Unit 3 Tutorials: Classes

INSIDE UNIT 3

Introduction to Classes

by Sophia

WHAT'S COVERED

In this lesson, you will explore the basics of object-oriented programming. You will also define its structure, including classes. Specifically, this lesson covers:

1. Recap and OOP

Before you explore object-oriented programming and classes, consider a few concepts that were discussed in previous tutorials.

In Units 1 and 2, you learned the four basic programming structures for writing code. They include:

- Sequential code - Lines of code that do something in sequence.
- Conditional code - Using selection statements such as if, else, and elif statements to check for certain conditions and act on those conditions when found, or not found.
- Repetitive code - Using for and while loops to repeat a process or move through a data collection type.
- Reusable code - Using either built-in or user-created methods to perform tasks that can be utilized either once or multiple times once they are created.

You also learned about creating variables by using proper data types and identifiers. Think back to the first one you created, `String myVar = "Hello World";`. You also explored various uses of data structures such as arrays and different collection types (`ArrayList`, `Set`, and `HashMap`).
You saw and discussed some poorly written code and practiced writing good code. You also optimized good code. It took three iterations for the Tic-Tac-Toe game, but in the end, it was a much better game and provided a much better player experience. Hopefully, at this point in the course, you are starting to see that there is an art and an aesthetic to writing code.

Finally, you have seen that your programs are getting longer and longer, as you continue to learn new functionality that exists within Java. As you become more experienced writing programs, and the solutions dictate the need, you will continue writing longer programs.

Some programs can get to be millions of lines long! The average modern high-end car can contain up to 100 million lines of code. Here is a fun statistics site that references average lines of code per application:

https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

THINK ABOUT IT

Why is it important for us to recap the previous challenges and tutorials at this point? For starters, up to this point in the course, you have relied more on static methods and logic coding. As you have used the four structures of code, you have defined your own variables and methods. You have also explored the use of common data collection types. You will now begin to consider and look at new strategies for organizing code. This includes bundling data and methods into objects.

Second, as your programs get larger, it becomes increasingly important to write code that is easy to understand and structured in a way to optimize it as best as possible. If you are working on a large program, you can never keep the entire program in mind at the same time. You will need to think of ways to break large programs into multiple smaller pieces. This allows you to focus on smaller pieces when solving a problem, fixing a bug, or adding a new feature. You noticed when optimizing previous coding examples and projects, it was performed to reduce repetition, clean up code, and thus help make the structure and functionality of the code clearer.

HINT

That *is* the art and aesthetic to writing code. Keep it clean and simple.

Finally, all of the variables, data structures, and methods that you have been using will all be used again in this next model of programming. It was essential that you knew the basics before moving into Lesson 3. Remember when you called out the first term “object”? You learned that an object is an instance of a class that has properties and methods that are encapsulated or part of it. Up to this point, you have actually been using objects and classes. As you learned, a Java program always consists of at least one class.

Turning to **object-oriented programming**, OOP for short, it is a programming model that organizes the design of code by bundling objects. Programming languages like Java, as well as Java, C++, Ruby, and others, are based on the OOP model.

Why use OOP? The structure and organization of the object-oriented programming model make code more reusable, more scalable, and more efficient. To do this, OOP structures programming into reusable pieces of code, or classes, that create individual instances of objects from. The goal of this challenge is to develop a basic understanding of how objects are constructed and how they function. Further, you will learn how to make use of the capabilities of objects that are provided to by Java. The basics of OOP will be touched on later, but know that there is much more capability to it than those capacities found in Java.

TERM TO KNOW

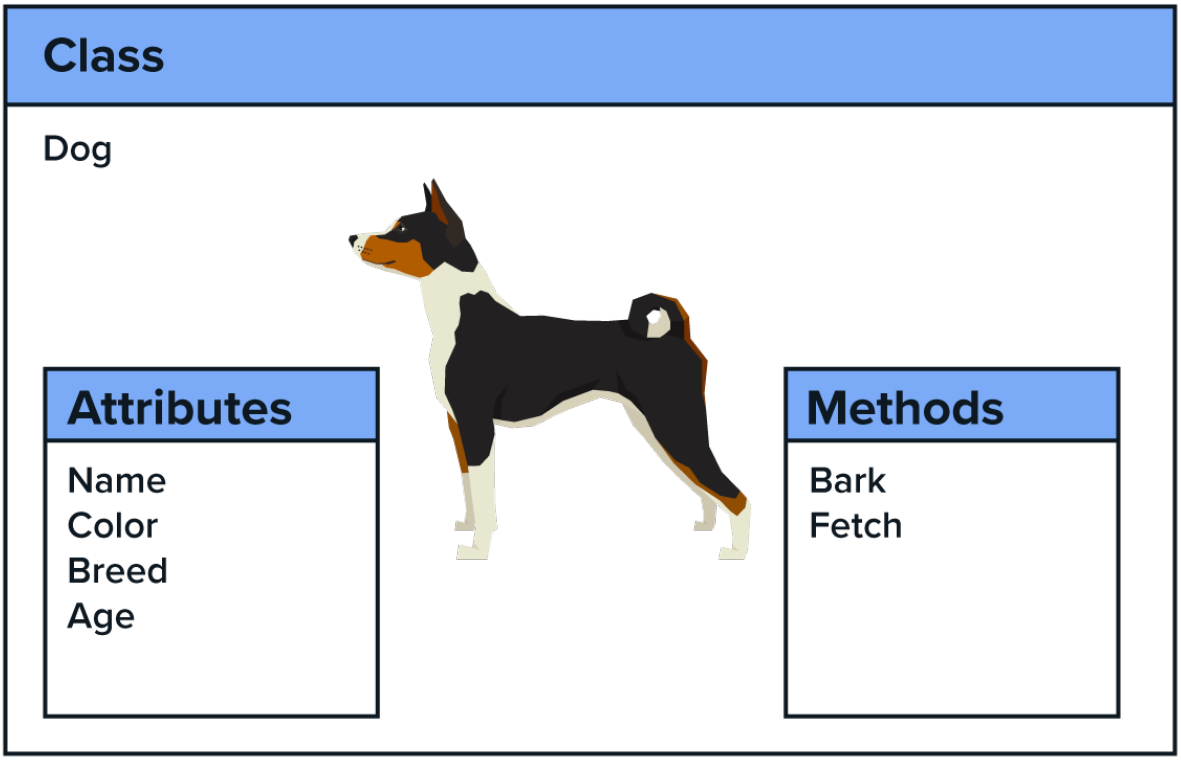
Object-Oriented Programming
Object-oriented programming (or OOP for short) is a programming model that organizes the design of code by bundling objects.

---

2. Understanding the Purpose of a Class

A **class** is a template for creating objects. Think of classes as a blueprint. In the example above,`Dog` is a class. However, it doesn’t contain any actual data. Classes contain fields for attributes and methods. The class, `Dog`, contains only the attributes and methods, or behaviors that are common to all dogs. The `Dog` class does specify that names, color, age, etc., are important to define a dog and that dogs should have behaviors like barking and fetching. However, it does not have any specific data about any particular dog.

A class has a name, and typically contains attributes and methods as demonstrated in the following graphic:



HINT

You will learn more about the code in upcoming tutorials.

For now, consider the following class:

```
// Note that the class is explicitly marked public
public class Dog {
  // Private attributes
  private String name;
  private String breed;
  private int age;
  private String color;

  // Public constructor with parameters
  public Dog(String name, String breed, int age, String color) {
    this.name = name;
    this.breed = breed;
    this.age = age;
    this.color = color;
  }

  // Accessor methods to provide read access to data in attributes
  public String getName() {
    return name;
  }
```

```
  public String getBreed() {
    return breed;
  }

  public int getAge() {
    return age;
  }

  public String getColor() {
    return color;
  }

  // Methods for behaviors
  public void bark() {
    System.out.println("Woof!");
  }

  public void fetch() {
    System.out.println(name + " went to fetch.");
  }
}
```

In order to use a class, you will need to create an instance of that class, called an object. This process is called **class instantiation**. Therefore, objects are an instance of a class that has properties (attributes) and methods that are encapsulated or part of it. The object, or instance, is uniquely created from the class and has data attributes that contain values that are unique to it. If you created an instance of the Dog class, this instance (object) is no longer a blueprint. This instance now contains actual data like a dog's name (Fluffy, for example), who is brown, a beagle, and is two years old.

The code that you have worked with so far in this course has been in a single class, in a single .java file. This single class has contained a `main()` method that runs when the program starts. The class with the main() method for an application is known as the program's "driver class."

It is a standard procedure when using a class like the `Dog` class above, that the code for the class is put in a separate .java file. It also includes a name matching the name of the class (`Dog.java`).


CONCEPT TO KNOW

Note that the class is explicitly marked `public`.


HINT

The code for the application's driver class (in this case, DogExample) is then put in its own .java file.


TRY IT

Directions: Type in the following code in a file named DogExample.java:

```
public class DogExample {
  public static void main(String[] args) {
    // Call constructor using keyword new.
    // Assign new Dog object to variable myDog.
```

```
    // Class names starts with capital letter
    // but variable name starts with lowercase
    Dog myDog = new Dog("Fluffy", "Beagle", 2, "Brown");
    // Call "get" methods to access data.
    // All methods called via the Dog object created above
    System.out.println("Name: " + myDog.getName());
    System.out.println("Breed: " + myDog.getBreed());
    System.out.println("Age: " + myDog.getAge());
    System.out.println("Color: " + myDog.getColor());
    System.out.print(myDog.getName() + " says ");
    // Call bark() method
    myDog.bark();
  }
}
```

REFLECT

Running this code in Replit is a two-step process. Move on to the next Try It to continue.

TRY IT

Directions: Compile the Dog class using the following command in the console:

EXAMPLE

```
javac Dog.java
```

REFLECT

Assuming there are no errors in the code for the Dog class, this will create a binary Dog.class file containing the Java byte code for a Dog object.

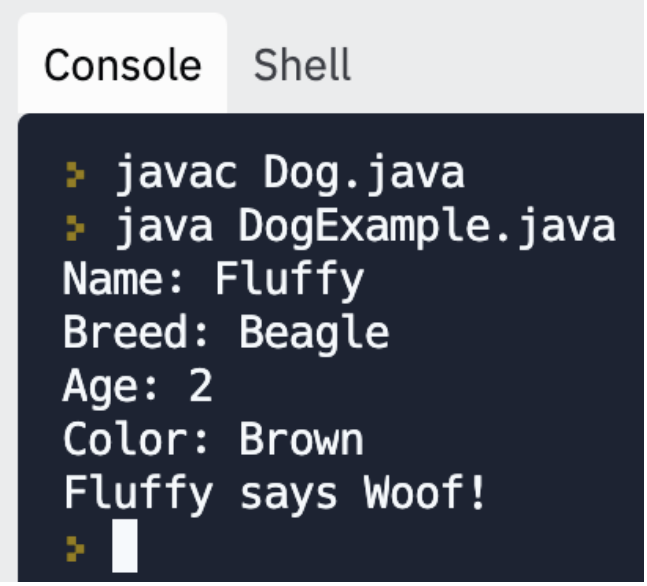
TRY IT

Directions: Once the Dog.java file has been compiled, run the driver class (and thus the program) using the Java command as usual:

EXAMPLE

```
java DogExample.java
```

The results should look like this:



TRY IT

Directions: After running the code, delete the Dog.class file by running the following command in the console:

EXAMPLE

```
rm Dog.class
```

While the Dog.java file is not directly executable, the presence of a .class file in the directory can prevent using the Java command to run a .java file with the same name.

REFLECT

So, what is the purpose of a class? Think of the class like a cookie-cutter and the objects created using the class are the cookies. You wouldn't likely put frosting on the cookie-cutter; you would put frosting on the cookies. And, you can put different frosting on each cookie. The frosting in this case represents the attributes or properties within the object. Each individual cookie with the frosting is considered as an object/instance of the class.

In the example of the Dog class, a template for creating dogs will not change. However, each instance created from it can. Because of this, you could create other instances like these:

EXAMPLE

```
Dog myDog1 = new Dog("Fluffy", "Beagle", 2, "Brown");
Dog myDog2 = new Dog("Mochi", "Mutt", 5, "White");
Dog myDog3 = new Dog("Wolfie", "Malteses", 10, "Black");
```

Of course, you could create instances like these, using an array:

EXAMPLE

```
Dog[] myDog = new Dog[3];
myDog[0] = new Dog("Fluffy", "Beagle", 2, "Brown");
myDog[1] = new Dog("Mochi", "Mutt", 5, "White");
```

```
myDog[2] = new Dog("Wolfie", "Malteses", 10, "Black");
```

In each one of these instances, it is not required to redefine the attributes and methods, since they were originally defined in the Dog class.

TERMS TO KNOW

Class
The class keyword defines the data and code that will make up each of the objects. When defining a class, it starts with the class keyword and is then followed by the name of the class and a colon.

Class Instantiation
The process of creating an instance of a class, which is called an object.

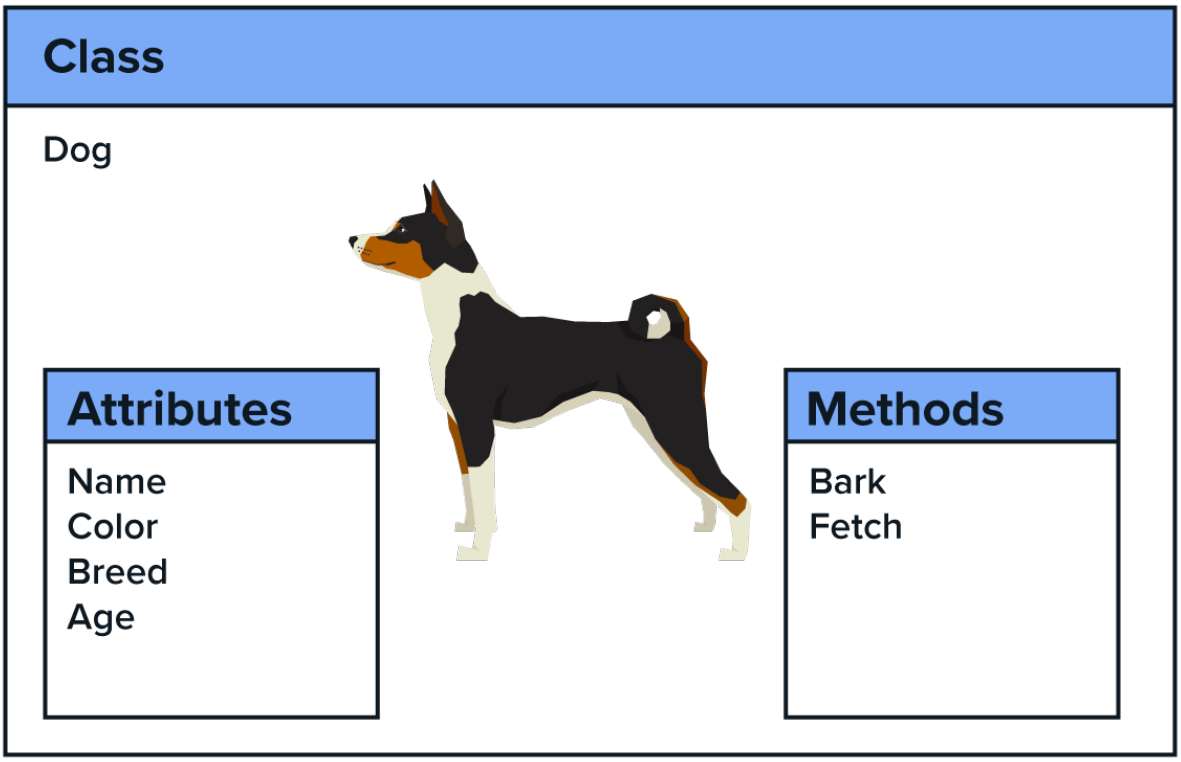
---

3. Attributes and Methods

In object-oriented programming, encapsulation is an important concept. It helps us describe the foundation that wraps attributes and methods that work on data in a single class. This allows us to add some restrictions to the access of attribute variables and methods directly. To prevent attributes from being directly changed, a class's attributes can be set to only be changed by its own class's method(s).

If you had a numeric grade attribute in a grade book object, you may want to force it to only be set between the numbers 0 and 100. A user should not be able to make a change to allow a negative number or have bonus points as part of the rules of encapsulation.

Back to our class example:



Now that you know that a class is a blueprint or template from which objects or instances are created, and that a class contains fields for attributes and methods, let's dive into those.

Attributes are sometimes also called properties of an object.**Attributes** are variables that are defined in a class template and are the characteristics or properties of an object. In the example, color, breed, and age will be properties of the class Dog. In a Java class, the attributes must always be explicitly declared `private` to ensure correct encapsulation. They should never be declared `public`, though the class itself should be declared `public`. The access modifier (`private`) should also never be left off.

Methods are specialized types of procedures that are directly associated with an object. They are called to perform a specific task by using the object name, a period, and then the method name. When individual objects are created from the class, these objects can call a method that was defined in the class. Since methods perform actions, a method might change or update an object's data (attribute) or possibly return some information. In our example, some behaviors or actions of the Dog class are methods, including barking and/or fetching.

The public methods that provide appropriate access to the data in the attributes have names that usually begin with "get" and are technically named accessor methods.

A Java class often includes a particular kind of method called a constructor. The constructor is used to initialize the attributes in an object when an instance of the class is created. The constructor is typically a public method. The constructor's name matches the name of the class in spelling and capitalization (this is important). The initial values for the class's attributes are passed via the parameters for the constructor.

Understanding the basics of a class is key to utilizing the OOP model.

3a. Example Class

Next, let's see a class and get to know some of the syntax aspects of it. Here is an example that you would break down.

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

In the first line of code, you could use the `class` keyword and give the class the access modifier `public`. A class that is defined as public can be used by any other code. By convention, a Java class name always begins with a capital letter. The name of the file in which the public class is defined must match the name of the class (with the .java file extension). The attributes and methods that make up the class are then defined within the pair of curly brackets that mark the beginning and end of the class.

The `PeopleCounter` class then declares a private attribute named `count`. The count attribute is declared as a long integer and is given an initial value of 0. The correct initialization of count is important because this allows the use of increment (++) operator later in the code.

Remember that correct data encapsulation requires that `count` be explicitly declared private.

There is no explicitly defined constructor, so the `PeopleCounter` class relies on the default constructor.

Next, you would declare a method and name it “anotherOne”. So, this class has one attribute (x) and one method (anotherOne).

This `anotherOne()` method is taking the attribute x and adding 1 to it each time the method is called.

3b. Using the Class

You have defined a class called `PeopleCounter`. What’s next? Does something happen if you run the code? No.

Just as the definition of a method does not cause the method code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a template (remember blueprints) indicating what data and code will be contained in each object of the class PeopleCounter. There are no executable lines of code so far in this program. Right now, just the class is defined with what attributes and methods you will want the instances of this class to use.

Remember the cookie-cutter analogy used earlier? You described a class as the cookie-cutter and what was created from it are cookies. Now, let’s take a look at how you can make a cookie from that cookie-cutter or, in this case, a “people” instance of the PeopleCounter class that was defined above. Remember that the PeopleCounter class needs to be defined in a file named PeopleCounter.java:

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

Before you can use the class, it needs to be compiled using this command:

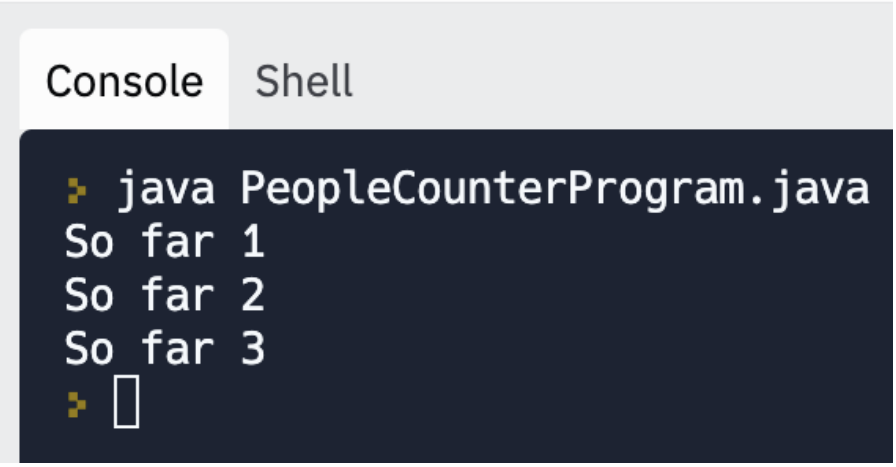
↗ EXAMPLE

```
javac PeopleCounter.java
```

The code in the application's `main()` method makes use of the `PeopleCounter` class. The driver class for the application is put into a file called PeopleCounterProgram.java.

```
class PeopleCounterProgram {

  public static void main(String[] args) {
    // Construct a PeopleCounter
    PeopleCounter pc = new PeopleCounter();
    // Count 3 people one by one
    pc.anotherOne();
    pc.anotherOne();
    pc.anotherOne();
  }
}
```

If you run this code now (using the command java PeopleCounterProgram.java), you would see the following output.



Continuing to break down this code, focus on this section of the code in the `main()` method in the driver class:

↱ EXAMPLE

```
// Construct a PeopleCounter
PeopleCounter pc = new PeopleCounter();
// Count 3 people one by one
pc.anotherOne();
pc.anotherOne();
pc.anotherOne();
```

As you continue to look at this program, consider the first executable line of code:

↱ EXAMPLE

```
PeopleCounter pc = new PeopleCounter();
```

This is where you tell Java to construct, or create, an object/instance of the class `PeopleCounter`. This line gives the object the variable name `pc`. An object can be given any valid variable name. It looks like a method call to the class itself. Java constructs the object with the right attributes (data) and methods (behaviors) and returns the object, which is then assigned to our variable `pc`.

CONCEPT TO KNOW

When the PeopleCounter class is used to construct an object, the variable `pc` is used to point to that object. Use the variable `pc` to access the methods to access the data and behaviors for that particular instance of the PeopleCounter class.

Each PeopleCounter object/instance contains within it a variable x and a method named `anotherOne()`. Call the `anotherOne()` method on this line:

↱ EXAMPLE

```
pc.anotherOne()
```

Notice that the standard method call structure is using the dot**operator** (.) before the method name.

TRY IT

Directions: Try typing in the PeopleCounter.java and PeopleCounterProgram.java files above. Remember that you need to compile the PeopleCounter class before you can use it by running this command:

EXAMPLE

```
javac PeopleCounter.java
```

To run the program that uses the PeopleCounter class, run this command:

EXAMPLE

```
java PeopleCounterProgram.java
```

The output should look like the screenshot above.

REFLECT

It is important to notice the relationship between the `PeopleCounter` class and the application's driver class. The code in the `PeopleCounter` class does not change, but the code in the`main()` or other method that uses `PeopleCounter` creates instances and uses them in ways that vary depending on what the code needs to do.

TERMS TO KNOW

Attributes
Attributes are data defined in the class template and are the characteristics or properties of an object.

Operator
The dot (.) operator connects the object (instance of a class) to the attributes and methods of that object.

SUMMARY

In this lesson, you started off with a **recap** of what you have learned so far. We discussed that if a program is to be more scalable, efficient, and especially more reusable, it requires you to move to a model of **object-oriented programming (OOP)**. You learned about the key **purpose of a class**, which is to set up this “template” and allow objects (instances) to be created from it. You identified the basics of the class. This includes **attributes**, or properties, and **methods**. Finally, you explored an **example class** and practiced **using the class** by creating a class called “PeopleCounter” that included one attribute and one method.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Attributes

Attributes are data defined in the class template and are the characteristics or properties of an object.

Class

The class keyword defines the data and code that will make up each of the objects. When defining a class, it starts with the class keyword and is then followed by the name of the class and a colon.

Class Instantiation

The process of creating an instance of a class, which is called an object.

Object-Oriented Programming

Object-oriented programming (or OOP for short) is a programming model that organizes the design of code by bundling objects.

Operator

The dot (.) operator connects the object (instance of a class) to the attributes and methods of that object.

Constructors

by Sophia

WHAT'S COVERED

In this lesson, you will explore initializing objects from classes and deleting objects when they are no longer needed. Specifically, this lesson covers:

1. Constructors

Each time an object is created from a class, through class instantiation, a Java constructor is invoked. The **constructor's** sole purpose is to initialize, or assign values to, the object's attributes. It is only used within a class. The Java constructor is similar to constructors in other programming languages like C# and C++.

In Java, there are two kinds of constructors. They are default and parameterized. The**default constructor** does not have any parameters. The default, or parameterless, constructor does the internal bookkeeping. This means that it allocates memory for an object and initializes the attributes to default values. An example of this is the use of 0 for an int or an empty String for a String attribute. If the default values for the attributes need to be changed, the code for the class will need to provide the methods to do so. This means that if a class uses a default constructor, the values in the attributes can be changed after the object has been created.

A **parameterized constructor**, as the name indicates, includes method parameters that are used to initialize the values of some or all of the attributes. If a class has a parameterized constructor, the default constructor without parameters is no longer automatically available, unless it is explicitly declared.

CONCEPT TO KNOW

A parameterized constructor is not required, but it is a good practice to use. To set attributes to specific values and not depend on system-defined default values, you would use this method.

Java has very specific requirements for the declaration of a constructor. As you have seen, the name of the constructor must always match the name and capitalization of the class name. Since a class name should always begin with a capital letter, the name of the constructor also should begin with a capital letter.

The second distinctive factor in the declaration of a constructor is that the constructor has no return type. This includes `void`.

CONCEPT TO KNOW

Before moving on, it is important to acknowledge another kind of constructor besides a parameterized constructor. The simplest form of a constructor is the default constructor that takes no parameters. In fact, if there is no parameterized constructor in a class, the default constructor does not need to be declared or defined. The compiler just generates the code needed to allocate memory and set up the class.

Here is a sample class (`PeopleCounter`) that does not have an explicitly defined constructor. It relies on the default constructor.

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

To start the PeopleCounter with a count value other than 0, you could create a parameterized constructor like the one shown in this version of the class:

```
public class PeopleCounter {
  private long count = 0;

  public PeopleCounter(long count) {
     this.count = count;
  }

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

When defining a parameterized constructor like this, though the default (parameterless) constructor is no longer available, include it along with the parameterized constructor.

CONCEPT TO KNOW

There can be more than one constructor in a class, as long as they take different types or numbers of parameters (as with any overloaded method):

```
public class PeopleCounter {
  private long count = 0;

  // Default parameterless constructor
  // Attribute count will have the initial value declared above
  public PeopleCounter() {}

  // Parameterized constructor
  // The count passed in will override initial value of count
  public PeopleCounter(long count) {
     this.count = count;
  }

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

You may recognize the syntax used to define constructors as it's somewhat similar to the creation of any method. First, though, it bears repeating that a constructor has no return type. Then comes the name of the method, which must match the name of the class in spelling and capitalization. Since the class name should start with the capital letter, the constructor's name must start with a capital letter. This is then followed by parentheses that may or may not contain parameters.

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

Here, the attribute is assigned a value of x to 0. This means that it will always start with the value 0. To have a different initial value for x would require using a constructor with an int parameter to pass in a different default value. Having a class with attributes with default value(s) is only unique to that class. So, for our example PeopleCounter class, the variable `x` is only available to the PeopleCounter class. Let's create a new class called "User" that can be used for an application/program that creates user login information.

First, you would create a class named `User`. This class contains private `String` attributes named `userName` and `password`. The values of these attributes are set via the `User()` constructor, which has two `String` parameters. In this case, a parameterless constructor (the default constructor) would not be very helpful because there are not likely to be good default values for the attributes in the `User` class.

```
public class User {
  private String userName;
  private String password;

  public User(String userName, String password) {
    this.userName = userName;
    this.password = password;
  }

  // Allow read-only access to user name
  public String getUserName() {
    return userName;
  }

  // There is no direct access to the password.
  // In real code there would be a method check
  // for a match with the stored password.
}
```

This code can't do a lot, but it can create a `User` object and also has a method to return the user name, `getUserName()`.

The method has been started; however, it does not define what comes next. At this point, if the code was executed, the constructor, `User()`, executes and it would set the values for the `userName` and `password` attributes. The constructor's parameters would hold whatever argument data that is being passed.

HINT

Remember that this is meant to be a `User` class about a user and looks to have attributes that reflect that kind of data. If you wanted each user to have a `userName` attribute that contains the user's username along with a password, the attribute would store the password.

You can define and populate those attributes by doing the following.

```
public class User {
  private String userName;
  private String password;

  public User(String userName, String password) {
    this.userName = userName;
    this.password = password;
  }

  // Allow read-only access to user name
  public String getUserName() {
    return userName;
  }

  // There is no direct access to the password.
  // In real code there would be a method check
  // for a match with the stored password.
}
```

The first indented line of the constructor sets the attribute named `userName` for the new instance to the value passed via the `userName` parameter. Note how the attribute `userName` is accessed via `this.userName` to distinguish it from the `userName` parameter. The keyword `this` refers to the current object being constructed or used. The second line in the body of the constructor (inside the curly brackets that mark the code executed when the constructor is called) sets the value of `this.password` to the `String` passed via the `password` parameter. A couple of conventions were used when setting up the new class. They include:

- Class names should be capitalized.
- Attributes must be declared private and use the same standard as used for creating variable names.

You will now focus on creating an instance of a class, using what you know regarding how a constructor is used to create an object.

TERMS TO KNOW

Constructor

The constructor's sole purpose is to initialize an object and, if it is a parameterized constructor, to set the object's attributes to specific (non-default) values. A constructor only exists inside a class and is only used to set up a new object based on the class's template.

Default Constructor

The default constructor does not have any parameters. It allocates memory for an object and initializes the attributes to default values.

Parameterized Constructor

Includes method parameters that are used to initialize the values of some or all of the attributes.

2. Creating an Object

Now that the class is created, you will learn how to create instances of it. The format would look something like this in relation to the defined User class:

↗ EXAMPLE

```
User userObject = new User("userNameString", "passwordString");;
```

To be able to use the User class shown above, you would need to compile it first, using this command:

↗ EXAMPLE

```
javac User.java
```

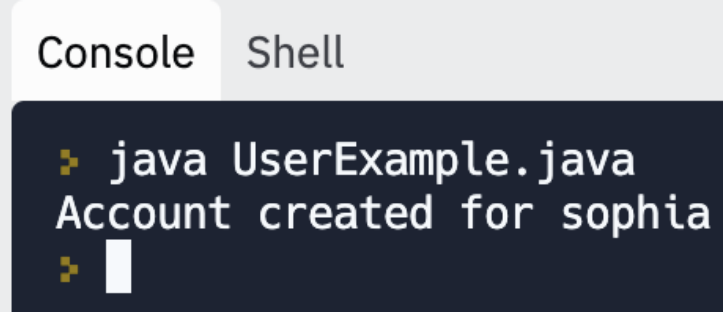
Once the User class has been compiled, you can use the class in an application with the class name UserExample (saved in a file named UserExample.java). You would need to replace the instanceName with the variable name that you choose and the usernameValue and passwordValue with their respective argument values. Let's say that you want to create a User object called "account" with the username "sophia" and the password as "myPass":

```
class UserExample {
  public static void main(String[] args) {
    User account = new User("sophia", "mypass");
  }
}
```

If the code is run as is, it will create an instance of theUser class passing the arguments "sophia" and "mypass" as argument values. It will also assign that returned object to the variable "account". It may help to see more details by printing the username out:

```
class UserExample {
  public static void main(String[] args) {
    User account = new User("sophia", "mypass");
    System.out.println("Account created for " + account.getUserName());
  }
}
```

The output screen should look like this:



BIG IDEA

It is important to note that you will see the term “object” and “instance” used through these tutorials. They mean the same thing. It’s important to note that an instance of an object is just a way to hold information about an element that’s similar to other elements of the same class.

The terms “properties” and “attributes” mean the same thing as well. The element's attributes may be unique to other instances. For example, in an application, it would make sense for each of the users to have unique usernames. However, some users may have the same value for their password.

TRY IT

Directions: Try creating an instance/object from a class. Start with our example above and see if you can modify it to something unique for yourself.

BRAINSTORM

What other sorts of data might the User class contain about the user? Which data types would you use for these new attributes? (When working in Java, it's always important to be mindful of data types.)

SUMMARY

In this lesson, you learned about using **constructors**. A constructor is a kind of special method which allows you to initialize and set up instance variables to be used within an object. You also learned how to **create a new object** (instance) from our class, access the variables, and determine the type of the object.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Constructor

The constructor's sole purpose is to initialize an object and, if it is a parameterized constructor, to set the object’s attributes to specific (non-default) values. A constructor only exists inside a class and is only used to set up a new object based on the class's template.

Default Constructor

The default constructor does not have any parameters. It allocates memory for an object and initializes the

attributes to default values.

Parameterized Constructor

Includes method parameters that are used to initialize the values of some or all of the attributes.

Object Attributes

by Sophia

WHAT'S COVERED

In this lesson, you will explore how to set and change values of object attributes and create default values. Specifically, this lesson covers:

1. Setting Values

In the prior tutorial, you created a `User` class with two attributes. It's important to remember that the attributes were declared as private in accordance with the principle of data encapsulation (the values in the attributes should not be directly read or modified from outside of the class; access is provided via appropriate public methods). Once you used the constructor to set the values, you did not make any changes to the attributes. Let's revisit that class:

```
public class User {
  private String userName;
  private String password;

  public User(String userName, String password) {
    this.userName = userName;
    this.password = password;
  }

  // Allow read-only access to user name
  public String getUserName() {
    return userName;
  }

  // There is no direct access to the password.
  // In real code there would be a method check
  // for a match with the stored password.
}
```

TRY IT

Directions: If you haven't already done so, go ahead and add the `User` class to Replit:

Continuing to use the example from the last tutorial, you created "account" as the instance of `User` class. Here is that full code again from the previous tutorial. Recall that there were two files: User.java (containing the `User` class) and UserExample.java (containing the UserExample class—the **driver class**—with the application's main() method).

Here is the code for the `UserExample` class:

```
class UserExample {
  public static void main(String[] args) {
    User account = new User("sophia", "mypass");
    System.out.println("Account created for " + account.getUserName());
  }
}
```

TRY IT

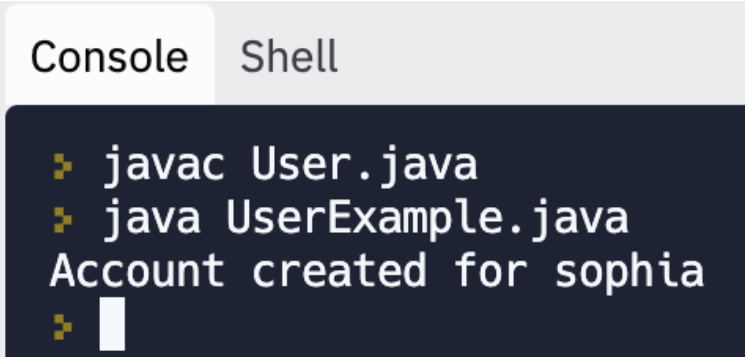
Directions: Remember that the User class needs to be compiled to be useful. Run the following command:

```
javac User.java
```

After compiling the class, you can run the application (via the driver class) using this command:

```
java UserExample.java
```

The output screen would display like this:



REFLECT

While the User class is quite simple, the way that it is compiled as a separate class provides a bit of perspective on how a Java class can be compiled and used in different programs. Just as the driver class, UserExample, makes use of the User class, other classes could do the same (with appropriate access to the .class file, but that aspect will not be addressed here). Think about how such classes allow building a large program out of many pieces.

Remember, with the new instance account, the call to the constructor is passing the arguments of “sophia” as the username and “mypass” as the password. This instance assigns these values to the parameters set in the attributes of the User class.

TERM TO KNOW

Driver Class
The class in an application that contains the main() method.

---

2. Setting Default Attribute Values

Up to this point, the focus has been on setting values via constructor parameters. However, it is not necessary to

pass in the value for every attribute when a new instance of an object is created—for example, if the value is always a set value when an attribute is declared, as in the PeopleCounter class that you have seen previously. In this class, the count attribute is always initialized to 0:

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

Let's expand on the User class to see how you can use attributes whose values are given default values. Perhaps now you would want to:

- Determine if the user's account is active.
- Determine when the user joined.

This provides three more attributes that you want to set. However, in all of these cases, there's no need to manually pass these values in, since:

- The account should automatically have the activity flag set to True.
- When a user is created, the created date should simply be set to the current date.

The last attribute deals with time. Anytime that you will do anything with dates and times, you will have to import one of the classes for working with dates provided by Java's standard libraries.


CONCEPT TO KNOW

In a number of previous examples, you imported classes such as java.util.Scanner for use in our code. Here you will need to import the datetime module to obtain this attribute's value. Importing classes from the library will be discussed in a later tutorial. Just know for now that importing of library classes allows you to incorporate additional functionality outside of what's available by default in Java. To import the library class, you will use the import command at the top of the program before the class is defined.

