Object-Oriented Programming in Java

Student Workbook

Version 1.1

Table of Contents

Module 1

Java Classes : A Review

Section 1–1

Classes

Object-Oriented Programming

- **Object-oriented programming is the practice of building an application by designing a set of classes to represent the "things" the application works with**

- **Classes are a way of packaging data and the operations that act upon that data in a single unit**
  - In JavaScript, we mostly used classes as a way to group data together
  - In TypeScript, we represented components, models and services using classes and defined data in them, as well as functionality like event handlers in components or methods that fetched API data in services

- **In Java, we can't have global functions so every piece of code must belong to a class**

- **When we typically talk about classes, they represent the nouns of a business**
  - A university would have classes for `Student`, `Course`, `Department`, `Faculty`, `Semester`, `Transcript` etc
  - An insurance company would have classes for `Customer`, `Policy`, `Claim`, `Agent`, etc.

- **We also have classes that represent software nouns**
  - For example, `String`, `File`, `Connection`, `Exception`, etc

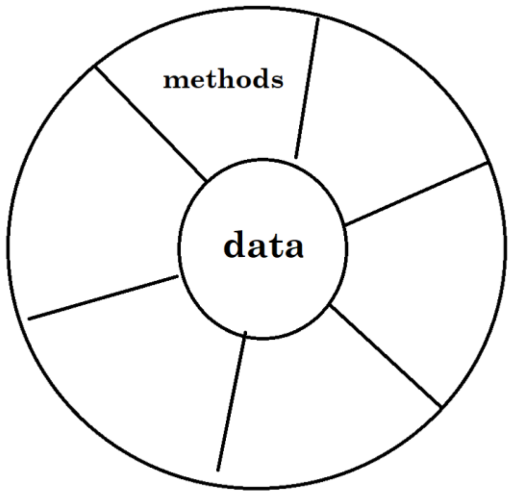
Object-Oriented Programming *cont'd*

- There are thousands of Java classes shipped as part of the JRE

- **Objects on the other hand are instances of classes**
  - Classes are blueprints; objects are the things the blueprints build
- **Thus far, we've been looking at our Java application class that has the static `main()` method**
  - This type of class isn't used to create objects -- but us used to bootstrap the application to start running
- **Let's figure out how to create a simple class in Java so that we can start becoming comfortable with terms like:**
  - encapsulation
  - class
  - object
  - constructor
  - instantiate an object
  - call a method
  - overload a method

Encapsulation

- **When we create classes in Java, we talk about the term *encapsulation***
  - An object-oriented programmer wants to design a class that hides how it works *inside* the class
  - They provide methods someone can call to interact with the class' objects
- **This "black box" approach allows the way a class works to be tweaked without affecting code that interacts with an object of that class**
- **Central to the concept of encapsulation are the access specifiers `public` and `private`**
  - Public members are accessible outside of the class
  - Private members are only accessible within an object



you can use an object's public methods which in turn access the object's private data

Creating the Class

- **In Java, a class has to be coded in a `.java` file of the same name**
  - Attributes are the variables, or data members, defined in a class
    - Each instance of the class contains its own copy of all the attributes
  - Methods define the behaviors of the objects

- **When we create a class, we typically:**
  - define private attributes
  - code a public constructor (with the same name as the class) that is used to create an instance of the class
  - add public getter and setter methods to provide access to an object's internals
  - add other methods deemed appropriate to the class

Example: A Java Class

Example

Person.java

```
public class Person {
   // attributes
   private String name;
   private int age;

   // constructor
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // getters and setters
   public String getName() {
      return this.name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return this.age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   // other methods
   public String convertToString() {
      return this.name + " is " + this.age + " year(s) old";
   }
}
```

– NOTE: The use of the keyword `this` in front of a variable identifies it as belonging to the object

  * That is, defined at the top of the class
  * It is not required unless there is a naming collision

