AP Computer Science — Object-Oriented Design, Inheritance, Interfaces

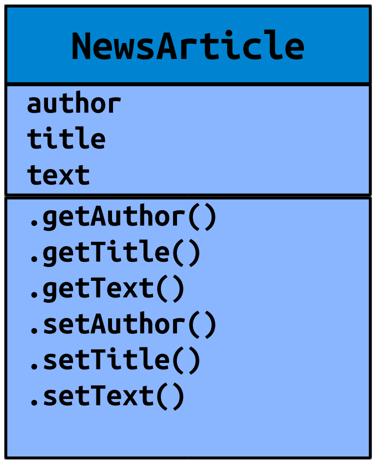
The Java programming language is often described as being an *object-oriented* language because of its strong support of *classes*, from which specific *instances* of a class can be constructed. These instances, or *objects*, or abstract representations of "things" as described by the class, which can be thought of as a blueprint or template for creating an object.

This unit develops some of the important object-oriented principles that you should take into consideration when you have been given the task of designing an object or collection of objects.

1. **Implementing a Class**
   When first learning an object-oriented language, you may be given a class description, complete with *instance variables* and *methods* for which you will write a Java-based implementation. We've been doing from the first day of this course, and if somebody says to write a given class, with these specific instance variables and these specific constructors and methods, you should be able to do so without much difficulty.

   Example: Write a class called `NewsArticle` which will be used to keep track of newspaper articles for a magazine or newspaper. It will have three instance variables—`author`, `title`, and `text`—along with methods for interacting with each of those variables. There should be two constructors: one that takes just the `author` as a parameter, and one that accepts both `author` and `title` (in that order) as parameters.

   Writing a class like this doesn't require any object-oriented *design*. You are simply writing the code that will implement a class that someone else has already designed for you.



2. **Designing a Class**

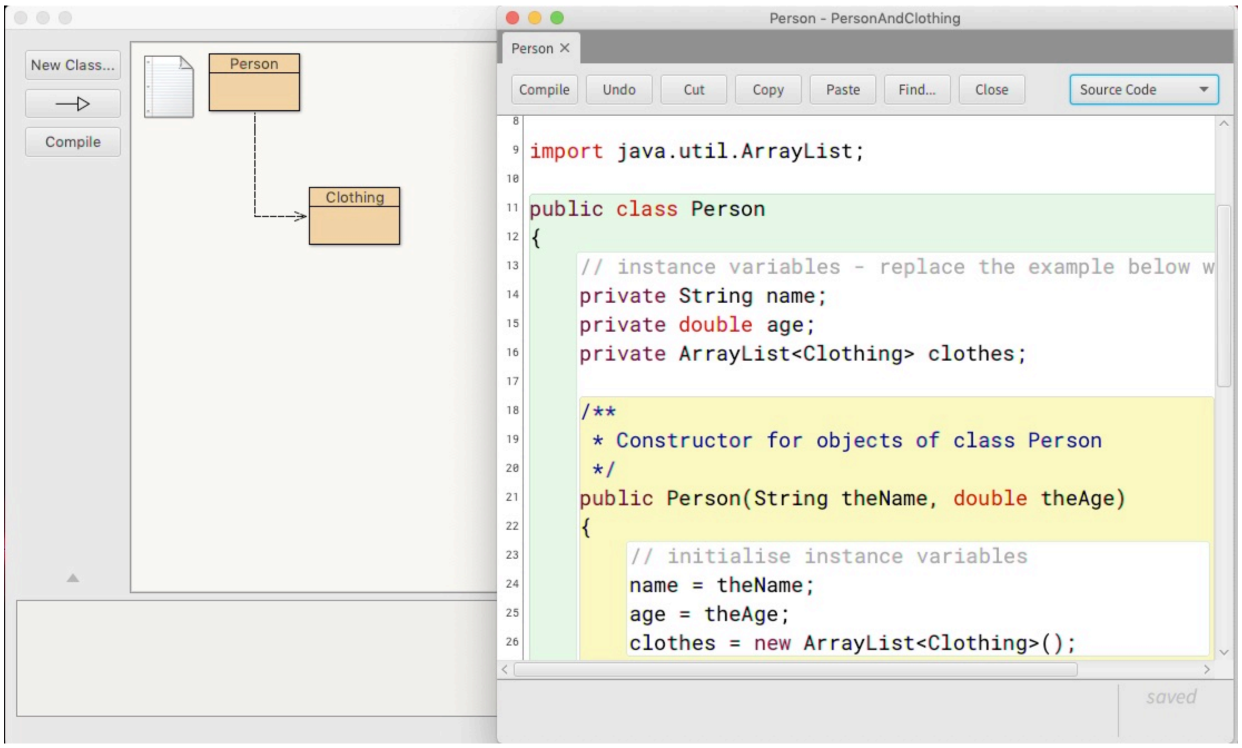
As you become more familiar with the way a class is designed—how its attributes (instance variables) and methods (accessors, or "getters," and mutators, or "setters") allow for interacting with objects of that class, you develop some idea of how to create or design a class to help solve a specific problem.