The class that needs to be imported here is java.time.LocalDate using this import statement:

```
import java.time.LocalDate;
```

```
public class UserAccount {
  private String userName;
  private String password;
  private LocalDate dateJoined = LocalDate.now();
  private boolean activeUser = true;

  public UserAccount(String userName, String password) {
    this.userName = userName;
    this.password = password;
  }
}
```

HINT

Note that in addition to setting these values when the attributes in the class are declared, you can set their initial values in the parameterized constructor, even though no values for these were passed:

```
public class UserAccount {
  private String userName;
  private String password;
  private LocalDate dateJoined;
  private boolean activeUser;

  public UserAccount(String userName, String password) {
    this.userName = userName;
    this.password = password;
    //this.dateJoined = LocalDate.now();
    //activeUser = true;
  }
```

TRY IT

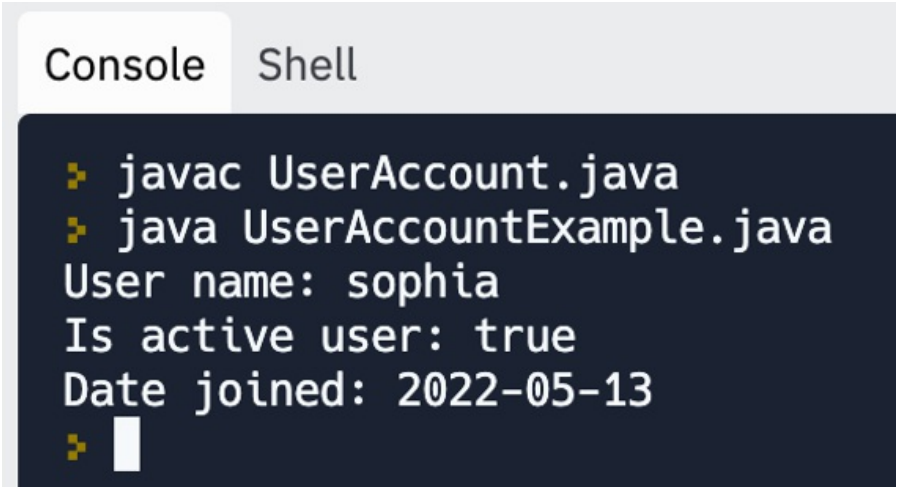
Directions: In Replit, add the new attributes and default values to your`User` class. Setting the values for the new attribute `activeUser` should be familiar. You would simply set the`activeUser` variable to true for every instance of the `UserAccount` class that is created.

The `dateJoined` attribute is using the `java.time.LocalDate` to get today's date. It is set using the `now()` method of the `LocalDate` class. Again, these are an object and method provided by the standard Java libraries. Once the class is imported, you would have access to use anything that class has prebuilt in it.

TRY IT

Directions: Let's add an output for each of these new attributes to our program and use them.

The output of each attribute of the account instance would look like this:



REFLECT

Notice that when creating our instance, nothing has changed, as you would just be passing in the username and password. All of the other attributes are automatically set.

SUMMARY

In this lesson, you learned how to **set values to attributes** and change those values after the instances have been created. You also learned how to **set default values for attributes**. Finally, you briefly looked at importing the java.time.LocalDate class to set the default value using the current date.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Driver Class

The class in an application that contains the main() method.

Object Methods

by Sophia

WHAT'S COVERED

In this lesson, you will explore how to create and use an object's method in a larger program. Specifically, this lesson covers:

1. Accessor Methods

Accessor methods are public methods in a class that provide read access to the data in the class's attributes (to the extent that read access is appropriate). Depending on the design requirements, an accessor method may just return the data from the attribute, or it may carry out any processing or conversion that is needed.

The convention in Java is that accessor methods should begin with the word "get" and then indicate which attribute's data is returned. For instance, the `UserAccount` code below has a `getUserName()` method that returns the username and a `getDateJoined()` method that returns the date the user joined. In the case of an accessor method that returns a boolean value, though, the name usually begins with the word "is." The method in the `UserAccount` class that returns the boolean indicating whether the user is active or not is called `isActiveUser()`, as demonstrated below:

```
import java.time.LocalDate;

public class UserAccount {
  private String userName;
  private String password;
  private LocalDate dateJoined;
  private boolean activeUser;

  public UserAccount(String userName, String password) {
    this.userName = userName;
    this.password = password;
    this.dateJoined = LocalDate.now();
    this.activeUser = true;
  }

  // Allow read-only access to user name
  public String getUserName() {
    return userName;
  }

  // Allow read-only access to date joined
  public LocalDate getDateJoined() {
    return dateJoined;
```

```
  }

  // Allow activeUser to be read & set (can change)
  public boolean isActiveUser() {
    return activeUser;
  }

  public void setActiveUser(boolean activeUser) {
    this.activeUser = activeUser;
  }
}
```

To define what the methods are in the class, use the same syntax that has already been seen:

↗ EXAMPLE

```
returnType <methodame>(self, <parameter(s)>) {
   /* Statements */
}
```

TERM TO KNOW

Accessor Method ("Getter" Method)
A public method that reads and returns the value in an attribute.

---

2. Mutator Methods

Mutator methods (also called "setter methods") are methods that are implemented to allow data in the class's attributes to be changed. Mutator methods are only provided if the value of the attributes is allowed to be changed after the object has been constructed. A mutator method requires a parameter for passing in the new value of the attribute.

In the `UserAccount` class that you have been looking at, there is a mutator method,`setActiveUser()`. The method is defined like this in the class with a `boolean` parameter that allows setting the value of the `activeUser` attribute:

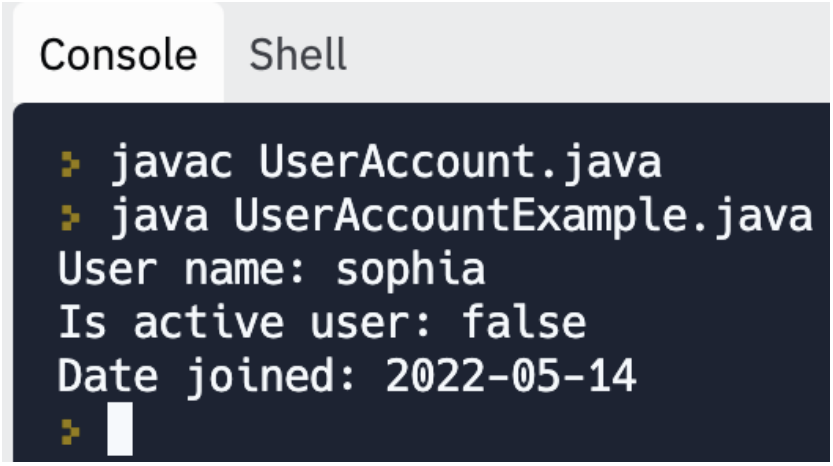
```
public void setActiveUser(boolean activeUser) {
   this.activeUser = activeUser;
}
```

To create an account with the username "sophia" and the password "mypass", but then set the activeUser attribute to false rather than the default true, you could write code like this in the application's driver class:

```
public class UserAccountExample {
  public static void main(String[] args) {
    UserAccount account = new UserAccount("sophia", "mypass");
    account.setActiveUser(false);
    System.out.println("User name: " + account.getUserName());
    System.out.println("Is active user: " + account.isActiveUser());
```

```
    System.out.println("Date joined: " + account.getDateJoined());
  }
}
```

If you type this code into a file named UserAccountExample.java and run it, the output should look like this:



REFLECT

As the output shows, the user's active status has been set to false (rather than the default true).

TERM TO KNOW

Mutator Method ("Setter" Method)
A public method that changes the value in an attribute.

---

3. Other Methods

In addition to the constructors, accessor methods, and mutator methods that have been discussed, a class can include other methods that carry out relevant operations on the data in the class's attributes. In an upcoming tutorial about debugging classes, you will look into a `toString()` method, but for the moment, let's look at another method that involves passing values via method parameters and getting back a value returned by the method. Here is a method that checks if the username and password match the values for an account:

EXAMPLE

```
public boolean checkCredentials(String userName, String password) {
   return this.userName.equals(userName) && this.password.equals(password);
}
```

Note that when comparing two String values for equality, the code needs to use the equals() method provided by the String class.

TRY IT

Directions: Try adding this method to the UserAccount class and then calling it from the `main()` in the application class.

BRAINSTORM

How could you use this to produce code that indicates if the login was successful using a username and password entered?

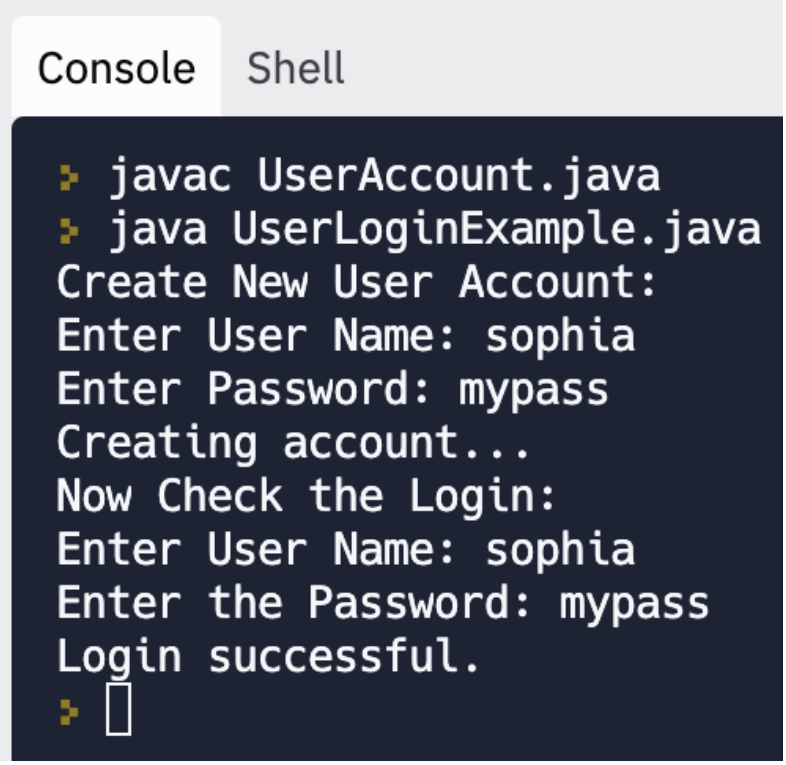
Let's write a small program that prompts the user to enter a username and password. The program then creates a new UserAccount object, prompts the user again to enter a username and password, and then checks to see if the login is correct. You have not read any user input for a while, so let's use a Scanner to read input in the console.

TRY IT

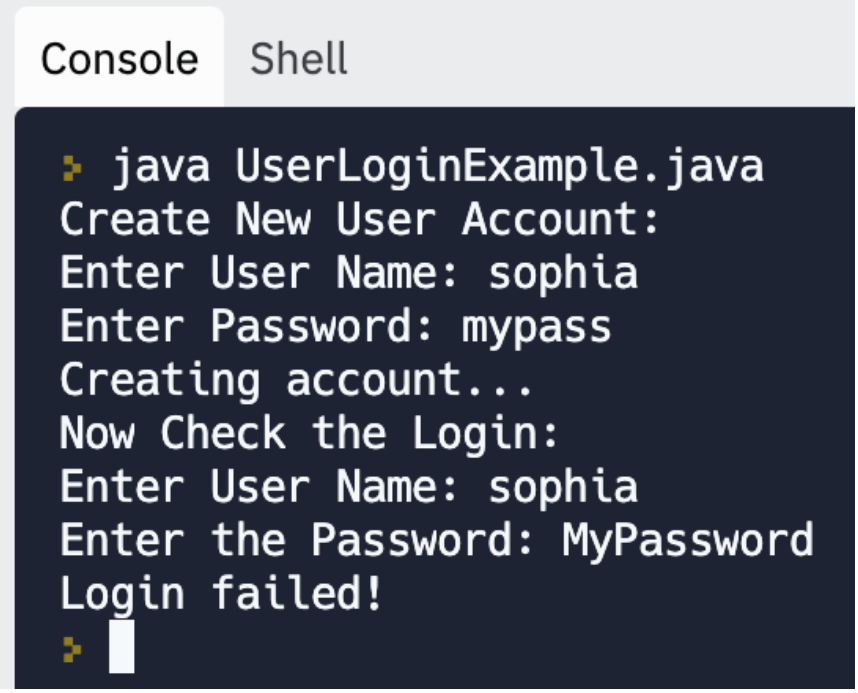
Directions: Type the following code into a file named UserLoginExample.java:

```
import java.util.Scanner;

class UserLoginExample {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("Create New User Account: ");
    System.out.print("Enter User Name: ") ;
    String user = input.nextLine();
    System.out.print("Enter Password: ");
    String passwd = input.nextLine();

    System.out.println("Creating account...");
    UserAccount acct = new UserAccount(user, passwd);

    System.out.println("Now Check the Login: ");
    System.out.print("Enter User Name: ");
    String userToCheck = input.nextLine();
    System.out.print("Enter the Password: ");
    String passwordToCheck = input.nextLine();

    boolean result = acct.checkCredentials(userToCheck, passwordToCheck);
    if(result) {
      System.out.println("Login successful.");
    }
    else {
      System.out.println("Login failed!");
    }
  }
}
```

Be sure to recompile the UserAccount class and then run the`UserLoginExample` program. The result of a valid login should look like this (keep in mind that both the username and password are case-sensitive):



A failed login attempt should look like this:



REFLECT

The code above shows how to write a method that checks the username and password stored in the object in a case-sensitive manner. While passwords are typically case-sensitive, think about how the `toUpperCase()` or `toLowerCase()` method provided by the `String` class could be used to check the username in a non-case-sensitive manner.

THINK ABOUT IT

In looking at the results, were they what you expected to see?

SUMMARY

In this lesson, you learned how to create methods that provide access to the data in attributes in a class (called **accessor methods**, also called "getter" methods). You have also seen how to**create mutator** or "setter" **methods** that allow the values of attributes to be changed after objects have been created. Finally, you have seen an example of **another method** that carries out a useful process using the data in an object.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Accessor Method ("Getter" Method)
A public method that reads and returns the value in an attribute.

Mutator Method ("Setter" Method)
A public method that changes the value in an attribute.

Instances of Objects

by Sophia

WHAT'S COVERED

In this lesson, you will learn about instantiating objects when using Java. A class defines a data type, and any object instantiated from a class has the type of that class. This allows programmers to use custom classes and the objects created from them—like another variable—to pass data from one method to another. Specifically, this lesson covers:

1. Object Scope

To illustrate the points made in this tutorial, let's use the relatively simple `PeopleCounter` class. It is defined like this, in a file named `PeopleCounter.java`:

```
public class PeopleCounter {
  private long count = 0;

  public void anotherOne() {
    count++;
    System.out.println("So far " + count);
  }
}
```

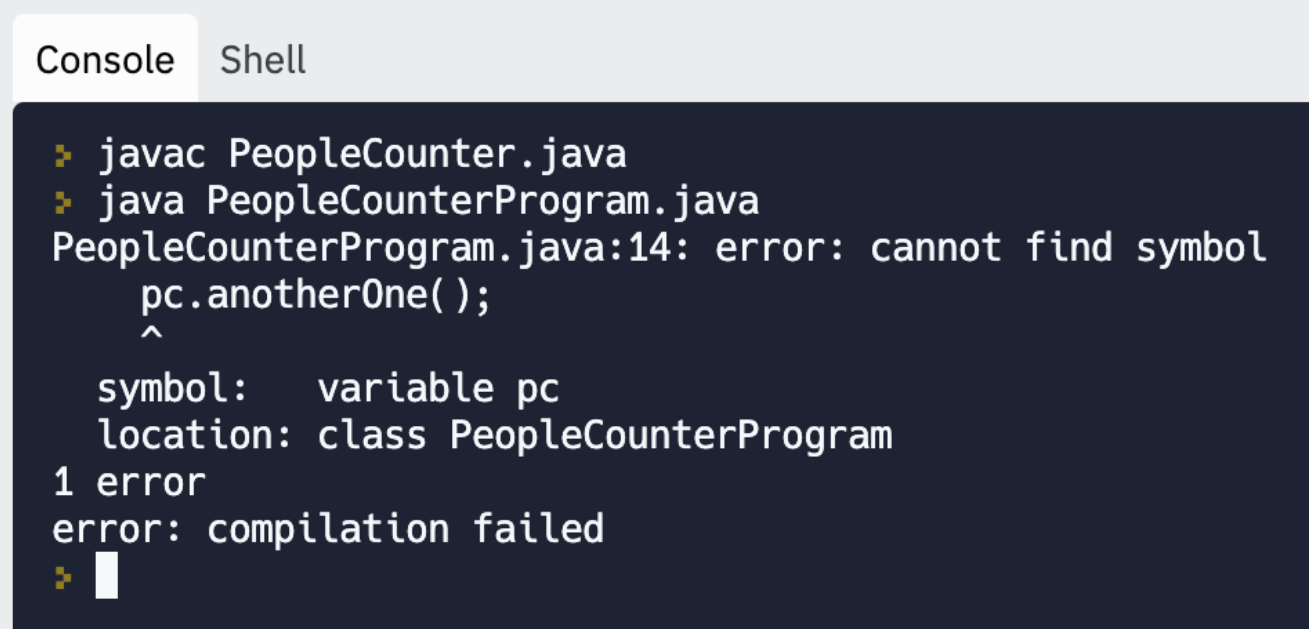
As with any variable, an object is local in scope (visibility). The **scope** of a variable or object refers to where it can be used in a program. If it is declared inside a method, the variable is in scope only in the body of that method. If it is declared at the class level (inside the class but not inside of a specific method), it is in scope anywhere in the class (in the application's driver class, such a variable needs to be declared as static, if it is to be accessed from within `main()`). Here is an example where a `PeopleCounter` object is constructed in the `main()` method:

```
class PeopleCounterProgram {

  public static void main(String[] args) {
    // Construct a PeopleCounter
    PeopleCounter pc = new PeopleCounter();
    // Count 3 people one by one
    pc.anotherOne();
    pc.anotherOne();
    pc.anotherOne();
  }
}
```

If there were a second method in the `PeopleCounterProgram`, the code in that method would not be able to access the instance named `pc` in the code above:

```
class PeopleCounterProgram {

  public static void main(String[] args) {
    // Construct a PeopleCounter
    PeopleCounter pc = new PeopleCounter();
    // Count 3 people one by one
    pc.anotherOne();
    pc.anotherOne();
    pc.anotherOne();
    anotherMethod();
  }

  public static void anotherMethod() {
    // This will not work - the instance pc is not in scope in this method
    pc.anotherOne();
  }
}
```

Trying to run this version of the program produces an error:



If the object needs to be used by another method, it needs to be passed as a parameter.

TERM TO KNOW

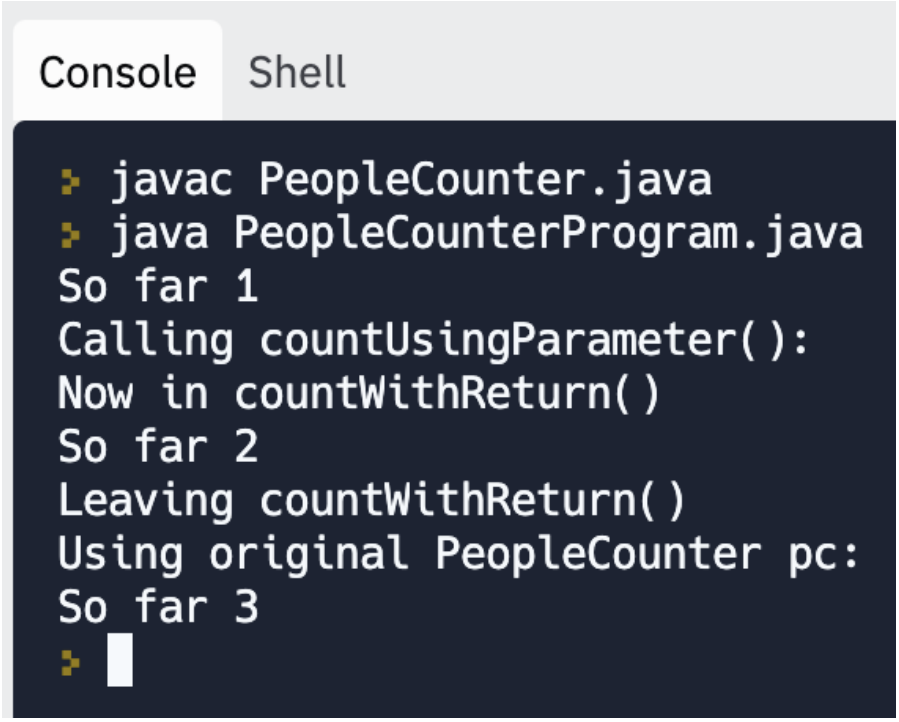
Scope
The scope of a variable or object refers to where it can be used in a program.

2. Passing Objects as Parameters

Java allows the programmer to pass objects as parameters to methods, assuming that the method being called has a signature that allows the specific type of object for a given parameter. Here is an example program that includes a method called:

```
class PeopleCounterProgram {

  public static void main(String[] args) {
    // Construct a PeopleCounter
    PeopleCounter pc = new PeopleCounter();
    // Count one
    pc.anotherOne();
    System.out.println("Calling countUsingParameter(): ");
    countUsingParameter(pc);
    System.out.println("Using original PeopleCounter pc: ");
    pc.anotherOne();
  }

  public static void countUsingParameter(PeopleCounter counter) {
    System.out.println("Now in countWithReturn()");
    counter.anotherOne();
    System.out.println("Leaving countWithReturn()");
  }
}
```

The output from this code allows you to see the flow through the code and where the PeopleCounter object is being used:



The fact that the count is in the correct sequence, even though the object is passed as a parameter, shows that the same object is being used, both in the body of the `main()` method and in the `countUsingParameter()` method.

---

3. Returning Objects from Methods

It is also possible to have an object returned from a method. Here is another program that uses the `PeopleCounter`. In this case, the `getPeopleCounter()` method creates a `PeopleCounter` object and returns it

to the caller:

```
class PeopleCounterProgram {
  public static void main(String[] args) {
    System.out.println("Getting PeopleCounter...");
    PeopleCounter counter = getPeopleCounter();
    System.out.println("Using new PeopleCounter...");
    counter.anotherOne();
  }

  public static PeopleCounter getPeopleCounter() {
    System.out.println("Returning new PeopleCounter...");
    return new PeopleCounter();
  }
}
```

HINT

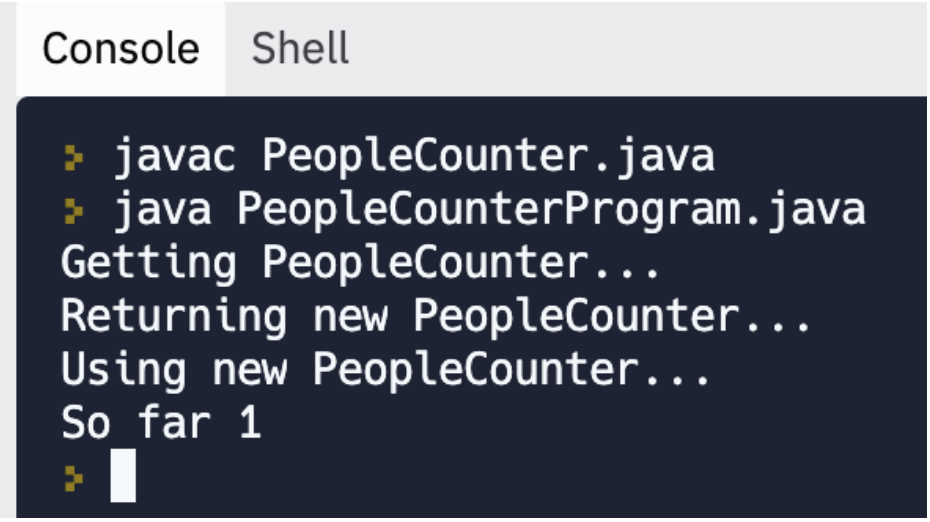
Note that the line that calls the `getPeopleCounter()` method also declares a `PeopleCounter` variable to hold the object returned by the method.

The caller is responsible for providing a variable to hold the resulting object:

→ EXAMPLE

```
PeopleCounter counter = getPeopleCounter();
```

The results from running this program should look like this:



SUMMARY

In this lesson, you learned how **objects** created from classes can be used like any other kind of variable to pass data between methods, noting that as with any variable, an object is local in **scope**. You also learned that objects that are instances of a class can be used as method parameters and return types; Java allows the programmer to **pass objects as parameters** to methods, as well as **return objects from methods**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Scope

The scope of a variable or object refers to where it can be used in a program.

Debugging Classes

by Sophia

WHAT'S COVERED

In this lesson, you will learn about debugging classes when using Java. Specifically, this lesson covers:

1. Implementing toString() to Check an Object's State

In previous lessons about debugging arrays and collections, you have seen how the `toString()` methods provided in Java's standard libraries can facilitate tracking down and fixing problems. Using the `toString()` method with objects can be helpful in debugging, by allowing us to see the values currently stored in the object (its state).

All Java objects have a default `toString()` method, but this default version may not be very helpful with classes that you create. Let's take a look at the `UserAccount` class that you have worked with previously.

Here is the code (which needs to be in a file in Replit called `UserAccount.java`):

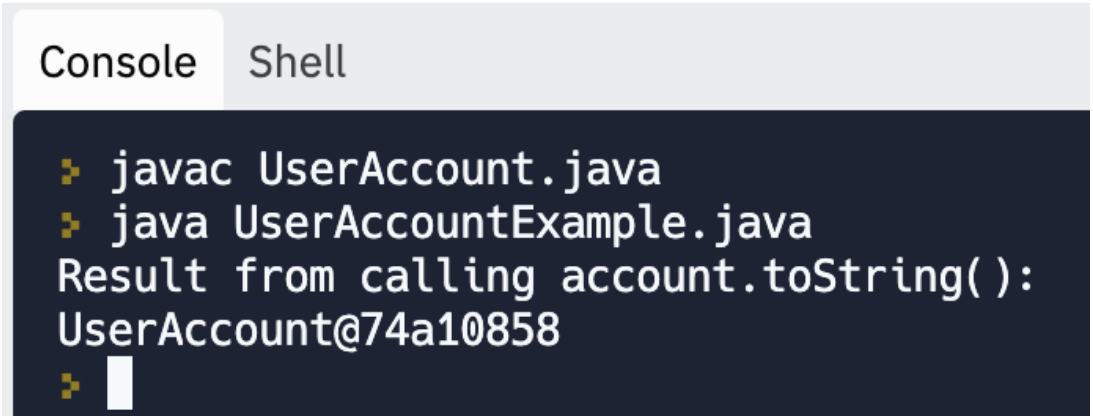
```
import java.time.LocalDate;
public class UserAccount {
  private String userName;
  private String password;
  private LocalDate dateJoined;
  private boolean activeUser;

  public UserAccount(String userName, String password) {
    this.userName = userName;
    this.password = password;
    this.dateJoined = LocalDate.now();
    this.activeUser = true;
  }

  // Allow read-only access to user name
  public String getUserName() {
    return userName;
  }

  // Allow read-only access to date joined
  public LocalDate getDateJoined() {
    return dateJoined;
  }
```

```
  // Allow activeUser to be read & set (can change)
  public boolean isActiveUser() {
    return activeUser;
  }

  public void setActiveUser(boolean activeUser) {
    this.activeUser = activeUser;
  }
}
```

Here is a simple driver class that makes use of the UserAccount class. Type this code into a file named EmployeeExample.java:

```
public class UserAccountExample {
  public static void main(String[] args) {
    UserAccount account = new UserAccount("Sophia2", "TestTest123");
    System.out.println("Result from calling account.toString():");
    System.out.println(account.toString());
  }
}
```

Compile the UserAccount class and then run the UserAccount example program. The results should look like this:



HINT

The exact **hexadecimal** (base 16) number in the last line of the output will vary from computer to computer, but that doesn't matter because it doesn't tell a human reader anything useful.

Here is the code for a toString() method to be added to the UserAccount class. This code produces a String with the names of the attributes and their current values:

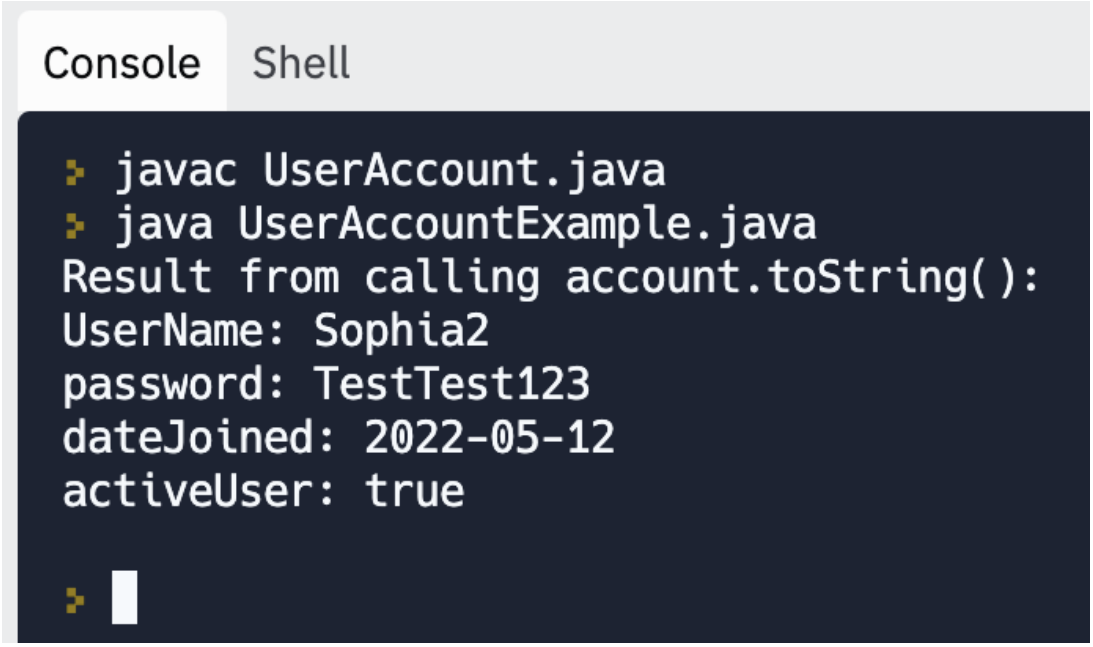
```
public String toString() {
  String state = "UserName: " + userName + "\n";
  state += "password: " + password + "\n";
  state += "dateJoined: " + dateJoined + "\n";
  state += "activeUser: " + activeUser + "\n";
  return state;
}
```

After adding this code to the UserAccount.java file, the code will need to be recompiled:

↗ EXAMPLE

```
javac UserAccount.java
```

The results of running the UserAccountExample program with this version of the `UserAccount` class produces this output:



This version of `toString()` provides a much more useful representation of the data in the object. The format of the date (year - month - day) may be a bit unexpected, but it is the default format. You can use another Java format class to put the date into a more familiar format. First, the code (in UserAccount.java) needs to add an import near the top of the file (with the other import for `java.time.LocalDate`):

↗ EXAMPLE

```
import java.time.format.DateTimeFormatter;
```

Then, in the body of the toString() method, add this statement to create the desired format:

↗ EXAMPLE

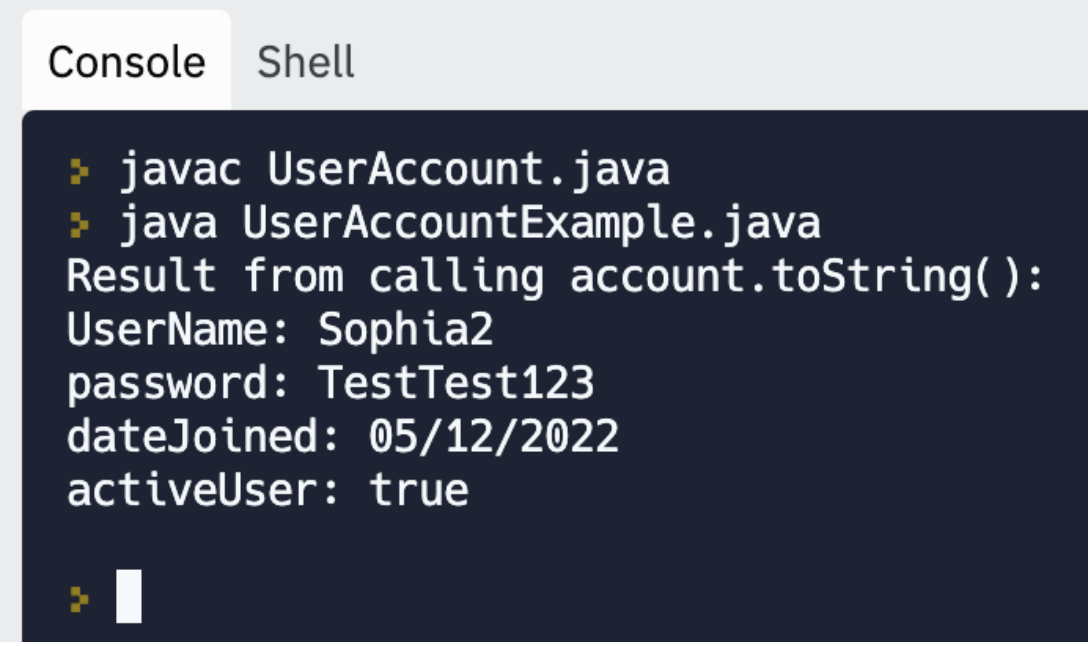
```
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("MM/dd/YYYY");
```

The method used to create a `DateTimeFormatter` is a bit different from how the constructor for `DecimalFormat` is used, but as you can hopefully make out, the code above formats the date as month/day/year. Calling `dateFormat`'s `format()` method outputs the date in the specified format. The `toString()` method should now look like this:

```
public String toString() {
  DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("MM/dd/YYYY");
  String state = "UserName: " + userName + "\n";
```

```
  state += "password: " + password + "\n";
  state += "dateJoined: " + dateFormat.format(dateJoined) + "\n";
  state += "activeUser: " + activeUser + "\n";
  return state;
}
```

The output from the program should now look like this:



Let's now look at an example of how this method might be useful. The constructor for the UserAccount class includes a line that sets the `activeUser` attribute to true, but let's imagine the programmer has made an error and the constructor looks like this instead:

↗ EXAMPLE

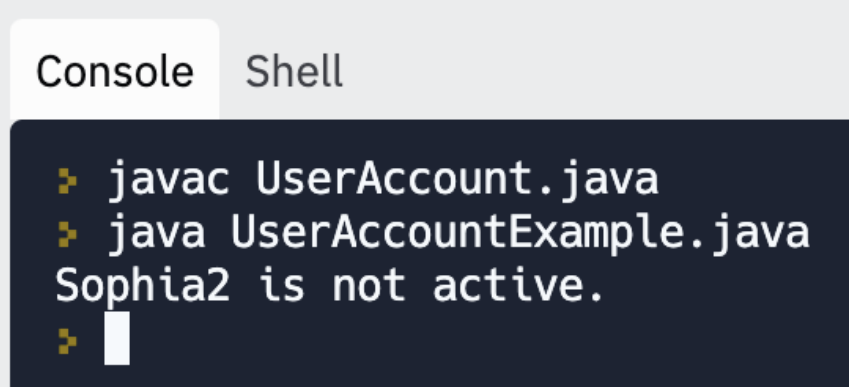
```
public UserAccount(String userName, String password) {
  this.userName = userName;
  this.password = password;
  this.dateJoined = LocalDate.now();
}
```

Since activeUser is a boolean, the default value is false, if it is not initialized to true. If the code in the application's `main()` were like this, the result would not be what you expected for a newly created account:

```
public class UserAccountExample {
  public static void main(String[] args) {
    UserAccount account = new UserAccount("Sophia2", "TestTest123");
    if(account.isActiveUser()) {
      System.out.println(account.getUserName() + " is active.");
    }
    else {
      System.out.println(account.getUserName() + " is not active.");
    }
  }
```

}

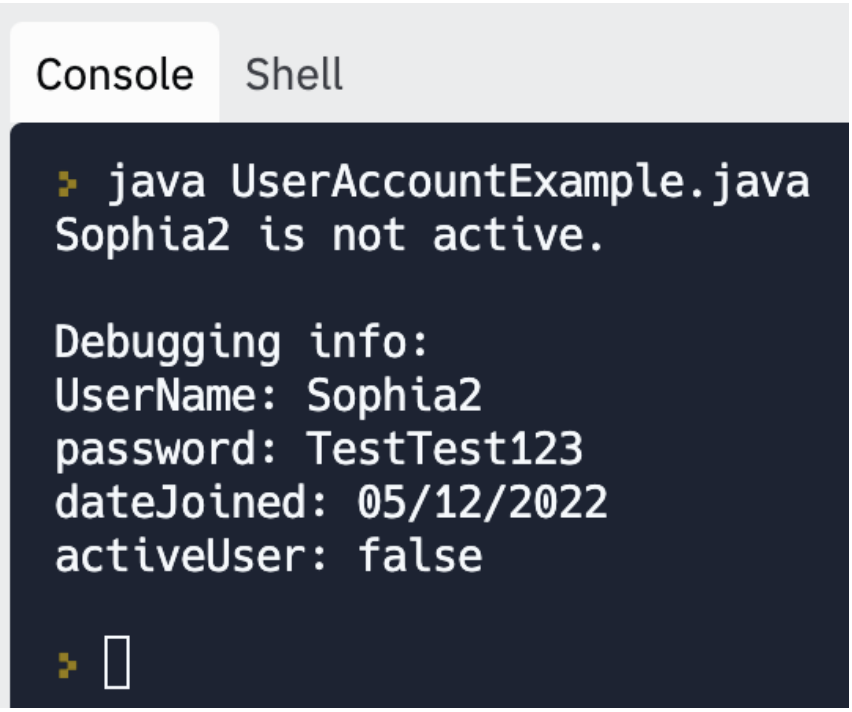
The results from running this code look like this:



This is not what is expected, so adding a call to the UserAccount object's toString() method can help us figure out the cause of the trouble:

```
public class UserAccountExample {
  public static void main(String[] args) {
    UserAccount account = new UserAccount("Sophia2", "TestTest123");
    if(account.isActiveUser()) {
      System.out.println(account.getUserName() + " is active.");
    }
    else {
      System.out.println(account.getUserName() + " is not active.");
    }
    // Added for debugging
    System.out.println("\nDebugging info:\n" + account.toString());
  }
}
```

Running the program now produces the following output:



The data displayed by the toString() method shows that the activeUser is set to false (the default value for a boolean variable in Java) since it is not assigned the value true by the constructor.

Putting back the line:

↱ EXAMPLE

```
this.activeUser = true;
```

This will get the program working as intended.

TERM TO KNOW

Hexadecimal
A value expressed in a base 16 number system (rather than base 10).