Instantiating an Object

- **Once your class is defined, you can use it to create objects that hold your data**
  - Creating objects is also called instantiating an object
- **You must declare a variable of your class type**
  - If declared at the class level, it defaults to `null`
  - `null` means it doesn't refer to an object
- **You then use your constructor to instantiate the object by using the `new` keyword**
  - You pass values to your constructor's parameters

Example

```
// create the variable
Person p1;

// instantiate the Person object
p1 = new Person("Dana", 63);
```

- The values you pass to the constructor might also come from variables

Example

```
String name = "Dana";
int age = 63;

Person p1;
p1 = new Person(name, age);
```

Instantiating an Object *cont'd*

- **Often people declare the variable and instantiate the object in one step**

Example

```
String name = "Dana";
int age = 63;

Person p1 = new Person(name, age);
```

- **You can also instantiate multiple objects**

Example

```
// create the variables
Person p1, p2, p3;

// instantiate the Person object
p1 = new Person("Dana", 63);
p2 = new Person("Natalie", 37);
p3 = new Person("Zachary", 31);
```

- **Once we know about arrays in Java, we can even have arrays of class objects**

Example

```
Person[] family = new Person[4];   // 0, 1, 2, 3 (4 - size)
family[0] = new Person("Dana", 63);
family[1] = new Person("Natalie", 37);
...
```

Working with Objects

- **In Java, we traditionally declare our instance variables as private**
  - This enforces the principle of encapsulation which says that we only expose data that we want from an object
  - This is why we write getter and setter methods

- **If we tried to access the private data members of the object through the object name, the compiler would generate an error**

Example

```
Person p1 = new Person("Dana", 63);

System.out.println("Your name is " + p1.name);  // error
```

- **We must use the getter method to retrieve a data member's value**

Example

```
Person p1 = new Person("Dana", 63);

System.out.println("Your name is " + p1.getName());
```

Default Access Modifier

- **If you don't use explicitly specify an access modifier, Java assumes a default called** *package-private*

- ***Package-private*** **which means the member is visible within the same package but isn't accessible from other packages**

Example

Person.java

```
public class Person {
   String name;
   int age;

   Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   String getName() {
      return this.name;
   }

   int getAge() {
      return this.age;
   }
}
```

- **Java programmers will argue whether the best practice is to explicitly specify your access modifier**
  - I fall in the group that prefers explicit specification

Overloading Methods

- **Java allows us to overload methods in a class**
  - Overload means we can have more than one method with the same name as long as the ***signature*** is different
  - The signature is determined by taking the name of a method and adding to it the type of each parameter

Example

Thingy.java

```
public class Thingy {

 public void foo() {
    // signature  foo
 }

 public void foo(int x) {
    // signature  foo_int
 }

 public void foo(int x, int y) {
    // signature  foo_int_int
 }

 public void foo(int x, String s) {
    // signature  foo_int_String
 }

 public void foo(String s, int x) {
    // signature  foo_String_int
 }

 public void foo(String x) {
    // signature  foo_String
 }
}
```

Overloading Methods *cont'd*

- **The compiler can deduce which overloaded method you wanted to call by examining the arguments**

Example

```
Thingy obj = new Thingy();

obj.foo(7);

obj.foo(7, "Xyz");

obj.foo("Xyz", 7);
```

- **Overloads allow us to use methods in different ways**
  - Sometimes on overload offers features that another overload doesn't
  - Sometimes it just gives us a different way of passing arguments

Example

```
int answer = myCalculator.add(x, y);
```

-- or --

```
MathOperands ops = new MathOperands(x, y);

int answer = myCalculator.add(ops);
```

Overloading Constructors

- **In this workbook, you may see situations where there is more than one way to create an object**
  - That is because the constructor is overloaded
  - Overloading the constructor helps provide a robust class that can be used by many people in many different situations

- **Let's overload the constructor**

Example

Person.java

```
public class Person {
   private String name;
   private int age;

   public Person() {

   }

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
      return this.name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return this.age;
   }
```

Overloading Constructors *cont'd*

```
   public void setAge(int age) {
      this.age = age;
   }
}
```