(a) **Designing to maximize *Cohesion***

A well-designed class is one that is *cohesive*: it represents a single "thing," more or less, and doesn't try to accommodate unrelated features that aren't directly related to that thing.

Example: A `Person` class might describe a person in terms of their `name` and `age`, which are more or less directly related to helping to identify a person. We could also define a person in terms of the clothing that they're wearing—top, pants or skirt, hat, shoes—but that's less connected to the idea of a person. From a cohesion perspective, it would make more sense to keep `Person` as they are, and create a new class `Clothing`, and we'd use that class with the `Person` class to manage what a person wears. The `Person` class might have an array of `Clothing` items as one of its instance variables, for example (see diagram below).

By making our classes more cohesive, writing our code is simplified as we focus on the aspects of a single, specific object.



```java
import java.util.ArrayList;

public class Person
{
    // instance variables - replace the example below w
    private String name;
    private double age;
    private ArrayList<Clothing> clothes;

    /**
     * Constructor for objects of class Person
     */
    public Person(String theName, double theAge)
    {
        // initialise instance variables
        name = theName;
        age = theAge;
        clothes = new ArrayList<Clothing>();
```

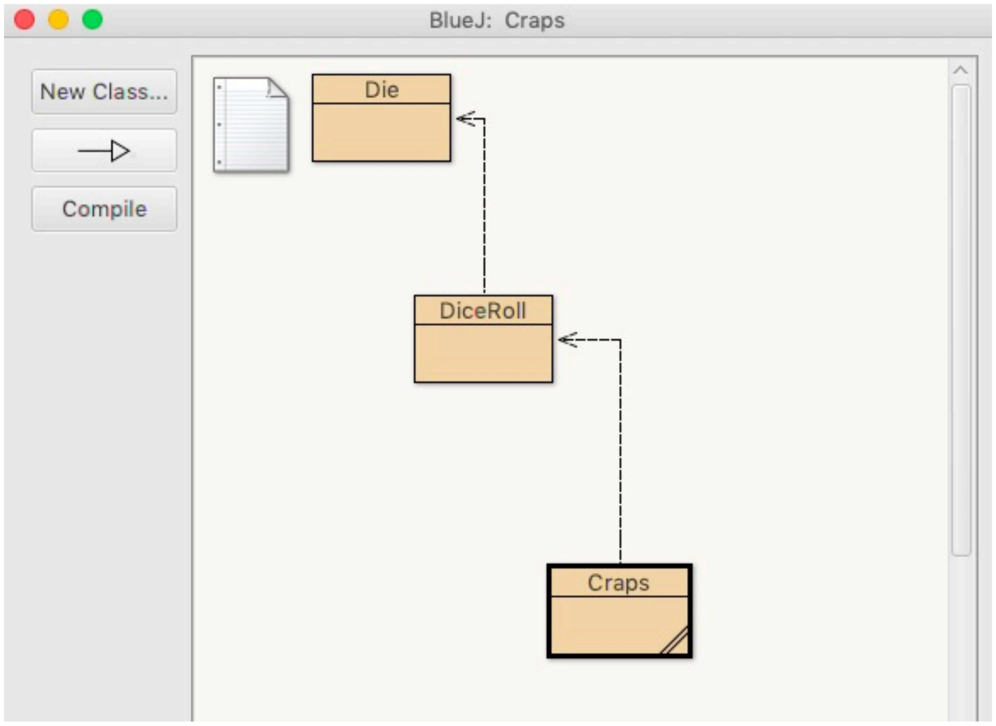
(b) **Designing to minimize *Coupling***