SUMMARY

In this lesson, you have learned how to **implement a toString()** method in a class that you have created, **to check an object's state**. As you saw when working with arrays and collections, the toString() method can be very helpful in tracking down and fixing problems that arise through programming mistakes and other errors.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Hexadecimal
A value expressed in a base 16 number system (rather than base 10).

The Employee Class Program

by Sophia

WHAT'S COVERED

In this lesson, you will explore the creation of a completed class that contains attributes and methods. Specifically, this lesson covers:

1. The Employee Class

Since Java is one of the computer languages that follows the object-oriented programming (OOP) model, you've basically been working with classes and objects since the beginning of all of the tutorials (every program has a declared class and all library methods involve classes). When it comes to classes and objects, it's important to note that an object should have everything about itself encapsulated in a class.

Let's first start by looking at an employee of an organization. For an employee, think about what attributes may be needed to describe them. You would have their first name, last name, title, salary, hire date and employee ID as their basic attributes. The start of the Employee class and its attributes should look like this (declared in a file named Employee.java). Note that this code is not a complete class:

EXAMPLE

```
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int emplId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;
```

Let's continue to define our class with an appropriate constructor. You will need to have each of those instance variables (first name, last name, etc.) set as part of the parameters of the constructor. The only item that won't be passed in will be the hire date, which will use the current day's date for when the object representing the employee is created. You would need to import `java.util.LocalDate` for that:

EXAMPLE

```
public Employee(String firstName, String lastName, int emplId, String jobTitle,
        double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
```

```
    this.emplId = emplId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
}
```

TRY IT

Directions: Create our Employee class with the attributes and a parameterized constructor.

Remember the needed import for java.time.LocalDate, as shown above. This gives us a starting point with the employee's details. Once the attributes are set, you typically won't be accessing these variables directly. Rather, you will use accessor methods to get the data from these variables. You will also want to set up mutator (or "setter") methods so you can set these values after a bit of error checking.

For example, for the first name, returning it may not have anything special or unique to check, but when you try to set the first name, you will make sure that the length of the value that's being passed in isn't empty. So, let's add a method called getFirstName() that returns the firstName variable when it is called by the new instance. And you will create another method called setFirstName() with an additional parameter of firstName to check that the argument passed is not an empty string. In order to use it, you must call it directly to set the firstName attribute. This could be done after the object has been defined, and the constructor has set the values already. Using this method, you will be able to update the attribute value afterwards.

↱ EXAMPLE

```
// Returns the first name
public String getFirstName() {
  return firstName;
}

// Sets the value of attribute firstName to value passed as parameter firstName
public void setFirstName(String firstName) {
  if(firstName.length() > 0) {
    this.firstName = firstName;
  }
}
```

TRY IT

Directions: Go ahead and add these two new methods. This of course doesn't check if the value passed into the parameter for the first name was an empty String (""). You would need to handle that separately. The last name and job title can use the same format for their respective methods to set and get those values.

↱ EXAMPLE

```
public String getLastName() {
  return lastName;
}
```

```
public void setLastName(String lastName) {
  if(lastName.length() > 0) {
    this.lastName = lastName;
  }
}
```

EXAMPLE

```
public String getJobTitle() {
  return jobTitle;
}</code>

public void setJobTitle(String jobTitle) {
  if(jobTitle.length() > 0) {
    this.jobTitle = jobTitle;
  }
}
```

TRY IT

Directions: Go ahead and add the four new methods for reading and setting the lastName and jobTitle. Let's see if you have what is currently created so far:

```
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int emplId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle,
          double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.emplId = emplId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
  }

  // Returns the first name
  public String getFirstName() {
    return firstName;
  }
```

```
  // Sets the value of attribute firstName to value passed as parameter firstName
  public void setFirstName(String firstName) {
    if(firstName.length() > 0) {
      this.firstName = firstName;
    }
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    if(lastName.length() > 0) {
      this.lastName = lastName;
    }
  }

  public String getJobTitle() {
    return jobTitle;
  }

  public void setJobTitle(String jobTitle) {
    if(jobTitle.length() > 0) {
      this.jobTitle = jobTitle;
    }
  }
}
```

If your class looks different that what is displayed above, check line by line to see what may be different. Make sure it looks identical before moving on.

REFLECT

Now you will have to set up the salary and employee ID.

For the employee ID (attribute named `empId`), you will not permit the value to be updated (once an employee has an employee ID, it is typically set), so you only need to create the get method to return the value. You will not need a set method for this variable:

EXAMPLE

```
// empId cannot be changed, so there is only accessor, no mutator method public int getEmpId() {   return empId; }
```

TRY IT

Directions: Add the `getEmpId()` method to your class. For the salary, the accessor method will return the annual salary as a double. You need a method that returns the salary as a numeric value that can be used in calculations. You can also declare a method that returns the salary as a formatted String, but as you will see, that method can't be called `getSalary()`.

EXAMPLE

```
public double getSalary() {
  return salary;
}
```

For the mutator or setter method, you will permit any value that's larger than 0:

↱ EXAMPLE

```
public void setSalary(double salary) {
   if(salary > 0.00) {
    this.salary = salary;
   }
}
```

To get the salary in a format that represents an amount as dollars and cents, you will use one of Java's format classes. You do not want to get into a lot of detail about formatting, but the Java DecimalFormat class makes it fairly easy to format a numeric value as an amount of money. Before the start of the Employee class, add this import:

↱ EXAMPLE

```
import java.text.DecimalFormat;
```

A DecimalFormat object is a pattern or template for displaying numeric values. This line of code (in the body of main()) defines a format that has a leading dollar sign, an initial 0 if there is no value to the left of the decimal point, and 2 decimal places (with 0's added to the right to fill out the pattern:

↱ EXAMPLE

```
DecimalFormat salaryFormat = new DecimalFormat("$0.00");
```

Here, the DecimalFormat formatting pattern is named salaryFormat. A DecimalFormat object includes a format() method that converts the numeric value into a String in the specified format. Here is the complete getSalaryAsString() method that returns the salary as a String:

```
public String getSalaryAsString() {
  // Format salary with leading dollar sign and 2 decimal places
  DecimalFormat salaryFormat = new DecimalFormat("$0.00");
  // Use getSalary to get numeric value and then format
  return salaryFormat.format(getSalary());
}
```

Note that while this may look somewhat like an override of the getSalary() method, it really isn't because only the return type varies. Since the return type is not part of a method signature in Java, the compiler won't let us just call this a version of getSalary(), so the distinct name getSalaryAsString() is needed.

TRY IT

Directions: Now add the three new methods for salary, the get method (for returning), and the set method to make sure the value is greater than 0.

Altogether, our code should look like the following:

```
import java.text.DecimalFormat;
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int emplId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle,
          double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.emplId = emplId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
  }

  // Returns the first name
  public String getFirstName() {
    return firstName;
  }

  // Sets the value of attribute firstName to value passed as parameter firstName
  public void setFirstName(String firstName) {
    if(firstName.length() > 0) {
      this.firstName = firstName;
    }
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    if(lastName.length() > 0) {
      this.lastName = lastName;
    }
  }

  public String getJobTitle() {
```

```
  return jobTitle;
 }

 public void setJobTitle(String jobTitle) {
  if(jobTitle.length() > 0) {
   this.jobTitle = jobTitle;
  }
 }

 public double getSalary() {
  return salary;
 }

 public void setSalary(double salary) {
  if(salary > 0.00) {
   this.salary = salary;
  }
 }

 public String getSalaryAsString() {
    // Format salary with leading dollar sign and 2 decimal places
    DecimalFormat salaryFormat = new DecimalFormat("$0.00");
    // Use getSalary to get numeric value and then format
    return salaryFormat.format(getSalary());
 }

 // emplId cannot be changed, so there is only accessor, no mutator method
 public int getEmplId() {
  return emplId;
 }
}
```

Now that our class is set up, it's time to create instances (objects) of the `Employee` class.

TRY IT

Directions: Under the Employee class, start to create the first object. Call it "sophia" or any employee name you would like as the first object. Start with the name and the equal sign (=), then our class Employee, then the first parenthesis:

EXAMPLE As a reminder, here is the signature for the `Employee()` constructor:

```
Employee(String firstName, String lastName, int emplId, String jobTitle,
        double salary)
```

EXAMPLE A sample call to the constructor looks like this:

```
Employee empl = new Employee("Jack", "Krichen", 1000, "Manager", 75000);
```

TRY IT

Directions: Let's write the code that creates an Employee object. You can use the same arguments as seen in

the code below, or create your own employee. After entering the arguments, make sure to add the print() functions with each variable, so you can see that the instance created, “sophia” in this case, did include our arguments:

```
public class EmployeeProgram {
  public static void main(String[] args) {
    Employee empl = new Employee("Jack", "Krichen", 1000, "Manager", 75000);
    System.out.println("First Name: " + empl.getFirstName());
    System.out.println("Last Name: " + empl.getLastName());
    System.out.println("EmplId: " + empl.getEmplId());
    System.out.println("Job Title: " + empl.getJobTitle());
    System.out.println("Salary: " + empl.getSalaryAsString());
  }
}
```

TRY IT

Directions: Go ahead and run the program to see if you get the same output, or the expected output if you used different employee values. Remember to compile the Employee class first, using this command:

EXAMPLE

```
javac Employee.java
```

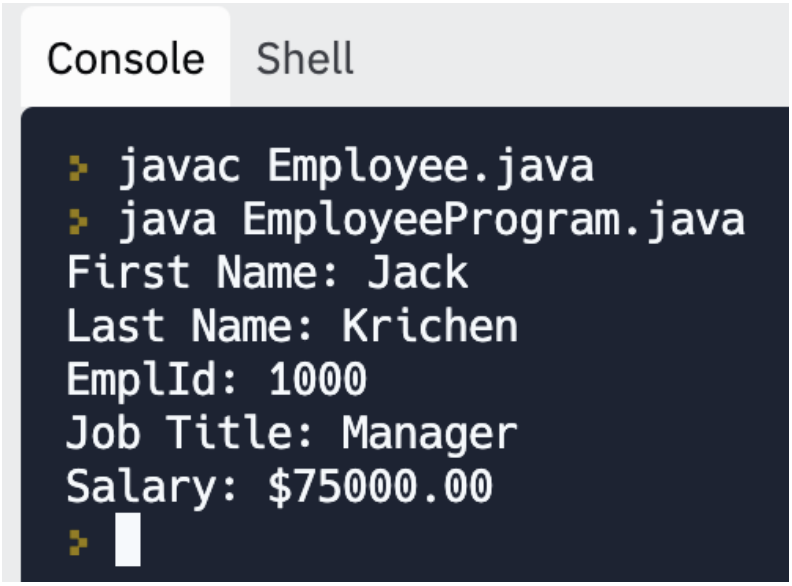
TRY IT

Directions: Then, run the program using this command:

EXAMPLE

```
java EmployeeProgram.java
```

The results should look like this:



REFLECT

This would be a good foundation for our Employee class. However, you could add some more functionality, such as allowing for increases in the employee's salary. You could pass in a percentage and the salary would automatically be increased by that amount:

EXAMPLE

```
// Method to increase salary by percent as decimal. 0.02 is a 2% raise
  public void increaseSalary(double percentAsDecimal) {
    if(percentAsDecimal > 0.0) {
      salary *= (1 + percentAsDecimal);
    }
  }
```

REFLECT

Notice that as part of the calculation of this new `increaseSalary()` method, you check if the percentage is greater than 0 or not. If it is, you will add 1 to the decimal representing the rate of increase and multiply the current salary by the sum. Using the **compound assignment operator for multiplication (*=)** takes the current value of the `salary` attribute and multiplies it by the factor for the raise and assigns the product back to the `salary` attribute.

TRY IT

Directions: Let's give it a shot by adding a 2% increase in salary. First, add this new method to the Employee class and then add the increase of salary code below to the program. Finally, run the program.

Here is the code for the Employee class (Employee.java):

```
import java.text.DecimalFormat;
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int empId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int empId, String jobTitle,
          double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.empId = empId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
```

```
}

// Returns the first name
public String getFirstName() {
  return firstName;
}

// Sets the value of attribute firstName to value passed as parameter firstName
public void setFirstName(String firstName) {
  if(firstName.length() > 0) {
    this.firstName = firstName;
  }
}

public String getLastName() {
  return lastName;
}

public void setLastName(String lastName) {
  if(lastName.length() > 0) {
    this.lastName = lastName;
  }
}

public String getJobTitle() {
  return jobTitle;
}

public void setJobTitle(String jobTitle) {
  if(jobTitle.length() > 0) {
    this.jobTitle = jobTitle;
  }
}

public double getSalary() {
  return salary;
}

public void setSalary(double salary) {
  if(salary > 0.00) {
    this.salary = salary;
  }
}

public String getSalaryAsString() {
    // Format salary with leading dollar sign and 2 decimal places
    DecimalFormat salaryFormat = new DecimalFormat("$0.00");
    // Use getSalary to get numeric value and then format
    return salaryFormat.format(getSalary());
}
```

```
  // emplId cannot be changed, so there is only accessor, no mutator method
  public int getEmplId() {
    return emplId;
  }

  // Method to increase salary by percent as decimal. 0.02 is a 2% raise
  public void increaseSalary(double percentAsDecimal) {
    if(percentAsDecimal > 0.0) {
      salary *= (1 + percentAsDecimal);
    }
  }
}
```

After entering this code in a file named Employee.java, remember to compile the class using this command:

→ EXAMPLE

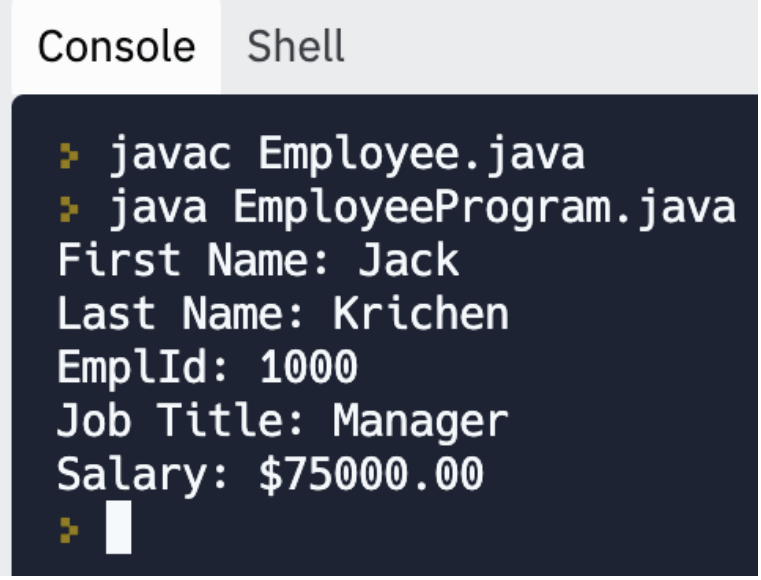
```
javac Employee.java
```

Here is the code for the application's driver class (EmployeeProgram) that uses the Employee class. This code needs to be entered in a file named EmployeeProgram.java:

```
import java.text.DecimalFormat;  // Needed for DecimalFormat

public class EmployeeProgram {
  public static void main(String[] args) {
    Employee empl = new Employee("Jack", "Krichen", 1000, "Manager", 75000);
    System.out.println("First Name: " + empl.getFirstName());
    System.out.println("Last Name: " + empl.getLastName());
    System.out.println("EmplId: " + empl.getEmplId());
    System.out.println("Job Title: " + empl.getJobTitle());
    System.out.println("Salary: " + empl.getSalaryAsString());
  }
}
```

REFLECT

Did you get the following as expected? Now our employee will get a bump of $1000.00.



THINK ABOUT IT

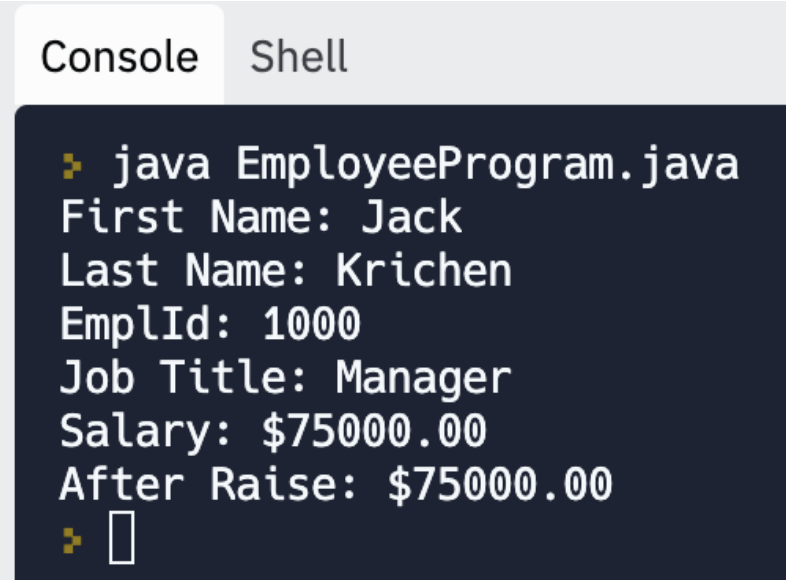
What happens in the case when the salary increase is less than 0?

TRY IT

Directions: Add the following two lines after the line that prints out the salary:

```
empl.increaseSalary(-0.02);
System.out.println("After Raise: " + salaryFormat.format(empl.getSalary()));
```

The result should look like this:



REFLECT

In this case, there is no change. But this was probably an error on the part of the user entering the values.

TRY IT

Directions: Go ahead and try a negative salary increase. This is not an ideal result, as you most likely would want

to inform the user that the value was not accepted. You can add that message to the user to inform them of the error. In the Employee class, revise theincreaseSalary() method to look like this:

```
// Method to increase salary by percent as decimal. 0.02 is a 2% raise
  public void increaseSalary(double percentAsDecimal) {
    if(percentAsDecimal > 0.0) {
      this.salary *= (1 + percentAsDecimal);
    }
    else {
      System.out.println("Salary increase must be greater than 0.");
    }
  }
}
```

REFLECT

Here you have added a selection statement to look to see if the value entered (passed as an argument) is greater than 0. If it is not, it will present an error.

TRY IT

Directions: Add the conditional statement lines to our increaseSalary() method.

Your program should look like this:

```
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int emplId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle,
          double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.emplId = emplId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
```

```
  if(firstName.length() > 0) {
    this.firstName = firstName;
  }
 }

 public String getLastName() {
  return lastName;
 }

 public void setLastName(String lastName) {
  if(lastName.length() > 0) {
    this.lastName = lastName;
  }
 }

 // EmplId cannot be changed, so there is only accessor, no mutator method
 public int getEmplId() {
  return emplId;
 }

 public String getJobTitle() {
  return jobTitle;
 }

 public void setJobTitle(String jobTitle) {
  if(jobTitle.length() > 0) {
    this.jobTitle = jobTitle;
  }
 }

 public double getSalary() {
  return salary;
 }

 public void setSalary(double salary) {
  if(salary > 0.00) {
    this.salary = salary;
  }
 }

public String getSalaryAsString() {
    // Format salary with leading dollar sign and 2 decimal places
    DecimalFormat salaryFormat = new DecimalFormat("$0.00");
    // Use getSalary to get numeric value and then format
    return salaryFormat.format(getSalary());
 }


 // Method to increase salary by percent as decimal. 0.02 is a 2% raise
 public void increaseSalary(double percentAsDecimal) {
  if(percentAsDecimal > 0.0) {
```

```
    this.salary *= (1 + percentAsDecimal);
   }
   else {
    System.out.println("Salary increase must be greater than 0.");
   }
  }
}
```

TRY IT

Directions: Compile the revised Employee class using this command:

EXAMPLE

```
javac Employee.java
```

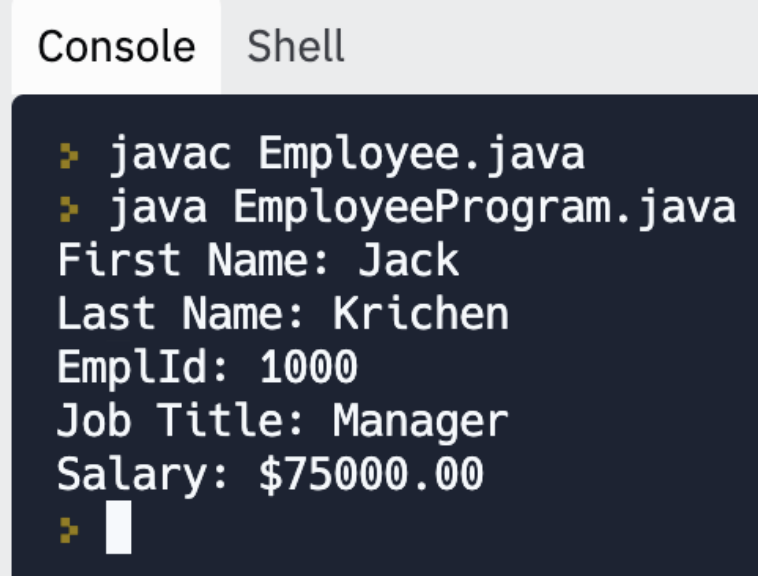
TRY IT

Directions: Try running the program again using the following in the shell:

EXAMPLE

```
java EmployeeProgram.java
```

You should now see the error message appear:



BRAINSTORM

Directions: This marks the end of this program for now. Try making some additional changes on your own. See if you can add any additional parameters to our Employee class.

REFLECT

We have built out the Employee class to the point where it is starting to resemble a class that would be useful in the real world. What other data and functionality might be appropriate parts of the Employee class? How could a

programmer determine what information and related functionality would or would not belong in a class?

TERM TO KNOW

Compound Assignment Operator for Multiplication (*=)
This operator multiplies the variable to the left by the value or expression to the right and assigns the product back to the variable to the left.

SUMMARY

In this lesson, you went through a program that creates an **Employee class** with all of the attributes and methods associated with it. You have included validation and verification of the object attributes to ensure that valid values are being set when they are being updated.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Compound Assignment Operator for Multiplication (*=)
This operator multiplies the variable to the left by the value or expression to the right and assigns the product back to the variable to the left.

Introduction to Inheritance

by Sophia

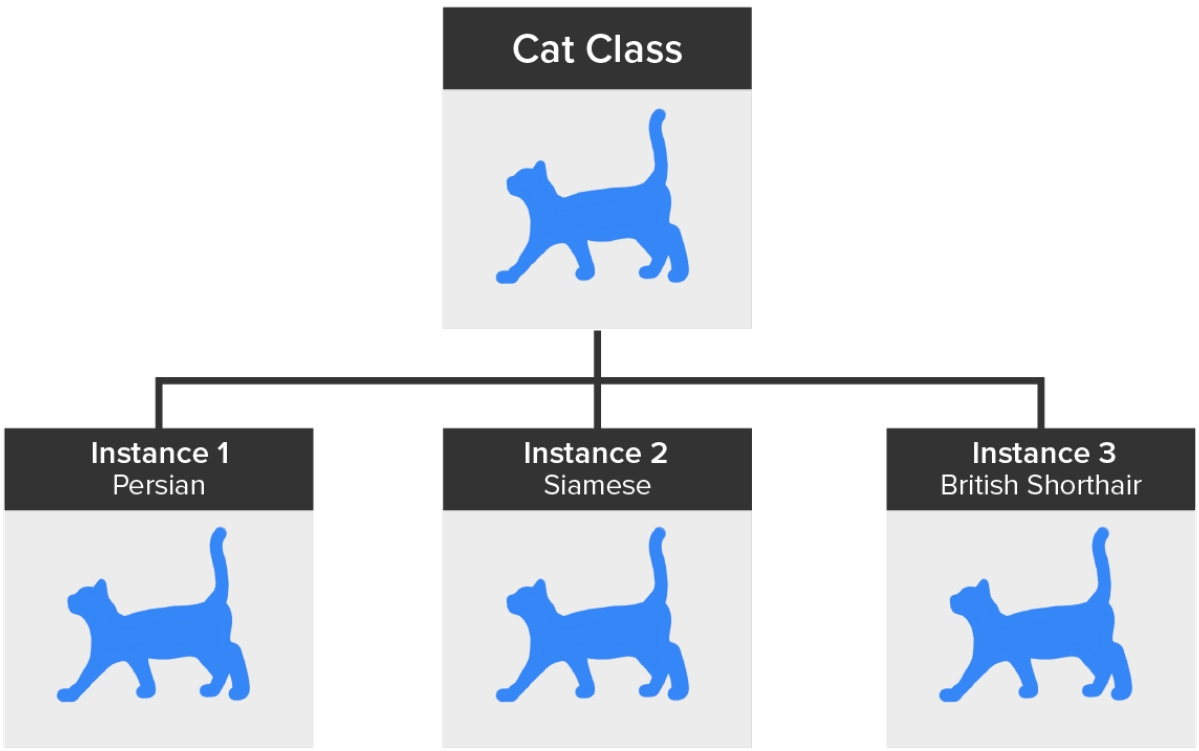
WHAT'S COVERED

In this lesson, you will learn about inheritance and how properties and methods are inherited from a parent class. Specifically, this lesson covers:

1. Real-World Example

Inheritance and subclasses are often topics of conversation when discussing object-oriented programming. While these terms may sound complex, these are concepts about things that you actually see in the real world.

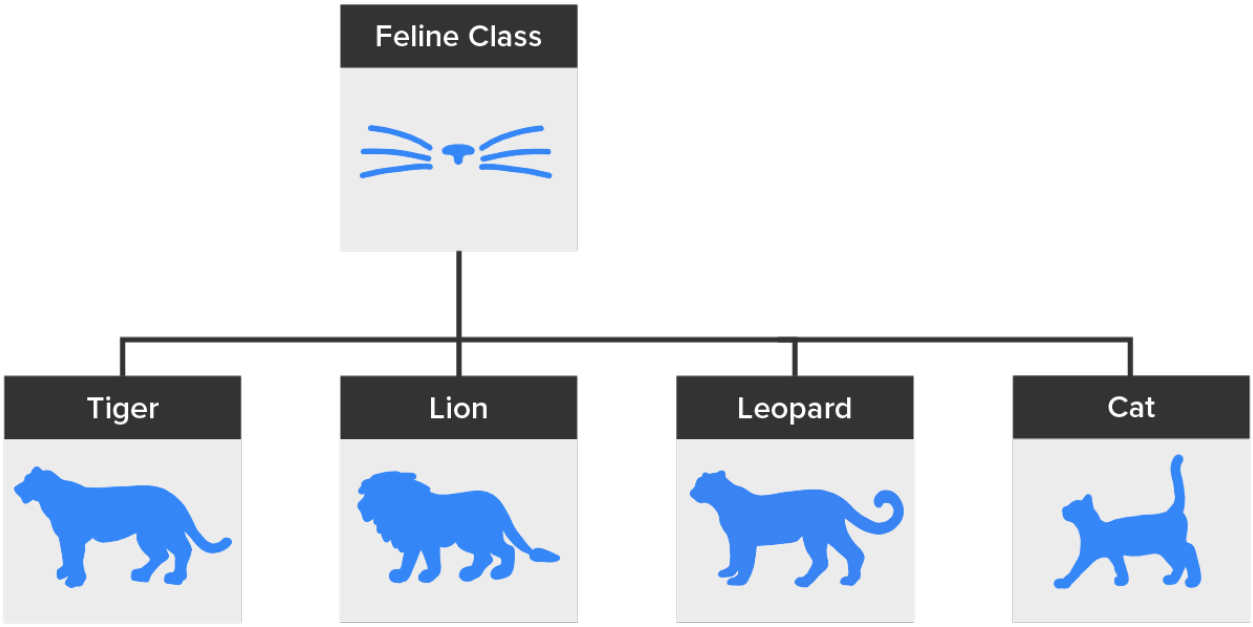
If you want to consider cats as a Java class, you could put together all of the possible cats into a class of animals that you would call `Cats`. Even though each cat may be unique, it is still a type of cat and would be a member of your `Cat` class.



Looking at the graphic above, Persian, Siamese, and British Shorthair are unique breeds of cats, but what makes cats similar to one another are the characteristics that they all inherit from the class `Cat`.

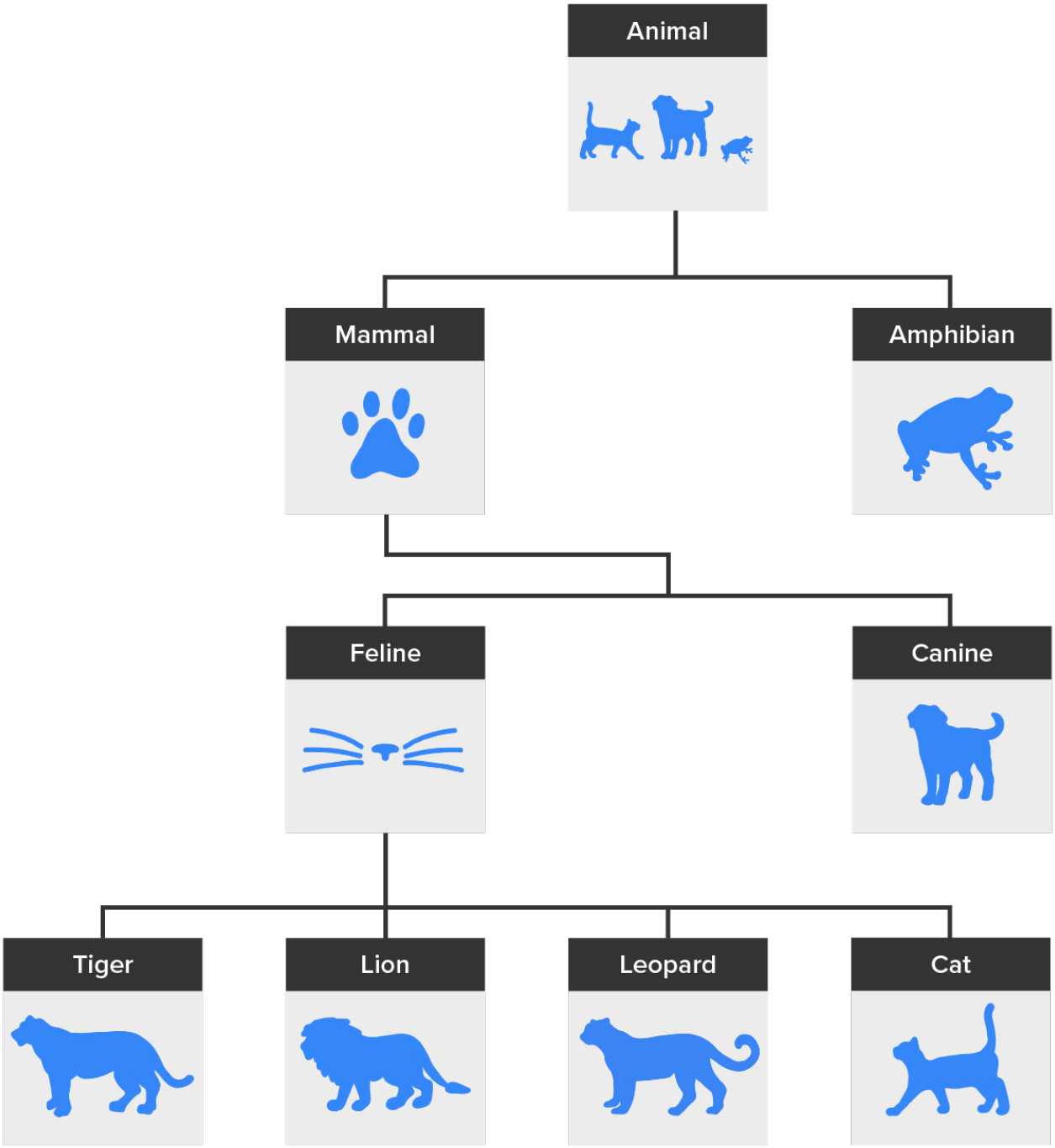
Classes and class inheritance were created to align with much of the information in the real world and how it can be stored, categorized, and understood by using classes and subclasses. For example, you may have noticed

other creatures similar to cats that are in the real world. They would be similar to cats in that they inherit features from a higher-level class that we could call Felines.



You could have the class Feline, and under it, you could have Tiger, Leopard, Lion, and Cat as subclasses.

Above Feline, you could have Mammals as a layer, as all felines are mammals. Then, on top of Mammals, you could have a layer called Animals since all mammals are animals.



This concept doesn't just apply to cats. It can apply to other animals like dogs or amphibians. If you search online to see the hierarchy of animals, you'll see that there are many ways to classify animals, and also see how inheritance works all the way down various lines to specific animals.

From a coding perspective, the easiest way to use inheritance is to create "child" classes (or subclasses) from "parent" classes or (base classes).

IN CONTEXT

You have your mother's eyes, and you have the same patience as your father...? These are things that are relevant to the term "inheritance." You inherited those objects or traits without having to do anything.

CONCEPT TO KNOW

In the world of Java programming (and programming more generally), inheritance means the same thing—a “child” class inheriting characteristics (data) and behaviors (methods) from a “parent” class without modifying anything.

You will hear a few terms in this Challenge that may be defined differently in other resources you may be using, so it makes sense to call them out.

Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

A **base class** is also known as a parent class. It is a class that is being inherited from.

A **subclass** is also known as a child class. It is a class that inherits from another class. A subclass can “extend” the base class, meaning that it can define additional data that will not affect the base class.

From here on, we will be using the terms "base class" and "subclass."

A base class defines all of the things that apply to all instances of that class. Therefore, a subclass automatically inherits what was defined in the base class. Once the subclass is created, though, any data/methods defined in that subclass are relevant only to that subclass. Anything that is defined in the subclass will not change or replace anything that is coming from the base class. These definitions will also not affect any other subclasses of the base class.

BIG IDEA

So, what is the benefit of inheritance in Java? Inheritance allows programmers to create subclasses that have the properties and methods of an existing class. This supports code reusability, maintenance, and optimization; and it speeds up development time. You will not have to “reinvent the wheel” with each class. You can create it once and use it multiple times. Back to our feline example. If you have a base class with all the common attributes and standard behaviors (methods) that every feline should have, you should never need to alter that base class when creating subclasses. You inherit those properties and methods automatically, so there is no need to define those again. Also, if something changes in the base class, once that change is made, it automatically cascades those changes to each subclass.

TERMS TO KNOW

Base Class
A base class is also known as a parent class or superclass. It is a class that is being inherited from.

Subclass
A subclass is also known as a child class. It is a class that inherits from another class. A subclass can “extend” the base class, meaning that it can define additional data and related behavior that will not affect the base class.

Inheritance
Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

---

2. Base Class

Subclasses inherit the public methods (behaviors/actions) of a higher-level class or the base class. A base class is no different than any other class, but it is simply a class that a subclass would inherit from.

To help define these inheritance concepts and start reviewing code, let's start with a class called `Member`:

TRY IT

Directions: Input the following into Replit:

```
import java.time.LocalDate;

public class Member {
  private String firstName;
  private String lastName;
  private int expiryDays = 365;
  private LocalDate expiryDate;

  public Member(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    expiryDate = LocalDate.now().plusDays(expiryDays);
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public LocalDate getExpiryDate() {
    return expiryDate;
  }

  public String getStatus() {
    return firstName + " " + lastName + " is a Member.";
  }
}
```

TRY IT

Directions: Compile the Member class by running the following command:

```
javac Member.java
```

REFLECT

In the code, you may find it quite familiar, as we used something very similar in a prior tutorial. We are importing the library class `java.time.LocalDate` to use the current date functionality. Again, you will be seeing module importation and usage in the next challenge. Then, we defined our class called `Member`. There is an attribute

declared called expiryDays, and we set it to "365". It is used to calculate the expiration date from the current date. We then defined the constructor with three parameters. The first is the “self” parameter, which is used to point to the instance being referenced. Next are the “first” and “last” parameters, which are then assigned to the variables first_name and last_name. The expiry_date variable is set to "365" days after today when we run the code. The .timedelta() method (included in the datetime module) calculates the difference of time between the two arguments.

Next, we will create an instance of this class with the variable TestMember and create some print() functions so we can see some output to the screen.

Let’s give this a test.

TRY IT

Directions: Create a MemberProgram class in a file named MemberProgram.java:

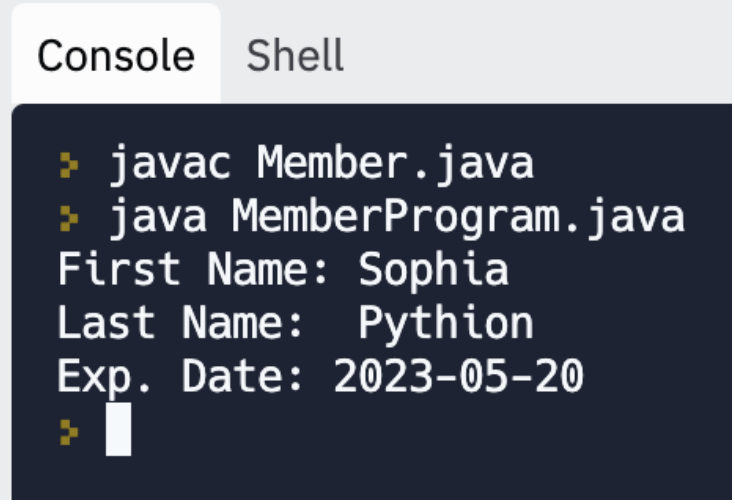
```
class MemberProgram {
  public static void main(String[] args) {
    Member testMember = new Member("Sophia", "Java");
    System.out.println("First Name: " + testMember.getFirstName());
    System.out.println("Last Name:  " + testMember.getLastName());
    System.out.println("Exp. Date: " + testMember.getExpiryDate());
  }
}
```

TRY IT

Directions: After compiling the Member class (as noted above), run the MemberProgram using this command:

```
java MemberProgram.java
```

The output table below is the result of this small program:



THINK ABOUT IT

See anything new? Not really. This is very similar to a class we built out in the last challenge. You have already created this class to use as a base class for the examples to follow. Now, the intent is to create two different kinds of subclasses of the Member class.

In the next topic, you will create an `Admin` and a `User` subclass. Both of these subclasses will have attributes that the `Member` class includes. These new classes will be subclasses of our `Member` class. They will automatically include the attributes and methods of our base class (`Member`).

