Learning Objectives - Advanced Topics

- **Set up and import an object defined in a separate file**
- **Create and manipulate a list of objects**
- **Define object composition**
- **Compare and contrast component and composite classes**
- **Identify when to use composition and when to use inheritance**
- **Represent an object as a string**
- **Create and implement an interface**
- **Compare the values of two objects**

Importing User-Defined Classes

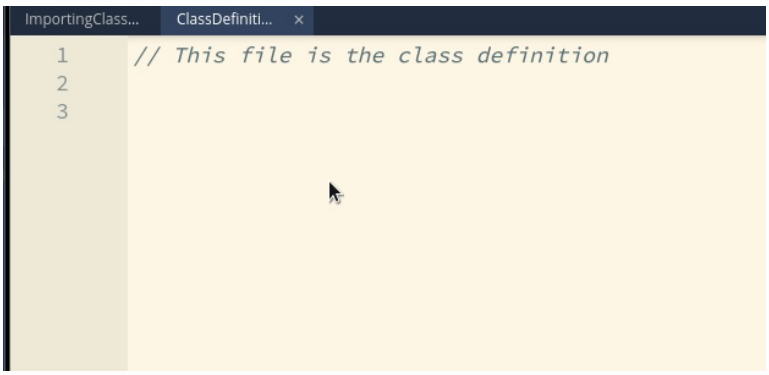
Importing a User-Defined Class

You may have noticed that writing your own classes adds many lines of code before your the logic of your program even begins. To better organize your code, define classes in a separate file. Then import the module into your program so you can use the class.

Make sure you are in the file for defining the class (look at the comments at the top of the files). Start by creating a simple `Employee` class.

▼ **Switching Between Files**

Notice that the IDE on the left now has more than one file. Click on the tabs on the top to switch between the files.



Switching Files

```
// This file is the class definition

class Employee {
  private String name;
  private String title;

  public Employee(String n, String t) {
    name = n;
    title = t;
  }

  public void display() {
    System.out.println("Employee: " + name);
    System.out.println("Title: " + title);
  }
}
```

Now go to the file for your program (look at the comments at the top of the files). Instantiate an `Employee` object with two strings as arguments. Then call the `display` method.

```
    //add code below this line

    Employee e = new Employee("Calvin", "CEO");
    e.display();

    //add code above this line
```

How Does This Work?

You may have noticed that the file that implements the `Employee` object did not use the `import` keyword. How does Java know to look in the `ClassDefinition.java` file to find the class definition for `Employee`? Java has a two-step process to run a program. First you compile it. This turns the source code (the code you wrote) and compiles (transforms) it to byte code. The next step is to run the byte code on the Java Virtual Machine. All of this is hidden behind the `TRY IT` buttons.

If compile two (or more) files that are in the same folder, Java now knows about all of the classes in all of the files. You do not need an `import` statement because the compilation process has already informed Java of all of the classes.

The terminal command `javac` means to compile Java source code. Give this command the path to each of the Java files `ImportingClasses.java` and `ClassDefinition.java`. Notice how both Java files are in the same folder. These files **must** be in the same folder. Copy and paste the command below into the terminal and press `Enter` on the keyboard. This is the point where Java would return any error messages. If Java does not do anything, then the compilation process was successful.

```
javac code/advanced/ImportingClasses.java
        code/advanced/ClassDefinition.java
```

Once the byte code has been made, go ahead and run the program. The command to run a Java program is `java`. The parameter `-cp` stands for class path. This is the folder in which the Java program lives. Finally `ImportingClasses` is the actual program (the file with the `main` method). This command will produce the output from your program.

```
java -cp code/advanced/ ImportingClasses
```

The rest of this module will continue to use the `TRY IT` button, it is important to understand how Java can use multiple files for a single program.

ArrayList of Objects

ArrayList of Objects

You may find yourself needing several instances of a class. Keeping these objects in an ArrayList is a good way to organize your code. It also simplifies interacting with the objects because you can iterate through the ArrayList as opposed to manipulating each object separately.

The first thing you need to do is to create a class. We are going to make a class to represent the apps on your smartphone.

```
// Define the App class

class App {
  private String name;
  private String description;
  private String category;

  public App(String n, String d, String c) {
    name = n;
    description = d;
    category = c;
  }

  public void display() {
    System.out.println(String.format("%s is a(n) %s app that is
        %s.", name, category, description));
  }
}
```

Next, we need to create a list with objects of the `App` class as each element. To speed this up, we are going to read from a CSV file that has the information for the `name`, `description`, and `category` attributes.

