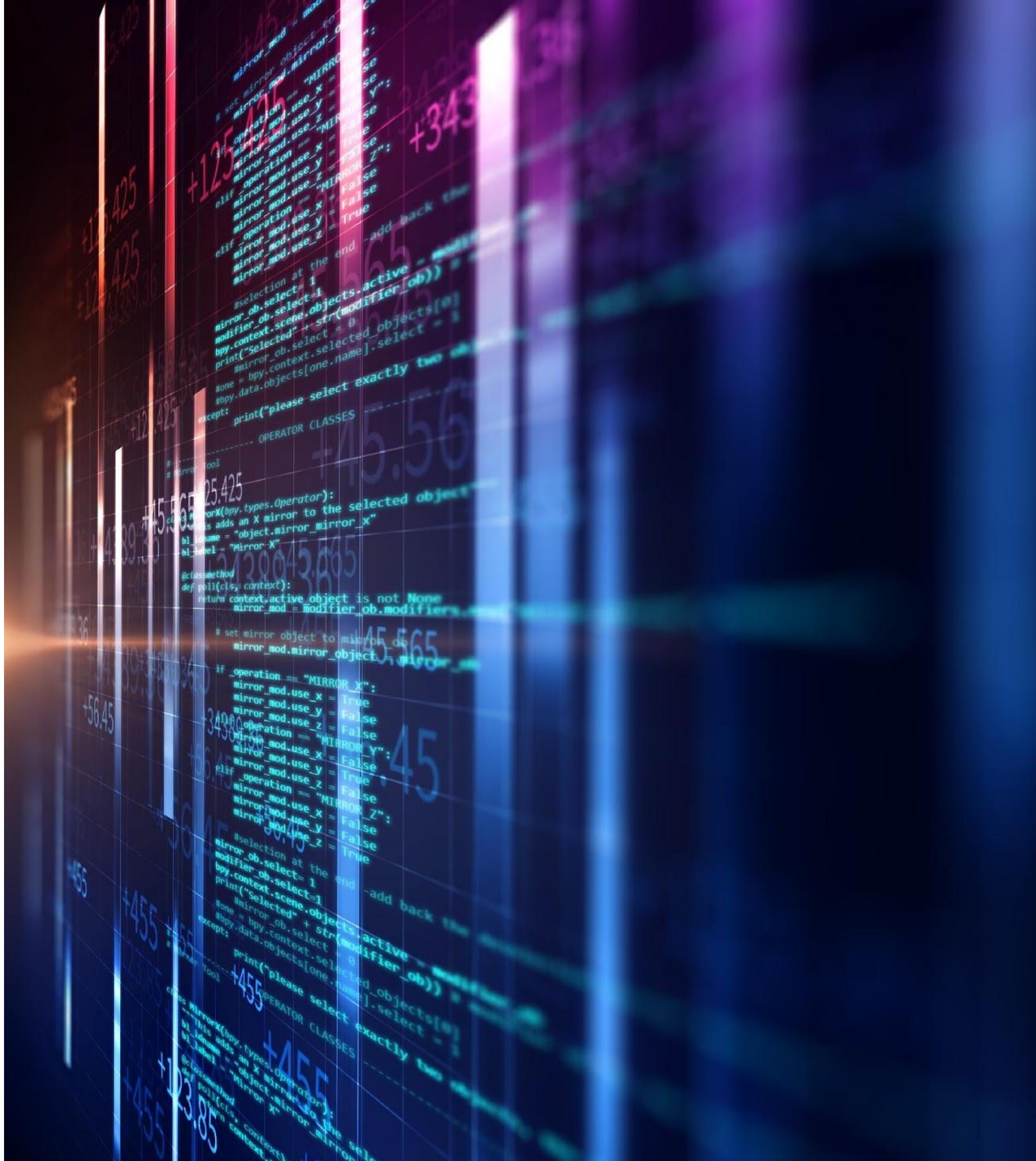
Static Variables & Methods

Brandon Krakowsky



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
- 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
  - 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

- Reference a *static variable* with a class name, not an instance of a class
- Even if you create multiple instances of Employee, every customer 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 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
- 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);
    }
}
```

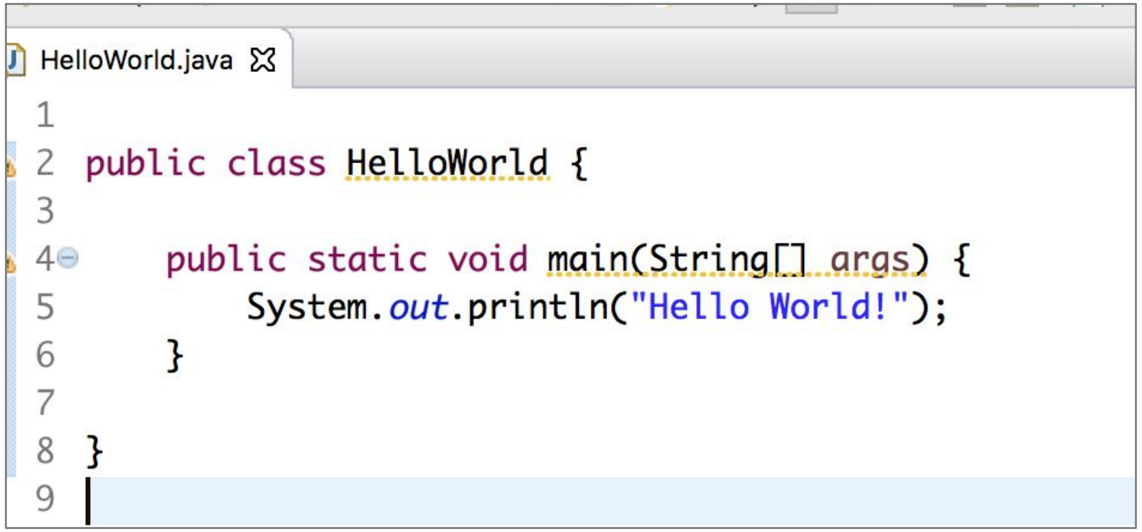
- You do not create an instance of HelperClass to call its static methods

```
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
- How does Java run the following program?