When you define those types of members as a subclass of `Member`, they will automatically be assigned the same attributes and methods if there are any defined.

---

3. Subclasses

To create and define a subclass, you will need to ensure that the subclass is below the base class and is not indented. The subclass is not part of or contained within the base class.

Therefore, you would use the following syntax:

```
class SubclassName extends BaseClassName {
```

It is important to note the use of the keyword `extends` between the name of the subclass and the base class.

You would replace `SubclassName` with what you want to name the subclass. Then, you would replace `BaseClassName` with the name of the base class. For example, to make a subclass of our base class `Member` and name it `Admin`, you would create a file named Admin.java and start with the following code:

```
public class Admin extends Member {
```

The code for the first version of this class should look like this:

```
public class Admin extends Member{
  public Admin(String firstName, String lastName) {
    super(firstName, lastName);
  }
}
```

At this point, you would keep the content and functionality of the subclasses simple. They don't have any attributes or methods of their own, other than a parameterized constructor. This simple structure will allow you to test the two subclasses.

You can do the same thing for the `User` class as well.

TRY IT

Directions: The following code should be entered in a file named User.java:

```
public class User extends Member {
  public User(String firstName, String lastName) {
    super(firstName, lastName);
  }
}
```

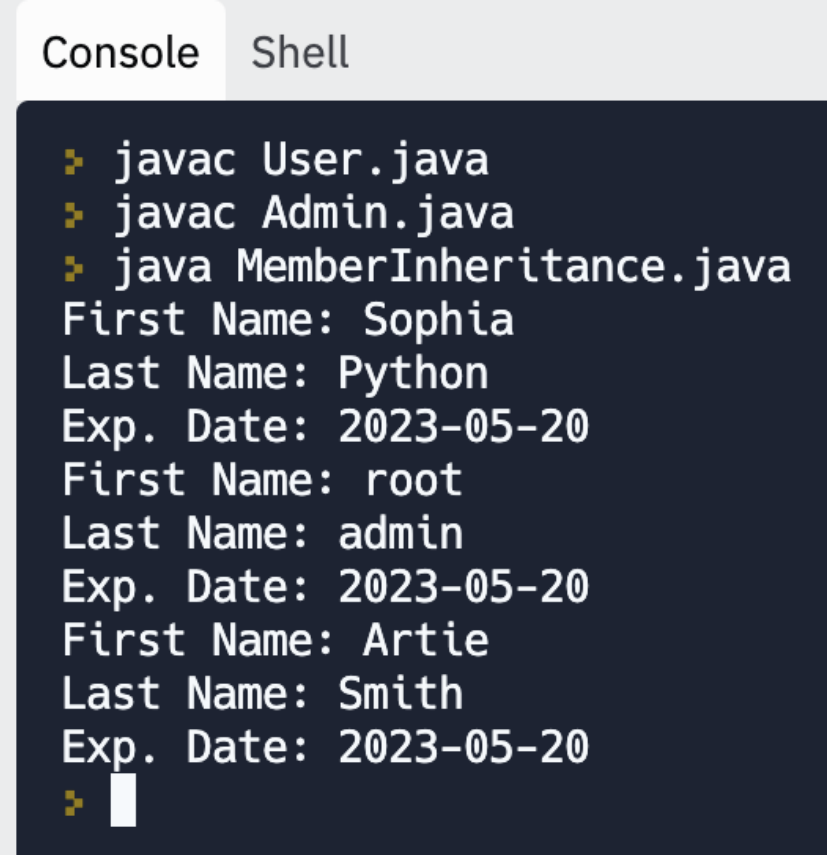
To do a quick test to ensure that this works, you would create an `Admin` and a `User` instance as subclasses that will simply inherit all of the methods from our base class (`Member`). You would also include some output statements to view and evaluate the logic of the program so far.

TRY IT

Directions: Save the following code in a file named MemberInheritance.java:

```
public class MemberInheritance {
  public static void main(String[] args) {
    Member testMember = new Member("Sophia", "Java");
    System.out.println("First Name: " + testMember.getFirstName());
    System.out.println("Last Name: " + testMember.getLastName());
    System.out.println("Exp. Date: " + testMember.getExpiryDate());

    Admin testAdmin = new Admin("root", "admin");
    System.out.println("First Name: " + testAdmin.getFirstName());
    System.out.println("Last Name: " + testAdmin.getLastName());
    System.out.println("Exp. Date: " + testAdmin.getExpiryDate());

    User testUser = new User("Artie", "Smith");
    System.out.println("First Name: " + testUser.getFirstName());
    System.out.println("Last Name: " + testUser.getLastName());
    System.out.println("Exp. Date: " + testUser.getExpiryDate());
  }
}
```

We added some different arguments to the subclass instances besides “Sophia” and “Java,” so the output will be clearer. Again, using the same output statements for each of the instances, the results of this would look like the following:



REFLECT

Here we see the outputs for the base class (`Member`) and both instances of the subclasses (`Admin` and `User`). At this point, all three classes produce the same output, but note that all of the attributes and the methods for accessing them are in the `Member` base class. The two subclasses just have parameterized constructors that call the superclass's constructor (via the `super()` constructor).

BRAINSTORM

Directions: Build out a base class and create a subclass. Practice on some of the code above or create your own.

SUMMARY

In this lesson, you learned about inheritance by looking at **real-world examples**. The use of inheritance and classes in a programming language increases the efficiency and supports the “build once and use many times” analogy. You saw that any defined class can be the **base class** whose attributes (properties) and methods (behaviors/actions) can be inherited from. **Subclasses** are defined as the classes that inherit from the base class. A subclass can “extend” the base class, meaning that it can define additional data that will not affect the base class. Finally, you saw the output of two subclass instances inherit attributes from a base class that we built.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Base Class

A base class is also known as a parent class or superclass. It is a class that is being inherited from.

Inheritance

Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

Subclass

A subclass is also known as a child class. It is the class that inherits from another class. A subclass can “extend” the base class, meaning that it can define additional data and related behavior that will not affect the base class.

Child Classes

by Sophia

WHAT'S COVERED

In this lesson, you will explore how subclasses are used and how they can extend functionality. Specifically, this lesson covers:

1. Customizing Subclasses

So far, you have learned that a subclass accepts all the different parameters of its base class. In a previous example, you created and defined a base class called Member. You then created two subclasses called Admin and User. The subclasses inherited the methods from the base class, and these, in turn, provided access to the data in the attributes. However, the Admin and User were just subclasses without any unique characteristics. If you recall, you declared the subclasses with parameterized constructors that just called the parent class's parameterized constructor using **super()**.

Let's look at the prior example again, starting with the Member class (in a file named Member.java).

```
import java.time.LocalDate;

public class Member {
  private String firstName;
  private String lastName;
  private int expiryDays = 365;
  private LocalDate expiryDate;

  public Member(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    expiryDate = LocalDate.now().plusDays(expiryDays);
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public LocalDate getExpiryDate() {
    return expiryDate;
  }
```

```
}
```

↷ EXAMPLE Here is the code for the Admin subclass (in Admin.java):

```
public class Admin extends Member{
  public Admin(String firstName, String lastName) {
    super(firstName, lastName);
  }
}
```

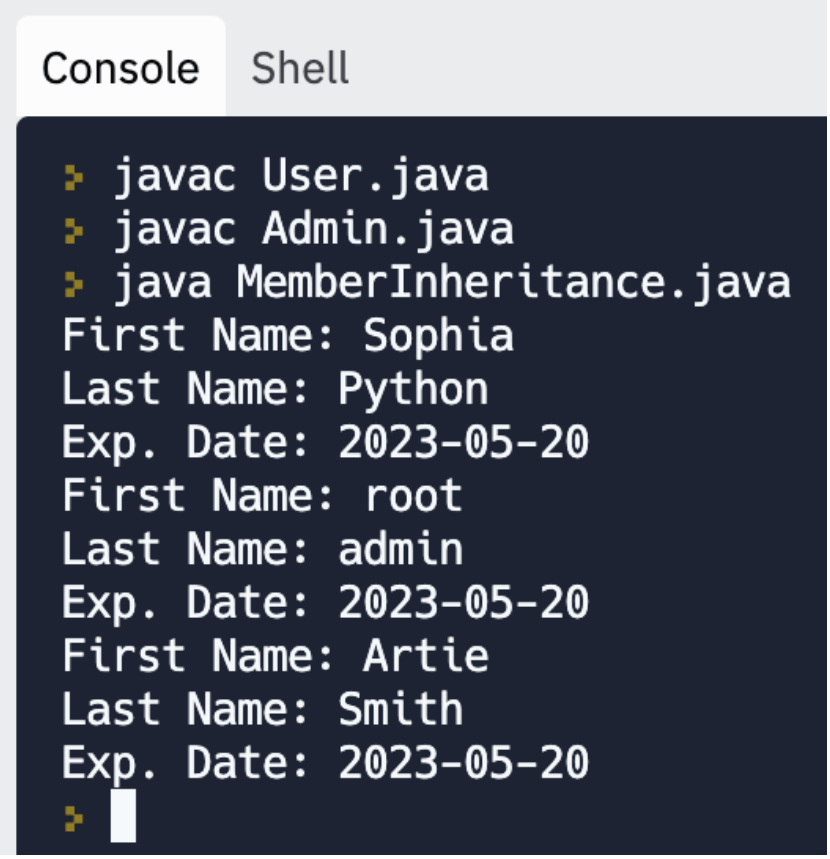
↷ EXAMPLE And here is the almost identical code for the User class (User.java):

```
public class User extends Member {
  public User(String firstName, String lastName) {
    super(firstName, lastName);
  }
}
```

↷ EXAMPLE Lastly, here is the driver class that is used to test these classes (MemberInheritance.java):

```
public class MemberInheritance {
  public static void main(String[] args) {
    Member testMember = new Member("Sophia", "Java");
    System.out.println("First Name: " + testMember.getFirstName());
    System.out.println("Last Name: " + testMember.getLastName());
    System.out.println("Exp. Date: " + testMember.getExpiryDate());

    Admin testAdmin = new Admin("root", "admin");
    System.out.println("First Name: " + testAdmin.getFirstName());
    System.out.println("Last Name: " + testAdmin.getLastName());
    System.out.println("Exp. Date: " + testAdmin.getExpiryDate());

    User testUser = new User("Artie", "Smith");
    System.out.println("First Name: " + testUser.getFirstName());
    System.out.println("Last Name: " + testUser.getLastName());
    System.out.println("Exp. Date: " + testUser.getExpiryDate());
  }
}
```

Compiling the Member, Admin, and User classes and then running the driver class produced output like this:



There is nothing specific defined in either subclass other than the constructor (and its call to the`super()`). To truly make the subclasses useful, we need them to have some differences.

One of the things we can do is to make an attribute that has a default value different from what's in the base class. For example, in the `Member` class, we set the `expiryDays` attribute to 365. However, if we want a rule in place that we don't want the `Admin` accounts to expire until 100 years from now, we could change the `expiryDays` value to be 365 * 100. We do this just by declaring the attribute with the desired default value in the `Admin` subclass. Since the `getExpiryDate()` method inherited from the base class will return the expiration date for a `Member`, we also need to override the `getExpiryDate()` method in the `Admin` subclass.

TRY IT

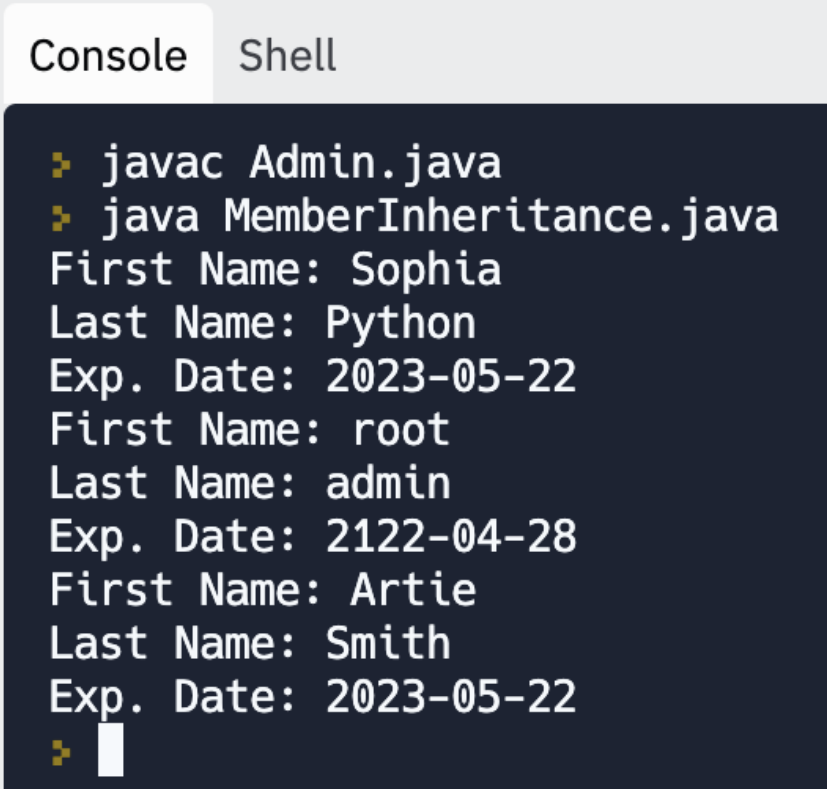
Directions: Enter this code in Replit and make sure you get the same output before moving on:

```
import java.time.LocalDate;

// Subclass of Member for administrators
public class Admin extends Member{
  private int expiryDays = 100 * 365;
  private LocalDate expiryDate;

  public Admin(String firstName, String lastName) {
    super(firstName, lastName);
    expiryDate = LocalDate.now().plusDays(expiryDays);
  }

  @Override
  public LocalDate getExpiryDate() {
```

```
    return expiryDate;
  }
}
```

The private `expiryDays` attribute in the base class is not inherited, so you can declare an`expiryDays` attribute in the subclass, but to get the public `getExpiryDays()` method to return the correct value for the subclass (rather than the base class), you would need to override the inherited `getExpiryDate()` method.

Compiling the `Admin` subclass and then running the MemberInheritance.java program in Replit should now produce results like these (the specific dates will vary):



TRY IT

Directions: Try modifying the Admin subclass as shown above with the following updates:

Sometimes a subclass has attributes that the base class does not. In that case, you may want to pass an argument to the subclass's constructor that doesn't exist in the base class. Doing so is a little more complicated; however, it is a common technique, so you should be aware of the steps to do that.

The `Admin` subclass already has its own constructor that calls the base class's constructor using`super()` and then initializes the attributes that are specific to the `Admin` class. In this case, you would need to add a`String` attribute named secret with a line like this:

EXAMPLE

```
private String secret;
```

This attribute does not have a default value, so the constructor for the`Admin` class will need to be modified to add a parameter to provide this value when an `Admin` object is created. The constructor's signature line now will look like this:

EXAMPLE

```
public Admin(String firstName, String lastName, String secret) {
```

See, it looks identical to the base class's constructor except for the last parameter named `secret`.

At this point, it's not important what the value of secret will be used for, but if the value is to be used for something, we will need to provide an accessor method:

EXAMPLE

```
public String getSecret() {
 return secret;  }
```

TRY IT

Directions: Revise the Admin class as shown above:

The revised version of the `Admin` class should now look like this:

```
import java.time.LocalDate;

// Subclass of Member for administrators
public class Admin extends Member{
  private int expiryDays = 100 * 365;
  private LocalDate expiryDate;
  private String secret;

  public Admin(String firstName, String lastName, String secret) {
    super(firstName, lastName);
    expiryDate = LocalDate.now().plusDays(expiryDays);
    this.secret = secret;
  }

  @Override
  public LocalDate getExpiryDate() {
    return expiryDate;
  }

  public String getSecret() {
    return secret;
  }
}
```

REFLECT

It is important to remember that while subclasses get access to the public methods in the base class (and thus get access to the attributes in the base class) for free, this doesn't mean that the subclass is limited to these. Subclasses can build on the functionality provided by the base class and customize it.

Let's test this altogether now with an updated `Admin` subclass and updated `Admin` instance using a new third argument for the secret parameter. We are leaving the `User` subclass as it was. Note: we placed a string of lines to separate each subclass for easier viewing.

TRY IT

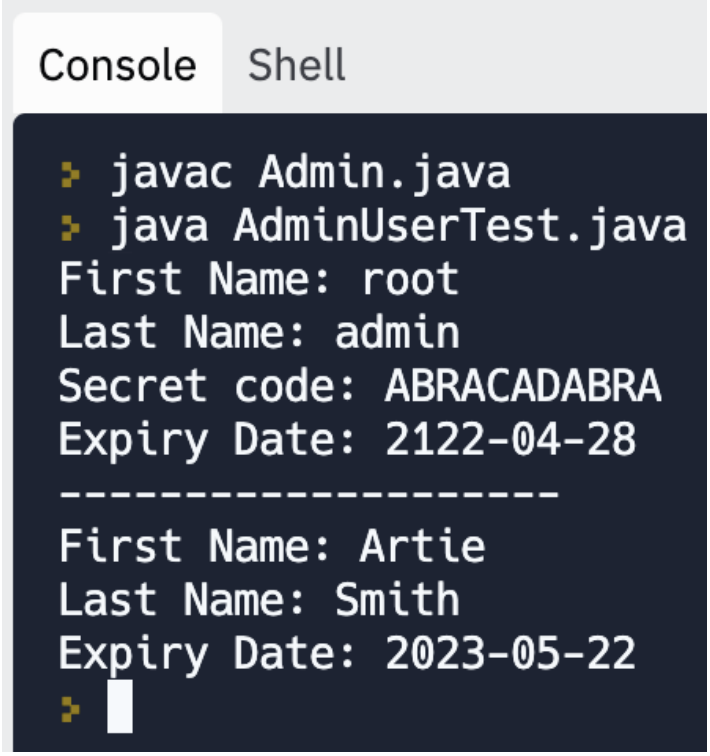
Directions: To test these classes, use the following code for the AdminUserTest class (saved in a file named AdminUserTest.java):

```
class AdminUserTest {
  public static void main(String[] args) {
    Admin testAdmin = new Admin("root", "admin", "ABRACADABRA");
    System.out.println("First Name: " + testAdmin.getFirstName());
    System.out.println("Last Name: " + testAdmin.getLastName());
    System.out.println("Secret code: " + testAdmin.getSecret());
    System.out.println("Expiry Date: " + testAdmin.getExpiryDate());

    System.out.println("--------------------");

    User testUser = new User("Artie", "Smith");
    System.out.println("First Name: " + testUser.getFirstName());
    System.out.println("Last Name: " + testUser.getLastName());
    System.out.println("Expiry Date: " + testUser.getExpiryDate());
  }
}
```

TRY IT

Directions: Enter the updated code to test the application. We are only creating instances of the subclasses. Be sure to compile the revised Admin class if you haven't done so already, using the javac compiler and then run the AdminUserTest.java program:

The output for this should look something like the following:



In this example, we have created an instance of the `Admin` subclass with arguments of “root” for firstname, “admin” as lastname, and “ABRACADABRA” as the secret code value.

In the output, we can see that all the attributes are present, including the updated attribute `expiryDate` from the `Admin` subclass. You can also see that the `User` subclass hasn’t been affected by any changes that were made to the `Admin` subclass.

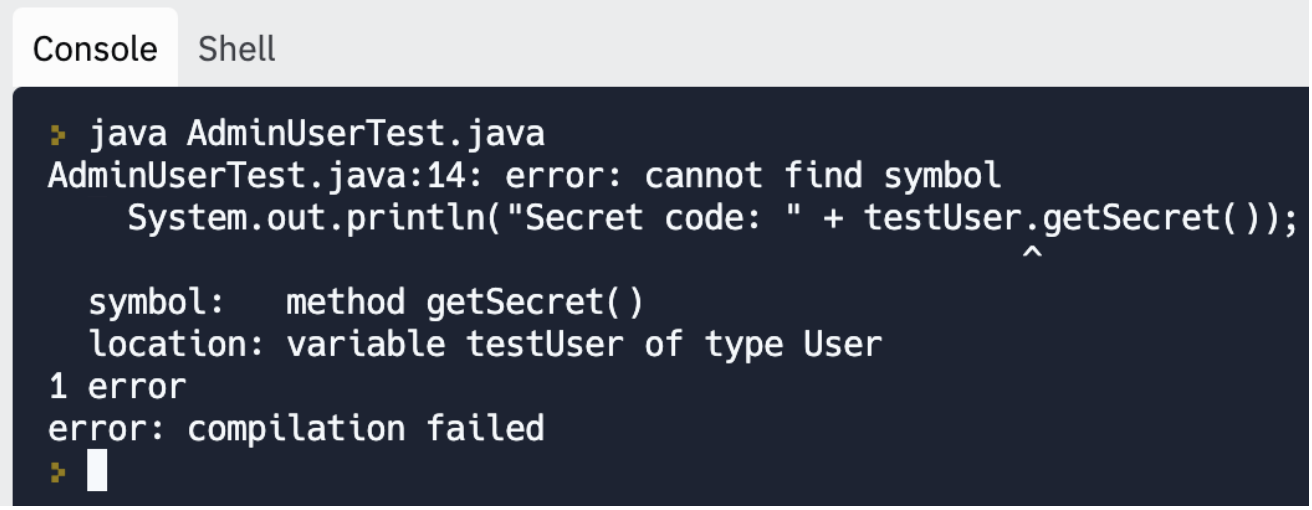
TRY IT

Directions: Test this by trying to call getSecret() from the User subclass using code like the following:

EXAMPLE

```
User testUser = new User("Artie", "Smith");
System.out.println("First Name: " + testUser.getFirstName());
System.out.println("Last Name: " + testUser.getLastName());
System.out.println("Secret code: " + testUser.getSecret());
System.out.println("Expiry Date: " + testUser.getExpiryDate());
```

Note that this code will not compile or run as demonstrated below:



REFLECT

It is important to keep in mind that subclasses "know" about the parent class, but the parent (or base) class doesn't "know" about the functionality that the subclasses add. While classes inheriting from a common base class have common functionality, they may customize or build on the base class in different ways that do not affect each other.

This is because the `getSecret()` method only belongs to the Admin subclass and not the User subclass. What we define in a subclass is not inherited by or added to the other subclasses of the same base class.

There are also instances where we may have the same method name in the base class and in a subclass. We saw this when we overrode the inherited `getExpiryDate()` method in the Admin subclass. When a subclass overrides an inherited method, Java will use the most specific one that's tied to the subclass. It will use the more generic method if nothing in that subclass overrides the method in the base class.

For example, we'll add a method called `getStatus()` to the base class and each subclass. The `getStatus()` method returns a concatenated string with first name, last name, and the class.

Here is the method as it needs to be implemented in the `Member` class:

↗ EXAMPLE

```
public String getStatus() {
 return firstName + " " + lastName + " is a Member.";
}
```

Since the `firstName` and `lastName` are private attributes in the Member class, they can be referenced by name in the method. In the subclasses, though, the methods that override the `getStatus()` method will need to use the `getFirstName()` and `getLastName()` methods inherited from the Member class.

Here is the version of the method for the `Admin` and `User` classes:

↗ EXAMPLE

```
public String getStatus() { return getFirstName() + " " + getLastName() + " is an Admin."; }
```

TRY IT

Directions: Compile the Member, Admin, and User classes using these commands:

EXAMPLE

```
javac Member.java
javac Admin.java
javac User.java
```

TRY IT

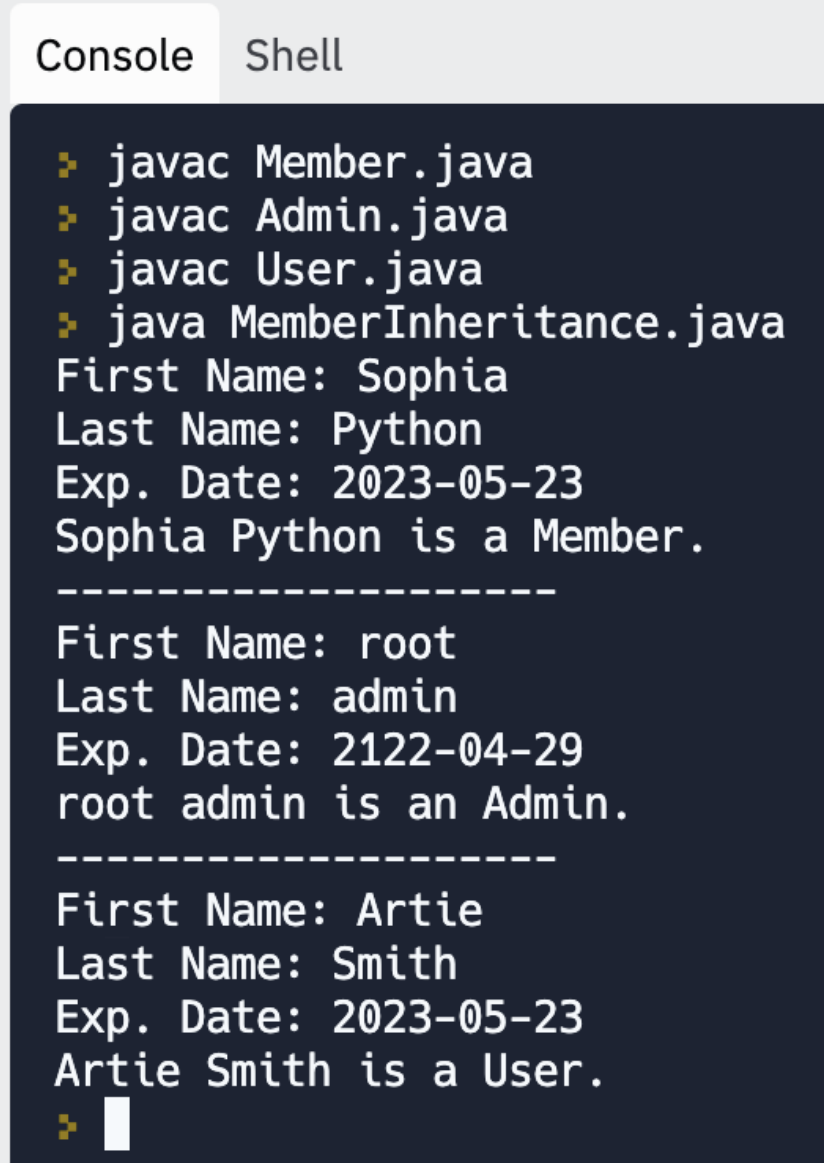
Directions: Then, to demonstrate how these classes work now, let's modify the code in the MemberInheritance.java file so that it reads as follows:

```
public class MemberInheritance {
  public static void main(String[] args) {
    Member testMember = new Member("Sophia", "Java");
    System.out.println("First Name: " + testMember.getFirstName());
    System.out.println("Last Name: " + testMember.getLastName());
    System.out.println("Exp. Date: " + testMember.getExpiryDate());
    System.out.println(testMember.getStatus());
    System.out.println("--------------------");

    Admin testAdmin = new Admin("root", "admin", "ABRACADABRA");
    System.out.println("First Name: " + testAdmin.getFirstName());
    System.out.println("Last Name: " + testAdmin.getLastName());
    System.out.println("Exp. Date: " + testAdmin.getExpiryDate());
    System.out.println(testAdmin.getStatus());
    System.out.println("--------------------");

    User testUser = new User("Artie", "Smith");
    System.out.println("First Name: " + testUser.getFirstName());
    System.out.println("Last Name: " + testUser.getLastName());
    System.out.println("Exp. Date: " + testUser.getExpiryDate());
    System.out.println(testUser.getStatus());
  }
}
```

TRY IT

Directions: After entering the new code for the `MemberInheritance` driver class, run the program in Replit:

The results should look like this (though the dates will vary):



TRY IT

Directions: Let's comment out the getStatus() method from the User class in the User.java file so that it reads like this:

```
public class User extends Member {
  public User(String firstName, String lastName) {
    super(firstName, lastName);
  }

  /* @Override
  public String getStatus() {
    // Need to use inherited methods to get name
    return getFirstName() + " " + getLastName() + " is a User.";
  } */
}
```

TRY IT

Directions: Recompile the User class using this command:

EXAMPLE

```
javac User.java
```

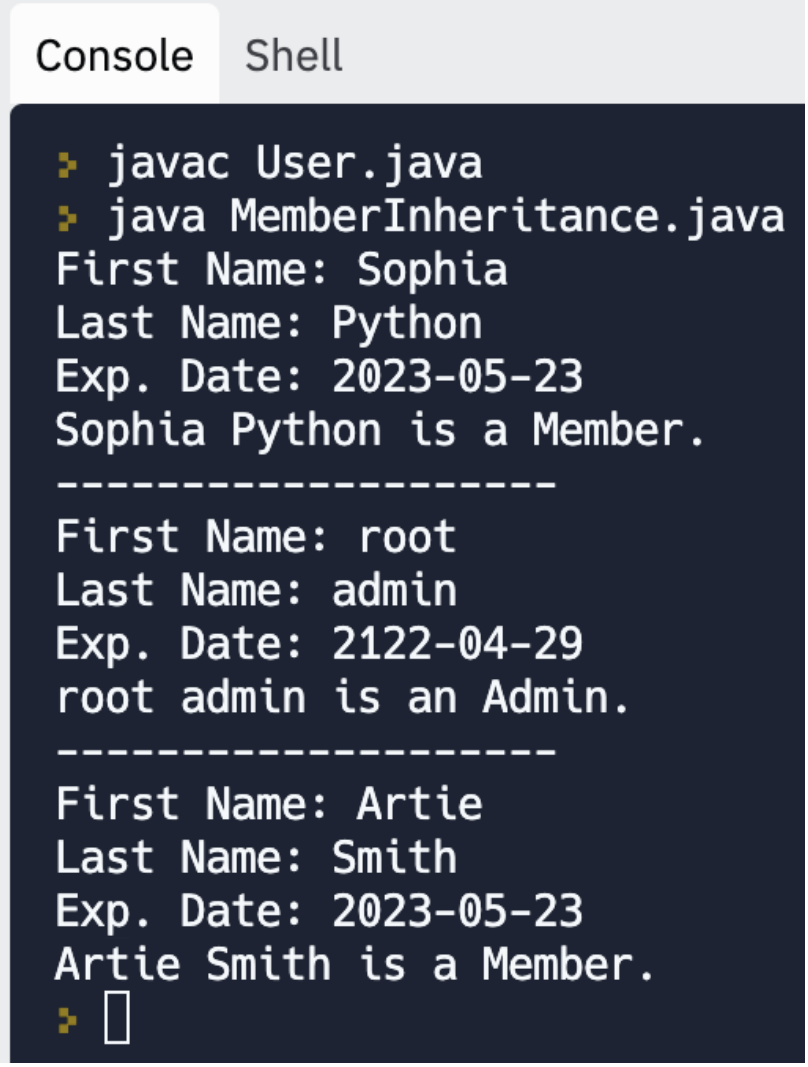
TRY IT

Directions: Run the program (MemberInheritance) like this:

EXAMPLE

```
java MemberInheritance.java
```

Notice that Artie Smith is showing the status of Member instead of User.



That's because the getStatus() method is no longer overridden in the User class. As such, the Java compiler has to look in the Member class (base class) and use the one found there.

BRAINSTORM

Directions: Try removing the method from the "User" class (or comment it out) and see if you get the same output (though the dates will vary).

TERM TO KNOW

super()
The super() method allows the constructor in a subclass to call the constructor for the base class.

SUMMARY

In this lesson, you learned that subclasses do not only have to use the methods that they inherit from the base class. **Subclasses can be customized** or extended from the base class. To do this, subclasses can have their own constructors. With the use of the super() method, a subclass can call the base class's constructor. You were able to build out subclasses with extra attributes and methods. We also learned that if a method is the same in the base class and subclasses, Java will use the most specific one that's tied to the subclass. It will use the base class method if nothing in that subclass has that method name.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from "Python for Everybody" By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

super()
The super() method allows the constructor in a subclass to call the constructor for the base class.

Introduction to Interfaces

by Sophia

WHAT'S COVERED

In this lesson, you will learn about relating groups of classes using interfaces, when using Java.

1. Class Interfaces

In Java and other programming languages, the way that an object interacts with other objects is referred to as its **interface**. For instance, a class's public and protected methods and attributes define the class's programming interface.

There is also a specific Java programming construct called an interface. This construct is designed to allow the way a class interacts with other objects to be abstracted and separate from the specifics of the implementation. An interface can be thought of as a programming contract that specifies behaviors that a class will make available when it implements an interface.

TERM TO KNOW

Interface
The defined ways in which one object interacts with another.

---

2. Defining an Interface

A Java interface is similar to a Java class, but they lack attributes or instance variables. An interface consists of methods, but most of the time these methods lack bodies (implementations) and the interface just specifies the signature and return type of the methods. This allows an interface to specify what classes that implement the interface can do, but not how they do it.

Below is an example of an interface that specifies some common methods (behavior) that classes using it must implement. This example also includes a constant DecimalFormat that all classes implementing the interface can use.

Just like a Java class, an interface needs to be declared in a .java file with a name matching the name of the interface (in this case, Beverage.java):

```
import java.text.DecimalFormat; // Needed for DecimalFormat for price

public interface Beverage {
  // Shared constant for price format. Available to all classes
  // that implement the Beverage interface.
  final DecimalFormat PriceFormat = new DecimalFormat("$0.00");
```

```
  // Common methods
  String getName();  // Coffee, Tea, etc...
  int getSize();     // Size in ounces
  double getPrice(); // Price in dollars
}
```

While the Beverage interface does not specify how the classes have to do it, any class that implements Beverage must have a method called `getName()` that returns a String, a method named`getSize()` that returns an integer, and a method called `getPrice()` that returns a double. Exactly how these methods do their work is up to the classes implementing the interface, but the interface is a kind of contract that stipulates that these methods are available and returns the data types specified (none of these methods have any parameters). The DecimalFormat constant named PriceFormat can also be used by any class that implements the Beverage interface.

The Beverage interface, like a Java class, needs to be compiled before it can be used (implemented) by other code:

↱ EXAMPLE

```
javac Beverage.java
```

---

3. Implementing an Interface

A Java interface by itself does not do anything. It engages when classes explicitly implement it. Here are a couple examples of classes that implement the Beverage interface. First, the `Coffee` class, which needs to be in a file called Coffee.java:

```
public class Coffee implements Beverage {
  private String roastType;
  private int size;
  private boolean decaf;
  private double price;

  public Coffee(String roastType, int size, boolean decaf, double price) {
      this.roastType = roastType;
      this.size = size;
      this.decaf = decaf;
      this.price = price;
  }

  // Required by Beverage interface
  public String getName() {
    return "coffee";
  }

  // Required by Beverage interface
  public int getSize() {
    return size;
```

```
  }

  // Required by Beverage interface
  public double getPrice() {
    return price;
  }

  public boolean isDecaf() {
    return decaf;
  }

  @Override // toString inherited from Java Object base class
  public String toString() {
    // PriceFormat is constant provided by Beverage interface
    String item = roastType + " coffee (" + size + " oz.) " + PriceFormat.format(price);
    if(decaf) {
      return "decaf " + item;
    }
    else {
      return item;
    }
  }
}
```

The Coffee class has the attributes and parameterized constructor that you might expect for such a class. Note, too, how the Coffee class has specific **implementations** for the getName(), getSize(), and getPrice() methods, along with the accessor methods for the attributes that are specific to coffee. The Beverage interface provides the PriceFormat that is used as part of the toString() method.

The Tea class also implements the Beverage interface:

```
public class Tea implements Beverage {
  private String teaType;
  private int size;
  private boolean iced;
  private double price;

  public Tea(String teaType, int size, boolean iced, double price) {
    this.teaType = teaType;
    this.size = size;
    this.iced = iced;
    this.price = price;
  }

  public String getName() {
    return "tea";
  }

  public int getSize() {
    return size;
  }
```

```
  public double getPrice() {
    return price;
  }

  public boolean isIced() {
    return iced;
  }

  public String toString() {
    String item = teaType + " tea (" + size + " oz.) " + PriceFormat.format(price);
    if(iced) {
      return "iced " + item;
    }
    else {
      return "hot " + item;
    }
  }
}
```

TERM TO KNOW

Implementation
The specific code written to produce a method that carries out the action described or specified in an interface.

---

4. Using an Interface as a Type

Classes that implement the same interface can be treated as having the same data type. This means that you can write methods that take classes that implement an interface as the same type for parameters or return values.

