomer(c1);
181
182         //print customers again
183         Customer.printAllCustomers();
184
185     }
```