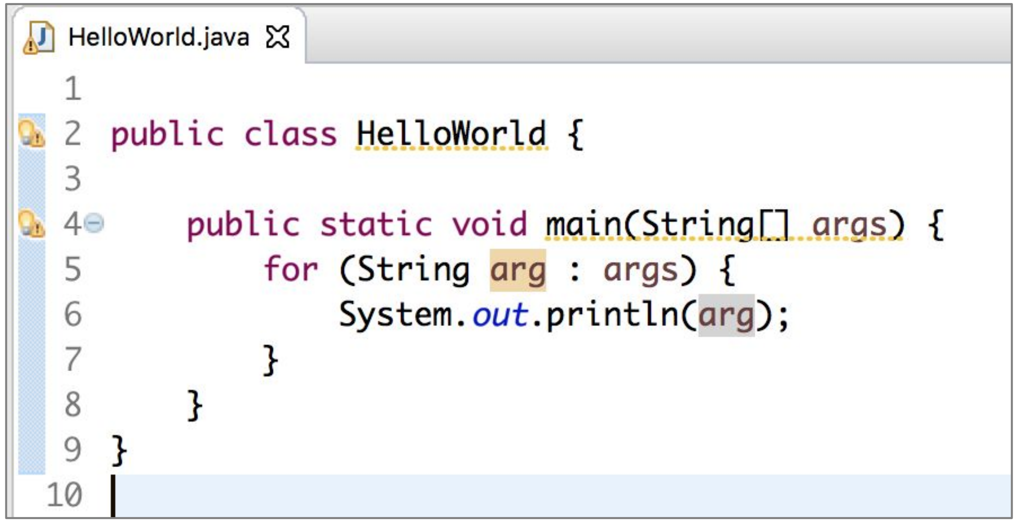


```

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }

}

```

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

public static void main (String[ ] args)

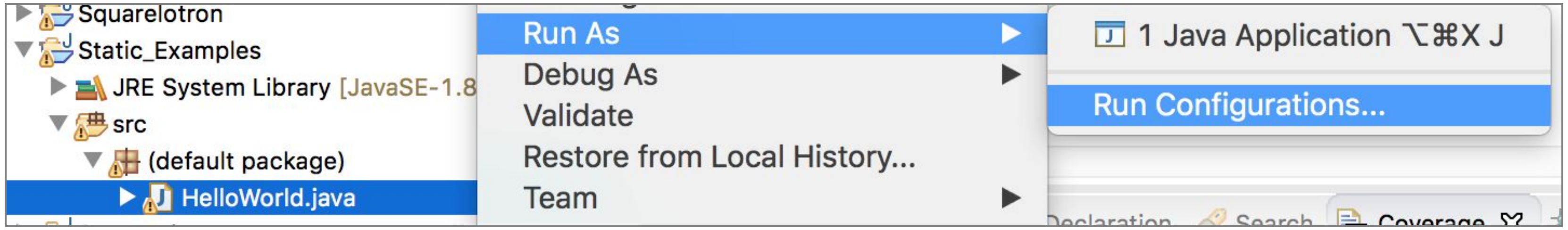
- Incidentally, the main method accepts a single argument of type String array
  - This is also called the *command line arguments*
  - It’s an array of String values passed to your *main* method, when running your program
- Here’s an example of how to access (and print) the command line arguments



```java

public class HelloWorld {

    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}

```

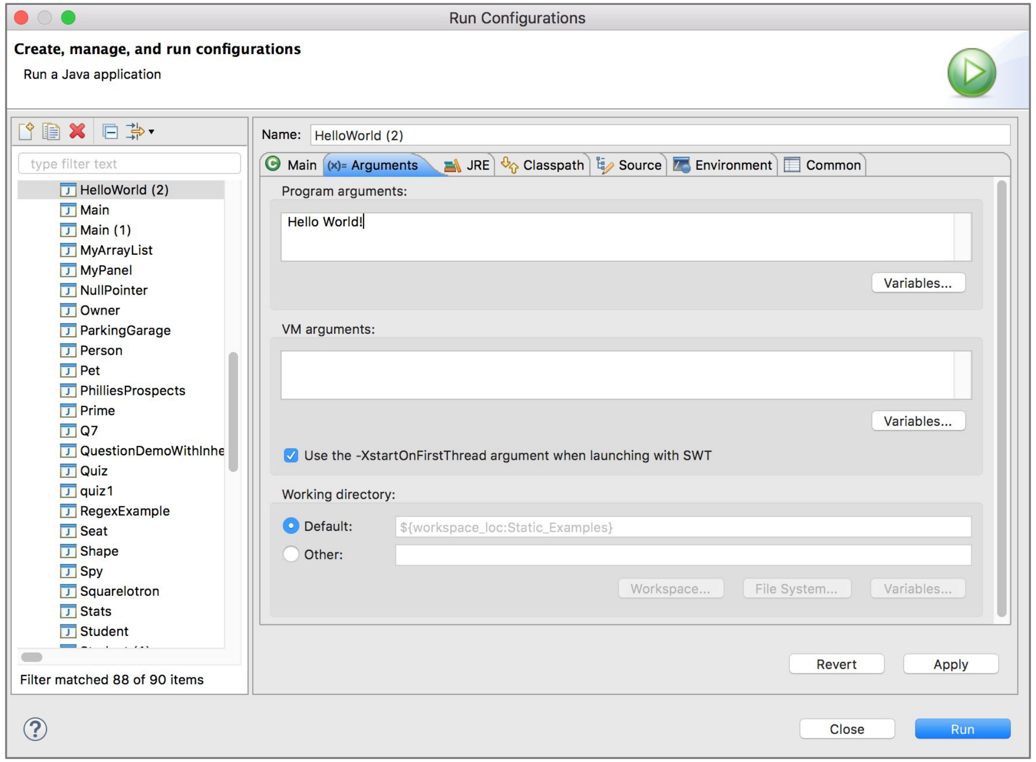
public static void main (String[ ] args)

- 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



Run Configurations

Create, manage, and run configurations

Run a Java application

Name: HelloWorld (2)

Program arguments:

```
Hello World!
```

VM arguments:

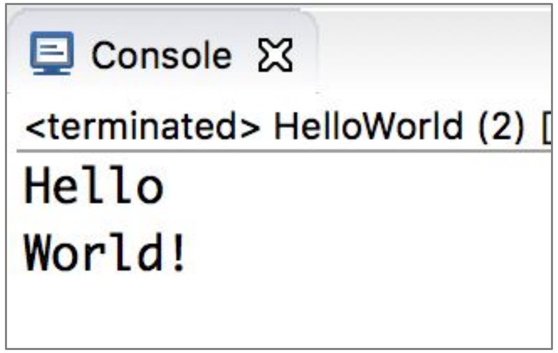
☑ Use the -XstartOnFirstThread argument when launching with SWT

Working directory:

● Default: ${workspace_loc:Static_Examples}

○ Other:

Filter matched 88 of 90 items



Console

```
<terminated> HelloWorld (2) [
Hello
World!
```

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

Penn Engineering

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
- 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));
    }
}
```

- 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

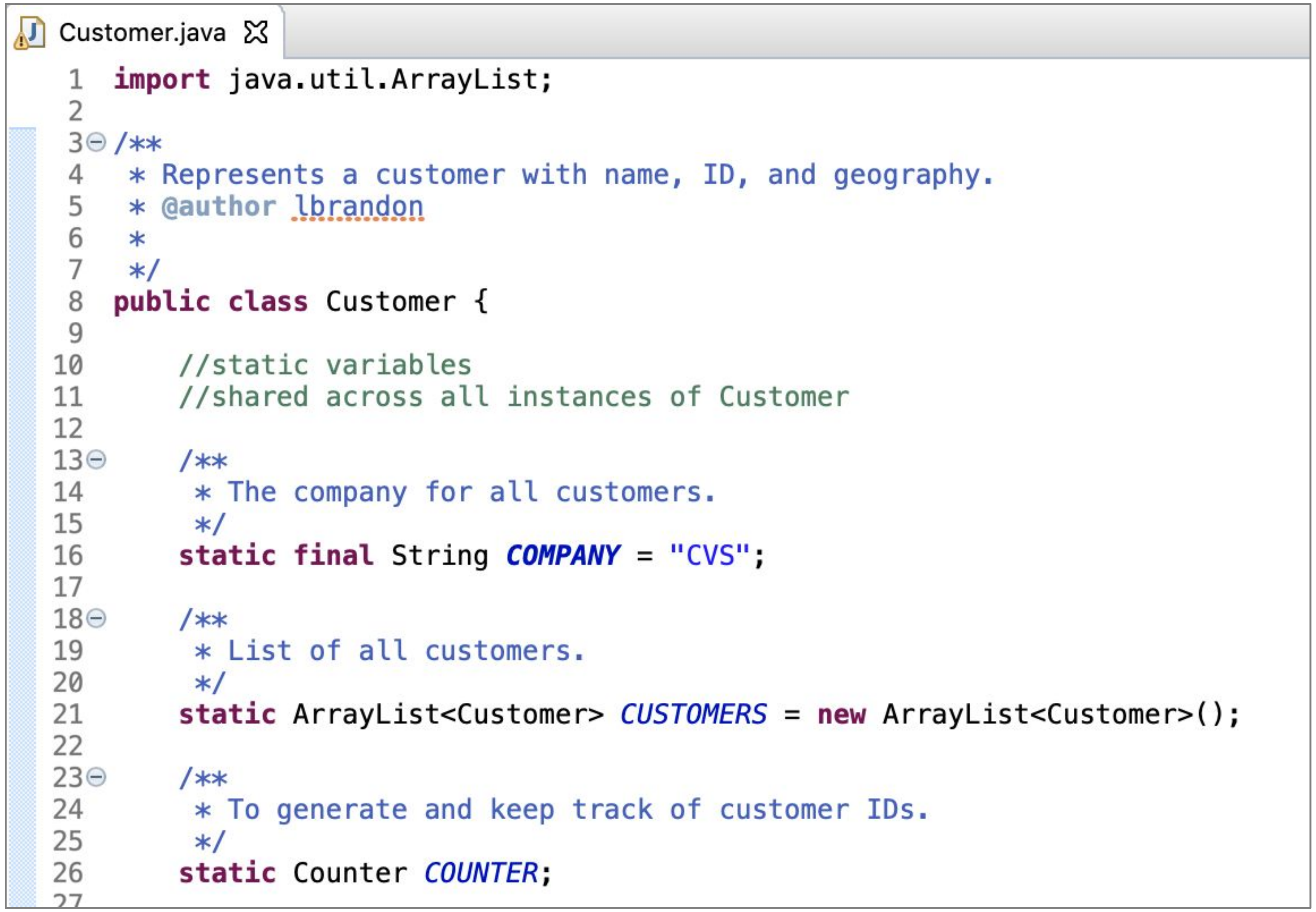
```
public class Car {
    //list of all created cars
    static ArrayList<Car> CAR_LIST = new ArrayList<Car>();

    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
- 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*