Here is a sample class called DrinkOrder that shows how Beverage can be used as a data type with a collection or with methods:

TRY IT

Directions: Enter this code in Replit:

```
import java.util.ArrayList;

public class DrinkOrder {
  // ArrayList can hold both Coffee & Tea objects since both implement the
  // Beverage interface
  private ArrayList<Beverage> order = new ArrayList<>();

  // add() method accepts an object any class that implements Beverage interface
  public void add(Beverage beverage) {
    order.add(beverage);
  }
```

```
  // Add up total for order
  public double getTotalPrice() {
    double total = 0;
    for(Beverage beverage : order) {
      total += beverage.getPrice();
    }
    return total;
  }

  // Return ArrayList of Beverages in order
  public ArrayList<Beverage> getOrder() {
    return order;
  }
}
```

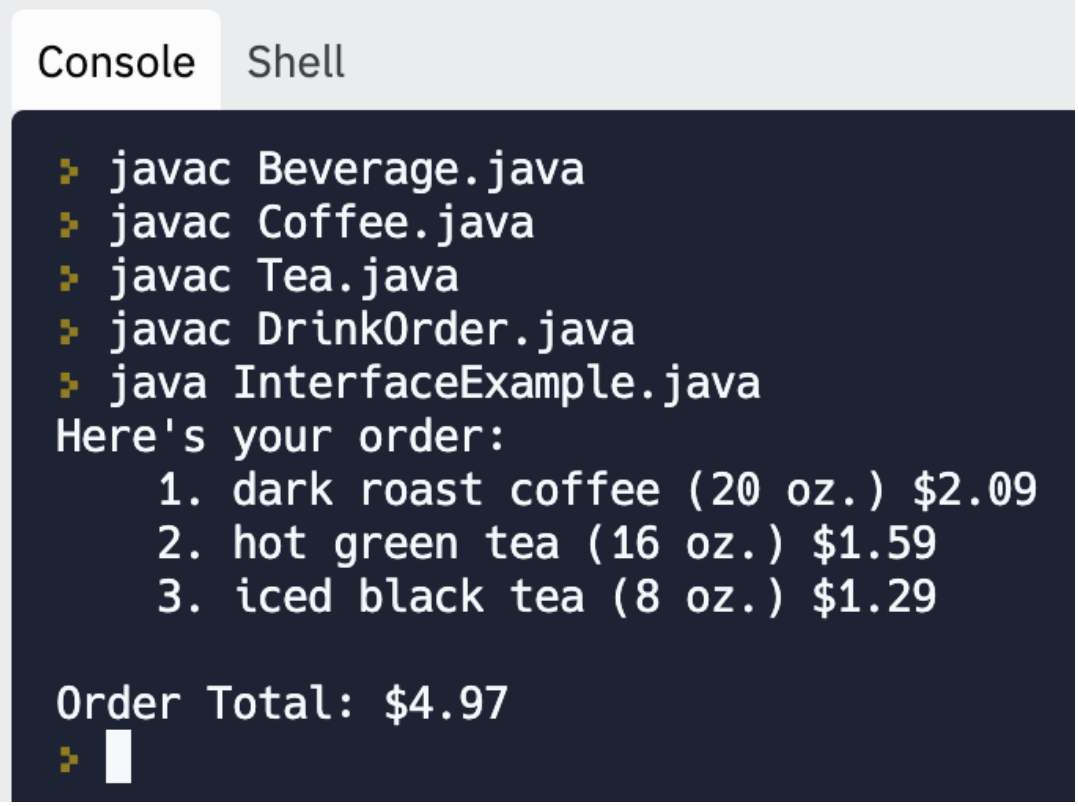
Here is a fairly simple program that uses this class (we'll leave out the user input and menu prompts that you may remember from an earlier drink order program).

TRY IT

Directions: Enter this code in Replit:

```
class InterfaceExample {
  public static void main(String[] args) {
    // Create order and add drinks
    DrinkOrder toGoOrder = new DrinkOrder();
    Coffee coffee = new Coffee("dark roast", 20, false, 2.09);
    toGoOrder.add(coffee);
    Tea greenTea = new Tea("green", 16, false, 1.59);
    toGoOrder.add(greenTea);
    Tea blackTea = new Tea("black", 8, true, 1.29);
    toGoOrder.add(blackTea);

    // Number to use for numbered list of drinks in order
    int itemNumber = 1;
    System.out.println("Here's your order: ");
    // getOrder() returns ArrayList<Beverage>
    for(Beverage bev : toGoOrder.getOrder()) {
      System.out.println("\t" + itemNumber++ + ". " + bev);
    }

    System.out.println("\nOrder Total: " +
        // Note how Beverage.PriceFormat can be used here, too
        Beverage.PriceFormat.format(toGoOrder.getTotalPrice()));
    }
}
```

The output from running this program (after compiling the interface and needed classes, if not already done) should look like this:



SUMMARY

In this lesson, you have learned how a Java **interface can be defined** as a kind of contract that specifies what behaviors (methods) and sometimes attributes a class will include when it implements a **class interface**. You learned that an interface specifies how classes can interact without specifying the internal details. You also learned that interfaces can serve as a common type for the classes that **implement the interface**. Finally, you learned that an**interface can be used with collections, as a method parameter type**, and as a return type.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Implementation
The specific code written to produce a method that carries out the action described or specified in an interface.

Interface
The defined ways in which one object interacts with another.

Composition

by Sophia

WHAT'S COVERED

In this lesson, you will learn ways that Java classes can be used as components of other classes. You will also learn about building classes that make use of other classes as a way to add common functionality without using inheritance. Specifically, this lesson covers:

1. Aggregation

Aggregation is the combining of Java objects into another class. For instance, in the lesson on interfaces, we worked with a DrinkOrder class that consisted of one or more Beverage objects in an ArrayList collection.

Just as a reminder, here is the DrinkOrder class:

```
import java.util.ArrayList;

public class DrinkOrder {
  // ArrayList can hold both Coffee & Tea objects since both implement the
  // Beverage interface
  private ArrayList<Beverage> order = new ArrayList<>();

  // add() method accepts an object any class that implements Beverage interface
  public void add(Beverage beverage) {
    order.add(beverage);
  }

  // Add up total for order
  public double getTotalPrice() {
    double total = 0;
    for(Beverage beverage : order) {
      total += beverage.getPrice();
    }
    return total;
  }

  // Return ArrayList of Beverages in order
  public ArrayList<Beverage> getOrder() {
    return order;
  }
}
```

Aggregation can be thought of as a structure where one class "has one or more" objects of another class. In the DrinkOrder<c/ode> example, a <code>DrinkOrder had one or more Beverage objects (which could have

been coffee or tea). This "ownership" was just one way, though. A `Coffee` or `Tea` object could not really be said to have a DrinkOrder. The objects in an aggregation are relatively independent in that they could exist or be useful even if not part of the larger object.

TERM TO KNOW

Aggregation
Aggregation is the combining of Java classes to create another class (instances of the combined classes are used as attributes in the new class).

---

2. Composition

Composition is a more specific type of aggregation. In the case of **composition**, the ties between a component object and the larger object are much tighter, so that the component can't really exist apart from the larger object. If the larger containing object is deleted or destroyed, the objects inside it can practically be said to go out of existence. Think of the rooms in a house or the sections of a book as examples. If a house is destroyed, the rooms don't have separate existences. The same can be said of the parts of a book. If the book is shredded, the table of contents, chapters, and bibliography also go out of existence (at least practically).

Let's apply the concept of composition to the Coffee and Tea classes that we have seen previously. The Beverage class encapsulates the data about the name of the drink, the size, and price. This Beverage data is part of the information about coffee and tea, but a Beverage object doesn't really exist apart from a specific drink (a Coffee or Tea object).

TRY IT

Directions: Enter this code for the Beverage class (in Beverage.java) in Replit:

```
import java.text.DecimalFormat;

public class Beverage {
  private String name;
  private int size;
  private double price;
  // The DecimalFormat for the price format is a rare public attribute, but since it's
  // final, the object is constant so it can't be modified (even though it's public)
  public final DecimalFormat PriceFormat = new DecimalFormat("$0.00");

  Beverage(String name, int size, double price) {
    this.name = name;
    this.size = size;
    this.price = price;
  }

  public String getName() {
    return name;
  }

  public int getSize() {
    return size;
```

```
  }

  public double getPrice() {
    return price;
  }
}
```

TRY IT

Directions: Enter this code for Embeds a Beverage object (saved in a file named Coffee.java) in Replit:

```
public class Coffee {
  // Embedded Beverage object
  private Beverage beverage;
  private String roastType;
  private boolean decaf;

  public Coffee(String roastType, int size, boolean decaf, double price) {
    // Construct the embedded Beverage object
    this.beverage = new Beverage("coffee", size, price);
    this.roastType = roastType;
    this.decaf = decaf;
  }

  // The Coffee class's getName() calls the embedded object's getName()
  public String getName() {
    return beverage.getName();
  }

  // The Coffee class's getSize() calls the embedded object's getSize()
  public int getSize() {
    return beverage.getSize();
  }

  // The Coffee class's getPrice() calls the embedded object's getPrice()
  public double getPrice() {
    return beverage.getPrice();
  }

  public String getRoastType() {
    return roastType;
  }

  public boolean isDecaf() {
    return decaf;
  }

  public String toString() {
    // Calls to accessor methods to access data in embedded Beverage object
    String item = roastType + " coffee (" + getSize() + " oz.) " + beverage.PriceFormat.format(getPrice());
    if(decaf) {
```

```
      return "decaf " + item;
    }
    else {
      return item;
    }
  }
}

public class Tea {
  private Beverage beverage;
  private String teaType;
  private boolean iced;

  public Tea(String teaType, int size, boolean iced, double price) {
    // Construct the embedded Beverage object
    this.beverage = new Beverage("tea", size, price);
    this.teaType = teaType;
    this.iced = iced;
  }

  // The Tea class's getName() calls the embedded object's getName()
  public String getName() {
    return beverage.getName();
  }

  // The Tea class's getSize() calls the embedded object's getSize()
  public int getSize() {
    return beverage.getSize();
  }

  // The Tea class's getPrice() calls the embedded object's getPrice()
  public double getPrice() {
    return beverage.getPrice();
  }

  public String getTeaType() {
    return teaType;
  }

  public boolean isIced() {
    return iced;
  }

  public String toString() {
    // Note use of accessor methods that forward calls to get info from embedded object
    String item = teaType + " tea (" + getSize() + " oz.) " + beverage.PriceFormat.format(getPrice());
    if(iced) {
      return "iced " + item;
    }
    else {
      return "hot " + item;
```

```
    }
  }
}
```

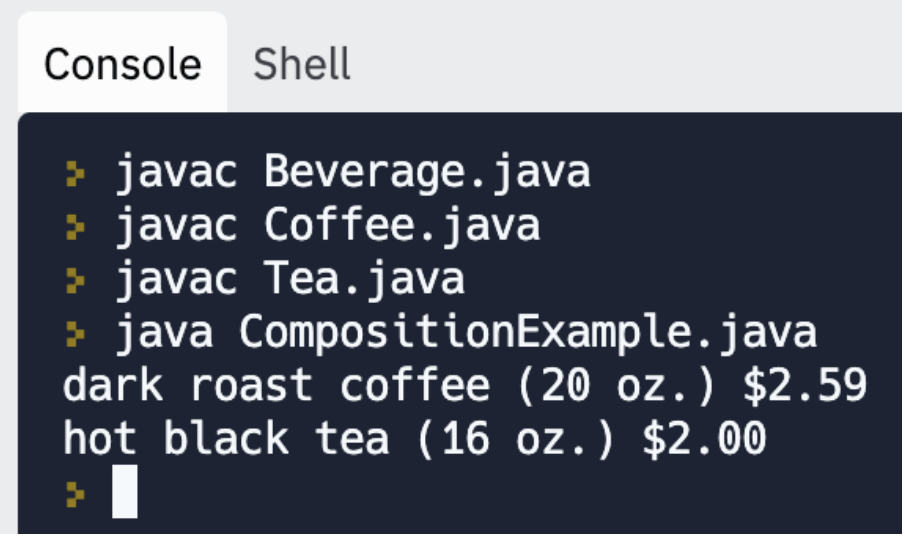
Here is a simple program for testing this version of the `Coffee` and `Tea` classes. Save this code in a file named CompositionExample.java.

TRY IT

Directions: Save this code in a file named CompositionExample.java:

```
class CompositionExample {
  public static void main(String[] args) {
    // Construct Coffee & Tea objects that have a Beverage objet embedded in them.
    // There's nothing unusual about the constructor calls.
    Coffee darkRoastCoffee = new Coffee("dark roast", 20, false, 2.59);
    Tea blackTea = new Tea("black", 16, false, 2.00);
    // Print out information about the drinks using their toString() methods
    System.out.println(darkRoastCoffee);
    System.out.println(blackTea);
  }
}
```

The output for this code looks like this:



REFLECT

Note how the code in the application's `main()` isn't visibly different, even though the code in the classes is different. This allows the programmer some latitude in designing a class without having an effect on how the class is used in other code.

TERM TO KNOW

Composition

Composition is a more specific type of aggregation where the component can't really exist apart from the larger object.

SUMMARY

In this lesson, you learned about approaches using **aggregation** and **composition** to class design that embed one object in another object. You also learned about using embedded objects as another way to provide common pieces of functionality, similar to how inheritance can be used to provide common functionality.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Aggregation

Aggregation is the combining of Java classes to create another class (instances of the combined classes are used as attributes in the new class).

Composition

Composition is a more specific type of aggregation where the component can't really exist apart from the larger object.

Debugging Inheritance

by Sophia

WHAT'S COVERED

In this lesson, you will learn about a couple of common errors in Java classes that make use of inheritance. Specifically, this lesson covers:

1. Common Errors With Inheritance

As we have seen, the base class does not differ visibly from any other class. This is not true of a subclass, though. Remember that the first line of a subclass must include the keyword `extends`, and this must be followed by the name of the base class as demonstrated below:

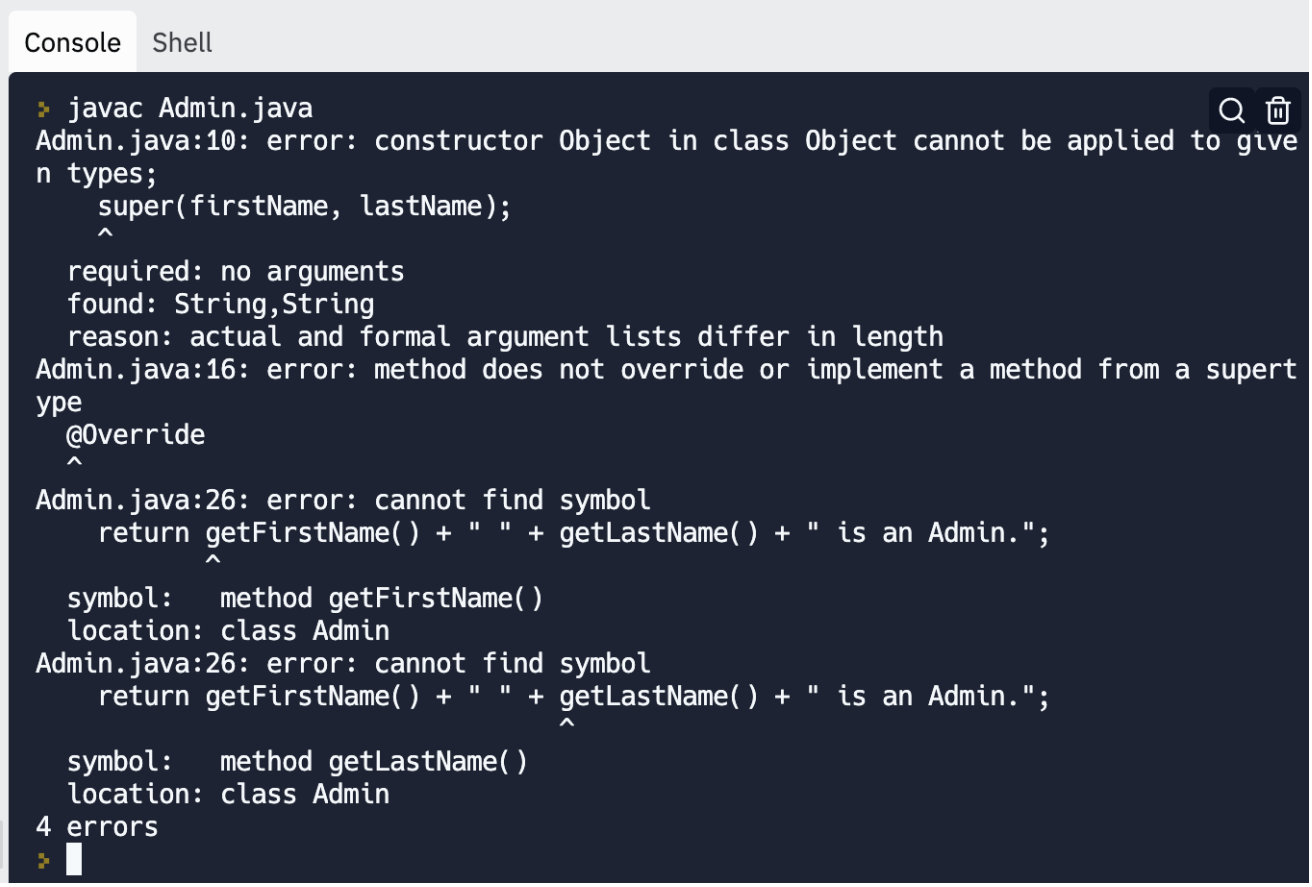
```
public class Admin extends Member{
```

If the `extends Member` is left out so that the `Admin` class looks like this, the code won't compile.

```
import java.time.LocalDate;

// Subclass of Member for administrators
public class Admin {
  private int expiryDays = 100 * 365;
  private LocalDate expiryDate;
  private String secret;

  public Admin(String firstName, String lastName, String secret) {
    super(firstName, lastName);
    expiryDate = LocalDate.now().plusDays(expiryDays);
    this.secret = secret;
  }

  @Override
  public LocalDate getExpiryDate() {
    return expiryDate;
  }

  public String getSecret() {
    return secret;
  }

  public String getStatus() {
    return getFirstName() + " " + getLastName() + " is an Admin.";
```

```
  }
}
```

The compiler output looks like this:



Since the `Admin` class does not explicitly extend the `Member` class, the compiler assumes that the call to `super()` refers to the Java **Object class**, since all classes are subclasses of `Object`, but an `Object` instance can't be created using the arguments passed. The compiler can't process the `@Override` annotation because there is nothing to override.

The error message also indicates that the compiler can't find the `getFirstName()` and `getLastName()` methods. These don't exist in the `Admin` class because it is supposed to inherit them from the `Member` base class. The error message doesn't state that the extends `Member` has been left out, but the clues indicate that the `Admin` class needs to extend a class other than `Object` and that the class to be extended must include the `getFirstName()` and `getLastName()` methods.

When working with a subclass, it is important to remember that the subclass's constructor must call `super()` to invoke the base class's (or superclass's) constructor. This call must include arguments that match the superclass's constructor.

Let's look at some code that you have worked with previously.

Here is the Member base class:

```
import java.time.LocalDate;

public class Member {
  private String firstName;
  private String lastName;
```

```
  private int expiryDays = 365;
  private LocalDate expiryDate;

  public Member(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    expiryDate = LocalDate.now().plusDays(expiryDays);
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public LocalDate getExpiryDate() {
    return expiryDate;
  }

  public String getStatus() {
    return firstName + " " + lastName + " is a Member.";
  }
}
```

Remember that the base class does not have any features that distinguish it from other classes. This is not true of a subclass, though, which is marked by the `extends` keyword between the name of the subclass and the base class:

```
import java.time.LocalDate;

// Subclass of Member for administrators
public class Admin extends Member{
  private int expiryDays = 100 * 365;
  private LocalDate expiryDate;
  private String secret;

  public Admin(String firstName, String lastName, String secret) {
    super(firstName, lastName);
    expiryDate = LocalDate.now().plusDays(expiryDays);
    this.secret = secret;
  }

  @Override
  public LocalDate getExpiryDate() {
    return expiryDate;
  }

  public String getSecret() {
```

```
  return secret;
 }

 public String getStatus() {
  return getFirstName() + " " + getLastName() + " is an Admin.";
 }
}
```

Note that the code in the subclass's constructor must call the base class's constructor via `super()` (with arguments to match the base class's arguments). This must be the first statement in the subclass's constructor. If the call to `super()` doesn't come first, the subclass won't compile. If the `Admin()` constructor's statements are ordered like this:

↗ EXAMPLE

```
public Admin(String firstName, String lastName, String secret) {
 expiryDate = LocalDate.now().plusDays(expiryDays);
 this.secret = secret;
 super(firstName, lastName); // In wrong place - should be first statement
}
```

The compiler will produce an error message like this:

Console Shell

```
> java EmployeeProgram.java
First Name: Jack
Last Name: Krichen
EmplId: 1000
Job Title: Manager
Salary: $75000.00
Vacation Days: 14
Taking 10 days of vacation...
Vacation Days: 4
Taking 10 more days of vacation...
Employee doesn't have sufficient vacation to take 10 days off.
Taking -1 days of vacation...
Requested vacation days must > 0
Increasing vacation days remaining...
Vacation Days: 18
>
```

This output may seem a bit hard to understand, but in the final couple of lines it indicates that the call to `super()` with the appropriate arguments must be the first statement in the constructor.

TERM TO KNOW

Object Class

Every class in Java automatically inherits from the Object class, so it is the base class for all other Java classes.

SUMMARY

In this lesson, you learned about **common errors when working with inheritance**. You learned how to recognize them when confronted with compiler errors, which are not always so clear. Finally, you learned about correct uses of the extends keyword when defining a subclass and that getting the order of statements in the constructor incorrect is an error that you can easily avoid.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Object Class

Every class in Java automatically inherits from the Object class, so it is the base class for all other Java classes.

Revisiting the Employee Class Program

by Sophia

WHAT'S COVERED

In this lesson, you will be extending on the Employee class to create subclasses. Specifically, this lesson covers:

1. Creating the Base Class Person

In a prior tutorial, we had created a basic Employee class that looks like the following:

```
import java.text.DecimalFormat;
import java.time.LocalDate;

public class Employee {
  private String firstName;
  private String lastName;
  private int emplId;
  private String jobTitle;
  private double salary;
  private LocalDate hireDate;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle,
          double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.emplId = emplId;
    this.jobTitle = jobTitle;
    this.salary = salary;
    this.hireDate = LocalDate.now();
  }

  // Returns the first name
  public String getFirstName() {
    return firstName;
  }

  // Sets the value of attribute firstName to value passed as parameter firstName
  public void setFirstName(String firstName) {
    if(firstName.length() > 0) {
      this.firstName = firstName;
```

```
  }
}

public String getLastName() {
  return lastName;
}

public void setLastName(String lastName) {
  if(lastName.length() > 0) {
    this.lastName = lastName;
  }
}

public String getJobTitle() {
  return jobTitle;
}

public void setJobTitle(String jobTitle) {
  if(jobTitle.length() > 0) {
    this.jobTitle = jobTitle;
  }
}

public double getSalary() {
  return salary;
}

public void setSalary(double salary) {
  if(salary > 0.00) {
    this.salary = salary;
  }
}

public String getSalaryAsString() {
    // Format salary with leading dollar sign and 2 decimal places
    DecimalFormat salaryFormat = new DecimalFormat("$0.00");
    // Use getSalary to get numeric value and then format
    return salaryFormat.format(getSalary());
}

// EmplId cannot be changed, so there is only accessor, no mutator method
public int getEmplId() {
  return emplId;
}

// Method to increase salary by percent as decimal. 0.02 is a 2% raise
public void increaseSalary(double percentAsDecimal) {
  if(percentAsDecimal > 0.0) {
    salary *= (1 + percentAsDecimal);
  }
}
```

```
}
```

TRY IT

Directions: If you don't already have this in Replit, enter it into a file named Employee.java since we will be modifying this example with an updated base class and new subclasses.

Although you have Employee as the prior class, you will want to consider other aspects about an employee. For example, you can have different types of employees. You could have full-time and part-time employees that get vacation hours and an annual salary, as we currently have in our “Employee” class. You could also have contractors that get an hourly wage but don’t accumulate vacation time or have an annual salary. Contractors could also have a contractor ID rather than an employee ID.

CONCEPT TO KNOW

In order to build a correctly defined base class, you need to pull in only the key information that would be consistent across both the contractor and employee classes. In our next example, we will define the base class as Person and only place in what is common. The Person class needs to be entered into a file named Person.java.

By removing all items related to salary and employee ID, you will have the following result:

```
import java.time.LocalDate;

public class Person {
  private String firstName;
  private String lastName;
  private String jobTitle;
  private LocalDate hireDate;

  // Parameterized constructor
  public Person(String firstName, String lastName, int empId, String jobTitle) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.jobTitle = jobTitle;
    this.hireDate = LocalDate.now();
  }

  // Returns the first name
  public String getFirstName() {
    return firstName;
  }

  // Sets the value of attribute firstName to value passed as parameter firstName
  public void setFirstName(String firstName) {
    if(firstName.length() > 0) {
      this.firstName = firstName;
    }
  }
```

```
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    if(lastName.length() > 0) {
      this.lastName = lastName;
    }
  }

  public String getJobTitle() {
    return jobTitle;
  }

  public void setJobTitle(String jobTitle) {
    if(jobTitle.length() > 0) {
      this.jobTitle = jobTitle;
    }
  }
}
```

TRY IT

Directions: Remove all aspects of salary and employee ID information from this updated base class called `Person` now. Make sure your program looks like the code above. Compile the `Person` class using this command:

EXAMPLE

```
javac Person.java
```

Now, with the `Person` base class, we are set up to create our unique subclasses.

REFLECT

Before we move on to classes that inherit from `Person`, it is important to note how the `Person` class (which will be our base or parent class) encapsulates data and functionality that will be common to all people working for a given company. This will help us avoid redundancy in the subclasses, and a good design avoids redundancy whenever possible.

---

2. Creating the Subclass Employee

In the `Person` base class, we have the common information defined as `firstName`, `lastName`, `jobTitle`, and the `hireDate` attributes. Now that you have that content in place, you can create the `Employee` subclass to extend the `Person` base class and have the custom content that makes it unique. You will be using most of the elements that you had in the prior base class that we initially set up.

The declaration of the Employee class indicates that it inherits from the `Person` class (using the `extends` keyword):

```
import java.text.DecimalFormat;

public class Employee extends Person {
  private int emplId;
  private double salary;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle, double salary) {
    super(firstName, lastName, jobTitle);
    this.emplId = emplId;
    this.salary = salary;
  }

  public double getSalary() {
    return salary;
  }

  public void setSalary(double salary) {
    if(salary > 0.00) {
      this.salary = salary;
    }
  }

  public String getSalaryAsString() {
      // Format salary with leading dollar sign and 2 decimal places
      DecimalFormat salaryFormat = new DecimalFormat("$0.00");
      // Use getSalary to get numeric value and then format
      return salaryFormat.format(getSalary());
  }

  // EmplId cannot be changed, so there is only accessor, no mutator method
  public int getEmplId() {
    return emplId;
  }

  // Method to increase salary by percent as decimal. 0.02 is a 2% raise
  public void increaseSalary(double percentAsDecimal) {
    if(percentAsDecimal > 0.0) {
      salary *= (1 + percentAsDecimal);
    }
  }
}
```

Note how the `Employee()` constructor has parameters that include the values needed to be passed to the

constructor for the Person base class. The constructor for the base class (also called the "superclass") is called using super(). The call to super() passes the parameters needed by the superclass's constructor—in this case, the String values for first name, last name, employee ID, and salary. It's important that the first statement in the subclass's constructor is the call to super(). The Employee() constructor then sets the values for the emplId and salary attributes. These are the attributes that are specific to the Employee subclass.

TRY IT

Directions: Now type in the code for the Employee subclass in a file named Employee.java. Compile the Employee class by running this command:

EXAMPLE

```
javac Employee.java
```

Attributes vacationDaysPerYear/ and vacationDaysRemaining

We did say that employees should receive vacation days. Let's say by default for this organization, all employees have 14 days of vacation. Now it is helpful to have two different attributes for vacations—one for the yearly total (14) and one for the actual days remaining for the specific employee. You will add the field called vacationDaysPerYear and set it to 14. Next, we will create the field called vacationDaysRemaining which is set to vacationDaysPerYear as the default. We will update the constructor to include these attributes as follows:

```
public class Employee extends Person {
  private int emplId;
  private double salary;
  private int vacationDaysPerYear = 14;
  private int vacationDaysRemaining;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle, double salary) {
    super(firstName, lastName, emplId, jobTitle);
    this.salary = salary;
    vacationDaysRemaining = vacationDaysPeryear;
}
```

TRY IT

Directions: Go ahead and add these additional attributes and update the constructor in the Employee subclass:

EXAMPLE

```
method: increaseVacationDaysPerYear()
```

There will be a few methods that will be specific to vacations. One method will be to increase the vacation days per year. Typically, this could be increased by negotiation or based on how long the employee has been at the company.

```
// Increase vacation days per year
public void increaseVacationDaysPerYear(int days) {
  if(days > 0) {
    this.vacationDaysPerYear += days;
  }
}
```

We defined a method called increaseVacationDaysPerYear() with a parameter for the number of days to add. That way, you can pass in an integer to change the default vacation. Next, we have an if() statement that checks if the number passed is larger than 0. If it is, you add that value to the vacationDaysPerYear attribute.

Directions: Go ahead and add the increaseVacationDaysPerYear() method to the Employee subclass:

↗ EXAMPLE

method: increaseVacationDaysRemaining()

The next method you will add will be one that will increase the actual vacation days remaining if the added days were granted. You will again check if days is greater than 0, and if so, you can add to the existing attribute vacationDaysRemaining:

```
// Increase remaining vacation days
public void increaseVacationDaysRemaining(int days) {
  if(days > 0) {
    this.vacationDaysRemaining += days;
  }
}
```

Directions: Go ahead and add the `increaseVacationDaysRemaining()` method to the Employee subclass. We will need to have a method that we will use when an employee wants to take some days off. This method will accept the number of requested days off. As long as the value is greater than 0 and the employee still has days left that's greater than the days requested, it will be permitted. Otherwise, if the days requested is less than or equal to 0, meaning the employee entered 0, we will inform the employee that their request must be greater than 0.

```
 // Use vacation days
 public void takeVacationDays(int days) {
   if(days > 0 && vacationDaysRemaining >= days) {
     this.vacationDaysRemaining -= days;
```

```
  }
  else if(days <= 0) {
    System.out.println("Requested vacation days must > 0");
  }
  else {
    System.out.println("Employee doesn't have sufficient vacation to take " +
                  days + " days off.");
  }
}
```

Here, we have defined a method called `takeVacationDays()` with the parameter days that will accept the requested days off.

TRY IT

Directions: Go ahead and add this `takeVacationDays()` method to the Employee subclass:

EXAMPLE

method: getVacationDaysRemaining()

We'll also have a simple accessor method to return the number of vacation days.

```
// Return number vacation days remaining
public int getVacationDaysRemaining() {
  return vacationDaysRemaining;
}
```

This `getVacationDaysRemaining()` method will return the value of `vacationDaysRemaining`.

TRY IT

Directions: Add the `getVacationDaysRemaining()` method to the Employee subclass. Before we add some instance calls to test this subclass, make sure your program looks like the following:

```
import java.text.DecimalFormat;

public class Employee extends Person {
  private int emplId;
  private double salary;
  private int vacationDaysPerYear = 14;
  private int vacationDaysRemaining;

  // Parameterized constructor
  public Employee(String firstName, String lastName, int emplId, String jobTitle, double salary) {
    super(firstName, lastName, emplId, jobTitle);
```

```
  this.salary = salary;
  this.emplId = emplId;
  vacationDaysRemaining = vacationDaysPerYear;
}

public double getSalary() {
  return salary;
}

public void setSalary(double salary) {
  if(salary > 0.00) {
    this.salary = salary;
  }
}

public String getSalaryAsString() {
    // Format salary with leading dollar sign and 2 decimal places
    DecimalFormat salaryFormat = new DecimalFormat("$0.00");
    // Use getSalary to get numeric value and then format
    return salaryFormat.format(getSalary());
}

// EmplId cannot be changed, so there is only accessor, no mutator method
public int getEmplId() {
  return emplId;
}

// Method to increase salary by percent as decimal. 0.02 is a 2% raise
public void increaseSalary(double percentAsDecimal) {
  if(percentAsDecimal > 0.0) {
    salary *= (1 + percentAsDecimal);
  }
}

// Increase vacation days per year
public void increaseVacationDaysPerYear(int days) {
  if(days > 0) {
    this.vacationDaysPerYear += days;
  }
}

// Increase remaining vacation days
public void increaseVacationDaysRemaining(int days) {
  if(days > 0) {
    this.vacationDaysRemaining += days;
  }
}
// Use vacation days
public void takeVacationDays(int days) {
  if(days > 0 && vacationDaysRemaining >= days) {
    this.vacationDaysRemaining -= days;
```

```
    }
    else if(days <= 0) {
      System.out.println("Requested vacation days must > 0");
    }
    else {
      System.out.println("Employee doesn't have sufficient vacation to take " +
                  days + " days off.");
    }
  }

  // Return number vacation days remaining
  public int getVacationDaysRemaining() {
    return vacationDaysRemaining;
  }
}
```

REFLECT

Since the Employee subclass inherits from the Person class, it will inherit the public methods from the Person class. Think about which methods these are that are inherited from the Person class and how they provide functionality to the Employee class "for free." Note how the attributes and methods in the Employee class add to what is provided by the Person base class.

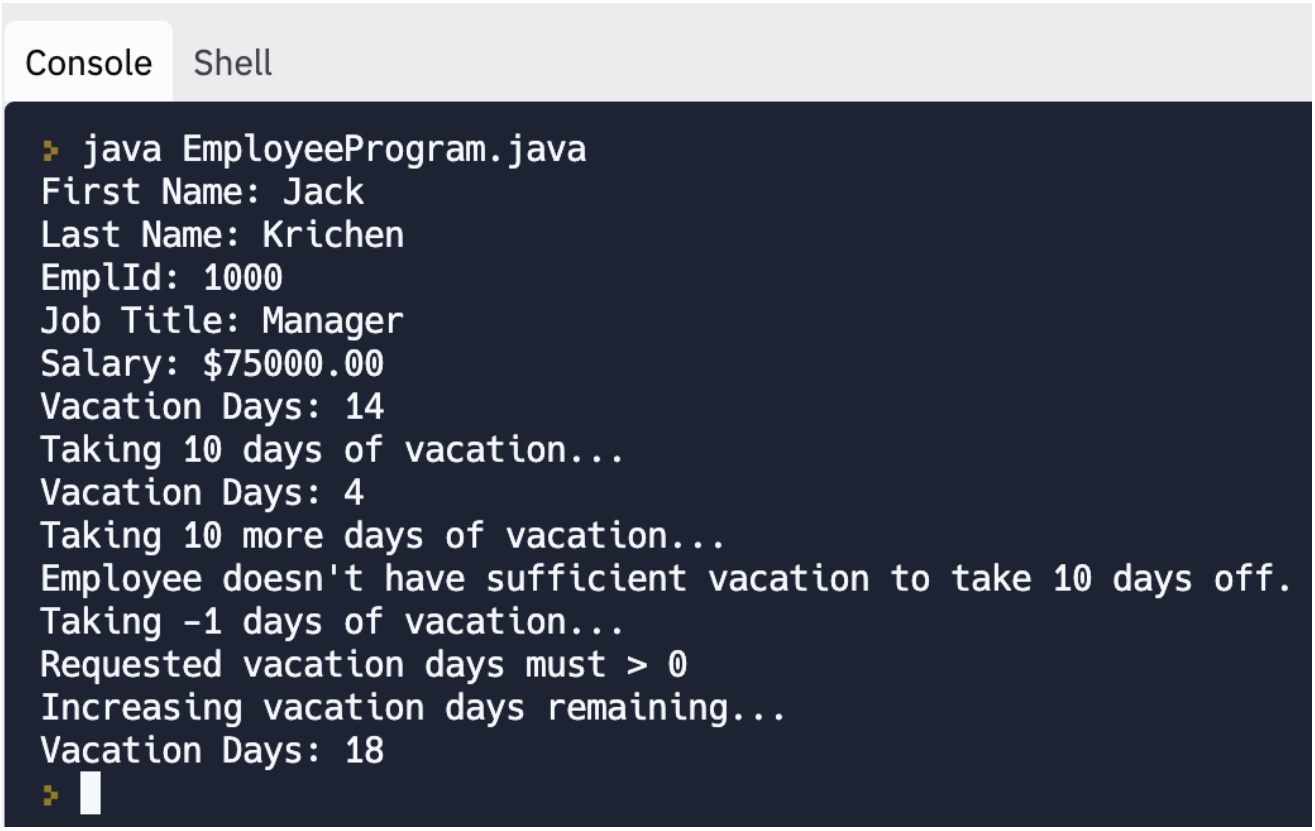
Let's test out the code, especially around the vacation days methods in the "Employee" subclass, to ensure all is working as expected. First, you will create an instance of the subclass called empl. You will pass some arguments for first name, last name, title, salary, and employee ID. Then, we will create some println() calls, so we can see output to the screen.

```
public class EmployeeProgram {
  public static void main(String[] args) {
    Employee empl = new Employee("Jack", "Krichen", 1000, "Manager", 75000);
    System.out.println("First Name: " + empl.getFirstName());
    System.out.println("Last Name: " + empl.getLastName());
    System.out.println("EmplId: " + empl.getEmplId());
    System.out.println("Job Title: " + empl.getJobTitle());
    System.out.println("Salary: " + empl.getSalaryAsString());
    // Now display vacation information
    System.out.println("Vacation Days: " + empl.getVacationDaysRemaining());
    System.out.println("Taking 10 days of vacation...");
    empl.takeVacationDays(10);
    System.out.println("Vacation Days: " + empl.getVacationDaysRemaining());
    System.out.println("Taking 10 more days of vacation...");
    empl.takeVacationDays(10);
    System.out.println("Taking -1 days of vacation...");
    empl.takeVacationDays(-1);
    System.out.println("Increasing vacation days remaining...");
    empl.increaseVacationDaysRemaining(14);
    System.out.println("Vacation Days: " + empl.getVacationDaysRemaining());
  }
}
```

TRY IT

Directions: Add the code above to your EmployeeProgram.java file. Give the employee a first name, last name, title, salary, and employee ID. To keep consistent with this example, test with the vacation days indicated. Once entered, run the program.

In the output, you should see the employee's first name, last name, employee ID, title, and salary for the first five `System.out.println()` calls.



REFLECT

Notice that the first output of the current vacation days is 14, which is correct since the attribute `vacationDaysRemaining` was initially set to the attribute `vacationDaysPerYear`, which has 14 as the default value. Then, we pass 10 vacation days as a request (argument) to the `takeVacationDays()` method and print out the value of vacationDaysRemaining once the subtraction is done, so 4 days left is also correct. We then try to take another 10 days of vacation; however, we get an error message since there aren't enough vacation days left (we only had 4 days left after the first request). Next, we try to take a negative number of vacation days, which also returns an error that the argument (request for days off) needs to be greater than 0. Lastly, we increase the vacation days based on the yearly increase and accurately see 18 days, as there were 4 days left and the yearly increase was 14 days.

BRAINSTORM

Directions: Now that you have a working "Employee" subclass, try changing a few arguments to see if you can change what is output to the screen.