▼ **Why use a CSV file?**

This page is about manipulating a list of objects. Instead of manually creating several objects, we are going to read information from the `apps.csv` file and use it to create several objects in a simple loop.

```
name,description,category
Gmail,the official app for Google's email
      service,communication
FeedWrangler,used to read websites with an RSS feed,internet
Apollo,used to read Reddit,social media
Instagram,the offical app for Facebook's Instagram
      service,social media
Overcast,used to manage and listen to podcasts,audio
Slack,the official app for Slack's email
      replacement,communication
YouTube,the official app for Google's video service,video
FireFox,used to browse the web,internet
OverDrive,used to checkout ebooks from the library,ebooks
Authenticator,used for two-factor authentication,internet
```

Make sure that you are altering the `ListOfObjects.java` file. Start by creating an ArrayList of type `App` and a variable that has the path to the CSV file. Then read the file and skip the header row. Iterate through each row of the CSV file. Add a new `App` object to the list. The first element is the name of the app, the second element is the description, and the third element is the category. Finally, print the ArrayList.

```
  //add code below this line

  ArrayList<App> apps = new ArrayList<App>();
  String path = "code/advanced/app.csv";

  try {
    CSVReader reader = new CSVReader(new FileReader(path));
    reader.skip(1);
    for (String[] row : reader) {
      apps.add(new App(row[0], row[1], row[2]));
    }
    reader.close();
    System.out.println(apps);
  } catch (Exception e) {
    System.out.println(e);
  } finally {
    System.out.println("Finished reading a CSV");
  }

  //add code above this line
```

▼ **Explaining the Output**

The output from the above print statement is an ArrayList of elements that look something like this:

```
App@378bf509
```

This is how Java represents an object. Each element is an `App` object. The `@` symbol and numbers is the location in memory where the object is stored (your memory locations will be different). If you see 10 of these, then your code is working properly.

Interacting with the Objects

Now that there is a list of objects, we can manipulate each object by iterating through the list. We no longer need the print statement in our program. Replace it with a for loop. On each iteration, call the `display` method.

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
  CSVReader reader = new CSVReader(new FileReader(path));
  reader.skip(1);
  for (String[] row : reader) {
    apps.add(new App(row[0], row[1], row[2]));
  }
  reader.close();
} catch (Exception e) {
  System.out.println(e);
} finally {
  System.out.println("Finished reading a CSV");
}

for (App app : apps) {
  app.display();
}

//add code above this line
```

challenge

Try these variations:

- Call the `display` method on only the third app.

  ▼

  Solution

  Normally, you would use a variable when instantiating an object. In this case, however, objects need to be referenced by the index in a list. Indexes start counting at 0, so the third element would be `.get(2)`:

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
  CSVReader reader = new CSVReader(new
    FileReader(path));
  reader.skip(1);
  for (String[] row : reader) {
    apps.add(new App(row[0], row[1], row[2]));
  }
  reader.close();
} catch (Exception e) {
  System.out.println(e);
} finally {
  System.out.println("Finished reading a CSV");
}

apps.get(2).display();

//add code above this line
```

- Call the `display` method for all objects that are have “social media” as the `category` attribute.

  ▼

  Solution

  Add the `getCAtegory` accessor method to the `App` class.

```
public String getCategory() {
  return category;
}
```

Iterate over the list and use a conditional to determine if the `category` attribute is “social media”. If true, call the `display` method.

```
//add code below this line

ArrayList<App> apps = new ArrayList<App>();
String path = "code/advanced/app.csv";

try {
  CSVReader reader = new CSVReader(new
    FileReader(path));
  reader.skip(1);
  for (String[] row : reader) {
    apps.add(new App(row[0], row[1], row[2]));
  }
  reader.close();
} catch (Exception e) {
  System.out.println(e);
} finally {
  System.out.println("Finished reading a CSV");
}

for (App app : apps) {
  if (app.getCategory().equals("social media")) {
    app.display();
  }
}

//add code above this line
```

Composition

Composition

Composition is a way to make a functional whole out of smaller parts. If you were to create a `Car` class, this would start out as a simple exercise. Every car has a make, a model, and a year it was produced. Representing this data is simple: two strings and an integer.

```
//add class definitions below this line

class Car {
  private String make;
  private String model;
  private int year;

  public Car(String ma, String mo, int y) {
    make = ma;
    model = mo;
    year = y;
  }

  public void describe() {
    System.out.println(String.format("%s %s %s", make, model,
        year));
  }
}

//add class definitions above this line
```

Create an instance of the `Car` class and call the `describe` method.

```
  //add code below this line

  Car car = new Car("De Tomaso", "Pantera", 1979);
  car.describe();

  //add code above this line
```