Coupling has to do with dependencies between classes. If a class has a lot of connections to other classes, then we say that there is a "high degree of coupling." A `Student` class is a member of a `Course` class which is part of a `Schedule`, and the `Student` is part of a `GradeLevel`, but is also a member of a `School`, where there are `Teachers,` and a `Teacher` teaches that `Course`,... In a complex problem, there is necessarily more coupling, more interdependence of classes.

There is always going to be some coupling between classes—that's the whole point of writing cohesive classes, so we can have them interact with each other—but you'll know you have too much coupling when you find that some of your classes are connected to too many other classes.

Example: A `Die` class might manage a six-sided die object, and a `Dice` class might manage two of those `Die` objects so that people can roll the dice in a `Craps` game. It's expected that the game class will interact with the `Dice` class—rolling dice is part of the game. But there should be no need for the `Craps` class to call the `Die` class directly. (See below.)

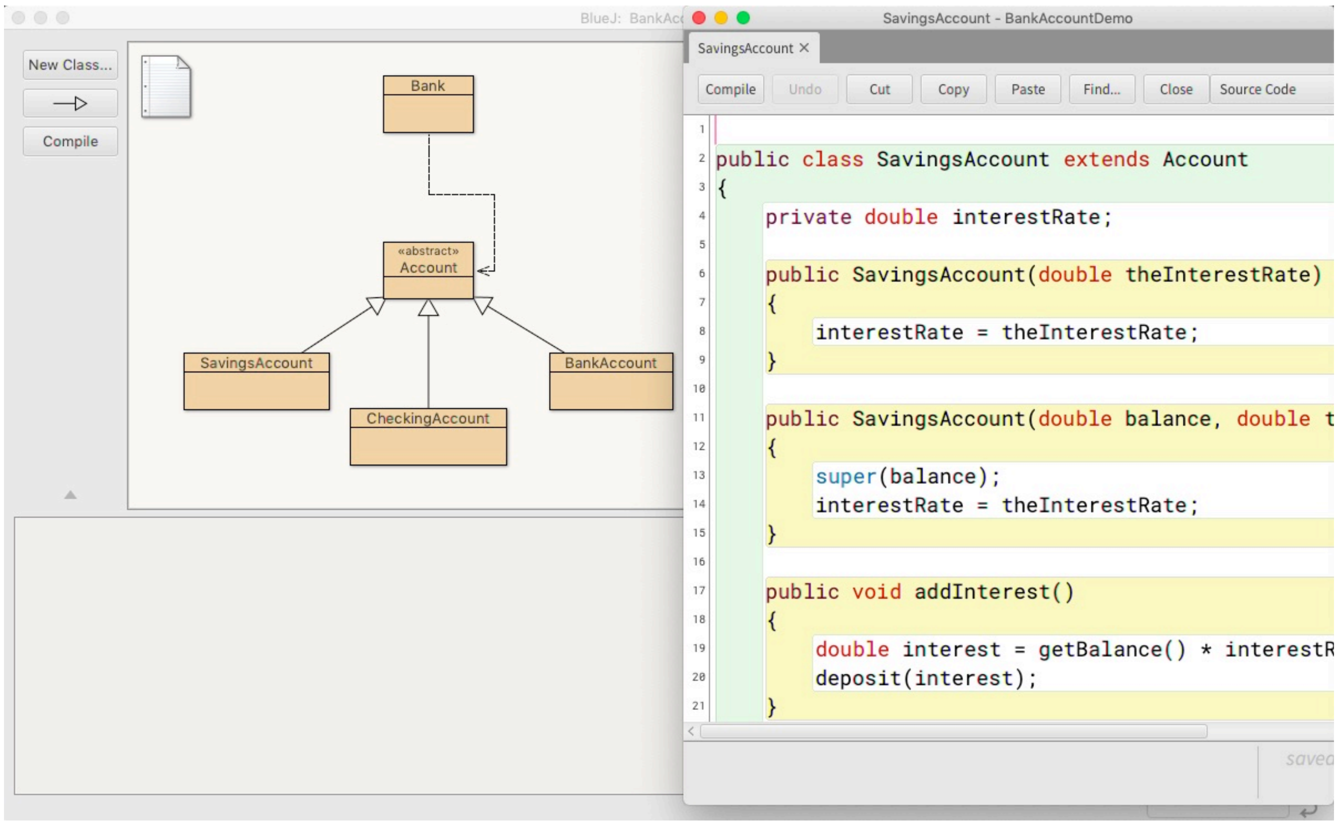
By reducing coupling where possible, we can keep the relationships between our classes simpler and better organized. Where coupling has been reduced, we are more easily able to make changes to one part of the system without affecting a large number of other parts of the system.