- **Now that we have two constructors, let's look at two different ways to create an object**

Example

```
// This uses the parameterized constructor
Person p1 = new Person("Dana", 63);

// This uses the constructor without parameters and
// then places data in the object using the setter methods
Person p2 = new Person();
p2.setName("Natalie");
p2.setAge(37);

System.out.println("P1's name is " + p1.getName());
System.out.println("P2's name is " + p2.getName());
```

Example: Working with a Class

Example

Person.java

```
public class Person {
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
      return this.name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return this.age;
   }

   public void setAge(int age) {
      this.age = age;
   }
}
```

MainApp.java

```
public class MainApp {

   public static void main(String[] args) {
      // create a Person object
      Person person = new Person("Dana", 63);

      // display the person's information
      System.out.printf("%s is %d years old\n",
          person.getName(), person.getAge());
   }
}
```

Example: Working with a Class *cont'd*

- **To compile this program, you will need to run the javac compiler on both files**
  - Then, run the class file that has the `main()` method in it

Example

```
➢ javac *.java
```

Module 2

Java Packages

Section 2–1

Java Packages

Java Packages

- **A Java package is a collection of related `.class` files**
  - It organizes the class files in a series of folders
- **Packages are used to avoid naming conflicts with classes**
  - They also encourage you to create more maintainable code by organizing related classes together
- **There are two types of packages:**
  - The Java API packages
  - User-defined packages
- **The Java API consist of a large class library that is organized into packages**
  - For example, we've used classes from the `java.util` and `java.io` packages
  - `https://docs.oracle.com/javase/8/docs/api/`
- **Developers can also create packages of the custom code**

Using `import`

- **You can use a class defined in a different package by using the `import` keyword**
  - You can import a single class

Example

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

  - You can import all classes from a specific package

Example

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

User-Defined Packages

- **You can create packages for your own classes**
  - This allows you to group related classes together in a way that helps organize your code

- **Package names are typically expressed in a reverse DNS format**

Example

```
com.hca.dealership
```

- **The package name above implies the following folder structure**

Example

```
source-folder
   |
   |-- com
     |
     |-- hca
       |
       |-- inventory
         |
         |-- java files go here
```

- **Class files would reside in a similar folder with a different parent (often named `bin`)**

User-Defined Packages *cont'd*

Example

```
project-folder
   |
   |-- src
       |
       |-- com
           |
           |-- hca
               |
               |-- inventory
                   |
                   |-- java files go here
   |
   |-- bin
       |
       |-- com
           |
           |-- hca
               |
               |-- inventory
                   |
                   |-- class files go here
```

Example: Defining Packages

- **To specify the name of the package, place a `package` statement as the first statement in the code file**

Example

Product.java

```
package com.hca.inventory;

public class Product {
   private int id;
   private String productName;
   private int qtyOnhand;
   private double price;

   // remaining code not shown
}
```

MainApp.java

```
package com.hca.inventory;

import java.util.ArrayList;

public class MainApp {

   public static void main(String[] args) {

      List<Product> inventory = new ArrayList<Product>();
      ...

      // remaining code not shown

   }
}
```

Compiling Packages

- **To compile the classes and create the package structure, use the command below**
  - The -d flag is used to specify the destination where the package structure containing the course files should be placed

Example

From the project folder, issue the following command

```
javac -d .\bin  .\com\hca\inventory\*.java
```

Launching the Program

- **To run the program, remember to use the full name of the class containing the main method**

Example

From the project folder, issue the following commands

```
cd bin
java com.hca.inventory.MainApp
```

Exercise

EXERCISE 1

In this exercise, you will take the last version of your dealership program and update it to place your classes in packages. The package name you will use in this project is `com.hca.dealership`.

Begin by copying the entire project folder to a new folder under this module. Name the project folder `dealership-package`. Within that folder, create two subfolders: `bin` and `src`.

Now, under the src folder, create a directory structure like the one below and copy your .java files to the dealership folder.

```
|-- src
    |
    |-- com
        |
        |-- hca
            |
            |-- dealership
                |
                |-- java files go here
```

Open each of your `.java` files and add a package statement to the top of each.