The `Car` class, however, is missing an important component: the engine. What data type would you use to represent an engine? Creating another class is the best way to do this. Modify the `Car` class so that it has an `engine` attribute and a getter for this attribute. Then create the `Engine` class with attributes for configuration (V8, V6, etc.), displacement, horsepower, and torque. Finally, add the `ignite` method to the `Engine` class.

```
//add class definitions below this line

class Car {
  private String make;
  private String model;
  private int year;
  private Engine engine;

  public Car(String ma, String mo, int y, Engine e) {
    make = ma;
    model = mo;
    year = y;
    engine = e;
  }

  public void describe() {
    System.out.println(String.format("%s %s %s", make, model,
        year));
  }

  public Engine getEngine() {
    return engine;
  }
}

class Engine {
  private String configuration;
  private double displacement;
  private int horsepower;
  private int torque;

  public Engine(String c, double d, int h, int t) {
    configuration = c;
    displacement = d;
    horsepower = h;
    torque = t;
  }

  public void ignite() {
    System.out.println("Vroom vroom!");
  }
}

//add class definitions above this line
```

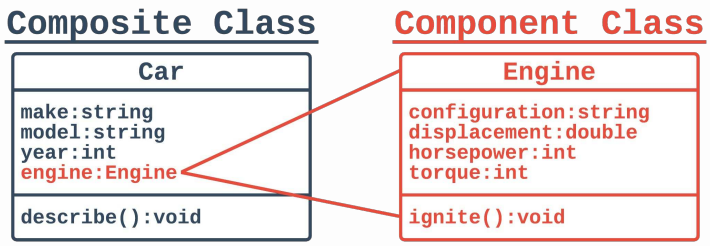
Since the `Car` class takes an `Engine` object, instantiate an `Engine` object first. Then pass that object to the constructor for the `Car` class. To call the `ignite` method, you need to call the getter method for the `engine` attribute and then call the method.

```
//add code below this line

Engine engine = new Engine("V8", 5.8, 326, 344);
Car car = new Car("De Tomaso", "Pantera", 1979, engine);
car.getEngine().ignite();

//add code above this line
```

The combination of the `Car` class and the `Engine` class lead to a better representation of an actual car. This is the benefit of object composition. Because the `Engine` class is a part of the `Car` class, we can say that the `Engine` class is the component class and the `Car` class is the composite class.



.guides/img/advanced/composite_component

challenge

Try these variations:

- In the `Car` class, add the `start` method then have the last line of the script call `start` instead of `ignite`:

```
// Car class
public void start() {
  engine.ignite();
}

// main method
Engine engine = new Engine("V8", 5.8, 326, 344);
Car car = new Car("De Tomaso", "Pantera", 1979, engine);
car.start();
```

- In the `Engine` class, create the `info` method then call `info` instead of `describe`:

```
// Engine class
public void info() {
  describe();
}

// main method
Engine engine = new Engine("V8", 5.8, 326, 344);
Car car = new Car("De Tomaso", "Pantera", 1979, engine);
car.getEngine().info();
```

▼ **Why is there an error?**
Composition is a one-way street. The composite class (the `Car` class) has access to all of the attributes and methods of the component classes (the `Engine` class). However, component classes cannot access the attributes and methods of the composite class.

Composition versus Inheritance

Assume you have the class `MyClass`. You want to use this class in your program, but it is missing some functionality. Do you use inheritance and extend the parent class, or do you use composition and create a component class? Both of these techniques can accomplish the same thing. This is a

complex topic, but you can use a simple test to help you decide. Use inheritance if there is an “is a” relationship, and use composition if there is a “has a” relationship.

For example, you have the `Vehicle` class and you want to make a `Car` class. Ask yourself if a car has a vehicle or if a car is a vehicle. A car is a vehicle; therefore you should use inheritance. Now imagine that you have a `Phone` class and you want to represent an app for the phone. Ask yourself if a phone is an app or if a phone has an app. A phone has an app; therefore you should use composition.