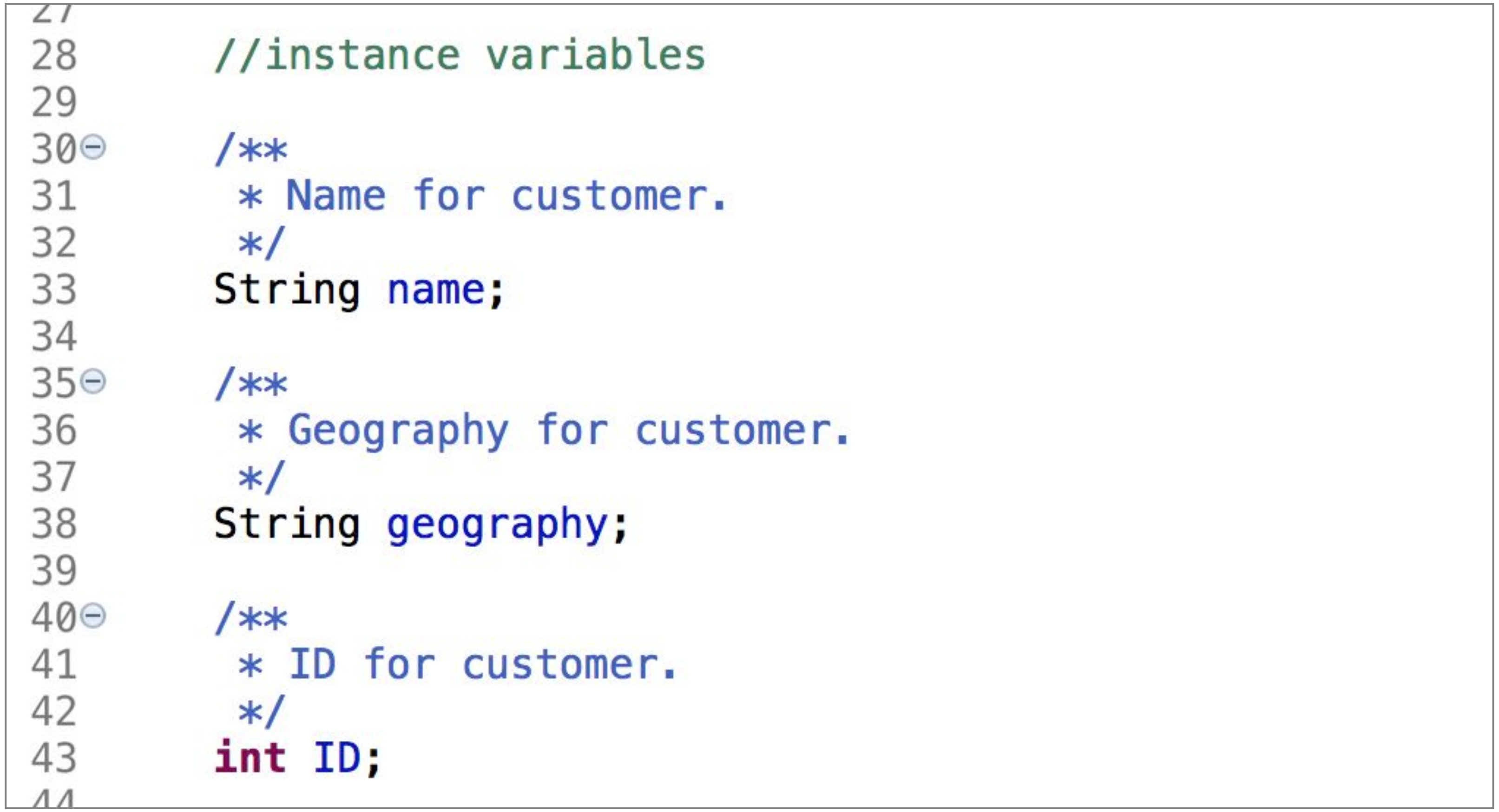
Customer Tracking Project

Customer Class



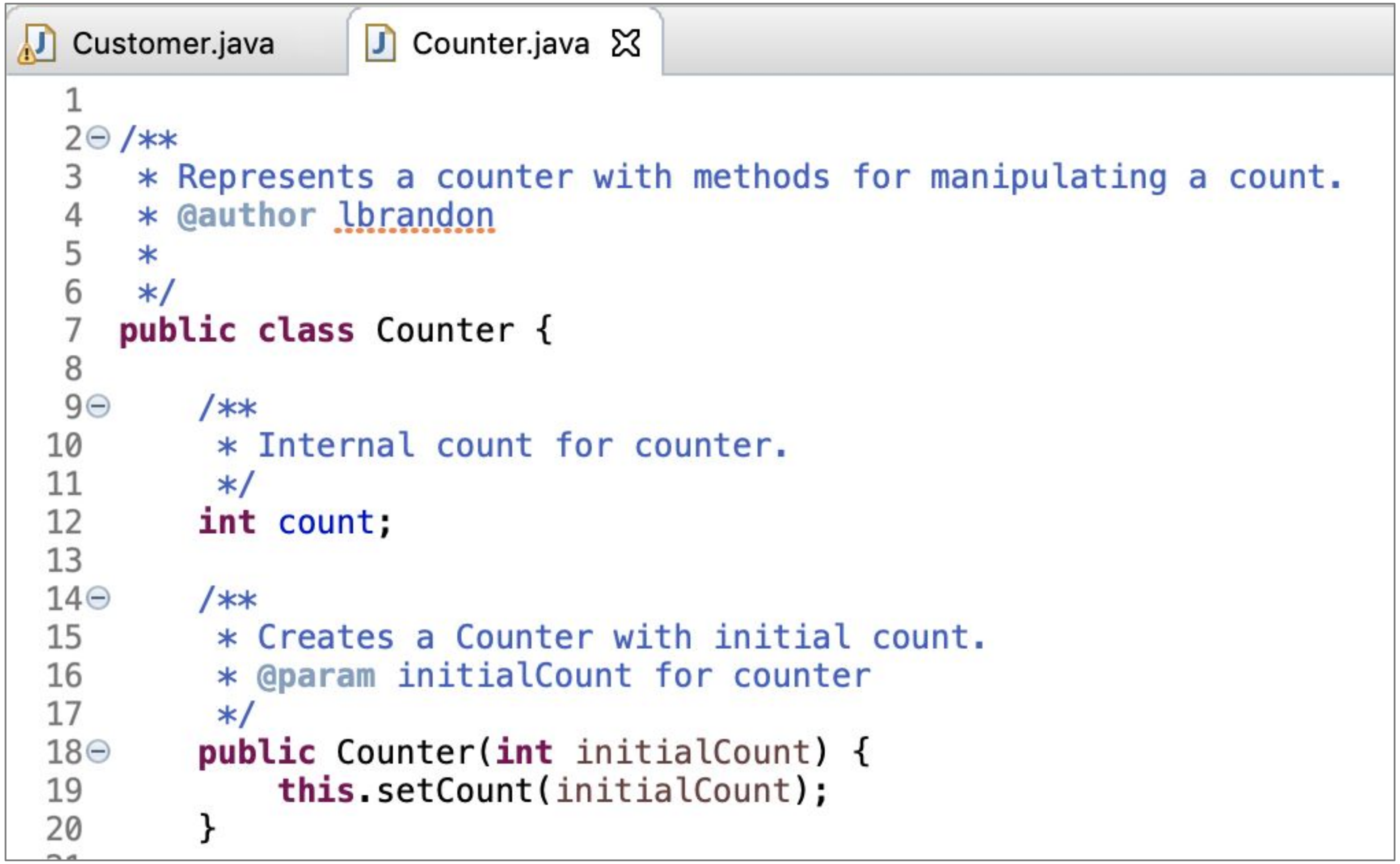
```java
import java.util.ArrayList;

/**
 * Represents a customer with name, ID, and geography.
 * @author lbrandon
 *
 */
public class Customer {

    //static variables
    //shared across all instances of Customer

    /**
     * The company for all customers.
     */
    static final String COMPANY = "CVS";

    /**
     * List of all customers.
     */
    static ArrayList<Customer> CUSTOMERS = new ArrayList<Customer>();

    /**
     * To generate and keep track of customer IDs.
     */
    static Counter COUNTER;
```

Customer Class



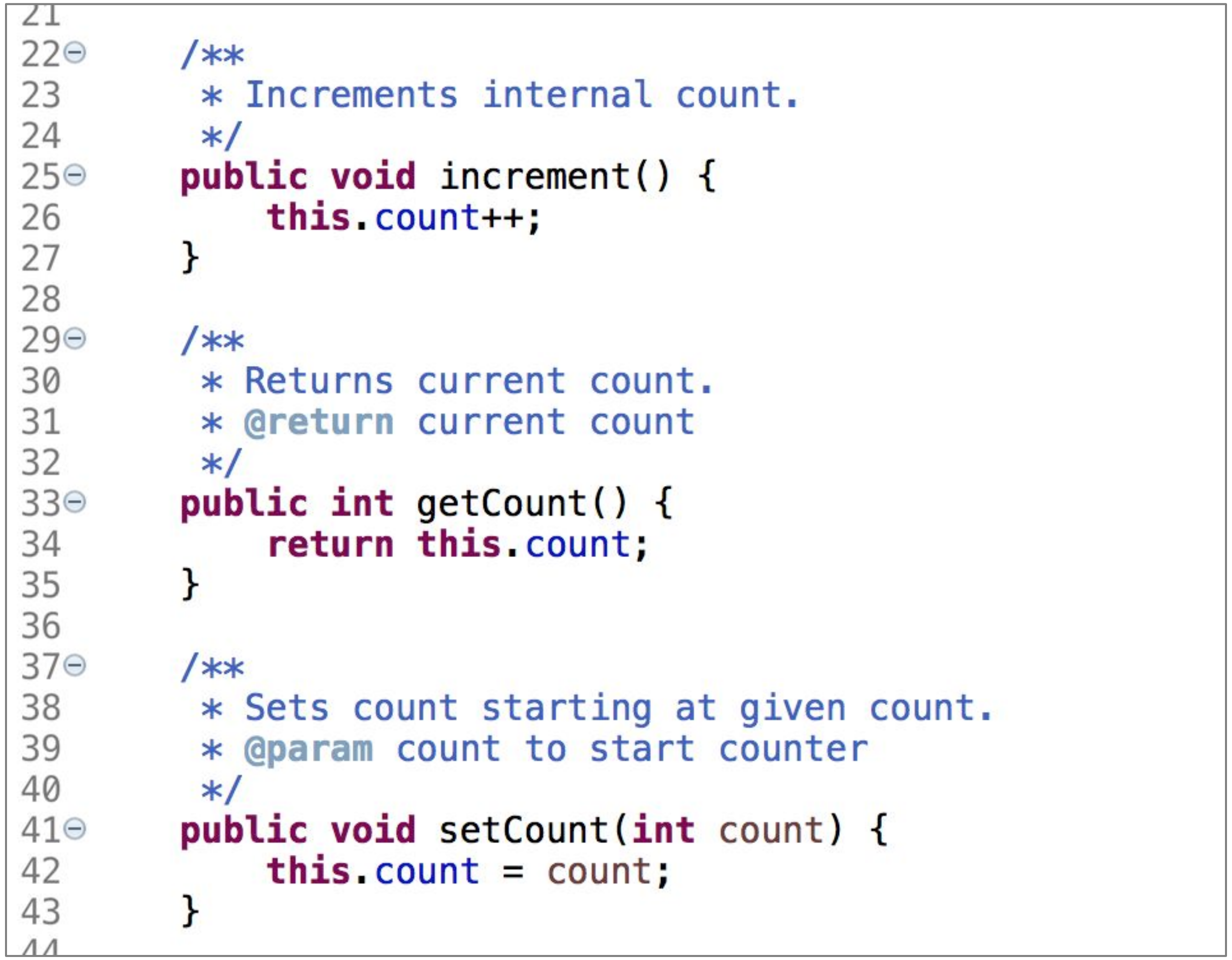
```
    //instance variables

    /**
     * Name for customer.
     */
    String name;

    /**
     * Geography for customer.
     */
    String geography;

    /**
     * ID for customer.
     */
    int ID;
```