Where should we place the data file? In this version of the project, we will place the data file in the root of the project folder and then pass the data file to our application on the command line when we run the file.

This means you will be able to access it using **`args[0]`**. Remember `args` is the name of the parameter that receives command line arguments.

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

}
```

Your `main()` function will need to pass the filename (`args[0]`) to your UserInterface object's constructor so that it can save it in a class-level variable and use it as needed.

You will compile your program from your ***project directory*** using the command:

```
javac -d .\bin .\src\com\hca\dealership\*.java
```

Exercise *cont'd*

Fix any compiler errors you have.

Then, run the program from your ***project's `bin` folder*** using the command:

```
java com.hca.dealership.Program c:\path\to\your\datafile.txt
```

where:

- `Program` is your class that has the `main` method
- `c:\path\to\your\datafile.txt` is your data file name

Creating / Running a JAR File

- **You can distribute a Java application or libraries using a JAR (Java Archive) file**

- **A JAR file is essentially a compressed, or zipped, set of directories**
  - It also contains the `.class` files, resources (audio files, images, etc), and some metadata
  - To see what's in a JAR file, rename it to `.zip` and use WinZip to extract view the contents

- **To create a JAR file, use the `jar` command**
  - the -cf flag says to create the file whose name follows the flag
  - the * indicates to add all folders/files in the current directory

Example

```
jar cf dealership.jar *
```

- **This is a non-executable JAR because it doesn't know the name of the class that has the main**
  - To run this type of JAR, you must specify the JAR and the name of the class with the main

Example

```
java -cp dealership.jar com.hca.dealership.Program
```

Specifying an Executable Jar

- **To create an "executable JAR", you must add a manifest file that specifies the entry point for the application**
  - This is the name of the class with the `main` method
- **The file should be named `MANIFEST.MF`**
  - It contains a single line with the fully qualified name of the entry point class

Example

```
Main-Class: com.hca.dealership.Program
```

- **When you package the JAR, use different flags and specify the manifest file**

Example

```
jar cvmf MANIFEST.MF dealership.jar *
```

- **You can run a JAR packaged application using the java command**
  - Include the -jar flag

Example

```
java -jar dealership.jar
```

Exercises

EXERCISE 1

Continue working in the previous project. Create an executable JAR file. To do this, create a MANIFEST.MF file that specifies your application's entry point.

Then create the JAR file.

Copy the JAR file and the data file to a new folder and run the application. Make sure to pass the name of the data file into the application.

Module 3

Inheritance

Section 3–1

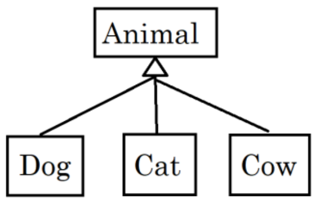
Inheritance

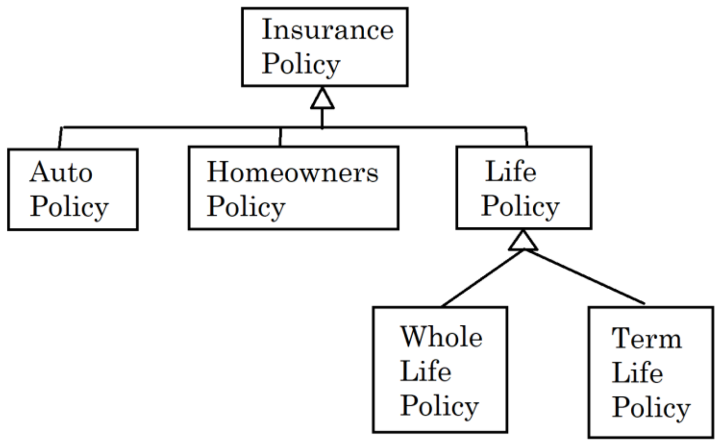
Inheritance

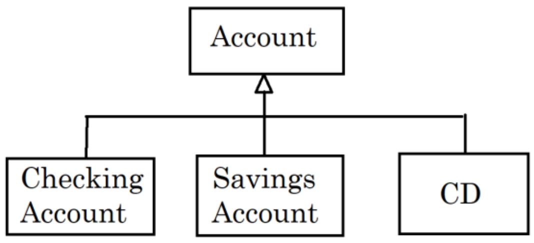
- **Object-oriented programming languages allow developers to create new classes by extending existing ones**
  - This process is called inheritance
  - Inheritance models "IS A" relationships: a ***dog*** is an ***animal***, a ***checking account*** is an ***account***, a ***house*** is an ***asset***

- **The new class *<u>inherits</u>* the attributes and methods from is base, or parent, class**
  - The ***parent class*** is also called the super class or base class and contains the features you want to extend
  - The new class is called the ***child class***, sub-class, or derived class and extends the parent class by adding additional features and overriding existing ones

- **Examples of inheritance include:**







Java Inheritance using `extends`

- **Java uses the `extends` keyword to indicate a class inherits from another class**

Syntax