---

3. Creating the Subclass Contractor

Our next step will be to create the "Contractor" subclass. The framework of this class will be the same structure

as the “Employee” subclass with some small differences. In particular, you will have a contractorid instead of the employeeid. There will also be an hourly wage instead of a salary, and no vacation.

```
public class Contractor extends Person {
  int contractorId;
  double hourlyWage;
  double totalWage;

  public Contractor(String firstName, String lastName, int contractorId, String jobTitle, double hourlyWage) {
    super(firstName, lastName, jobTitle);
    this.contractorId = contractorId;
    this.hourlyWage = hourlyWage;
  }

  public int getConstractorId() {
    return contractorId;
  }

  public double getHourlyWage() {
    return hourlyWage;
  }
  public void setHourlyWage(double hourlyWage) {
    if(hourlyWage > 0) {
      this.hourlyWage = hourlyWage;
    }
  }
}
```

Most of this should be quite familiar, as the coding structure is the same in this subclass, with some slight differences from the names of the attributes in the Employee subclass.

The `getContractorId()` method was modeled on the `getEmpId()` method. The `setHourlyWage()method` is based on the `setSalary()` method and `getHourlyWage()` is a version of the `getSalary()` method.

TRY IT

Directions: Enter the Contractor subclass in Replit in a file named Contractor.java. Compile the subclass using this command:

EXAMPLE

```
javac Contractor.java
```

Now, let's write some code to test the Contractor class and save it in a file named ContractorProgram.java:

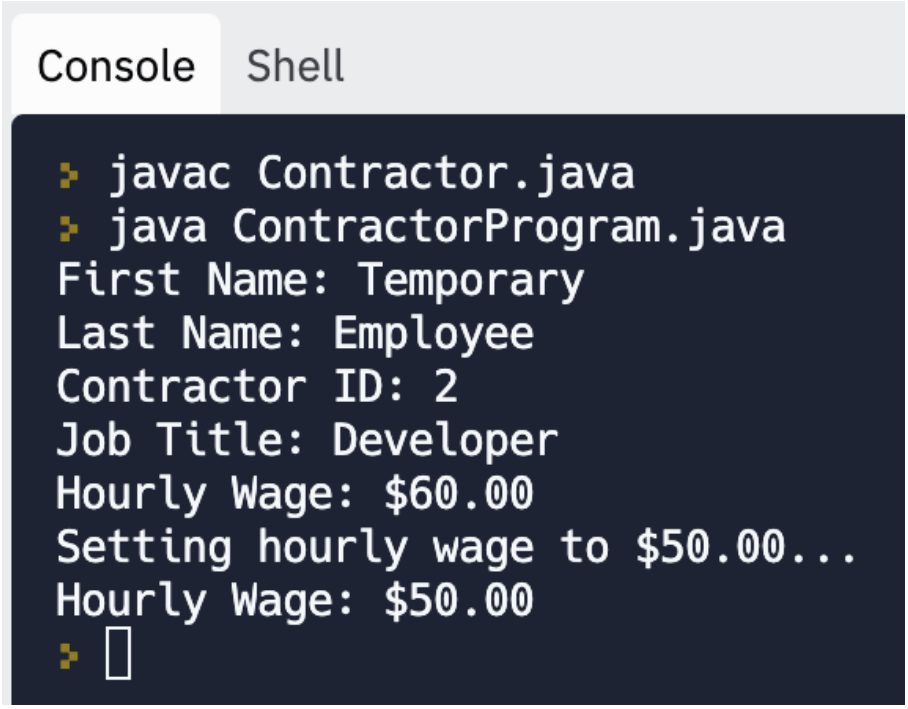
```
import java.text.DecimalFormat;

class ContractorProgram {
```

```
  public static void main(String[] args) {
    Contractor contractor = new Contractor("Temporary", "Employee", 2, "Developer", 60.00);
    System.out.println("First Name: " + contractor.getFirstName());
    System.out.println("Last Name: " + contractor.getLastName());
    System.out.println("Contractor ID: " + contractor.getConstractorId());
    System.out.println("Job Title: " + contractor.getJobTitle());
    DecimalFormat wageFormat = new DecimalFormat("$0.00");
    System.out.println("Hourly Wage: " + wageFormat.format(contractor.getHourlyWage()));
    System.out.println("Setting hourly wage to $50.00...");
    contractor.setHourlyWage(50.00);
    System.out.println("Hourly Wage: " + wageFormat.format(contractor.getHourlyWage()));
  }
}
```

TRY IT

Directions: Add the code needed to construct a Contractor object and display its information in a class named ContractorProgram (in a file named ContractorProgram.java) to your program. Give the contractor a first name, last name, title, hourly wage, and contractor ID. To keep consistent with this example, test with the hourly wage indicated. Once entered, run the program:

As we see, the output and contents are slightly different:



In the output, we should see the contractor's first name, last name, contractor ID, title, and hourly wage for the first five `System.out.println()` calls.

Then, we changed the hourly wage to $50 an hour using the`setHourlyWage()` method and reprinted the hourly wage again using the `getHourlyWage()` method.

THINK ABOUT IT

As you look at the code to test, think about how else you would test the code to ensure that it works correctly. What values would you try to set?

SUMMARY

In this lesson, you moved the standard attributes and methods to a **Person base class** that we wanted to exist globally. Then, we took the Employee specific attributes and methods and placed those into a new **Employee subclass**. We added methods to the “Employee” subclass to account for salary and vacation days, and tested our program for vacation requests against what an employee has in their current vacation bank. Finally, we **created the Contractor subclass** and introduced an hourly wage as opposed to a salary. In both subclasses, we added a unique ID method only associated with those subclasses.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

Introduction to Libraries

by Sophia

WHAT'S COVERED

In this lesson, you will review the use of Java libraries that you have already learned about. You will then learn about the use of a few new libraries that are part of the standard Java language. Specifically, this lesson covers:

1. Using import to Access Library Classes

From early in the course, you have been making use of Java's standard libraries. To begin working with a `Scanner` to read user input, you have needed to add a line like this to the code:

EXAMPLE

```
import java.util.Scanner;
```

This statement makes the `Scanner` class from the standard **library classes** available for use in the program. The import keyword is followed by the package `java.util`, which identifies the section of the `Java` library we are accessing, and the capitalized class name, `Scanner`, indicates the specific library class.

There are thousands of classes included in the libraries that come with Java. The **packages** are both a way of organizing the libraries and a way to avoid name collisions.

IN CONTEXT
You might think of the package name as being equivalent to an area code in a phone number. A number of people around the country (or the world) may have the local number 234-5678. To make sure the correct person is reached, the area code is included. The 212 area code reaches the person in Manhattan with the phone number 234-5678, while the 312 area code is used to reach the subscriber with that number in Chicago.

CONCEPT TO KNOW

Instead of specifying the specific class in a package that needs to be imported, we can use a wildcard character to import all classes in a package.

It is also possible to use the wildcard character * to specify that all classes in a package should be imported, as demonstrated in the following example:

EXAMPLE

```
import java.util.*;
```

This line imports the Scanner class along with everything else in the java.util package. It is sometimes the case that different packages may contain classes with the same name. In such cases, using the wildcard * to import all classes in a package may lead to "name collisions." It may not be clear which class is actually needed, so the compiler can't finish its work. In such cases, using more specific imports of specific classes can be useful.

TERMS TO KNOW

Package
A package is a collection of interrelated classes in a particular section of the Java class library.

Library Class
A class that is included in the libraries that come with the Java Virtual Machine and developer tools and is available on all machines running a given version of Java.

---

2. Working with Standard Library Classes and Libraries Used Already

Once a class (or all of the classes in a package) have been imported, the objects and methods provided by the class are available for use like local classes in the application. From early on in the course, you have made use of the java.util Scanner class (part of the java.util package).

↷ EXAMPLE A number of programs have featured this statement:

```
import java.util.Scanner;
```

In an earlier discussion of exception handling using try and catch, the code used a very general Exception in the catch clause (which did not require any further imports).

↷ EXAMPLE You might, though, have added the import for:

```
import java.util.InputMismatchException;
```

If you added the import, it would have been possible to catch a more specific exception.

Your program could look like this:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ScannerException {

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
```

```
    System.out.print("Please enter your age in years: ");
    int age = 0;
    try {
      age = input.nextInt();
    }
    catch(InputMismatchException ex) {
      System.out.println("You did not enter a valid integer.");
      System.out.println("Please run again & enter a valid age.");
    }
    System.out.println("You are " + age + " years old.");
  }
}
```

As noted in the previous lesson, this code could be simplified by one line if it used a wildcard when importing library classes, as demonstrated below:

→ EXAMPLE

```
import java.util.*;
```

We have also met the `java.util.Arrays` utility class when printing the contents of arrays, sorting them, and copying them. Note the use of the following line that you used earlier in the import class:

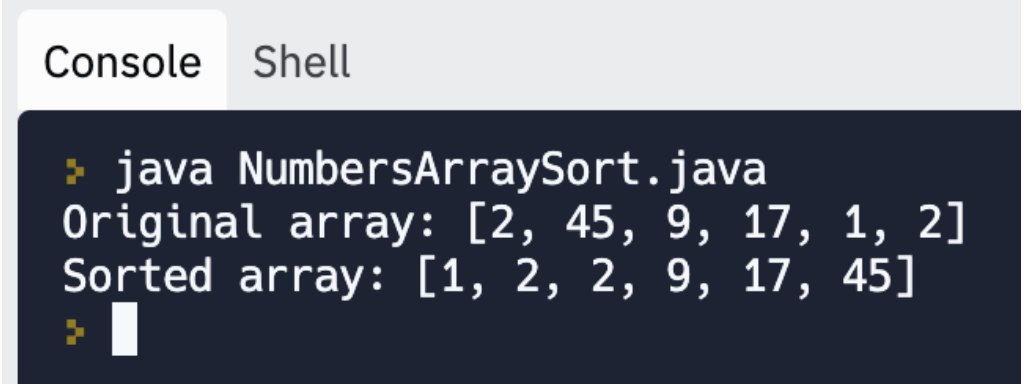
→ EXAMPLE

```
import java.util.Arrays;
```

As a reminder, here is a brief program that makes use of `Array.sort()` and `Arrays.toString()`.

```
import java.util.Arrays;

class NumbersArraySort {
  public static void main(String[] args) {
    int[] numArray = {2, 45, 9, 17, 1, 2};
    System.out.println("Original array: " + Arrays.toString(numArray));
    Arrays.sort(numArray);
    System.out.println("Sorted array: " + Arrays.toString(numArray));
  }
}
```

If you enter this code in Replit (in a file named NumbersArraySort.java) and run it, the results should look like this:



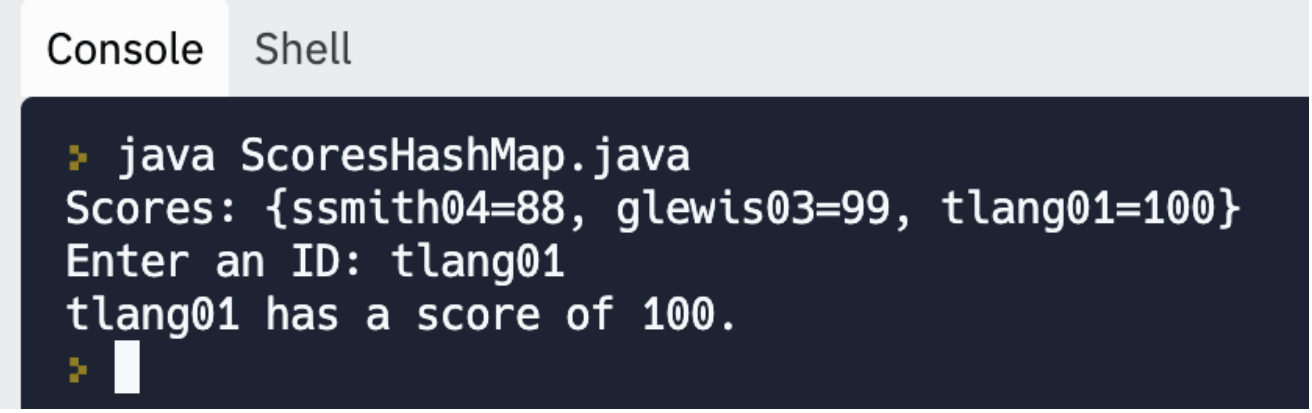
The Arrays class has a plural name to indicate that it includes utility methods that are useful when working with Java arrays. In the next section, we will be briefly introduced to a Files class that includes common utility methods for working with files.

The java.util library package also includes collections that we have worked with, such asArrayList, HashMap, and HashSet.

Here is an example that we saw back in lesson 2.1.5 that uses aHashSet, and so has to have the import statement for java.util.HashSet:

```
import java.util.HashMap;
import java.util.Scanner;

public class ScoresHashMap {

  public static void main(String[] args) {
    // HashMap holds key-value pairs.
    // The key (user ID) is a String (case sensitive).
    // The value (score) is an Integer (int)
    HashMap<String, Integer> scores = new HashMap<>();
    scores.put("ssmith04", 88);
    scores.put("tlang01", 100);
    scores.put("glewis03", 99);
    System.out.println("Scores: " + scores.toString());

    Scanner input = new Scanner(System.in);

    System.out.print("Enter an ID: ");
    String id = input.nextLine();
    // Check if the HashMap contains the key (id)
    if(scores.containsKey(id)) {
      // Only safe to use get() to retrieve value if key exists in HashMap
      int score = scores.get(id);
      System.out.println(id + " has a score of " + score + ".");
    }
    else {
      System.out.println("There is no score for " + id + ".");
    }
  }
}
```

If entered into a file named ScoresHashMap.java in Replit, the program should produce output like the following:



As the examples from earlier in the course show, the standard libraries included with Java have already been useful.

---

3. The java.io and java.nio Library Packages

In its history of more than 25 years, the Java libraries have evolved and changed. The devices that run Java have changed as well. The upcoming lessons will deal with file input and output. The libraries for working with files have changed over the years. To accommodate this growth and change, Java has ended up with classes for working with files that are organized into two library packages: `java.io` and `java.nio`.

The `java.io` package is the older of the two and goes back to the first version of Java. This package includes the File class that we will use in many of the upcoming programs.

The `java.io` package also provides important exception types when working with files:`FileNotFoundException` and `IOException`. Since the File object will be used for all of the code we will write for working with files, this library will play a key role.

BIG IDEA

The `java.nio` package provides the `Files` class. The plural name of this class indicates that it provides common utility methods for working with files. The functionality provided by java.nio provides simpler approaches to working with files that we will cover in the upcoming lessons. As you move further on in Java (after this course), you will want to look into older approaches to working with files using input and output streams, but for our purposes, the simpler approaches provided by `java.nio` `Files` will be ideal.

HINT

The File and Files classes are two different classes in different packages—even though the names are similar.

SUMMARY

In this lesson, you learned how to **access Java library packages and classes**. You learned how they are added to a program to be used. You have worked with classes provided by the **standard libraries** from early on in the course, by necessity. In this tutorial, you learned about a few common package examples. Finally, we looked ahead to a couple of key libraries that we will use for working with files in the upcoming lessons, which include **java.io** and **java.nio**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

TERMS TO KNOW

Library Class

A class that is included in the libraries that come with the Java Virtual Machine and developer tools and is available on all machines running a given version of Java.

Package

A package is a collection of interrelated classes in a particular section of the Java class library.

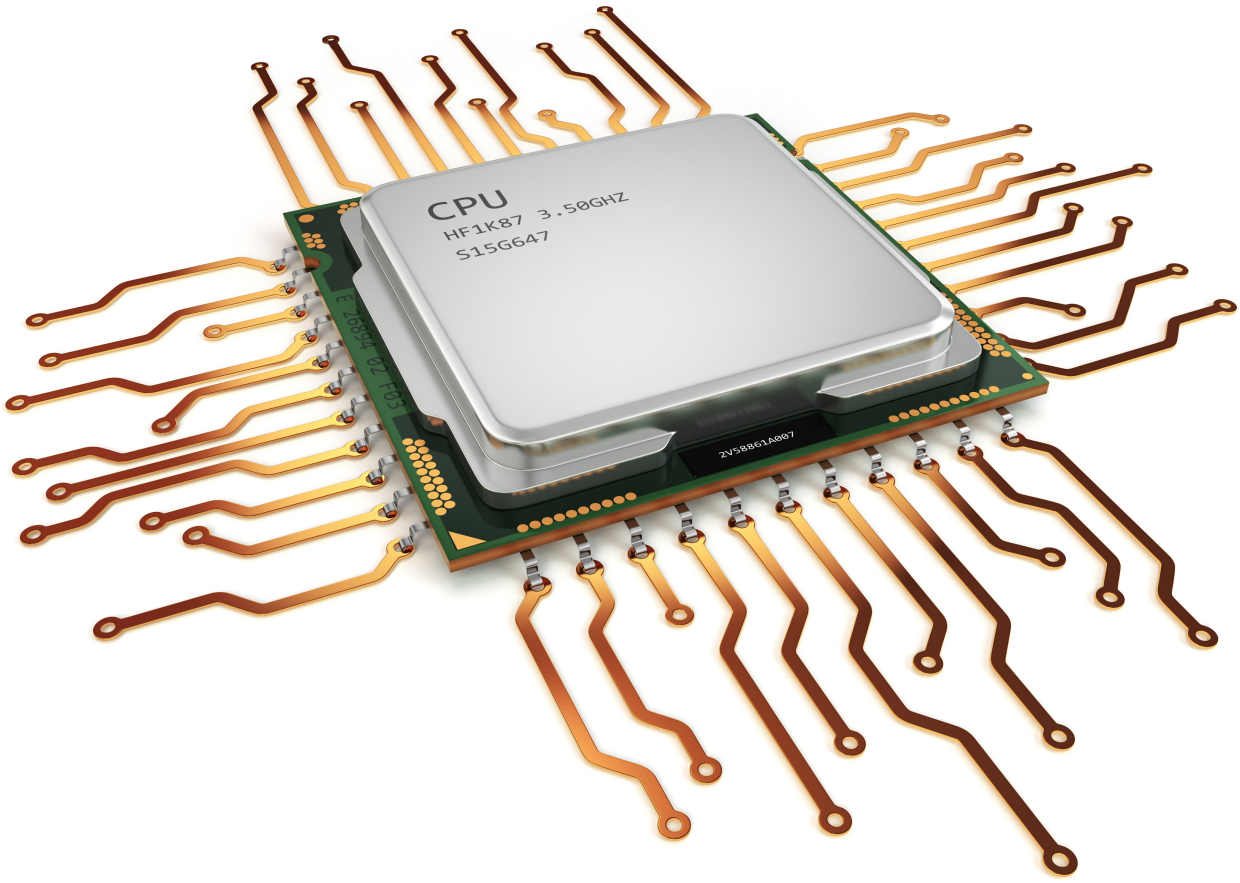
Introduction to File I/O

by Sophia

WHAT'S COVERED

In this lesson, you will learn about some of the basic ways to work with files. These are reading, writing, and closing files. Specifically, this lesson covers:

1. Working With Files



So far, you have learned how to write programs and communicate your intentions to the Central Processing Unit using conditional statements, functions, and iterations. The **Central Processing Unit** is the brain of any computer. It is what runs the software that we write. It's also called the "CPU" or the "processor." You have learned how to create and use data structures that are stored in the main memory of a computer. The **main memory** is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. The CPU and main memory are where our software works and runs. It is where all of the "thinking" happens.

Once the power is turned off, anything stored in either the CPU or main memory is erased. So, up to now, our programs have just been transient fun exercises to learn Java. Now you will start to work with secondary memory. **Secondary memory** is also used to store information, but it is much slower than main memory. The advantage of secondary memory is that it can store information even when there is no power to the computer.

HINT

Examples of secondary memory devices are a USB flash drive or hard drive.

By storing information on a USB flash drive, the information can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor. To read or write a file on a drive, like your hard drive, you must first open the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to first find the file by name and make sure the file exists, and then make its contents available (open) for reading or writing.

IN CONTEXT

In the example below, we open the file *poem.txt*, which should be stored in the same folder as the Java file that will access it. In Replit, we can create a file, name it *poem.txt*, and add in the following text:

Roses Are Red,
Violets are blue,
Sugar is sweet,

```
And so are you.
```

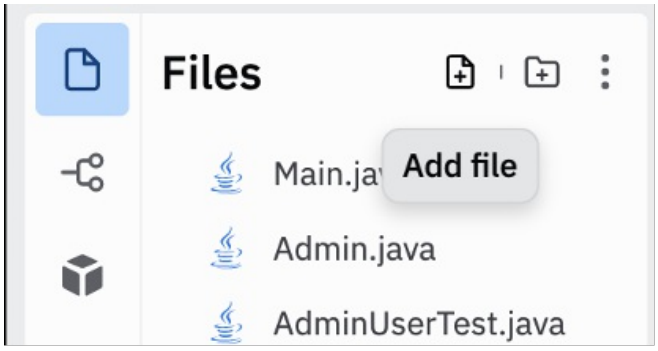
TRY IT

Directions: Try following the steps below to create a new file and add some code.

STEP BY STEP

1. Click on the 'Add file' icon at the top of the Files panel.

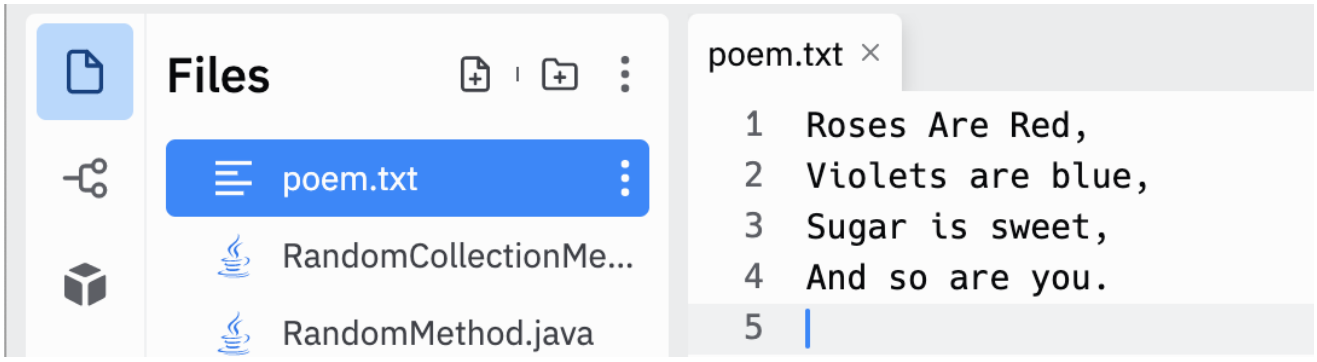
EXAMPLE



2. Next, we name the file; in this case, we will call it `poem.txt`. Remember, if you do not give a filename an extension, the default extension of `.java` will be added. We do not want to create a `.java` file, so we needed to add the `.txt` to make it a text file.

3. Then, we can paste in the code.

EXAMPLE



REFLECT

Notice that the icon for our *poem.txt* text file looks different than the Java code files (ending with the .java file extension).

Now that our *poem.txt* file has information inside of it, let's try to read the file.

TRY IT

Directions: First, we need to move back to the FileExample.java file to add the following code:

```
import java.io.File;

public class FileExample {
```

```
  public static void main(String[] args) {
    File poemFile = new File("poem.txt");
    System.out.println("File Information: ");
    System.out.println("File Name: " + poemFile.getName());
    System.out.println("File Path: " + poemFile.getPath());
    System.out.println("Full Path: " + poemFile.getAbsolutePath());
    System.out.println("File Size: " + poemFile.length());
    System.out.println("Can Read: " + poemFile.canRead());
    System.out.println("Can Write: " + poemFile.canWrite());
    System.out.println("Can Execute: " + poemFile.canExecute());

  }
}
```

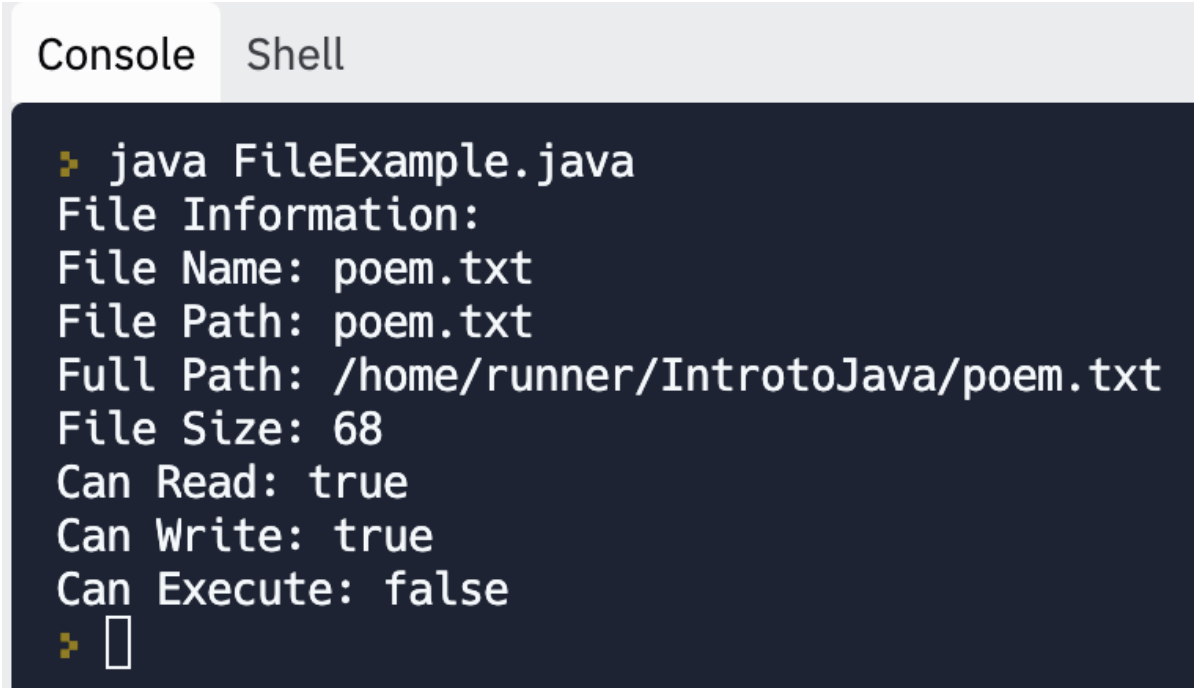
REFLECT

It is worth noting that the Java `File` object is an abstraction of a file on disk. The file that it refers to may or may not exist. Depending on how the `File` object is used later in the code, the file may be created, opened, or read.

TRY IT

Directions: Next, run the program using the command java FileExample.java.

The output screen should look like this:



REFLECT

Not seeing the poem? That is because at this point we only accessed information about the file and we haven't read any data from the file. The output we see is just information about the file pointed to by the File object that we created.

TERMS TO KNOW

Central Processing Unit

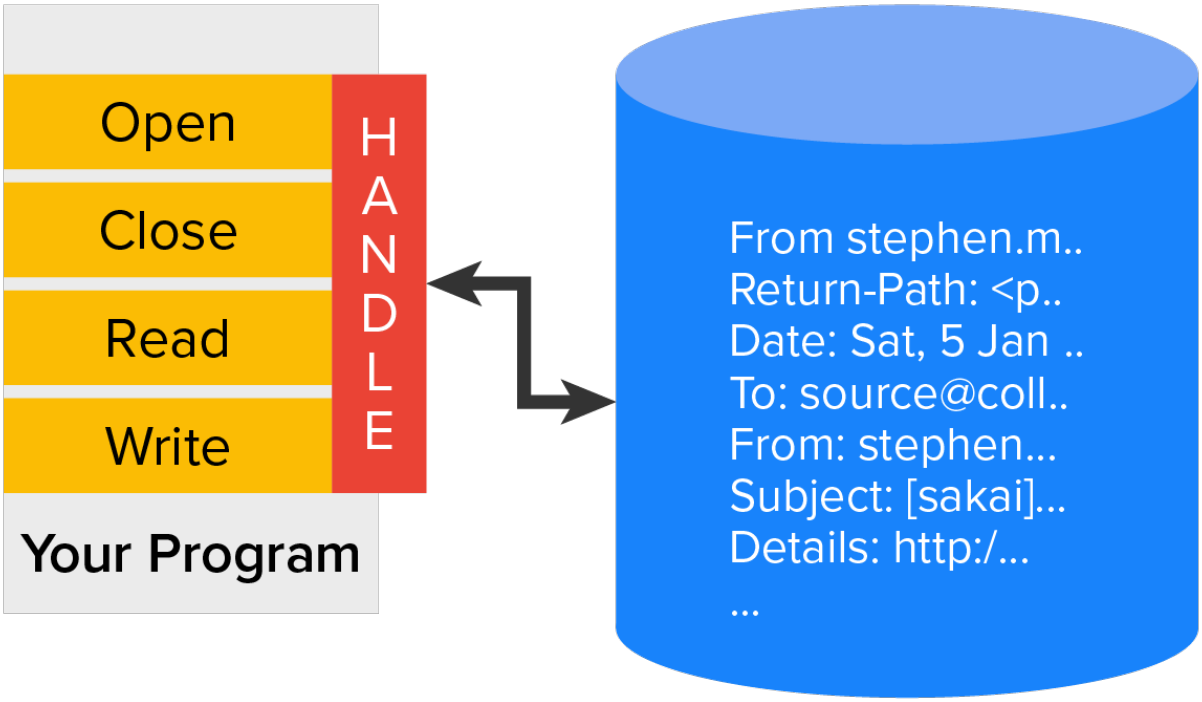
The Central Processing Unit is the heart of any computer. It is what runs the software that we write; it is also called the “CPU” or the “processor.”

Main Memory
The main memory is used to store information that the CPU needs in a hurry. Main memory is nearly as fast as the CPU. The CPU and memory are where our software works and runs.

Secondary Memory
Secondary memory is also used to store information, but it is much slower than main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer.

---

2. Reading Files



If opening a file is successful, the operating system returns a file handle. The **file handle** is not the actual data contained in the file, but instead it is a “handle” that can be used to read the data.

BIG IDEA

Think of the file handle as being an address to a house. With the address to a house, you know where the house is, but the address itself is just pointing to the house location in the same way that a file handle is pointing to a file.

A text file can be thought of as a sequence of lines, much like a Java String can be thought of as a sequence of characters. The *poem.txt* file simply contains four lines of text for the poem.

While the file handle does not itself contain the data in the file, it is quite easy to write code using the `Files` utility class that reads the contents of the whole file and stores the lines in a `List` collection of `String` objects. The size of the `List` is the number of lines in the file.

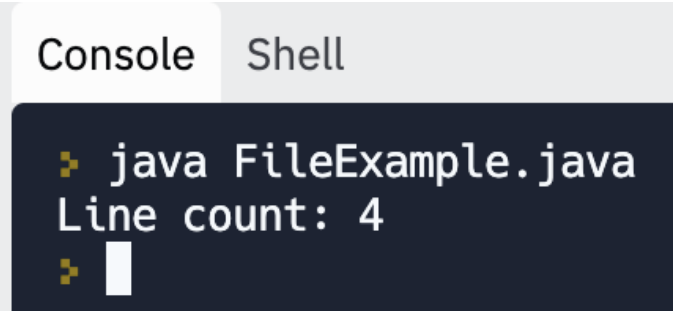
When working with the methods in the `Files` utility class, the `File` object's `toPath()` method will be called. A `Path` object has many similarities to a `File` object, and both classes include methods for converting objects back and forth.

TRY IT

Directions: Revise the code in the FileExample.java file so that it looks like the following code:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

public class FileExample {
  public static void main(String[] args) {
    // File object for text file
    File poemFile = new File("poem.txt");
    try {
      // Files.readAllLines() reads entire file & puts lines in the
      // List<String>
      List<String> lines = Files.readAllLines(poemFile.toPath());
      System.out.println("Line count: " + lines.size());
    }
    // If Files.readAllLines() can't find or read file, it throws an
    // IOException
    catch(IOException ex) {
      System.out.println("Error accessing file: " + ex.getMessage());
    }
  }
}
```

This simple program should produce the following output:



TRY IT

Directions: Try using the `Files.readAllLines()` method to count the lines of our *poem.txt* file.

↷ EXAMPLE

Use the `Files.readAllLines()` method to count the lines of our *poem.txt* file.

CONCEPT TO KNOW

Note that `Files.readAllLines()` does what the name indicates and reads all of the lines of the file into memory.

This is fine for cases like this one where the file is small. In cases where the file is larger, it might be more appropriate to use a Scanner to read from the file using a while loop.

The Scanner's nextLine() method reads each line until the new line character (\n) is encountered:

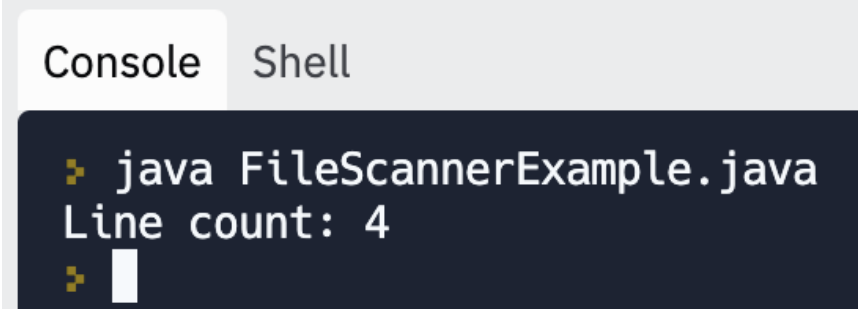
```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileScannerExample {
  public static void main(String[] args) {
    // Scanner can read from File pointing to file
    File poemFile = new File("poem.txt");
    int lineCount = 0;

    try {
      // Throws FileNotFoundException if file not found
      Scanner fileScanner = new Scanner(poemFile);
      // while loop runs as long another line is available
      while(fileScanner.hasNextLine()) {
        // Read line as String
        String line = fileScanner.nextLine();
        lineCount++;
      }
      // Must close Scanner when reading from a file.
      fileScanner.close();
      System.out.println("Line count: " + lineCount);
    }
    catch(FileNotFoundException ex) {
      System.out.println("Error accessing file: " + ex.getMessage());
    }
  }
}
```

TRY IT

Directions: Type in the code above in a file named FileScannerExample.java and run it using the command:

```
java FileScannerExample.java
```

This example should produce the following output, which is the same as the version using Files.readAllLines():



REFLECT

We have seen a couple of the approaches that Java provides for reading files. If you continue on in the language past this course, you will undoubtedly learn more. Sometimes this variety reflects the evolution of the language, and sometimes it reflects demands by programmers. Think about how a choice of methods can make life both better and worse for programmers using the language.

CONCEPT TO KNOW

The newline character is a whitespace character similar to the space and tab character; it is not shown as a '\' (space) or a '\t' (tab). Even though the newline character is stored as a '\n' (newline), you would not see that character as a '\n' but rather you would see the text moving to the next line.

Because the `while` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `Files.readAllLines()` method. What constitutes a large or small file is relative and depends on the amount of memory available and the number of files that may be open at one time.

Rather than reading the lines in the file to count them, it would be useful to actually display the contents. Since the code above that uses File.readAllLines() stores the lines of the file in a list, we can use a for loop to cycle through the list and print out the lines.

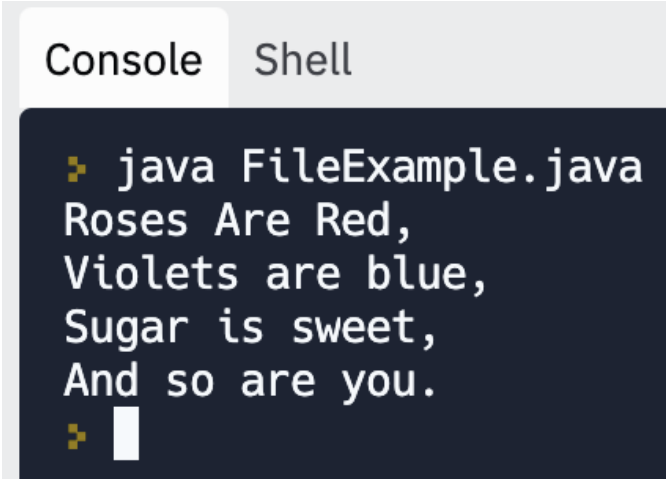
TRY IT

Directions: Modify the FileExample.java file so the code reads as follows:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

public class FileExample {
  public static void main(String[] args) {
    // File object for text file
    File poemFile = new File("poem.txt");
    try {
      // Files.readAllLines() reads entire file & puts lines in the
      // List<String>
      List<String> lines = Files.readAllLines(poemFile.toPath());
      // Loop through the list and print out the lines
      for(String line : lines) {
        System.out.println(line);
      }
    }
    // If Files.readAllLines() can't find or read file, it throws an
    // IOException
    catch(IOException ex) {
      System.out.println("Error accessing file: " + ex.getMessage());
    }
```

```
  }
}
```

This would be the expected output:



REFLECT

In this example, the entire contents of the file *poem.txt* are read directly into the List. Each line is stored as a String object in the List. The range-based for loop iterates over the List and prints out each line.

BRAINSTORM

Now try to use the `.read()` method and see if you can output the contents of our *poem.txt* file.

TERM TO KNOW

File Handle
The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data.

---

3. Writing to Files

The `Files` class includes a method called `write()` that writes an iterable collection (like an ArrayList) to an output file. Depending on the options passed when the method is called, the data in the collection may be written to a new file, overwrite the current contents of a file, or be appended to the contents of an existing file.

Here is a short program that creates a file and writes three lines of text to it. The code creates an `ArrayList<String>` containing the lines to be written to the file. Since there is the possibility of exceptions when working with a file, the relevant statements are in a `try` block, and there is a `catch` block for an `IOException`.

↷ EXAMPLE

```
import java.io.*;
import java.nio.file.*;
import java.util.ArrayList;

public class WriteTextToFile {
  public static void main(String[] args) {
    // Create ArrayList of Strings & add lines of text
```

```
    ArrayList<String> lines = new ArrayList<>();
    lines.add("Line 1");
    lines.add("Line 2");
    lines.add("Line 3");
    // File object pointing to output.txt file (which may not exist yet)
    File outputFile = new File("output.txt");

    try {
      // StandardOpenOption.CREATE creates a new file. It will create the file
      // if it doesn't exist or overwrite it if it does.
      Files.write(outputFile.toPath(), lines, StandardOpenOption.CREATE);
    }
    catch(IOException ex) {
      System.out.println("Error writing to file: " + ex.getMessage());
    }
  }
}
```

CONCEPT TO KNOW

If the file already exists, opening it in StandardOpenOption.CREATE mode (StandardOpenOption is an example of a Java **enumeration**) clears out the old data and starts fresh, so be careful! If the file doesn't exist already, it will be created.

TERM TO KNOW

Enumeration
StandardOpenOption is an example of a Java enumeration. An enumeration is a named collection of constant values.

---

4. Closing Files

In the sample program showing how to use a Scanner to read a text file's contents, the Scanner object needs to be closed (which then closes the underlying file). Note the following line:

```
fileScanner.close();
```

It carries out this task in the code:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileScannerExample {
  public static void main(String[] args) {
    // Scanner can read from File pointing to file
    File poemFile = new File("poem.txt");
    int lineCount = 0;
```

```
    try {
      // Throws FileNotFoundException if file not found
      Scanner fileScanner = new Scanner(poemFile);
      // while loop runs as long another line is available
      while(fileScanner.hasNextLine()) {
        // Read line as String
        String line = fileScanner.nextLine();
        lineCount++;
      }
      // Must close Scanner when reading from a file.
      fileScanner.close();
      System.out.println("Line count: " + lineCount);
    }
    catch(FileNotFoundException ex) {
      System.out.println("Error accessing file: " + ex.getMessage());
    }
  }
}
```

BIG IDEA

When using the methods for reading and writing text to and from a file, it is not necessary to close the file explicitly, since the methods provided by java.nio.Files take care of this step.