3. **Reusing Code with Inheritance (Subclasses that "extend" a Superclass)**
   It is often the case that a number of components of a large program may be considered as different categories of the same thing.

   Examples: The idea of `Computer` might include being a `Laptop`, a `Desktop`, or a `Phone`; a `BankAccount` might actually be a `SavingsAccount` or a `CheckingAccount`, and `Money` might be considered as a `Coin` or a `Bill`.

   In each case, the larger, more general class is a *superclass*, while the smaller, more specific class, *subclass* "extends" the superclass. Subclasses inherit the instance variables and methods of the superclass, but also have the ability to add their own instance variables and methods, and even override methods described by the superclass.

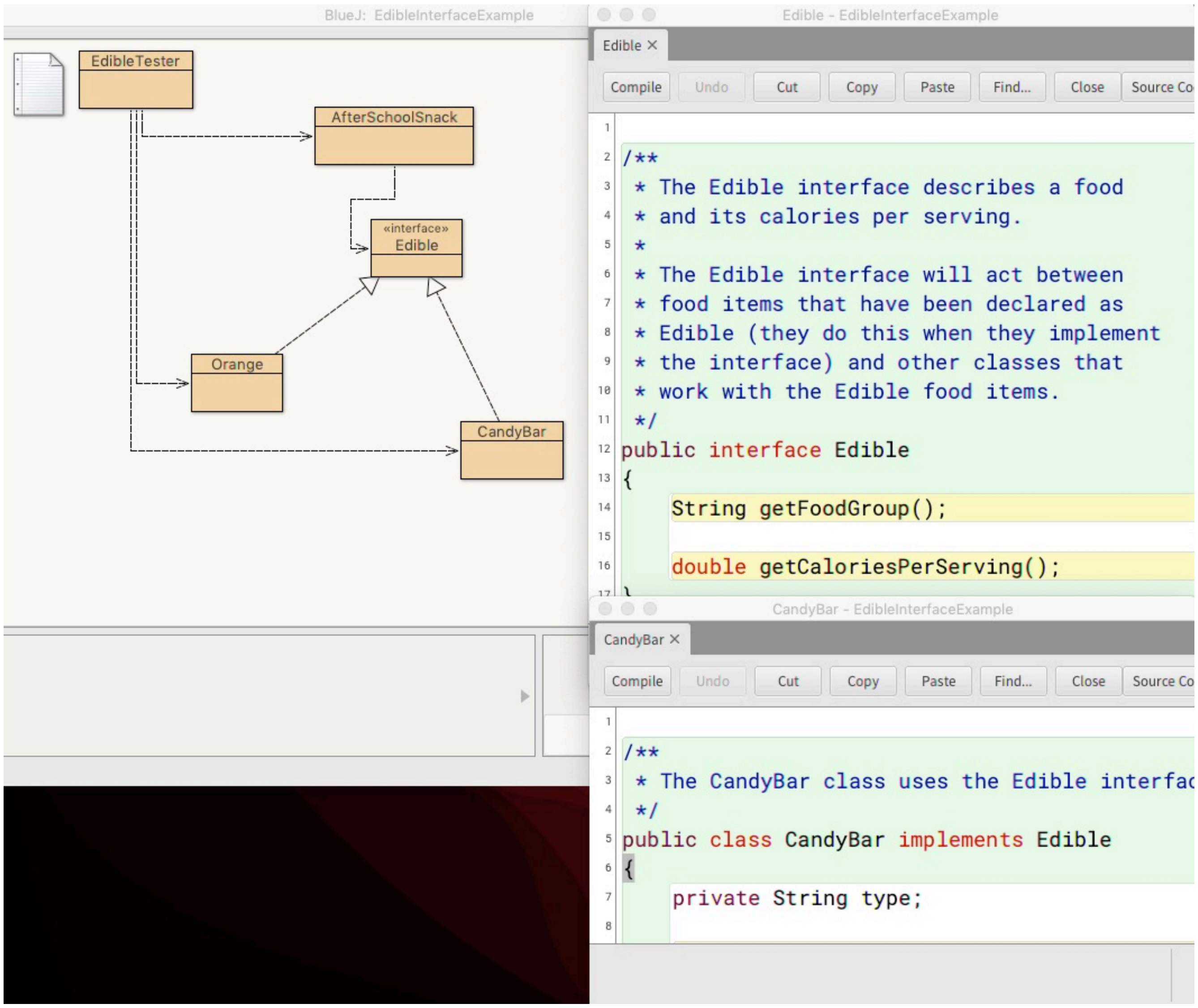


```
public class SavingsAccount extends Account
{
    private double interestRate;

    public SavingsAccount(double theInterestRate)
    {
        interestRate = theInterestRate;
    }

    public SavingsAccount(double balance, double t
    {
        super(balance);
        interestRate = theInterestRate;
    }

    public void addInterest()
    {
        double interest = getBalance() * interestR
        deposit(interest);
    }
```

4. **Reusing Code with Interfaces (Classes that "implement" an Interface)**
Another type of code reuse occurs when a class "implements" an interface. An interface is "abstract"—you can't make an instance of it—and is usually given a name in the form of an adjective: `Comparable`, `Measurable`, `Edible`, `Drivable`. The interface itself describes at least one method, for which only the header is given; it's up to a class that implements the interface to actually describe the code that will be used when that interface method is used.

Example: The `Comparable` interface is described by the official Java API, and requires that any class implementing the `Comparable` interface implement the `compareTo()` method. There are lots of different ways to compare things: numbers are compared according to their values, `String` objects are compared according to their alphabetical order, basketball players might be compared according to their height, ...

Why would you want to implement an interface? Couldn't you just write your own `compareTo()` method? You certainly could, but it turns out that Java provides support for `Comparable` objects. There are libraries of programs that work with `Comparable` objects—that allow you to sort them, for example—*but they only work with Comparable objects*. If you want to take advantage of any library that works with a particular type of interface, you'll want to be sure to implement that interface.



```

/**
 * The Edible interface describes a food
 * and its calories per serving.
 *
 * The Edible interface will act between
 * food items that have been declared as
 * Edible (they do this when they implement
 * the interface) and other classes that
 * work with the Edible food items.
 */
public interface Edible
{
    String getFoodGroup();

    double getCaloriesPerServing();
}
```

```

/**
 * The CandyBar class uses the Edible interfac
 */
public class CandyBar implements Edible
{
    private String type;

```

In addition to these large-scale themes in this unit, there are a number of other details and ideas with which you should be familiar.