```
class SomeSuperClcass {
   ...
}

class SomeChildClass extends SomeSuperClcass {
   ...
}
```

- **All of the features except constructors are inherited into the child class**
  - Child classes must define their own constructors

Syntax

```
class Asset {
   ...
}

class House extends Asset {
   ...
}

class Stock extends Asset {
   ...
}

class Cow extends Asset {
   ...
}
```

Example: Simple Inheritance

Example

Animal.java

```
public class Animal {
   private String name;

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

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

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Program.java

```
public class Program {

   public static void main(String[] args) {

      Animal ani = new Animal();
      ani.setName("Bob");
```

Example: Simple Inheritance   *cont'd*

```
        Dog dog = new Dog();
        dog.setName("Rubby");     // inherited method
        dog.setBreed("Corgi");
        dog.bark();

    }
}
```

`protected` Access Specifier

- **When a parent class member is declare as `private`, child classes inherit it… but they can't access it**

- **Occasionally, a programmer will declare a parent class member with the `protected` access specifier**
  - Child classes can access the member
  - Outside classes can't access the member (unless it is public!)

Example

Animal.java

```
public class Animal {
   protected String name;

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

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

   public void setBreed(String breed) {
      this.breed = breed;
   }

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

`protected` Access Specifier *cont'd*

```
      public void bark() {
         System.out.println(name + " says ruff, ruff!");
      }
   }
```

- **Although this is possible, this isn't done often**

- **Why?**
   - When a class makes a data member protected, it can't control what child classes do to it (i.e manage the range or value or format of the data)

Constructors and Inheritance

- **You might recall that, in Java, if a class doesn't have a constructor then a "default constructor" is generated under the hood**
  - It has no parameters
  - This is what allows you to instantiate an object

Example

Animal.java

```
public class Animal {
   private String name;

   // there is no explicit constructor in this class

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

elsewhere

```
Animal ani = new Animal();
```

- **Child class constructors automatically call their super class' parameterless constructor -- regardless of whether the compiler generated it or the programmer**
  - This happens at the beginning of the execution of the child class construction process

Example

Animal.java

```
public class Animal {
   private String name;

   public Animal() {
      System.out.println("Trace -- in Animal c'tor");
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

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

   // no constructor

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Constructors and Inheritance *cont'd*

Program.java

```
public class Program {

   public static void main(String[] args) {

      Animal ani = new Animal();

      Dog dog = new Dog();
   }
}
```

OUTPUT

```
Trace -- in Animal c'tor
Trace -- in Animal c'tor
```

- **When you code a constructor in your child class, you should *explicitly* call the parent class constructor using the function `super()`**
  - It must be the first line in the constructor
  - If you are calling the parent class' parameterized constructor, you can pass arguments to it

Example

Dog.java

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

   public Dog(String name, String breed) {
      super(name);        // calls Animal constructor
      this.breed = breed
   }
   public void setBreed(String breed) {
      this.breed = breed;
   }
```