Represent an Object as a String

The `toString` Method

When you print out the instance of a user-created class, Java returns only the class name and its location in memory.

```
//add class definitions below this line

class Animal {
  private int age;

  public Animal(int a) {
    age = a;
  }
}

//add class definitions above this line
```

Instantiate an `Animal` object and print it.

```
  //add code below this line

  Animal a = new Animal(3);
  System.out.println(a);

  //add code above this line
```

This is not very helpful. That is why we have seen code examples where classes have a method called `describe` or `display` that print out a description of the object. However, a better way of representing an object as a string is to override the `toString` method. **Note**, it is not necessary to explicitly call the `toString` method. This is automatically done with `System.out.println`.

```
//add class definitions below this line

class Animal {
  private int age;

  public Animal(int a) {
    age = a;
  }

  public String toString() {
    return getClass().getName() + "[age=" + age + "]";
  }
}

//add class definitions above this line
```

String Representation and Inheritance

The example above uses `getClass().getName()` to print the name of the class instead of manually printing `Animal`. This makes printing a string representation of a subclass much easier. Create the `Dog` class which extends the `Animal` class.

```
//add class definitions below this line

class Animal {
  private int age;

  public Animal(int a) {
    age = a;
  }

  public String toString() {
    return getClass().getName() + "[age=" + age + "]";
  }
}

class Dog extends Animal {
  private String name;
  private String breed;

  public Dog(String n, String b, int a) {
    super(a);
    name = n;
    breed = b;
  }
}

//add class definitions above this line
```

Change the object creation from an `Animal` to a `Dog`. When you run the program, Java will print `Dog` as the class name. However, it only prints out the `age` attribute but not the `name` or `breed` attributes. That is because `toString` is defined in `Animal` which does not have `name` or `breed` attributes.

```
  //add code below this line

  Dog d = new Dog("Rocky", "Pomeranian", 3);
  System.out.println(d);

  //add code above this line
```

To get a string representation of the `Dog` class, override the `toString` method. Call `toString` from the superclass and append the attributes (`name` and `breed`) from the subclass. You should see the name of the class (`Dog`), a set of square brackets with the attribute from the superclass, and another set of square brackets with attributes from the subclass.

```
class Dog extends Animal {
  private String name;
  private String breed;

  public Dog(String n, String b, int a) {
    super(a);
    name = n;
    breed = b;
  }

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

Interfaces

Interfaces

Interfaces are similar to abstract classes in that they cannot be instantiated and methods must be defined by subclasses. However, interfaces force the user to implement (write code for) all of the methods. The `Dog` interface is defined in the IDE to the left. Notice that there is no access modifier for the `bark` method. Methods in an interface are designed to be used by other classes, so they are public by default. There is no need to add the `public` access modifier.

To use an interface, create a class with the `implements` keyword. The `Chihuahua` class **must** override the `bark` method. Be sure to use the `public` keyword so objects can call the method.

```
//add class definitions below this line

class Chihuahua implements Dog {
  public String bark() {
    return "woof woof";
  }
}

//add class definitions above this line
```

Instantiate a `Chihuahua` object and print the output of the `bark` method.

```
    //add code below this line

    Chihuahua c = new Chihuahua();
    System.out.println(c.bark());

    //add code above this line
```

challenge

Try this variation:

- Comment out the definition for the `bark` method in the `Chihuahua` class.

```
//add class definitions below this line

class Chihuhua implements Dog {
//   public String bark() {
//     return "woof woof";
//   }
}

//add class definitions above this line
```

▼ **Why does this cause an error?**
The `Dog` interface requires that classes override the `bark` method. Since the method is commented out, Java throws an error.

Extending a Class and Implementing an Interface

Another benefit to interfaces is the ability to inherit from a class as well as implement an interface. For example, some people consider their pets to be a member of their family. Create the `FamilyMember` class with attributes for `name` and `age` and the `info` method. Then have the `Chihuahua` class extend the `FamilyMember` class and implement the `Dog` class.

```
//add class definitions below this line

class FamilyMember {
  private String name;
  private int age;

  public FamilyMember(String n, int a) {
    name = n;
    age = a;
  }

  public String info() {
    return String.format("%s is %d years old.", name, age);
  }
}

class Chihuahua extends FamilyMember implements Dog {
  public Chihuahua(String name, int age) {
    super(name, age);
  }

  public String bark() {
    return "woof woof";
  }
}

//add class definitions above this line
```

```
    //add code below this line

    Chihuahua c = new Chihuahua("Henry", 5);
    System.out.println(c.bark());
    System.out.println(c.info());

    //add code above this line
```

challenge

Try This Variation:

- Create the `movieStar` interface with the `movieDetails` method. Modify the `Chihuahua` class so that it extends the `FamilyMember` class and implements the `Dog` and `MovieStar` interfaces. Add the `film` and `revenue` attributes to help with overriding the `movieDetails`

method.