1. **Subclasses delivering constructor parameters to a superclass**
   Use the super(parameter); statement as the first line in the subclass constructor.

2. **Subclasses inherit instance variables and methods from the superclass**
   That's the reason we use subclasses: to take advantage of the methods that have already been written.

3. **Subclasses can override superclass methods**
   Although a superclass may have a method we can use, we're not stuck using that version of the method: we can rewrite it to do something unique with our subclass. One common strategy is to override the toString() method so that we can use it to reveal information about our specific subclass. Likewise, the equals() method may be overridden so that two objects that inherit from the "cosmic superclass" Object can be compared to each other.

4. **Converting between subclasses and superclasses**
   A superclass reference can be used when a subclass is expected... but a subclass reference can't be substituted for a superclass reference.

   Example: The Person superclass has a Student subclass. We can make an ArrayList<Person> and put Student objects in there, but we can't make an ArrayList<Student> and put Person objects in there.

   If you have an ArrayList<Person> and you know that one of the elements is a Student, you can always cast that Person object into a Student object, and then interact with that reference.

5. ***Abstract* classes, with abstract methods, are used to indicate methods that must be implemented**
   This is similar in some ways to the idea of an interface, but with some important differences.
   (a) Abstract classes have instance variables, constructors, instance variables, and concrete methods. These all apply to all classes that extend the abstract class. They also have abstract methods that are not defined in the abstract class, but must be defined by the extending class. You can't create an object from an abstract class, only from classes that extend the abstract class, and thus inherit its qualities. We say that the abstract class does have a *state* that is defined by its variables.
   (b) An interface has no instance variables and no state. It only defines methods that must be defined by classes that implement the interface.
   (c) A class may inherit from only one superclass, but a class may implement multiple interfaces.

EXERCISES

1. Write a class `ElectronicDevice` that has a name and a boolean variable `turnedOn`. Include constructors, accessor methods, and mutator methods that will allow a client program to interact with the class.

2. Some electronic devices have batteries that can be recharged. Write an interface `Chargeable` that includes three methods: `addEnergy(double percent)`, which will allow some amount of energy (as a percent of a battery's maximum capacity) to be added, `getEnergy()`, which will reveal the battery's remaining energy (as a percent of battery's maximum capacity), and `useEnergy(double percent)`, which will reduce the amount of energy stored in the battery by a percentage of its maximum capacity when the device is used.