Recall how this version of the code for reading the file's contents using the Files.readAllLines() does not have a separate statement to close the file:

EXAMPLE

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

public class FileExample {
  public static void main(String[] args) {
    // File object for text file
    File poemFile = new File("poem.txt");
    try {
      // Files.readAllLines() reads entire file & puts lines in the
      // List<String>
      List<String> lines = Files.readAllLines(poemFile.toPath());
      System.out.println("Line count: " + lines.size());
    }
    // If Files.readAllLines() can't find or read file, it throws an
    // IOException
    catch(IOException ex) {
      System.out.println("Error accessing file: " + ex.getMessage());
    }
  }
}
```

REFLECT

If you continue with Java, you will encounter some of the older methods using the input and output streams mentioned above. When using these pre-`java.nio` approaches, it will be important to handle the closing of the files. This is another advantage of using the newer methods provided by the `Files` class.

SUMMARY

In this lesson, you learned about different ways that you can **work with files**, including **reading a text file's** contents using the `Files` class and a `Scanner`. You also learned how to use the Files utility class to **write text to a file**. Finally, you learned about exception handling using `try` and `catch` when working with and **closing files**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from "Python for Everybody" By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Central Processing Unit
The Central Processing Unit is the heart of any computer. It is what runs the software that we write; it is also called the "CPU" or the "processor."

Enumeration
StandardOpenOption is an example of a Java enumeration. An enumeration is a named collection of constant values.

File Handle
The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data.

Main Memory
The main memory is used to store information that the CPU needs in a hurry. Main memory is nearly as fast as the CPU. The CPU and memory are where our software works and runs.

Secondary Memory
Secondary memory is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer.

Creating, Reading, Writing and Deleting a File

by Sophia

WHAT'S COVERED

In this lesson, you will learn more about how to read and write to files. Specifically, this lesson covers:

1. Reading and Writing to Files

In the previous tutorial, you worked with a File object that represents an abstraction of a file in that it may point to a file that already exists or to a file that doesn't exist yet (but will be created).

1a. Writing to a File

Since working with files is generally more interesting when they have content, let's first look a bit more into writing text to a file.

In the previous tutorial, you used this code to write to a new file using the Files.write() method (in the java.nio library package):

```
import java.io.*;
import java.nio.file.*;
import java.util.ArrayList;

public class WriteTextToFile {
  public static void main(String[] args) {
    // Create ArrayList of Strings & add lines of text
    ArrayList<String> lines = new ArrayList<>();
    lines.add("Line 1");
    lines.add("Line 2");
    lines.add("Line 3");
    // File object pointing to output.txt file (which may not exist yet)
    File outputFile = new File("output.txt");

    try {
      // StandardOpenOption.CREATE creates a new file. It will create the file
      // if it doesn't exist or overwrite it if it does.
      Files.write(outputFile.toPath(), lines, StandardOpenOption.CREATE);
    }
    catch(IOException ex) {
      System.out.println("Error writing to file: " + ex.getMessage());
    }
  }
}
```

This code creates a File object named outputFile that points to a file called output.txt:

EXAMPLE

```
File outputFile = new File("output.txt");
```

This next line then writes the text in the ArrayListcalled lines to the output.txt file:

EXAMPLE

```
Files.write(outputFile.toPath(), lines, StandardOpenOption.CREATE);
```

Note how the File object, outputFile, is converted to a Path object using the File class's toPath() method. The StandardOpenOption.CREATE option tells the JVM to create the file on disk, if the file doesn't already exist.

If there is a collection of lines of text to write to the file, the Files.writeString() method can be used like this:

```
class WriteStringToFile {
  public static void main(String[] args) {
    File output = new File("output.txt");
    try {
      Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);
    }
    catch(IOException ex) {
      System.out.println("Error: " + ex.getMessage());
    }
  }
}
```

This program creates a file named output.txt and writes the string "Hello, world" to it. If the file already exists, the contents are overwritten when using the StandardOpenOption.CREATE option.

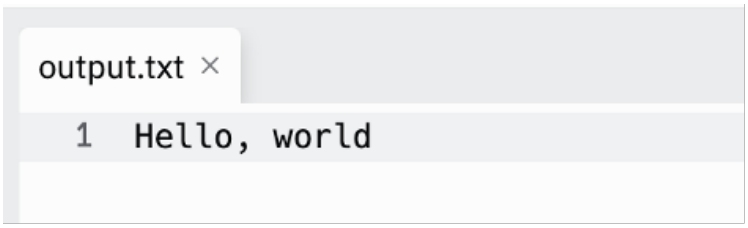
BIG IDEA

If the same file is written to more than once using multiple calls to Files.writeString(), there will only be one line of text in the output file because StandardOpenOption.CREATE mode will either create a new file or overwrite an existing file.

```
import java.io.*;
import java.nio.*;
import java.nio.file.*;

public class WriteStringToFile {
  public static void main(String[] args) {
    File output = new File("output.txt");
    try {
      // Try writing 2 lines to the file in StandardOpenOption.CREATE mode
      Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);
      Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);
```

```
    }
    catch(IOException ex) {
      System.out.println("Error: " + ex.getMessage());
    }
  }
}
```

Each time we run this example, our output.txt file will just have the output file always looking like the following (remember we need to move to the output.txt file to see this output):



TRY IT

Directions: Enter this code in a file named WriteStringToFile.java. If your Replit directory already contains an output.txt file, delete it, and then run the program (java WriteStringToFile.java). Look for the new file called output.txt and click on it to see the output. Try running it a few times to see if the output is the same.

REFLECT

The output file always contains one line because each time you write to the file, you are using the StandardOpenOption.CREATE option. This means that each time the file is opened, it creates a new file and overwrites all of the old content.

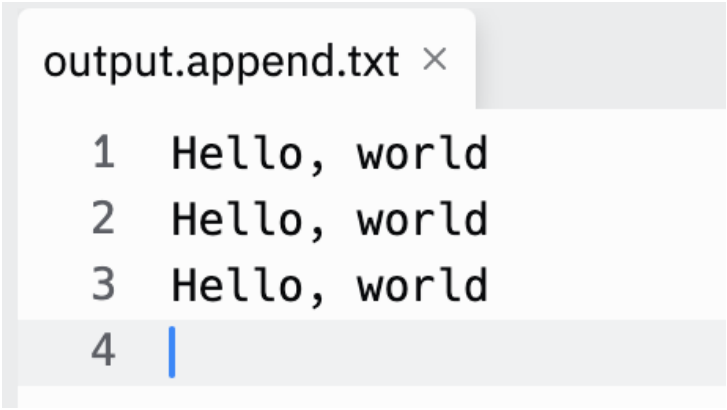
1b. Appending to a File

When using a combination of StandardOpenOption.CREATE and StandardOpenOption.APPEND, a new file will be created if the file doesn't exist. It will append a line to the file, if the file already exists.

```
import java.io.*;
import java.nio.*;
import java.nio.file.*;

public class AppendStringToFile {
  public static void main(String[] args) {
    File output = new File("output.append.txt");
    try {
      // Use both StandardOpenOption.CREATE and StandardOpenOption.APPEND so that
      // file is created if it doesn't exist. If the file already exists, text is
      // appended.
      Files.writeString(output.toPath(), "Hello, world\n", StandardOpenOption.CREATE,
                  StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error: " + ex.getMessage());
    }
  }
```

```
}
```

Each time the file is opened and written to, a new line is added. If we run it three times, the results will look like this:



TRY IT

Directions: Try entering the code above in a file in Replit named AppendStringToFile.java. Run the program again. Check the output.append.txt file to see the output.

REFLECT

Notice you see a new appended line each time you run the AppendStringToFile.java file.

1c. Writing Integers and Floats

Writing to a file works the same as we have been doing it. The only exception is that integers or floats must be converted to a string before the string can be written to the file. Since this conversion needs to be done for numbers one at a time, the program needs to use Files.writeString().

Let's try an example using integers. In this example, it will be years that need to be converted to strings. First, we will define an ArrayList of Integers, then we will write this List using StandardOpenOption.CREATE and StandardOpenOption.APPEND within a for loop:

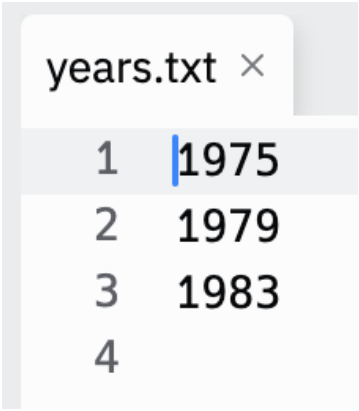
```
import java.io.*;
import java.io.IOException;
import java.nio.file.*;
import java.util.ArrayList;

public class WriteNumbersToFile {

    public static void main(String[] args) {
        ArrayList<Integer> years = new ArrayList<>();
        years.add(1975);
        years.add(1979);
        years.add(1983);
        File numbersOutput = new File("years.txt");
        // Iterate over ArrayList of years
        for(int year : years) {
```

```
        try {
           // Use Integer.toString() to convert years to strings.
           // Need to convert years one at a time, so they have to be
           // written one at a time using Files.writeString(). Add \n after each
           Files.writeString(numbersOutput.toPath(), Integer.toString(year) + "\n",
                 StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        }
        catch(IOException ex) {
           System.out.println("Error: " + ex.getMessage());
        }
     }
  }
}
```

TRY IT

Directions: Try adding the code above to create a new file called years.txt. Run the program, then see if the years as strings show up in the years.txt file as below:

EXAMPLE



REFLECT

It is important to remember that when reading and writing values to and from text files, all of the data is read and written as strings. This means that numeric data will need to be converted to and from Java Strings, since the rest of the code needs these to be of the correct data type.

1d. Reading Integers

An example reading our just written data works the same as before, except that the year string must be converted to an integer:

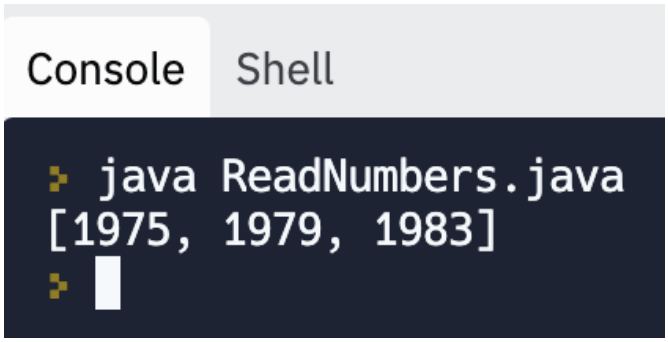
```
import java.io.*;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.List;

public class ReadNumbers {

  public static void main(String[] args) {
    File yearsFile = new File("years.txt");
```

```
    ArrayList<Integer> years = new ArrayList<>();
    try {
      List<String> yearsAsStrings = Files.readAllLines(yearsFile.toPath());
      for(String yearString : yearsAsStrings) {
        years.add(Integer.parseInt(yearString));
      }
    }
    catch(IOException ex) {
      System.out.println("Error reading file: " + ex.getMessage());
    }
    catch(NumberFormatException ex) {
      System.out.println("Number format error: " + ex.getMessage());
    }
    System.out.println(years.toString());
  }
}
```

TRY IT

Try typing the code above into a file named ReadIntegers.java and run the program.

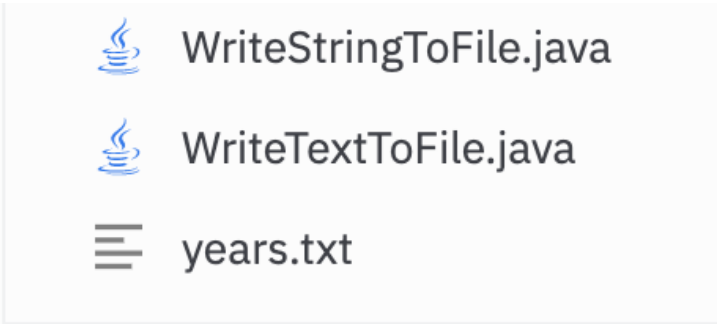
You should get the following result (based on the years written to the file previously):



1e. Deleting a File

The java.nio package's Files.deleteIfExists() handles deleting a file (assuming that the file exists). This method returns true or false, indicating if the file was deleted successfully or not. Notice that before using the following code example, our File structure looks like this (from Replit's left side File panel):

EXAMPLE



To delete the *years.txt* file that we have created, we could use the following code.

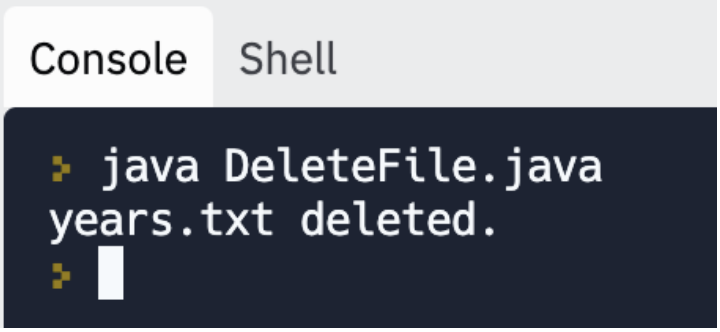
```
import java.io.File;
```

```
import java.io.IOException;
import java.nio.file.Files;

public class DeleteFile {

  public static void main(String[] args) {
    File yearsFile = new File("years.txt");
    boolean fileDeleted = false;
    try {
      fileDeleted = Files.deleteIfExists(yearsFile.toPath());
    }
    catch(IOException ex) {
      System.out.println("Error deleting file: " + ex.getMessage());
    }
    if(fileDeleted) {
      System.out.println(yearsFile.getName() + " deleted.");
    }
    else {
      System.out.println(yearsFile.getName() + " not deleted.");
    }
  }
}
```

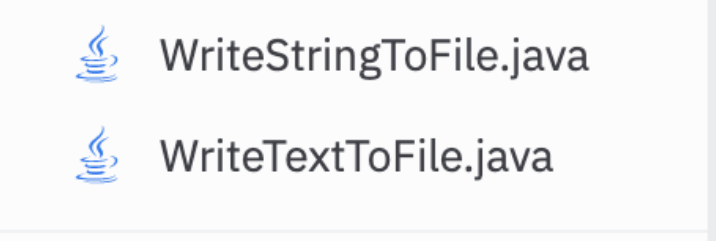
TRY IT

Directions: Try typing the code above into a file named ReadIntegers.java and run the program. Now type in the code above (in a file named DeleteFile.java) and see if you can remove the years.txt file.

The output from running the program should look like this:



Notice that after running the code, the file is removed from the Files directory.



SUMMARY

In this lesson, you had a chance to **read and write to files**. You learned how to create a file, **write to a**

file, and **append a file**. We also looked at an example of **writing numeric data or integers and floats** to a file. You also learned how to **read integers stored as strings in a text file** Finally, you learned how to **delete a file**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

Debugging Files

by Sophia

WHAT'S COVERED

In this lesson, you will learn about ways to make working with files more flexible and less susceptible to errors. Specifically, this lesson covers:

1. Working With Files

It is not efficient to edit code every time that you want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs. This would ensure that they could use the program on different files without changing the Java code.

This is fairly simple to do by reading the file name from the user, using input as follows:

```
import java.io.*;
import java.nio.file.*;
import java.util.List;
import java.util.Scanner;

class EnterFileName {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter existing file name: ");
    String fileName = input.nextLine();
    File inputFile = new File(fileName);
    try {
      List<String> lines = Files.readAllLines(inputFile.toPath());
      System.out.println(inputFile.getName() + " contains " + lines.size() + " lines.");
    }
    catch(IOException ex) {
      System.out.println("File error: " + ex.getMessage());
    }
  }
}
```

You would read the file name from the user and place it in a variable named `fileName`. The `Files.readAllLines()` method reads the contents of the file into a `List` named lines. The size of the `List` corresponds to the number of lines in the file. Now the program can run repeatedly on different files to count the number of lines in the file.

CONCEPT TO KNOW

Keep in mind that the file names are case sensitive (and spelling counts).

2. File Name and Permissions Errors

What if our user types something that is not a file name? In the following example, the user has typed the name of a file that doesn't exist:

↱ EXAMPLE

```
nonexistantFile.txt
```

This is the expected result:

As you have seen, using the methods provided by the Files utility class can result in an `IOException` being thrown, so using `Files.readAllLines()` requires using try and catch blocks for exception handling. If the file entered by the user does not exist, an `IOException` is thrown and the `System.out.println()` statement in the corresponding catch block runs and produces the message:

↱ EXAMPLE

```
File error: nonexistantFile.txt
```

CONCEPT TO KNOW

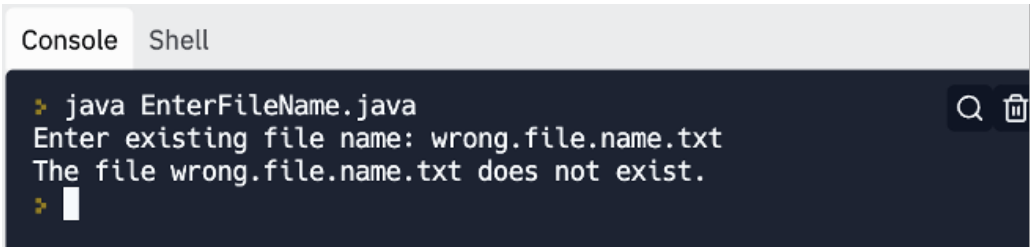
Note that an `IOException` may also be thrown if the file exists, but the user does not have read permission on the file. If we wanted to check that the file exists before trying to read it, we could use the `Files.exists()` method, which returns a boolean.

To check if the user running the program has read permission on the file, we could use the `Files.isReadable()` method, which also returns a `boolean`. These methods allow us to produce better error messages and also avoid the overhead cost of throwing an `IOException`.

```
import java.io.*;
import java.nio.file.*;
import java.util.List;
import java.util.Scanner;

class EnterFileName {
```

```
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter existing file name: ");
    String fileName = input.nextLine();
    File inputFile = new File(fileName);
    // Warn if file doesn't exist
    if(! Files.exists(inputFile.toPath())) {
      System.out.println("The file " + fileName + " does not exist.");
    }
    // Then warn if user doesn't have read permission
    else if(! Files.isReadable(inputFile.toPath())) {
      System.out.println("User doesn't have read permission on " + fileName + ".");
    }
    // If file exists and can be read by user, try to read it.
    // Other I/O errors may be possible, sot try/catch needed, but such
    // errors are much less likely.
    else {
      try {
        List<String> lines = Files.readAllLines(inputFile.toPath());
        System.out.println(inputFile.getName() + " contains " + lines.size() + " lines.");
      }
      catch(IOException ex) {
        System.out.println("File error: " + ex.getMessage());
      }
    }
  }
}
```

The result should look like this:



The specifics of setting file and directory permissions varies by operating system, but Replit's Linux environment uses the chmod command to set permissions. If you run the following command on a text file that already exists:

↱ EXAMPLE

```
chmod a-r wrong.permissions.txt
```

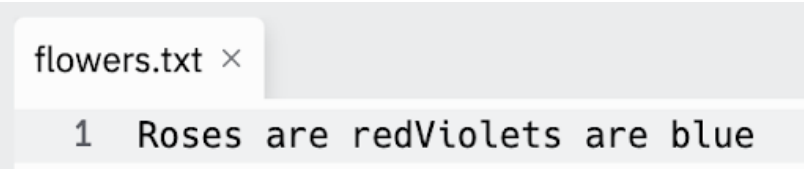
Read access is taken away for all users, so running the code above should produce output like this:

3. Text Written to File as Block Rather Than as Lines

When writing text to a file, it is important to be mindful of the presence or absence of newline (\n) characters in the String values being written to the file. Unlike the System.out.println() method for writing text to the terminal, the Files.write() and Files.writeString() methods do not automatically add a newline character.

The following code writes two statements about flowers to a file called flowers.txt:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;

public class FileNewLine {
  public static void main(String[] args) {
    File flowerFile = new File("flowers.txt");
    try {
      // Write each line without added new line
      Files.writeString(flowerFile.toPath(), "Roses are red", StandardOpenOption.CREATE);
      Files.writeString(flowerFile.toPath(), "Violets are blue", StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error: " + ex.getMessage());
    }
  }
}
```

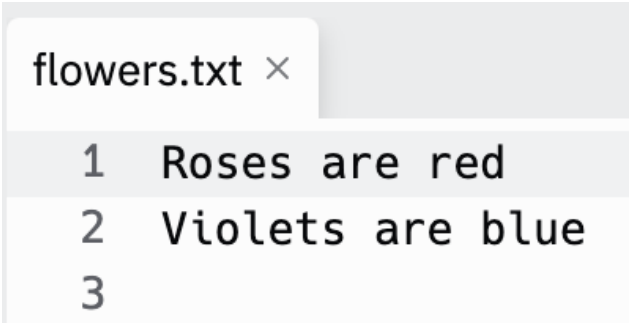


If the two calls to Files.writeString() add “\n” at the end of each spring, the text is written on separate lines. Here are the two modified lines:

↷ EXAMPLE

```
Files.writeString(flowerFile.toPath(), "Roses are red\n", StandardOpenOption.CREATE);
Files.writeString(flowerFile.toPath(), "Violets are blue\n", StandardOpenOption.APPEND);
```

To see the new results, delete the existing flowers.txt file. Since the text includes newline characters, the results will appear like this:



4. StandardOpenOption Modes

When writing to a file using Files.write() and Files.writeString(), one of the arguments passed is a value defined in the StandardOpenOption enumeration. An enumeration is a set of constant values with names.

We have already seen StandardOpenOption.CREATE, which creates a file if it doesn't already exist. If the file already exists, the contents will be overwritten. Note, though, that if the new text is shorter than the previous file contents, any lines not overwritten by new text will be unchanged. This may not be the behavior you would expect.

The argument StandardOpenOption.APPEND will cause the text to be written starting at the end of an existing file, but it will fail if the file doesn't already exist. As we saw in the previous lesson, these modes can be combined simply by passing both arguments, as shown in this program:

```
import java.io.*;
import java.nio.*;
import java.nio.file.*;

public class AppendStringToFile {
  public static void main(String[] args) {
    File output = new File("output.append.txt");
    try {
      // Use both StandardOpenOption.CREATE and StandardOpenOption.APPEND so that
      // file is created if it doesn't exist. If the file already exists, text is
      // appended.
      Files.writeString(output.toPath(), "Hello, world\n", StandardOpenOption.CREATE,
                        StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error: " + ex.getMessage());
    }
  }
}
```

When using StandardOpenOption.CREATE to overwrite an existing file, it is commonly paired with the StandardOpenOption.TRUNCATE_EXISTING so that the current contents are cleared before the new text is written to the file. This line shows how to pass these two as arguments:

EXAMPLE

```
Files.writeString(flowerFile.toPath(), "Roses are red\n", StandardOpenOption.CREATE,
                  StandardOpenOption.TRUNCATE_EXISTING);
```

It is worth noting that `StandardOpenOption.READ` and `StandardOpenOption.WRITE` are also defined, but the `Files.ReadAllLines()` and `File.Write()` methods that we have used handle opening the file in the correct read or write mode behind the scenes, so the programmer doesn't have to be concerned with them.

There are a few more modes defined in the `StandardOpenOption` enumeration. `StandardOpenOption.CREATE_NEW` is another one for beginners to be aware of. When using this mode with `Files.write()`, the file is created only if it does not already exist. It throws an`IOException` if the file already exists.

SUMMARY

In this lesson, you learned about handling file-related issues and ways to make **working with files** more flexible. One common issue is dealing with files that don’t exist. We can use Java's normal exception handling, but the Files class also provides methods that can help us steer clear of common **errors related to file names and permissions**. You learned that when **writing text to a file as block rather than as lines**, it is important to be mindful of the presence or absence of newline (\n) characters in the String values being written to the file. Lastly, you learned about the various **StandardOpenOption enumeration modes** used when writing to a file using `Files.write()` and `Files.writeString()`.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

The Company Employee Program

by Sophia

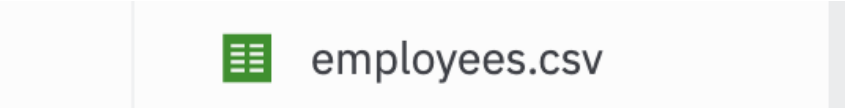
WHAT'S COVERED

In this lesson, you will construct a program by manipulating files using libraries and file inputs when using Java. Specifically, this lesson covers:

1. Storing Information to a File

In this lesson, we will give the employee class program some final touches! You will begin by creating a program that stores the employee ID, salary, last name, and first name for our employees into a file. Rather than having it entered every single time, you will write that information to a file so that we can easily recall that information. You could extend this program to include in the additional details of the employee, but instead, you will focus on the key elements (namely employee ID and salary). Let's first start by creating an *employees.csv* file in Replit by clicking on the 'Add file' icon at the top of the Files panel. Name it *employees.csv*.

EXAMPLE



TRY IT

Directions: Create a new file in Replit called *employees.csv*.

The icon will automatically change to a file that looks like a spreadsheet, just like our text files earlier looked like a few lines of text as an icon. Our file has a .java file extension since it is a Java file, and it has the Java icon.

DID YOU KNOW

The new file is a spreadsheet file that has a suffix of csv. What is that? Comma separated value (or .csv) is a file format where each of the data elements for a row of data would be split by a comma. This is very common to spreadsheet applications like Microsoft Excel. In a .csv file, we would see data that is saved in a format like this example (it's important to have a blank line at the end of the file):

EXAMPLE

```
10001,75000,Johnson,Mary
10002,68500,Doe,John
```

This would be a file that contains the employee ID, salary, first name, and last name. As more information (individuals) is added, the data separation would continue.

Later, you will review the *employees.csv* file to see what that looks like as data is added to it. For now, the file

should be completely empty.

As this is meant to be a larger program, start by defining a CompanyEmployee class to encapsulate data about an employee.

TRY IT

Directions: Type this code into a file named CompanyEmployee.java:

```
public class CompanyEmployee {
  private String lastName;
  private String firstName;
  int id;
  int salary;

  public CompanyEmployee(String lastName, String firstName, int id, int salary) {
    this.lastName = lastName;
    this.firstName = firstName;
    this.id = id;
    this.salary = salary;
  }

  public String getLastName() {
    return lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public int getId() {
    return id;
  }

  public int getSalary() {
    return salary;
  }

  // toString() method provides format for printing company employee data:
  // Smith, John ID: 10123 ($85000)
  public String toString() {
    return lastName + ", " + firstName + " ID: " + id + " ($" + salary + ")";
  }
}
```

TRY IT

Directions: After you have typed in the code above, remember that you need to compile the CompanyEmployee class so that it will be available for use in the program. Run the following command in the Replit shell:

↷ EXAMPLE

```
javac CompanyEmployee.java
```

Next, start the driver class for the application in a file named CompanyEmployeesProgram.java. We will start with an empty main() method.

TRY IT

Directions: Type in this code to start with, in a file named CompanyEmployeesProgram.java:

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

  }
}
```

You will return to the code in main() later, but let's continue at this point with a method to read the data from the .csv file and convert it into an ArrayList of CompanyEmployee objects. You will call the method to read the data readEmployees(). Since it will be part of the driver class and called from main(), it will need to be declared as a static method. This method takes a String parameter for passing in the name of the file. When done processing the data in the file, it will return an ArrayList<CompanyEmployee> collection. The line with the return type and the signature for the method should look like this:

EXAMPLE

```
public static ArrayList readEmployees(String csvFile) {
```

Since the code will work with an ArrayList, the code will need to import java.util.ArrayList. Since we will be working with files, we also need the File and Files classes (along with associated exception types). We might as well go ahead and add the other needed import statements at the top of the file. The complete list reads:

EXAMPLE

```
import java.util.ArrayList; import java.io.File; import java.io.FileNotFoundException; import
java.io.IOException; import java.nio.file.Files;
```

This list could be simplified a bit using a wildcard (*), if you prefer:

EXAMPLE

```
import java.util.ArrayList; import java.io.*; import java.nio.file.Files;
```

As part of the code statement of the readEmployees() *method*, we create an empty ArrayList of CompanyEmployee objects called employeeList. Then, we open up the data file using a File object and use a Scanner to read the data from the file. The code then takes a line from the file and splits the data into individual data elements (an array of String data).

The data is read from the file as String data, but the first two items (the employee ID and the salary) need to be converted to integer values. The Integer wrapper class that we met when discussing generic types includes a

method named Integer.parseInt(), which takes a String value and converts it to an int value.

Here is a partial initial draft of the method:

```
public static ArrayList<CompanyEmployee> readEmployees(String csvFile) {
  // Create an empty ArrayList of CompanyEmployee objects
  ArrayList<CompanyEmployee> employeeList = new ArrayList<>();

  // File object for accessing the CSV file
  File inputDataFile = new File(csvFile);
  List<String> lines = new ArrayList<>();
  // Because the following statements can through exceptions, they are in a try block
  try {
    lines = Files.readAllLines(inputDataFile.toPath());
    for(String line : lines) {
      String[] employeeData = line.split(",");
      int id = Integer.parseInt(employeeData[0]);
      int salary = Integer.parseInt(employeeData[1]);
      // Last name & first name don't need conversion to another datatype.
      String lastName = employeeData[2];
      String firstName = employeeData[3];
      // Now construct a CompanyEmployee object for each employee
      CompanyEmployee empl = new CompanyEmployee(lastName, firstName, id, salary);
      // Add CompanyEmployee object to ArrayList
      employeeList.add(empl);
    }
  }
  catch(FileNotFoundException ex) {
    System.out.println("File not found: " + ex.getMessage());
  }
  catch(IOException ex) {
    System.out.println("I/O error: " + ex.getMessage());
  }
  catch(NumberFormatException ex) {
     System.out.println("Number Format Error: " + ex.getMessage());
  }
  return employeeList;
}
```

TRY IT

Directions: Add the readEmployees() method to your CompanyEmployeesProgram.java. Don't forget the import statements at the top of the file.

Here is the first line of data from the .csv file:

EXAMPLE

```
10001,75000,Johnson,Mary
```

REFLECT

This CSV data shows an employee ID of 10001, a salary of $75000, a last name of Johnson, and a first name of Mary. Each line in the file corresponds to another employee. However, there are some potential errors and problems that could exist. Let's give it a try by switching the name slightly for the file to *wrong.csv* file instead of *employees.csv* file.

TRY IT

Directions: Add this code for the main() method:

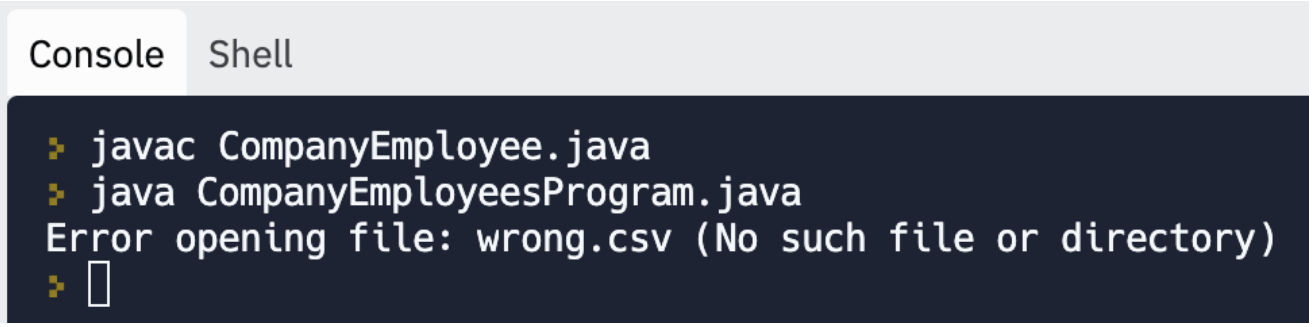
```
public static void main(String[] args) {
   // Try using the wrong file name
   ArrayList<CompanyEmployee> employees = readEmployees("wrong.csv");
 }
```

TRY IT

Directions: Next, switch out the filename for *wrong.csv* and try running the program.

REFLECT

Do you see the same output as shown below?



This output is produced by the first catch block (rather than the rather messy output from an uncaught exception).

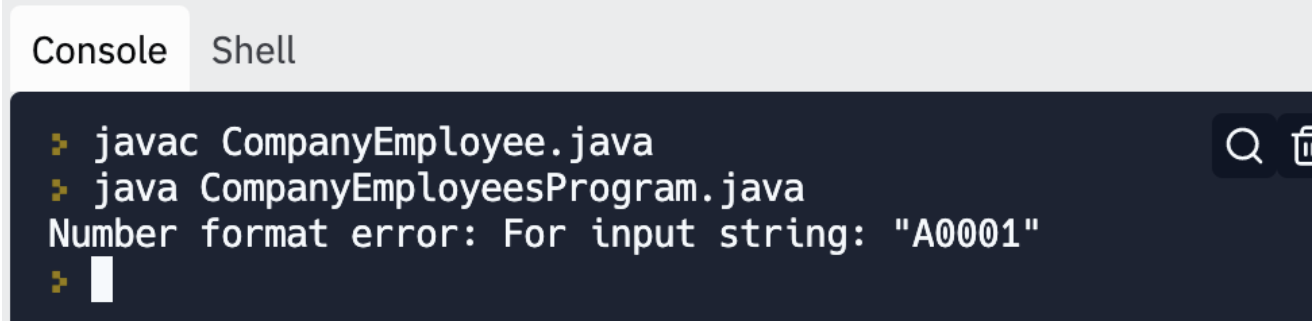
TRY IT

Directions: Change the file name back to the correct name employees.csv before continuing. The second catch block handles NumberFormatExceptions. This type of exception occurs if the Integer.parseInt() can't convert a String to an int because it contains non-numeric characters. Open the employees.csv file and edit the first line so that the ID begins with the letter A (which doesn't occur in a base 10 integer) like this:

EXAMPLE

```
A0001,75000,Johnson,Mary
```

Running the program now triggers the second catch block, so the output looks like this:



TRY IT

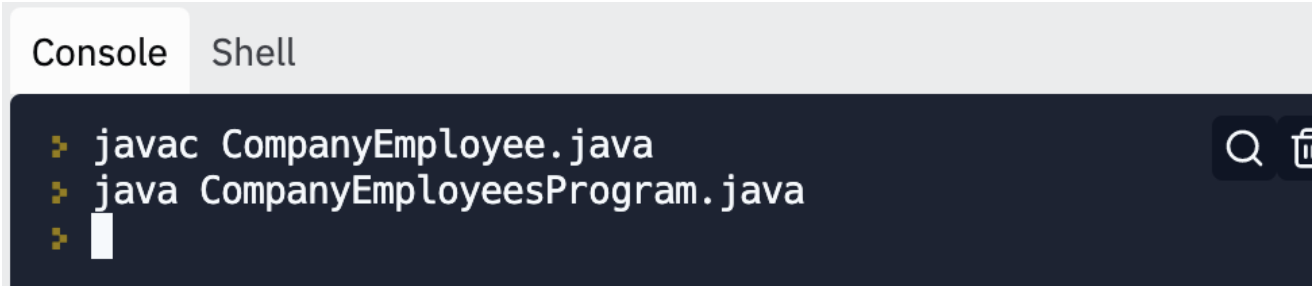
Directions: Edit the first line in the CSV file to correct the entry so that it starts with 1 rather than A:

EXAMPLE

```
10001,75000,Johnson,Mary
```

TRY IT

Directions: Go ahead and run the code when the file name and the data are correct. No exceptions should be thrown, so now there should be no output from the program:



REFLECT

There is no output as there were no errors. The program up to this point shouldn't have any errors to begin with. Now you can move on to the next step to be able to write to the file:

```
public static void writeEmployees(String csvFile, ArrayList<String> employees) {
  File outputFile = new File(csvFile);
  try {
    // Write to output file in APPEND mode
    Files.write(outputFile.toPath(), employees, StandardOpenOption.APPEND);
  }
  catch(IOException ex) {
    System.out.println("Error writing to file: " + ex.getMessage());
  }
}
```

REFLECT

In this writeEmployees() method, you have the parameters set to take the`employees` list. Similar to the readEmployees() method, you will also incorporate the `try` and `catch` blocks since you are setting this method

up for error handling as well. Within the try block, you are opening up the employees.csv file and writing to it in append mode, thanks to the argument StandardOpenOption.APPEND.

Here is an example of how writing to the file is handled by this line in the method:

↗ EXAMPLE

```
Files.write(outputFile.toPath(), employees, StandardOpenOption.APPEND);
```

REFLECT

The first argument is provided by the call to OutputFile.toPath() that converts the File object to the Path object that the Files.write() method expects. The second argument is the iterable ArrayListwith the lines for the employees to write to the file. The last argument, StandardOpenOption.APPEND, puts the write() method into append mode.

TRY IT

Directions: Next, add the *writeEmployees()* method to your program:

Let's take the next step to create a method that prompts the user for the new employee's information.

TRY IT

Directions: We will name this addEmployees() and pass in a Scanner to read from the console the name of the CSV file for the output:

```
public static void addEmployees(Scanner input, String csvFile) {
   ArrayList<CompanyEmployee> employeesToAdd = new ArrayList<>();
   char keepGoing = 'Y';
   while(keepGoing == 'Y') {
     System.out.print("Enter Employee Last Name: ");
     String last = input.nextLine();
     System.out.print("Enter Employee First Name: ");
     String first = input.nextLine();
     System.out.print("Enter Employee ID#: ");
     int id = input.nextInt();
     System.out.print("Enter Salary: ");
     int salary = input.nextInt();
     // Remove new line remaining in input buffer
     input.nextLine();
     CompanyEmployee employee = new CompanyEmployee(last, first, id, salary);
     employeesToAdd.add(employee);

     // Check if user want to continue adding employees
     System.out.print("Continue adding? (Y/N): ");
     // Read line as a String but just grab 1st char
     keepGoing = input.nextLine().charAt(0);
   }

   // Now call method to write new data to file
```

```
    writeEmployees("employee.csv", employeesToAdd);
  }
```

REFLECT

After reading in the data entered by the user, we create a CompanyEmployee object and add it to the ArrayList of employees to add to the file.

BIG IDEA

You may have noticed that we have been creating variables that are very similar, employee and employees, inside our functions. As long as these variables are within local scope, we will have no issues. Remember, if we were to pull one or two of these out from the function scope, we could potentially have scope errors like naming collisions. Always keep in mind that when creating variables, you should know what they are called and where they exist.

This newly created variable employee is appended to the *employees* list that was passed into the function. After this is done, we call the *write_employees()* function passing in the employees list. After that has been returned, we output to the screen that the employee was added:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class CompanyEmployeesProgram {
  public static void main(String[] args) {
    ArrayList<CompanyEmployee> employees = readEmployees("employees.csv");
    for(CompanyEmployee empl : employees) {
      System.out.println(empl);
    }
    ArrayList<String> newEmployees = new ArrayList<>();
  }

  public static ArrayList<CompanyEmployee> readEmployees(String csvFile) {
    // Create an empty ArrayList of CompanyEmployee objects
    ArrayList<CompanyEmployee> employeeList = new ArrayList<>();

    // File object for accessing the CSV file
    File inputDataFile = new File(csvFile);
    List<String> lines = new ArrayList<>();
    // Because the following statements can through exceptions, they are in a try block
    try {
      lines = Files.readAllLines(inputDataFile.toPath());
      for(String line : lines) {
        String[] employeeData = line.split(",");
```

```
      int id = Integer.parseInt(employeeData[0]);
      int salary = Integer.parseInt(employeeData[1]);
      // Last name & first name don't need conversion to another datatype.
      String lastName = employeeData[2];
      String firstName = employeeData[3];
      // Now construct a CompanyEmployee object for each employee
      CompanyEmployee empl = new CompanyEmployee(lastName, firstName, id, salary);
      // Add CompanyEmployee object to ArrayList
      employeeList.add(empl);
    }
   }
   catch(FileNotFoundException ex) {
    System.out.println("File not found: " + ex.getMessage());
   }
   catch(IOException ex) {
    System.out.println("I/O error: " + ex.getMessage());
   }
   catch(NumberFormatException ex) {
    System.out.println("Number Format Error: " + ex.getMessage());
   }
   return employeeList;
  }

  public static void addEmployees(Scanner input, String csvFile) {
   ArrayList<CompanyEmployee> employeesToAdd = new ArrayList<>();
   char keepGoing = 'Y';
   while(keepGoing == 'Y') {
    System.out.print("Enter Employee Last Name: ");
    String last = input.nextLine();
    System.out.print("Enter Employee First Name: ");
    String first = input.nextLine();
    System.out.print("Enter Employee ID#: ");
    int id = input.nextInt();
    System.out.print("Enter Salary: ");
    int salary = input.nextInt();
    // Remove new line remaining in input buffer
    input.nextLine();
    CompanyEmployee employee = new CompanyEmployee(last, first, id, salary);
    employeesToAdd.add(employee);

    // Check if user want to continue adding employees
    System.out.print("Continue adding? (Y/N): ");
    // Read line as a String but just grab 1st char
    keepGoing = input.nextLine().charAt(0);
   }

   // Now call method to write new data to file
   writeEmployees("employee.csv", employeesToAdd);
  }

  public static void writeEmployees(String csvFile, ArrayList<CompanyEmployee> employees) {
```

```
    // Convert ArrayList<CompanyEmployee> to ArrayList<String>
    ArrayList<String> newEmployees = new ArrayList<>();
    for(CompanyEmployee empl : employees) {
      newEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
                  "," + empl.getFirstName());
    }
    File outputFile = new File(csvFile);
    try {
      // Write to output file in APPEND mode
      Files.write(outputFile.toPath(), newEmployees, StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error writing to file: " + ex.getMessage());
    }

  }
}
```

Let's put this together with what we have so far by first returning the employees list from`read_employees()` function, and then we will call the `add_employee()` function.