```
//add class definitions below this line

interface MovieStar {
  String movieDetails();
}

class FamilyMember {
  private String name;
  private int age;

  public FamilyMember(String n, int a) {
    name = n;
    age = a;
  }

  public String info() {
    return String.format("%s is %d years old.", name, age);
  }
}

class Chihuahua extends FamilyMember implements Dog,
        MovieStar {
  private String film;
  private String revenue;

  public Chihuahua(String n, int a, String f, String r) {
    super(n, a);
    film = f;
    revenue = r;
  }

  public String bark() {
    return "woof woof";
  }

  public String movieDetails() {
    return String.format("The move %s grossed %s
        worldwide.", film, revenue);
  }
}

//add class definitions above this line
```

Change the instantiation of the `Chihuahua` object so that the new arguments are passed to the constructor. Then call the `movieDetails` method.

```
    //add code below this line

    Chihuahua c = new Chihuahua("Henry", 5, "Beverly Hills
        Chihuahua", "$149,281,606");
    System.out.println(c.bark());
    System.out.println(c.info());
    System.out.println(c.movieDetails());

    //add code above this line
```

Interface vs Abstract Class

You may have noticed that there are many similarities between interfaces and abstract classes. The table below highlights some of the similarities and differences.

Category	Abstract Class	Interface
Keyword:	`extends`	`implements`
Inheritance:	Extend one superclass	Can extend a superclass and implement an interface
Access Modifier:	Use `public`, `private`, and/or `abstract`	All methods are `public` by default, no need to use an access modifier
Implementation:	Must override `abstract` methods	Must override **all** methods

With these concepts being so much alike, how do you know when to use one or the other? First, we need to understand two words: behavior and implementation. Behavior means the name but not the code of a method, while implementation means pre-written code for a method. Abstract classes allow for behavior (abstract methods) **and** implementation (concrete methods). Interfaces, however, only allow for behavior. You cannot create a concrete method in an interface. So if you want to define only behavior, use an interface. If you want to define behavior and implementation, use an abstract class.

Object Equality

Object Equality

The equality operator (==) is overloaded, which means it can compare two integers, two floats, etc. It can even compare two objects. Create the `ExampleClass` that has two attributes.

```
//add class definitions below this line

class ExampleClass {
  private int attribute1;
  private String attribute2;

  public ExampleClass(int a1, String a2) {
    attribute1 = a1;
    attribute2 = a2;
  }
}

//add class definitions above this line
```

Create the `ExampleClass` object `example1` with `7` and `"hello"` as the attributes. Now make a copy of `example1` and save it to the variable `example2`. Comparing the two objects with `==` should return `true`.

```
    //add code below this line

    ExampleClass example1 = new ExampleClass(7, "hello");
    ExampleClass example2 = example1;
    System.out.println(example1 == example2);

    //add code above this line
```

Instead of making `example2` a copy of `example1`, create a new object with the same values passed to the constructor. The program should now print `false`.

```
    //add code below this line

    ExampleClass example1 = new ExampleClass(7, "hello");
    ExampleClass example2 = new ExampleClass(7, "hello");
    System.out.println(example1 == example2);

    //add code above this line
```

If the objects have the same type and the same values for their attributes, are they not the same? When comparing user-defined functions, the equality operator compares memory addresses. When `example2` is a shallow copy of `example1`, the two objects share the same memory address. When `example2` is created with the constructor, Java gives this object its own memory address.

Override the `eqauls` method if you want to compare attributes of different objects. If each attribute in one object is the same as the corresponding attribute in another object, the method should return `true`. Use a compound boolean expression to compare both attributes. Since `attribute1` is an integer, you can use `==` for the comparison. `attribute2` is a string, so you need to use the `equals` method.

```
//add class definitions below this line

class ExampleClass {
  private int attribute1;
  private String attribute2;

  public ExampleClass(int a1, String a2) {
    attribute1 = a1;
    attribute2 = a2;
  }

  public boolean equals(ExampleClass other) {
    return attribute1 == other.attribute1 &&
        attribute2.equals(other.attribute2);
  }

}

//add class definitions above this line
```

Finally, check for equality using the `equals` method. Java should return `true`.

```
    //add code below this line

    ExampleClass example1 = new ExampleClass(7, "hello");
    ExampleClass example2 = new ExampleClass(7, "hello");
    System.out.println(example1.equals(example2));

    //add code above this line
```

challenge

Try this variation:

- Change the values passed to the constructor for the `example2` object.

```
    //add code below this line

    ExampleClass example1 = new ExampleClass(7, "hello");
    ExampleClass example2 = new ExampleClass(-32,
        "goodbye");
    System.out.println(example1.equals(example2));

    //add code above this line
```

Advanced Topics Formative Assessment 1

Advanced Topics Formative Assessment 2