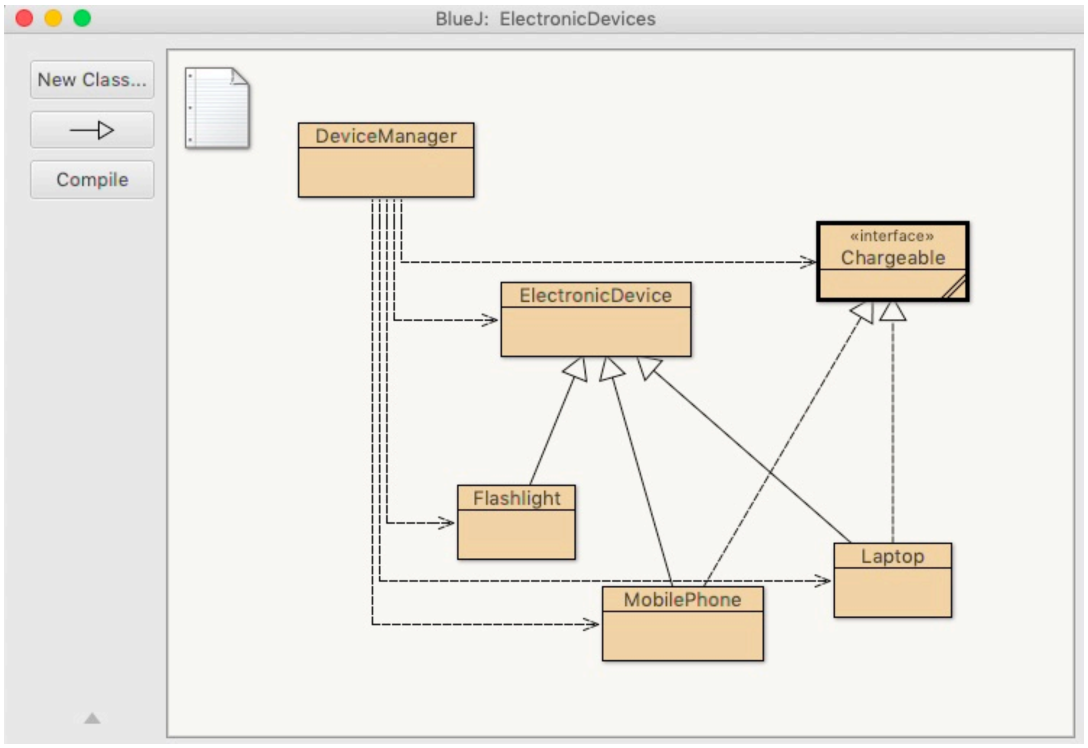
3. Write a class `MobilePhone` that inherits from `ElectronicDevice` and implements `Chargeable`. What additional instance variables and methods might be needed for this new class to work?

4. Write a class `Flashlight` that inherits from `ElectronicDevice`. This class should have a String instance variable `batterySize`.

5. Write `toString()` methods for each class so that we can easily observe their state.

6. You'd like to write classes for three more types of devices: a `LaptopComputer`, a `TICalculator`, and a `DesktopComputer`. How do these fit into the classes you've already written? Should they inherit from any classes? Should classes inherit from them? Which of these new devices should implement the `Chargeable` interface?

7. Write a `main()` program called `DeviceManager` that creates an `ArrayList` of `ElectronicDevice` objects and demonstrates interactions with them.