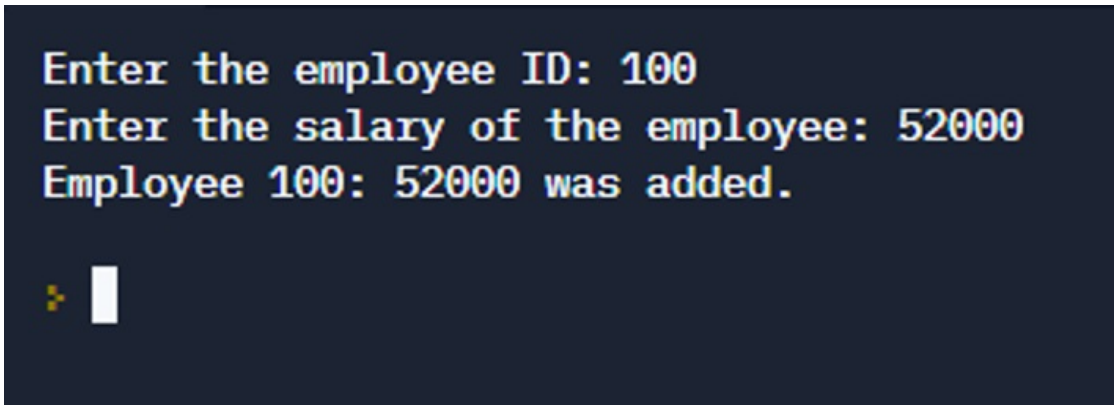
TRY IT

Directions: Add the `add_employee()` function to your program. Remember to also add the calls to the `read_employees()` and `add_employee()` functions at the bottom. When finished, run the program and give input for the employee ID and the salary of your first employee.

EXAMPLE

> Use 100 as ID and 52,000 as salary.

The output screen would reflect:



On screen, it looks correct so far. Let's take a look at what is in the*employees.csv* file.

TRY IT

Directions: See if your employee.csv file is showing the correct data too.

Here are the expected results:

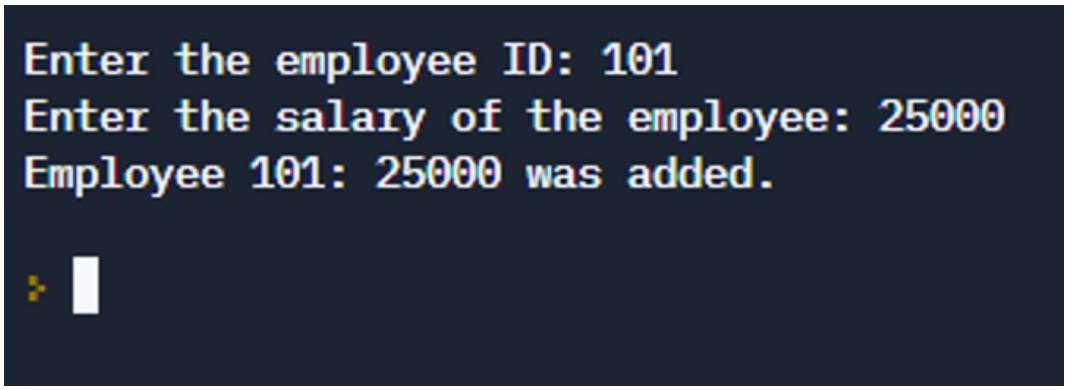
REFLECT

This looks good, as well as the data was saved correctly.

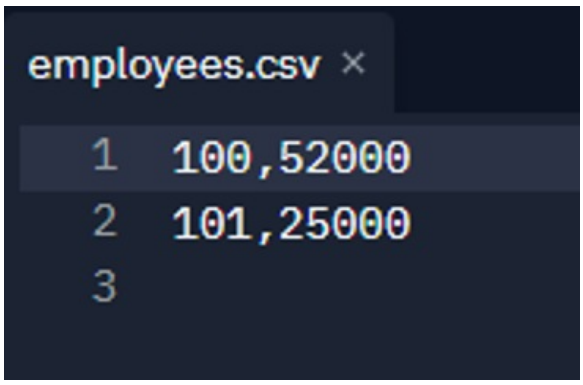
TRY IT

Directions: Run the program again now.

We expect to see the second employee ID as 101 and salary of 25000.



Looking back again at the employees.csv file, we should see:



REFLECT

This is useful, but it's probably not always ideal to have to keep looking at the *employees.csv* file. As an alternative, you could create a function to output the list of employees. Remember back when you first learned about loops, that an iterable is any object that can return its members one at a time. The `enumerate()` function from Java allows you to loop in the same way with the data.

Here is a method called listEmployees() that prints out a numbered list based on the data in the CSV file (which is passed in as a parameter):

```
public static void listEmployees(String csvFile) {
   ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
   int menuNumber = 1;
   for(CompanyEmployee empl : employees) {
     System.out.println(menuNumber++ + ". " + empl);
   }
 }
```

REFLECT

Let's break down this method. Here, you are passing the name of the file as a string. The contents of the CSV file are read into an ArrayList, which is, in turn, iterated over by an enhanced `for` loop, The variable `menuNumber` is initialized to 1 (since the menu numbers need to count the way people do, not starting with 0 as the computer does) and incremented on each pass through the loop. The `println()` relies on the CompanyEmployee class's toString() method (implicitly, since it is not called directly) to format the output so that it is simple to read. In this line, note how the ++ increment operator is placed right after the variable name and is then followed by a space before the + that does the concatenation:

EXAMPLE

```
System.out.println(menuNumber++ + ". " + empl);
```

The current value of menuNumber is concatenated first and then incremented (so the new value will be in place for the next iteration).

Let’s review what happens when we run this function with the rest of our program.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class CompanyEmployeesProgram {
  public static void main(String[] args) {
    listEmployees("employees.csv");
  }

  public static ArrayList<CompanyEmployee> readEmployees(String csvFile) {
    // Create an empty ArrayList of CompanyEmployee objects
    ArrayList<CompanyEmployee> employeeList = new ArrayList<>();

    // File object for accessing the CSV file
    File inputDataFile = new File(csvFile);
    List<String> lines = new ArrayList<>();
    // Because the following statements can through exceptions, they are in a try block
```

```
  try {
    lines = Files.readAllLines(inputDataFile.toPath());
    for(String line : lines) {
      String[] employeeData = line.split(",");
      int id = Integer.parseInt(employeeData[0]);
      int salary = Integer.parseInt(employeeData[1]);
      // Last name & first name don't need conversion to another datatype.
      String lastName = employeeData[2];
      String firstName = employeeData[3];
      // Now construct a CompanyEmployee object for each employee
      CompanyEmployee empl = new CompanyEmployee(lastName, firstName, id, salary);
      // Add CompanyEmployee obejct to ArrayList
      employeeList.add(empl);
    }
  }
  catch(FileNotFoundException ex) {
    System.out.println("File not found: " + ex.getMessage());
  }
  catch(IOException ex) {
    System.out.println("I/O error: " + ex.getMessage());
  }
  catch(NumberFormatException ex) {
    System.out.println("Number Format Error: " + ex.getMessage());
  }
  return employeeList;
}

public static void addEmployees(Scanner input, String csvFile) {
  ArrayList<CompanyEmployee> employeesToAdd = new ArrayList<>();
  char keepGoing = 'Y';
  while(keepGoing == 'Y') {
    System.out.print("Enter Employee Last Name: ");
    String last = input.nextLine();
    System.out.print("Enter Employee First Name: ");
    String first = input.nextLine();
    System.out.print("Enter Employee ID#: ");
    int id = input.nextInt();
    System.out.print("Enter Salary: ");
    int salary = input.nextInt();
    // Remove new line remaining in input buffer
    input.nextLine();

    CompanyEmployee employee = new CompanyEmployee(last, first, id, salary);
    employeesToAdd.add(employee);

    // Check if user want to continue adding employees
    System.out.print("Continue adding? (Y/N): ");
    // Read line as a String but just grab 1st char
    keepGoing = input.nextLine().charAt(0);
  }
```

```
    // Now call method to write new data to file
    writeEmployees("employee.csv", employeesToAdd);
  }

  public static void writeEmployees(String csvFile, ArrayList<CompanyEmployee> employees) {
    // Convert ArrayList<CompanyEmployee> to ArrayList<String>
    ArrayList<String> newEmployees = new ArrayList<>();
    for(CompanyEmployee empl : employees) {
      newEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
                  "," + empl.getFirstName());
    }
    File outputFile = new File(csvFile);
    try {
      // Write to output file in APPEND mode
      Files.write(outputFile.toPath(), newEmployees, StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error writing to file: " + ex.getMessage());
    }

  }

  public static void listEmployees(String csvFile) {
    ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
    int menuNumber = 1;
    for(CompanyEmployee empl : employees) {
      System.out.println(menuNumber++ + ". " + empl);
    }
  }
}
```

TRY IT

Directions: Add the list_employees() function to your program. Notice that we replaced the call to add_employee() function with the new list_employees() function. This time, the program will not have input, only listing the contents of the employee.csv. Remember to keep the read_employees() and list_employees() functions at the bottom. When finished, run the program.

The output is much cleaner now, as we have a centralized place to see the list of employees with the employee ID and the salary for each.

2. Delete From File

Next, you will create a method that will remove an employee from the file based on the employee ID.

```
public static boolean deleteEmployee(String csvFile, int emplID) {
   // Read contents of existing file
   ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
   // Track if employee has been found and deleted
   boolean employeeDeleted = false;
   CompanyEmployee emplToDelete = null;
   for(CompanyEmployee empl : employees) {
     if(empl.getId() == emplID) {
       emplToDelete = empl;
     }
   }
   if(emplToDelete != null) {
     employees.remove(emplToDelete);
     employeeDeleted = true;
     ArrayList<String> remainingEmployees = new ArrayList<>();
     for(CompanyEmployee empl : employees) {
       remainingEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
         "," + empl.getFirstName());
     }
     File outputFile = new File(csvFile);
     try {
       // Write to output file - overwrite if it already exists
       Files.write(outputFile.toPath(), remainingEmployees, StandardOpenOption.TRUNCATE_EXISTING);
     }
     catch(IOException ex) {
       System.out.println("Error writing to file: " + ex.getMessage());
     }
   }
   return employeeDeleted;
 }
```

In this method, we will prompt the user for an employee ID. Next, we will loop through the list to find that employee ID. If we find it in the list, we will remove it using the `remove()` method and set `employeeDeleted` to true. If, after looping through the entire list, the employeeDeleted variable is still false, it means that the employee was not found. Otherwise, the method writes an updated list with the item removed. Let's give it a try now.

TRY IT

Directions: Input the following to loop through the list:

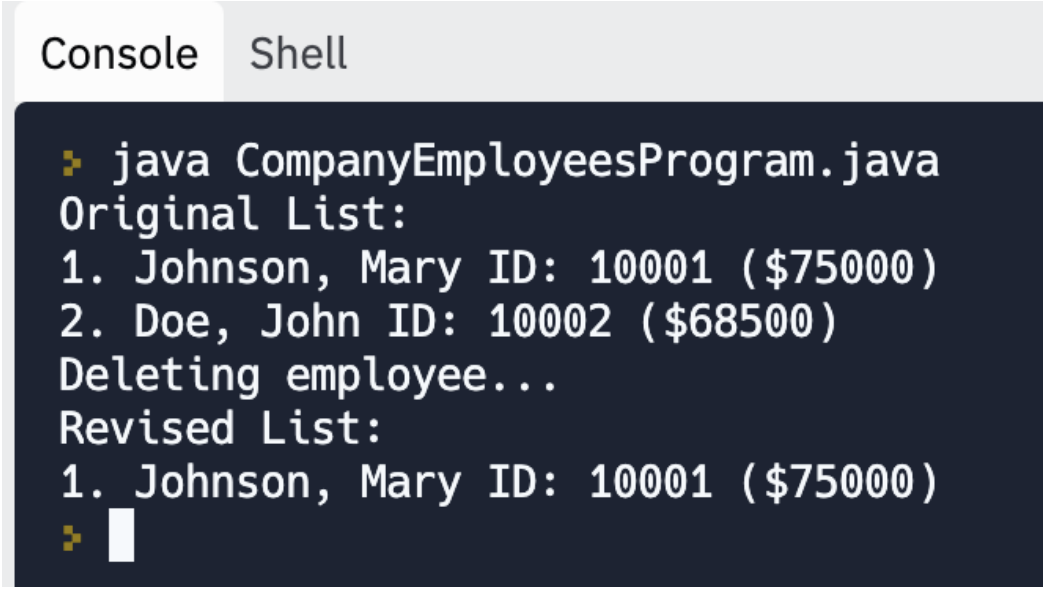
```
import java.io.*;
import java.nio.file.*;
import java.util.ArrayList;
```

```
import java.util.List;
import java.util.Scanner;

class CompanyEmployeesProgram {
  public static void main(String[] args) {
    System.out.println("Original List: ");
    listEmployees("employees.csv");
    System.out.println("Deleting employee...");
    deleteEmployee("employees.csv", 10002);
    System.out.println("Revised List: ");
    listEmployees("employees.csv");
  }

  public static ArrayList<CompanyEmployee> readEmployees(String csvFile) {
    // Create an empty ArrayList of CompanyEmployee objects
    ArrayList<CompanyEmployee> employeeList = new ArrayList<>();

    // File object for accessing the CSV file
    File inputDataFile = new File(csvFile);
    List<String> lines = new ArrayList<>();
    // Because the following statements can through exceptions, they are in a try block
    try {
      lines = Files.readAllLines(inputDataFile.toPath());
      for(String line : lines) {
        String[] employeeData = line.split(",");
        int id = Integer.parseInt(employeeData[0]);
        int salary = Integer.parseInt(employeeData[1]);
        // Last name & first name don't need conversion to another datatype.
        String lastName = employeeData[2];
        String firstName = employeeData[3];
        // Now construct a CompanyEmployee object for each employee
        CompanyEmployee empl = new CompanyEmployee(lastName, firstName, id, salary);
        // Add CompanyEmployee obejct to ArrayList
        employeeList.add(empl);
      }
    }
    catch(FileNotFoundException ex) {
      System.out.println("File not found: " + ex.getMessage());
    }
    catch(IOException ex) {
      System.out.println("I/O error: " + ex.getMessage());
    }
    catch(NumberFormatException ex) {
      System.out.println("Number Format Error: " + ex.getMessage());
    }
    return employeeList;
  }

  public static void addEmployees(Scanner input, String csvFile) {
    ArrayList<CompanyEmployee> employeesToAdd = new ArrayList<>();
    char keepGoing = 'Y';
```

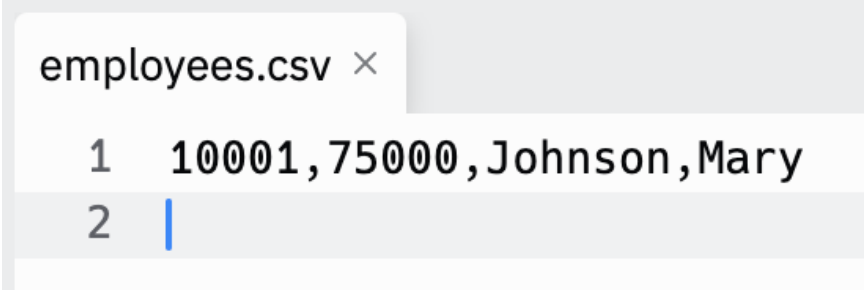
```
  while(keepGoing == 'Y') {
    System.out.print("Enter Employee Last Name: ");
    String last = input.nextLine();
    System.out.print("Enter Employee First Name: ");
    String first = input.nextLine();
    System.out.print("Enter Employee ID#: ");
    int id = input.nextInt();
    System.out.print("Enter Salary: ");
    int salary = input.nextInt();
    // Remove new line remaining in input buffer
    input.nextLine();
    CompanyEmployee employee = new CompanyEmployee(last, first, id, salary);
    employeesToAdd.add(employee);

    // Check if user want to continue adding employees
    System.out.print("Continue adding? (Y/N): ");
    // Read line as a String but just grab 1st char
    keepGoing = input.nextLine().charAt(0);
  }

  // Now call method to write new data to file
  writeEmployees("employee.csv", employeesToAdd);
}

public static void writeEmployees(String csvFile, ArrayList<CompanyEmployee> employees) {
  // Convert ArrayList<CompanyEmployee> to ArrayList<String>
  ArrayList<String> newEmployees = new ArrayList<>();
  for(CompanyEmployee empl : employees) {
    newEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
                "," + empl.getFirstName());
  }
  File outputFile = new File(csvFile);
  try {
    // Write to output file in APPEND mode
    Files.write(outputFile.toPath(), newEmployees, StandardOpenOption.APPEND);
  }
  catch(IOException ex) {
    System.out.println("Error writing to file: " + ex.getMessage());
  }

}

public static void listEmployees(String csvFile) {
  ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
  int menuNumber = 1;
  for(CompanyEmployee empl : employees) {
    System.out.println(menuNumber++ + ". " + empl);
  }
}

public static boolean deleteEmployee(String csvFile, int emplID) {
```

```
    // Read contents of existing file
    ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
    // Track if employee has been found and deleted
    boolean employeeDeleted = false;
    CompanyEmployee emplToDelete = null;
    for(CompanyEmployee empl : employees) {
      if(empl.getId() == emplID) {
        emplToDelete = empl;
      }
    }
    if(emplToDelete != null) {
      employees.remove(emplToDelete);
      employeeDeleted = true;
      ArrayList<String> remainingEmployees = new ArrayList<>();
      for(CompanyEmployee empl : employees) {
        remainingEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
          "," + empl.getFirstName());
      }
      File outputFile = new File(csvFile);
      try {
        // Write to output file - overwrite if it already exists
        Files.write(outputFile.toPath(), remainingEmployees, StandardOpenOption.TRUNCATE_EXISTING);
      }
      catch(IOException ex) {
        System.out.println("Error writing to file: " + ex.getMessage());
      }
    }
    return employeeDeleted;
  }
}
```

The output should look like the following:



Looking at the employees file, the result is correct:



3. Adding Main Menu

Now that we have this much of the program written, next we will create a method and revise the code in the application's main() so that it runs more like a real application. You will set up a basic menu that will allow the user to know how to run the program to call each of these functions with a basic menu:

```
public static void displayMenu() {
  System.out.println("\nlist - List all employees");
  System.out.println(" add - Add an employee");
  System.out.println(" del - Delete an employee");
  System.out.println("exit - Exit program");
  System.out.print("Enter Command: ");
}
```

This method allows the user to enter list, add, del, or exit to work through the program. Now, we can set up a basic program that will loop through the program to display the user the menu, and allow the user to perform actions on our employee list using the functions we have created. Here is the revised main() method:

```
public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  String cmd = "";
  System.out.print("Enter the name of the CSV file: ");
  String csvFile = input.nextLine();

  while(!cmd.toLowerCase().equals("exit")) {
    displayMenu();
    cmd = input.nextLine();
    if(cmd.toLowerCase().equals("list")) {
      listEmployees(csvFile);
    }
    else if(cmd.toLowerCase().equals("add")) {
      addEmployees(input, csvFile);
    }
    else if(cmd.toLowerCase().equals("del")) {
       System.out.print("Employee ID to delete: ");
       int emplId = input.nextInt();
        deleteEmployee(csvFile, emplId);
    }
  }
```

}

The only difference here is that we're using a while loop to iterate until the exit is entered.

The completed program will look like the following:

```
import java.io.*;
import java.nio.file.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class CompanyEmployeesProgram {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String cmd = "";
    System.out.print("Enter the name of the CSV file: ");
    String csvFile = input.nextLine();

    while(!cmd.toLowerCase().equals("exit")) {
      displayMenu();
      cmd = input.nextLine();
      if(cmd.toLowerCase().equals("list")) {
        listEmployees(csvFile);
      }
      else if(cmd.toLowerCase().equals("add")) {
        addEmployees(input, csvFile);
      }
      else if(cmd.toLowerCase().equals("del")) {
        System.out.print("Employee ID to delete: ");
        int emplId = input.nextInt();
        // Remove remaining \n in input
        input.nextLine();
        deleteEmployee(csvFile, emplId);
      }
    }
  }

  public static void displayMenu() {
    System.out.println("\nlist - List all employees");
    System.out.println(" add - Add an employee");
    System.out.println(" del - Delete an employee");
    System.out.println("exit - Exit program");
    System.out.print("Enter Command: ");
  }

  public static ArrayList<CompanyEmployee> readEmployees(String csvFile) {
    // Create an empty ArrayList of CompanyEmployee objects
    ArrayList<CompanyEmployee> employeeList = new ArrayList<>();

    // File object for accessing the CSV file
```

```
  File inputDataFile = new File(csvFile);
  List<String> lines = new ArrayList<>();
  // Because the following statements can through exceptions, they are in a try block
  try {
    lines = Files.readAllLines(inputDataFile.toPath());
    for(String line : lines) {
      String[] employeeData = line.split(",");
      int id = Integer.parseInt(employeeData[0]);
      int salary = Integer.parseInt(employeeData[1]);
      // Last name & first name don't need conversion to another datatype.
      String lastName = employeeData[2];
      String firstName = employeeData[3];
      // Now construct a CompanyEmployee object for each employee
      CompanyEmployee empl = new CompanyEmployee(lastName, firstName, id, salary);
      // Add CompanyEmployee obejct to ArrayList
      employeeList.add(empl);
    }
  }
  catch(FileNotFoundException ex) {
    System.out.println("File not found: " + ex.getMessage());
  }
  catch(IOException ex) {
    System.out.println("I/O error: " + ex.getMessage());
  }
  catch(NumberFormatException ex) {
    System.out.println("Number Format Error: " + ex.getMessage());
  }
  return employeeList;
}

public static void addEmployees(Scanner input, String csvFile) {
  ArrayList<CompanyEmployee> employeesToAdd = new ArrayList<>();
  char keepGoing = 'Y';
  while(keepGoing == 'Y') {
    System.out.print("Enter Employee Last Name: ");
    String last = input.nextLine();
    System.out.print("Enter Employee First Name: ");
    String first = input.nextLine();
    System.out.print("Enter Employee ID#: ");
    int id = input.nextInt();
    System.out.print("Enter Salary: ");
    int salary = input.nextInt();
    // Remove new line remaining in input buffer
    input.nextLine();
    CompanyEmployee employee = new CompanyEmployee(last, first, id, salary);
    employeesToAdd.add(employee);

    // Check if user want to continue adding employees
    System.out.print("Continue adding? (Y/N): ");
    // Read line as a String but just grab 1st char
    keepGoing = input.nextLine().charAt(0);
```

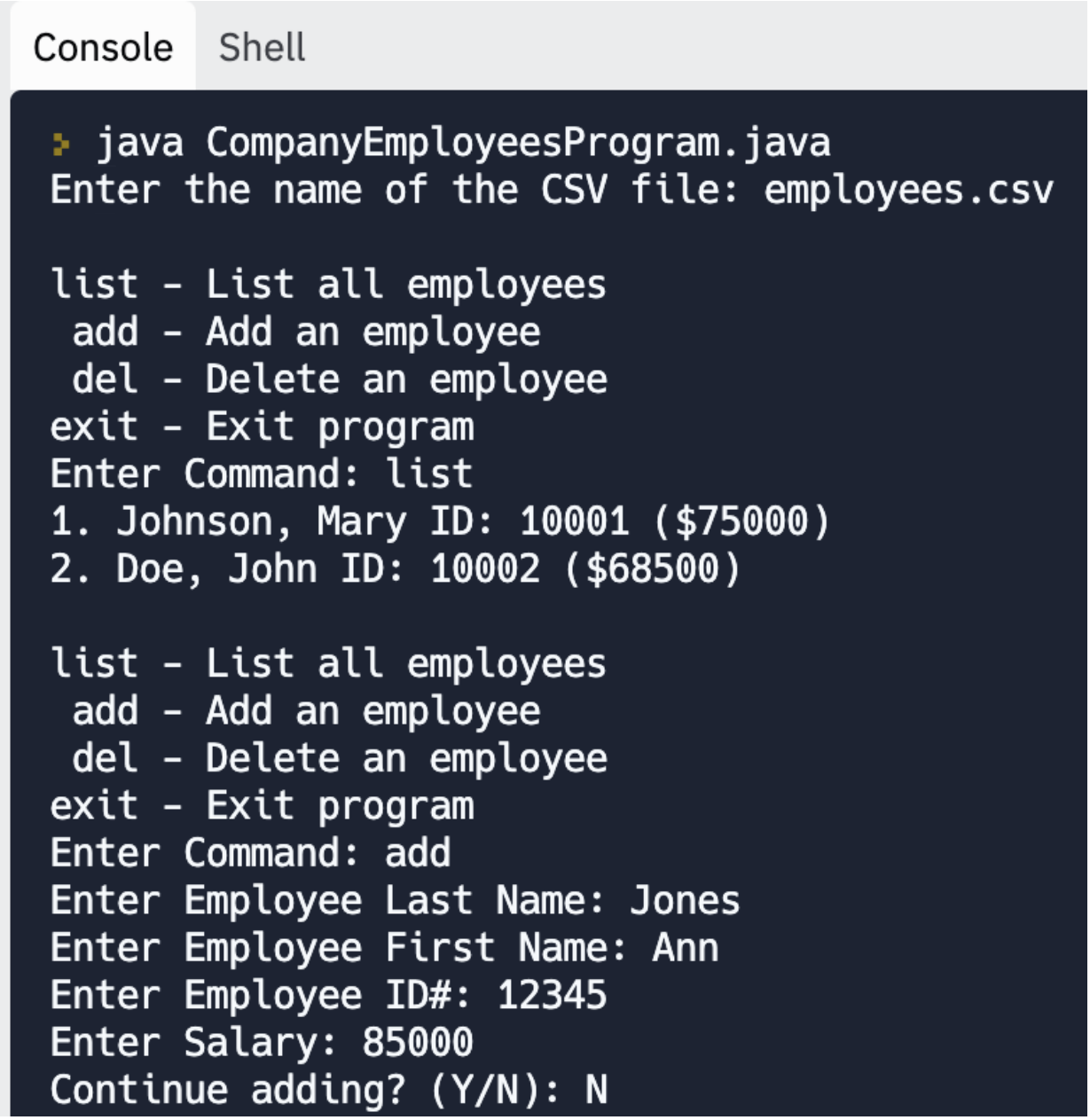
```
  }

  // Now call method to write new data to file
  writeEmployees(csvFile, employeesToAdd);
 }

 public static void writeEmployees(String csvFile, ArrayList<CompanyEmployee> employees) {
  // Convert ArrayList<CompanyEmployee> to ArrayList<String>
  ArrayList<String> newEmployees = new ArrayList<>();
  for(CompanyEmployee empl : employees) {
   newEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
                "," + empl.getFirstName());
  }
  File outputFile = new File(csvFile);
  try {
   // Write to output file in APPEND mode
   Files.write(outputFile.toPath(), newEmployees, StandardOpenOption.APPEND);
  }
  catch(IOException ex) {
   System.out.println("Error writing to file: " + ex.getMessage());
  }

 }

 public static void listEmployees(String csvFile) {
  ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
  int menuNumber = 1;
  for(CompanyEmployee empl : employees) {
   System.out.println(menuNumber++ + ". " + empl);
  }
 }

 public static boolean deleteEmployee(String csvFile, int emplID) {
  // Read contents of existing file
  ArrayList<CompanyEmployee> employees = readEmployees(csvFile);
  // Track if employee has been found and deleted
  boolean employeeDeleted = false;
  CompanyEmployee emplToDelete = null;
  for(CompanyEmployee empl : employees) {
   if(empl.getId() == emplID) {
    emplToDelete = empl;
   }
  }
  if(emplToDelete != null) {
   employees.remove(emplToDelete);
   employeeDeleted = true;
   ArrayList<String> remainingEmployees = new ArrayList<>();
   for(CompanyEmployee empl : employees) {
    remainingEmployees.add(empl.getId() + "," + empl.getSalary() + "," + empl.getLastName() +
     "," + empl.getFirstName());
   }
```

```
      File outputFile = new File(csvFile);
      try {
        // Write to output file - overwrite if it already exists
        Files.write(outputFile.toPath(), remainingEmployees, StandardOpenOption.TRUNCATE_EXISTING);
      }
      catch(IOException ex) {
        System.out.println("Error writing to file: " + ex.getMessage());
      }
    }
    return employeeDeleted;
  }
}
```

TRY IT

Directions: Try and see if the results from the employee is as expected:

The output should reflect the following:



```
list - List all employees
 add - Add an employee
 del - Delete an employee
exit - Exit program
Enter Command: list
1. Johnson, Mary ID: 10001 ($75000)
2. Doe, John ID: 10002 ($68500)
3. Jones, Ann ID: 12345 ($85000)

list - List all employees
 add - Add an employee
 del - Delete an employee
exit - Exit program
Enter Command:
```

```
list - List all employees
 add - Add an employee
 del - Delete an employee
exit - Exit program
Enter Command: del
Employee ID to delete: 12345

list - List all employees
 add - Add an employee
 del - Delete an employee
exit - Exit program
Enter Command: list
1. Johnson, Mary ID: 10001 ($75000)
2. Doe, John ID: 10002 ($68500)

list - List all employees
 add - Add an employee
 del - Delete an employee
exit - Exit program
Enter Command: exit
>
```

REFLECT

Everything looks accurate. Now the next time we launch the program, it should work exactly as expected with the employee list available to start with.

SUMMARY

In this lesson, you learned about how to create a program that allows the user to open a .csv file, add items to the .csv file, read from the .csv file and delete them. This included **storing information to a file, deleting from a file**, and **adding a main menu program** to help iterate through that entire process.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**

Terms to Know

Accessor Method ("Getter" Method)

A public method that reads and returns the value in an attribute.

Aggregation

Aggregation is the combining of Java classes to create another class (instances of the combined classes are used as attributes in the new class).

Attributes

Attributes are data defined in the class template and are the characteristics or properties of an object.

Base Class

A base class is also known as a parent class or superclass. It is a class that is being inherited from.

Central Processing Unit

The Central Processing Unit is the heart of any computer. It is what runs the software that we write; it is also called the “CPU” or the “processor.”

Class

The class keyword defines the data and code that will make up each of the objects. When defining a class, it starts with the class keyword and is then followed by the name of the class and a colon.

Class Instantiation

The process of creating an instance of a class, which is called an object.

Composition

Composition is a more specific type of aggregation where the component can't really exist apart from the larger object.

Compound Assignment Operator for Multiplication (*=)

This operator multiplies the variable to the left by the value or expression to the right and assigns the product back to the variable to the left.

Constructor

The constructor's sole purpose is to initialize an object and, if it is a parameterized constructor, to set the object’s attributes to specific (non-default) values. A constructor only exists inside a class and is only used to set up a new object based on the class's template.

Default Constructor

The default constructor does not have any parameters. It allocates memory for an object and initializes the attributes to default values.

Driver Class

The class in an application that contains the main() method.

Enumeration

StandardOpenOption is an example of a Java enumeration. An enumeration is a named collection of constant values.

File Handle

The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data.

Hexadecimal

A value expressed in a base 16 number system (rather than base 10).

Implementation

The specific code written to produce a method that carries out the action described or specified in an interface.

Inheritance

Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

Interface

The defined ways in which one object interacts with another.

Library Class

A class that is included in the libraries that come with the Java Virtual Machine and developer tools and is available on all machines running a given version of Java.

Main Memory

The main memory is used to store information that the CPU needs in a hurry. Main memory is nearly as fast as the CPU. The CPU and memory are where our software works and runs.

Mutator Method ("Setter" Method)

A public method that changes the value in an attribute.

Object Class

Every class in Java automatically inherits from the Object class, so it is the base class for all

other Java classes.

Object-Oriented Programming

Object-oriented programming (or OOP for short) is a programming model that organizes the design of code by bundling objects.

Operator

The dot (.) operator connects the object (instance of a class) to the attributes and methods of that object.

Package

A package is a collection of interrelated classes in a particular section of the Java class library.

Parameterized Constructor

Includes method parameters that are used to initialize the values of some or all of the attributes.

Scope

The scope of a variable or object refers to where it can be used in a program.

Secondary Memory

Secondary memory is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer.

Subclass

A subclass is also known as a child class. It is the class that inherits from another class. A subclass can “extend” the base class, meaning that it can define additional data and related behavior that will not affect the base class.

super()

The super() method allows the constructor in a subclass to call the constructor for the base class.