Counter Class



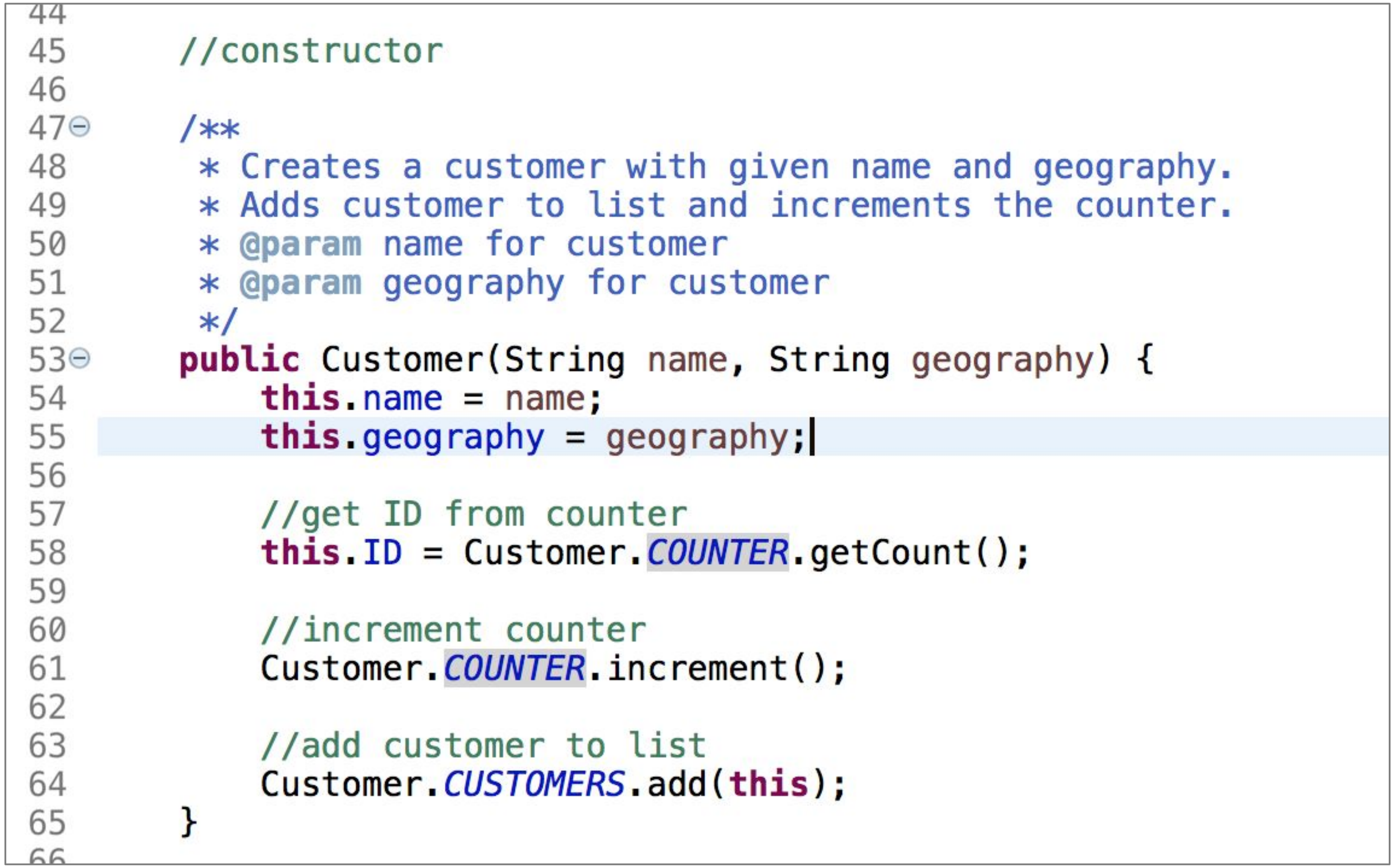
```
/**
 * Represents a counter with methods for manipulating a count.
 * @author lbrandon
 *
 */
public class Counter {

    /**
     * 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



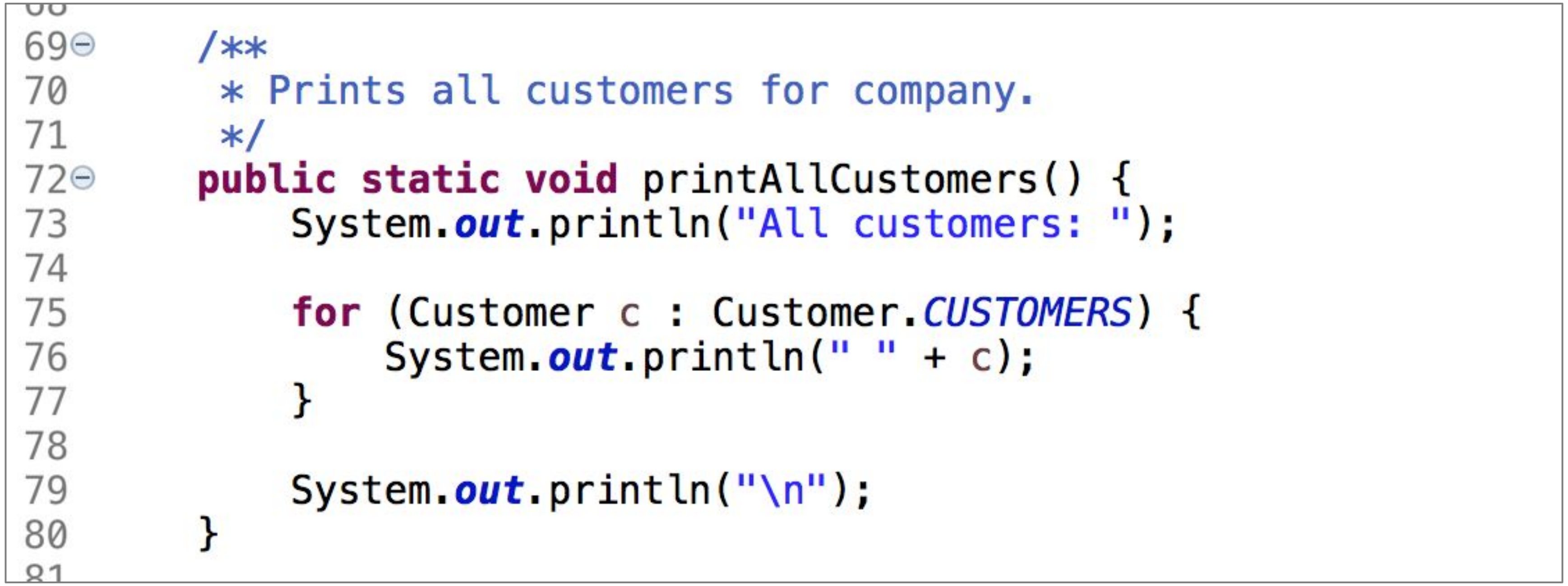
```
    /**
     * Increments internal count.
     */
    public void increment() {
        this.count++;
    }