EXERCISE SOLUTIONS

```java
/**
 * The class ElectronicDevice here.
 */
public class ElectronicDevice
{
    private String name;
    private boolean turnedOn;

    public ElectronicDevice(String theName)
    {
        name = theName;
        turnedOn = false;
    }

    public String getName()
    {
        return name;
    }

    public void turnOn()
    {
        turnedOn = true;
    }

    public void turnOff()
    {
        turnedOn = false;
    }

    public boolean isTurnedOn()
    {
        return turnedOn == true;
    }

    public String toString()
    {
        return super.toString() + "[name=" + name +
                                  ",turnedOn=" + turnedOn + "]";
    }
}
```

```java
/**
 * Write a description of interface Chargeable here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public interface Chargeable
{
    void addEnergy(double percent);

    double getEnergy();

    void useEnergy(double percent);
}
```

```java
/**
 * Describes the class MobilePhone.
 */
public class MobilePhone extends ElectronicDevice implements Chargeable
{
    private double batteryLevel;

    /**
     * Constructor for objects of class MobilePhone
     */
    public MobilePhone(String name)
    {
        super(name);
        batteryLevel = 100; // 100%
    }

    public void addEnergy(double percent)
    {
        batteryLevel += percent;
    }

    public double getEnergy()
    {
        return batteryLevel;
    }

    public void useEnergy(double percent)
    {
        batteryLevel -= percent;
    }

    public void makeCall(double minutes)
    {
        batteryLevel -= minutes * 0.5;
    }

    public void sendText()
    {
        batteryLevel -= 0.1;
    }

    public String toString()
    {
        return super.toString() + "[batteryLevel=" + batteryLevel + "]";
    }
}


/**
 * Describes the class Flashlight here.
 */
public class Flashlight extends ElectronicDevice
{
    private int name;
    private String batterySize;

    /**
     * Constructor for objects of class Flashlight
     */
    public Flashlight(String name, String batterySize)
    {
        super(name);
        this.batterySize = batterySize;
    }

    /**
     * An example of a method - replace this comment with your own
     */
    public String getBatterySize()
    {
        return batterySize;
    }
}
```

```java
/**
 * Describes the class MobilePhone.
 */
public class Laptop extends ElectronicDevice implements Chargeable
{
    private double batteryLevel;

    /**
     * Constructor for objects of class MobilePhone
     */
    public Laptop(String name)
    {
        super(name);
        batteryLevel = 100; // 100%
    }

    public void addEnergy(double percent)
    {
        batteryLevel += percent;
    }

    public double getEnergy()
    {
        return batteryLevel;
    }

    public void useEnergy(double percent)
    {
        batteryLevel -= percent;
    }

    public void surfInternet(double minutes)
    {
        batteryLevel -= minutes * 0.5;
    }

    public void sendEmail()
    {
        batteryLevel -= 0.1;
    }
}
```

```java
/**
 * The DeviceManager demonstrates the use of all the devices.
 */

import java.util.ArrayList;

public class DeviceManager
{
    public static void main(String[] args)
    {
        System.out.println("Initializing devices into ArrayList...\n");
        ArrayList<ElectronicDevice> devices = new ArrayList<ElectronicDevice>();
        devices.add(new Laptop("MacBookPro"));
        devices.add(new Flashlight("Maglite", "D"));
        devices.add(new MobilePhone("iPhone6"));
        devices.add(new ElectronicDevice("Nintendo Switch"));

        System.out.println("Interacting with machines...\n");
        Laptop myLaptop = (Laptop) devices.get(0);
        myLaptop.turnOn();
        myLaptop.surfInternet(2.5);
        myLaptop.sendEmail();

        Flashlight myMag = (Flashlight) devices.get(1);
        myMag.turnOn();

        MobilePhone myPhone = (MobilePhone) devices.get(2);
        myPhone.turnOn();
        myPhone.makeCall(60); // one-hour call
        for (int i = 0; i < 100; i++)   // sending 100 texts
            myPhone.sendText();

        System.out.println("All devices:");
        for (ElectronicDevice device : devices)
        {
            System.out.println(device);
        }

        System.out.println("\nDevices that have been turned on:");
        for (ElectronicDevice device : devices)
        {
            if (device.isTurnedOn())
                System.out.println(device);
        }

        System.out.println("\nBattery state of Rechargeables:");
        for (ElectronicDevice device : devices)
        {
            if (device instanceof Chargeable)   // only getEnergy for rechargeables
            {
                Chargeable chargeableDevice = (Chargeable) device; // cast
                System.out.println(device.getName() + ", " +
                                   chargeableDevice.getEnergy());
            }
        }
    }
}
```

```
OUTPUT:

Initializing devices into ArrayList...

Interacting with machines...

All devices:
Laptop@788ec784[name=MacBookPro,turnedOn=true]
Flashlight@42273aff[name=Maglite,turnedOn=true]
MobilePhone@6cc5499[name=iPhone6,turnedOn=true][batteryLevel=60.000000000000284]
ElectronicDevice@5a888156[name=Nintendo Switch,turnedOn=false]

Devices that have been turned on:
Laptop@788ec784[name=MacBookPro,turnedOn=true]
Flashlight@42273aff[name=Maglite,turnedOn=true]
MobilePhone@6cc5499[name=iPhone6,turnedOn=true][batteryLevel=60.000000000000284]

Battery state of Rechargeables:
MacBookPro, 98.65
iPhone6, 60.000000000000284
```