    /**
     * Returns current count.
     * @return current count
     */
    public int getCount() {
        return this.count;
    }

    /**
     * Sets count starting at given count.
     * @param count to start counter
     */
    public void setCount(int count) {
        this.count = count;
    }
```

Customer Class



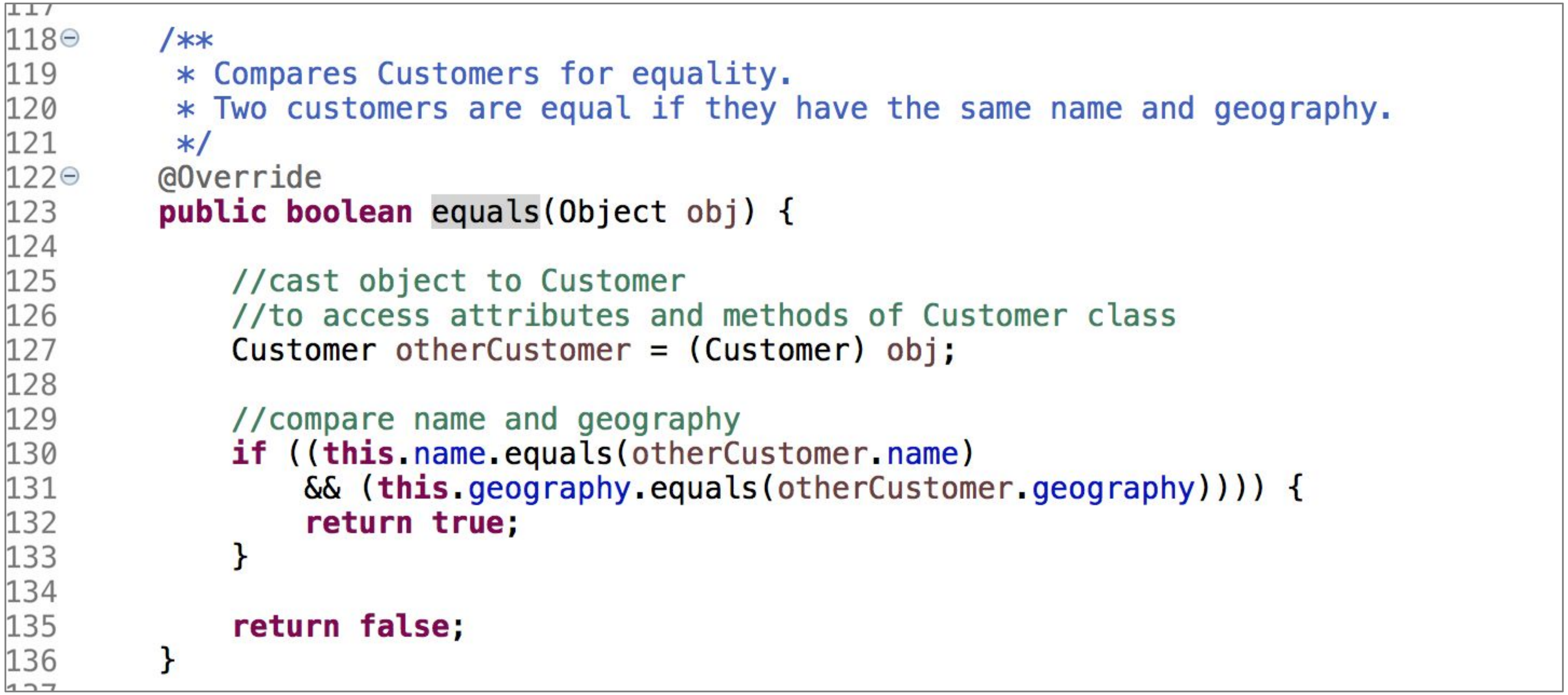
```
    //constructor

    /**
     * Creates a customer with given name and geography.
     * Adds customer to list and increments the counter.
     * @param name for customer
     * @param geography for customer
     */
    public Customer(String name, String geography) {
        this.name = name;
        this.geography = geography;

        //get ID from counter
        this.ID = Customer.COUNTER.getCount();

        //increment counter
        Customer.COUNTER.increment();

        //add customer to list
        Customer.CUSTOMERS.add(this);
    }
```

Customer Class



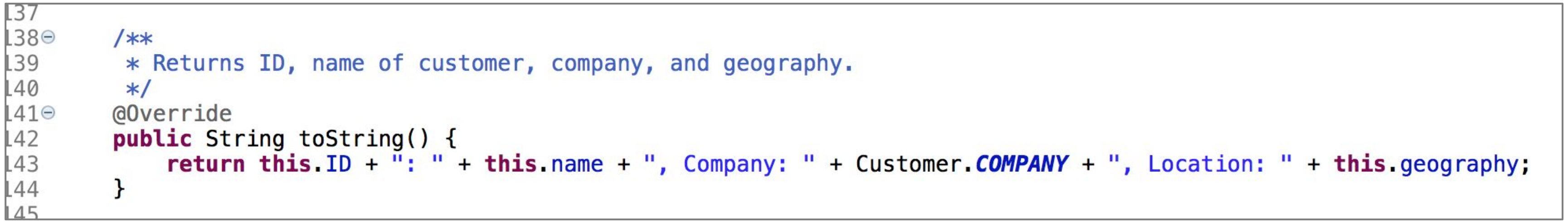
```
    /**
     * Prints all customers for company.
     */
    public static void printAllCustomers() {
        System.out.println("All customers: ");

        for (Customer c : Customer.CUSTOMERS) {
            System.out.println(" " + c);
        }

        System.out.println("\n");
    }
```

Customer Class



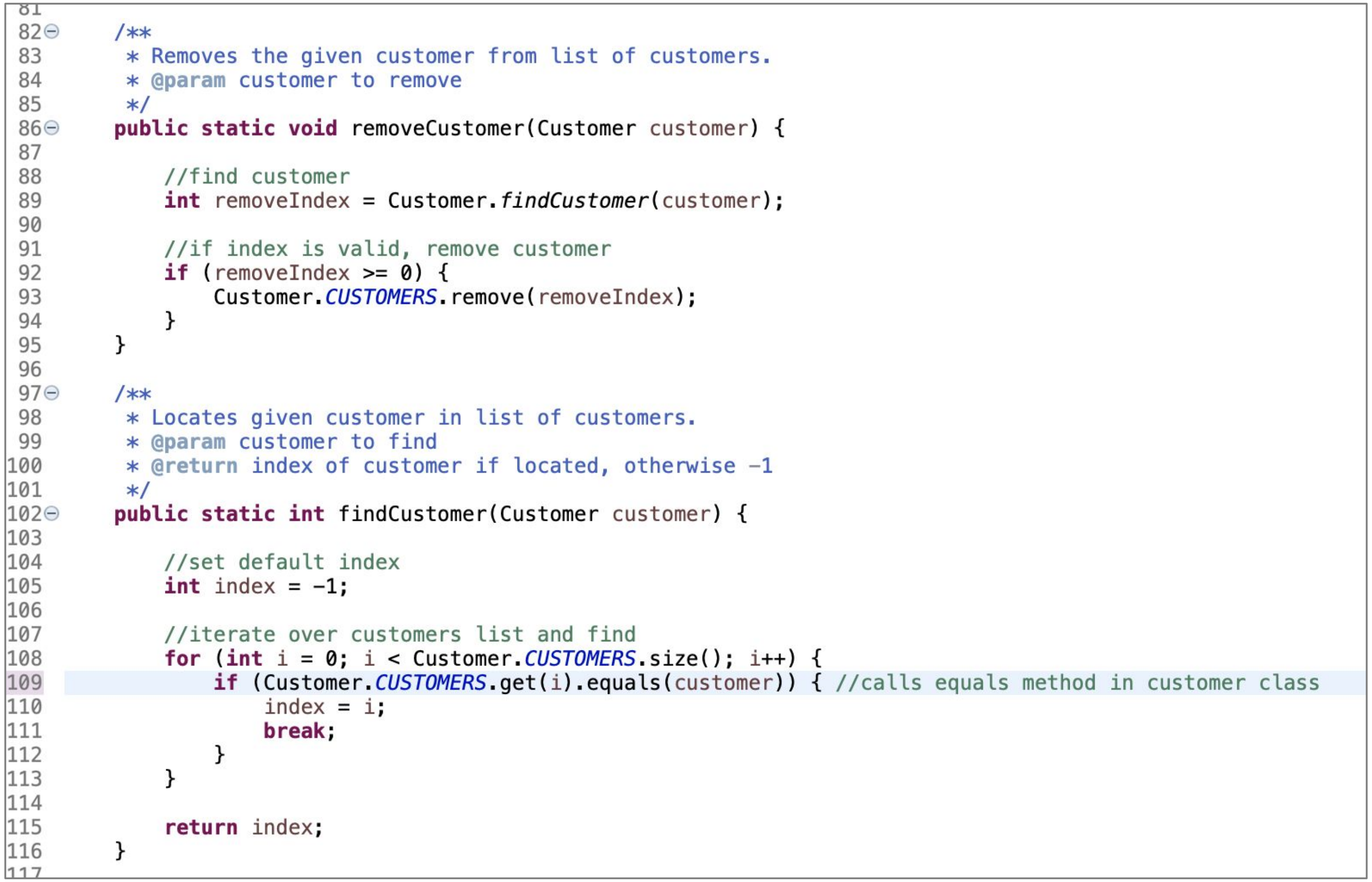
```java
    /**
     * Compares Customers for equality.
     * Two customers are equal if they have the same name and geography.
     */
    @Override
    public boolean equals(Object obj) {

        //cast object to Customer
        //to access attributes and methods of Customer class
        Customer otherCustomer = (Customer) obj;

        //compare name and geography
        if ((this.name.equals(otherCustomer.name)
            && (this.geography.equals(otherCustomer.geography)))) {
            return true;
        }

        return false;
    }
```

Customer Class



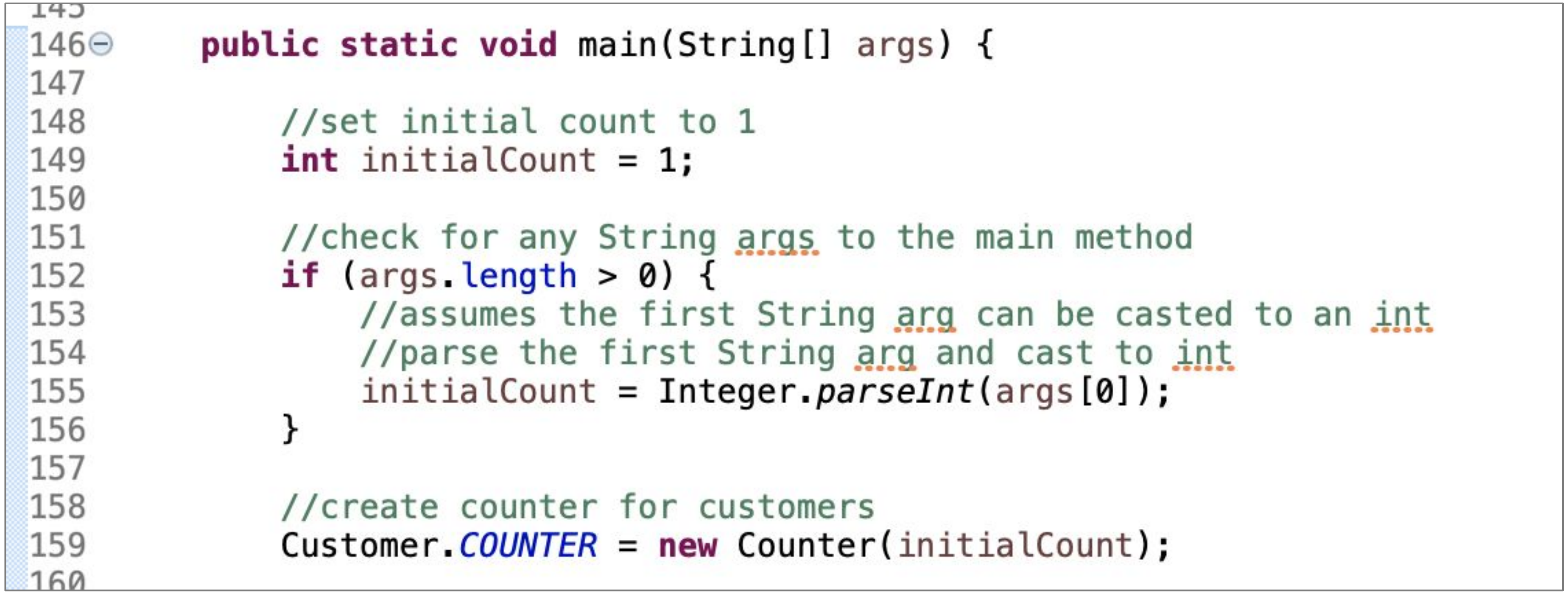
```
    /**
     * Returns ID, name of customer, company, and geography.
     */
    @Override
    public String toString() {
        return this.ID + ": " + this.name + ", Company: " + Customer.COMPANY + ", Location: " + this.geography;
    }
```

Customer Class



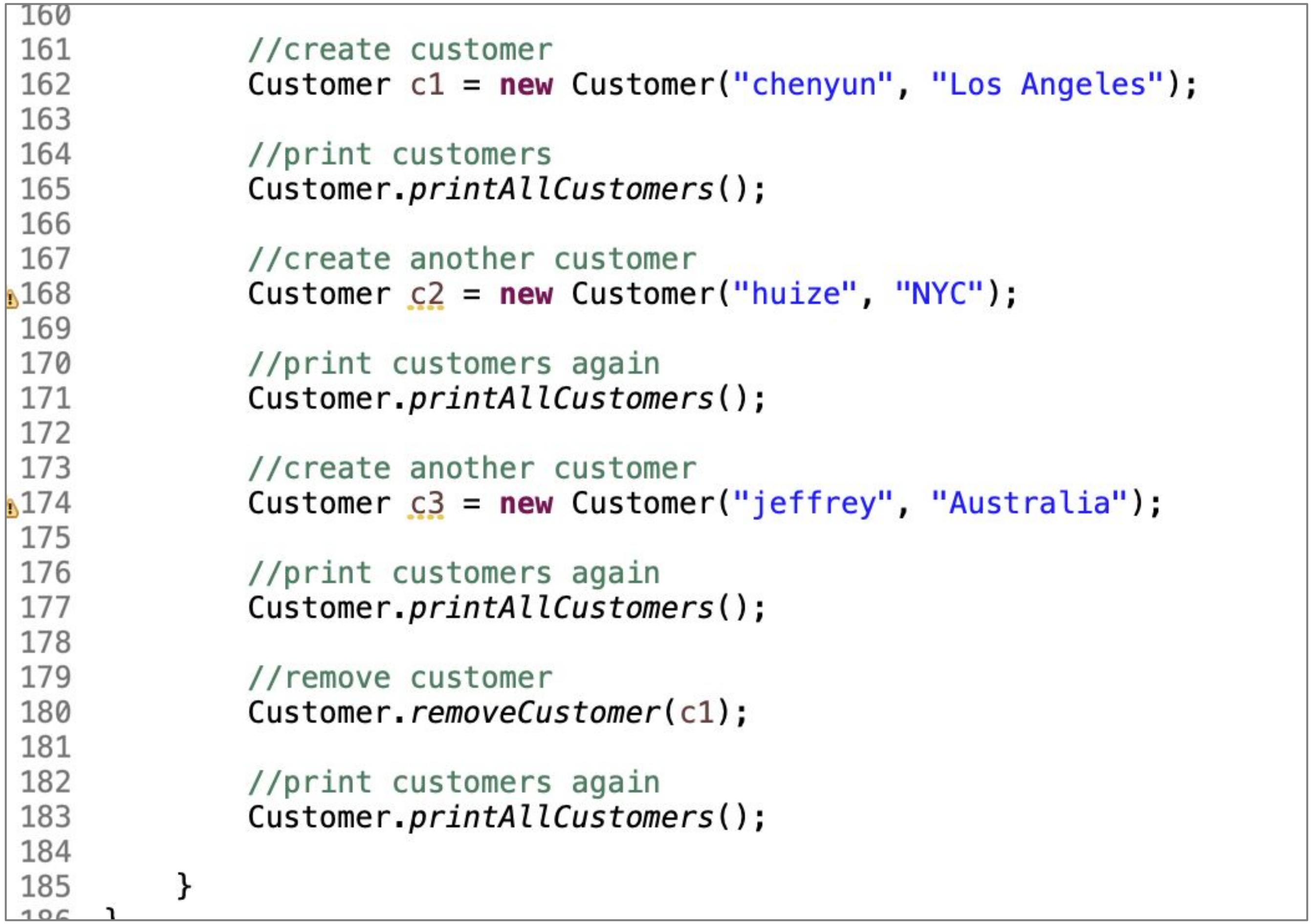
```java
    /**
     * Removes the given customer from list of customers.
     * @param customer to remove
     */
    public static void removeCustomer(Customer customer) {

        //find customer
        int removeIndex = Customer.findCustomer(customer);

        //if index is valid, remove customer
        if (removeIndex >= 0) {
            Customer.CUSTOMERS.remove(removeIndex);
        }
    }

    /**
     * Locates given customer in list of customers.
     * @param customer to find
     * @return index of customer if located, otherwise -1
     */
    public static int findCustomer(Customer customer) {

        //set default index
        int index = -1;

        //iterate over customers list and find
        for (int i = 0; i < Customer.CUSTOMERS.size(); i++) {
            if (Customer.CUSTOMERS.get(i).equals(customer)) { //calls equals method in customer class
                index = i;
                break;
            }
        }

        return index;
    }
```

Customer Class



```java
    public static void main(String[] args) {

        //set initial count to 1
        int initialCount = 1;

        //check for any String args to the main method
        if (args.length > 0) {
            //assumes the first String arg can be casted to an int
            //parse the first String arg and cast to int
            initialCount = Integer.parseInt(args[0]);
        }

        //create counter for customers
        Customer.COUNTER = new Counter(initialCount);
```

Customer Class



```
        //create customer
        Customer c1 = new Customer("chenyun", "Los Angeles");

        //print customers
        Customer.printAllCustomers();

        //create another customer
        Customer c2 = new Customer("huize", "NYC");

        //print customers again
        Customer.printAllCustomers();

        //create another customer
        Customer c3 = new Customer("jeffrey", "Australia");

        //print customers again
        Customer.printAllCustomers();

        //remove customer
        Customer.removeCustomer(c1);

        //print customers again
        Customer.printAllCustomers();

    }
}
```