Example: Passing Arguments to Parent Class Constructor

Example

Animal.java

```
public class Animal {
   private String name;

   public Animal() {
      System.out.println("Trace -- in Animal() c'tor");
   }

   public Animal(String name) {
      System.out.println(
         "Trace -- in Animal(name) c'tor w/ name");
      this.name = name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

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

   public Dog() {
      super();
      System.out.println("Trace -- in Dog() c'tor");
   }

   public Dog(String name, String breed) {
      super(name);
      System.out.println(
         "Trace -- in Dog(name, breed) c'tor w/ name");
      this.breed = breed;
   }
```

Example: Passing Arguments to Parent Class Constructor *cont'd*

```
   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Program.java

```
public class Program {

   public static void main(String[] args) {

       Animal animal_1 = new Animal();
       System.out.println(
         "animal_1's name is " + animal_1.getName() + "\n");

       Animal animal_2 = new Animal("Elmo");
       System.out.println(
          "animal_2's name is " + animal_2.getName() + "\n");

       Dog dog_1 = new Dog();
       System.out.println("dog_1's name is " + dog_1.getName()
          + " and their breed is " + dog_1.getBreed() + "\n");

       Dog dog_2 = new Dog("Fido", "Mutt");
       System.out.println("dog_2's name is " + dog_2.getName()
          + " and their breed is " + dog_2.getBreed() + "\n");
   }
}
```

OUTPUT

```
Trace -- in Animal() c'tor
animal_1's name is null
```

Example: Passing Arguments to Parent Class Constructor   *cont'd*

```
Trace -- in Animal(name) c'tor w/ name
animal_2's name is Elmo

Trace -- in Animal() c'tor
Trace -- in Dog() c'tor
dog_1's name is null and their breed is null

Trace -- in Animal(name) c'tor w/ name
Trace -- in Dog(name, breed) c'tor w/ name
dog_2's name is Fido and their breed is Mutt
```

Overriding Inherited Methods

- **Child classes can override inherited methods**
  - Overriding means they can code their own version and it replaces the inherited one

Example

Human.java

```
public class Human {

    public void eat() {
        System.out.println("Me hungry...");
    }
}
```

Caveman.java

```
public class Caveman extends Human {

    public void eat() {
        System.out.println("Me hunt lion and eat!");
    }
}
```

ModernPerson.java

```
public class ModernPerson extends Human {

    public void eat() {
        System.out.println("Me going to McDonalds and eat!");
    }
}
```

Overriding Inherited Methods *cont'd*

Program.java

```
public class Program {

    public static void main(String[] args)() {
        Human h = new Human();
        h.eat();
        Caveman c = new Caveman();
        c.eat();

        ModernPerson m = new ModernPerson();
        m.eat();
    }
}
```

OUTPUT

```
Me hungry...
Me hunt lion and eat!
Me going to McDonalds and eat!
```

Polymorphism and Base Class References

- **The ability to override methods introduces polymorphism into our class library**
  - Polymorphism, from the Greek "many forms", allows us to perform a single action in different ways
  - That is, we can have many forms of the same method

- **It is most notable when we use a base class variable to reference a derived class object**

- **A base class reference variable can refer to any object of a derived type**
  - For example, an animal variable can reference a cat or a dog object because "a cat is-an animal" and "a dog is-an animal"

Example

```
Animal animal;

animal = new Dog();   // okay if Dog extends Animal
animal = new Cat();   // okay if Cat extends Animal
```

- **However, the opposite isn't true**
  - You cannot assign a base class object to a derived class variable
  - What if the assignment isn't valid?

Polymorphism and Base Class References *cont'd*

Example

```
Animal animal = /* some value not shown */;

Dog dog = animal;   // ERROR -- what if animal references a Cat
```

- **If you are 100% sure that the base class variable references the right type of object, then you can force the assignment by downcasting**
  - This will cause a runtime error if you are wrong

Example

```
Animal animal = /* some value not shown */;

Dog dog = (Dog) animal;
```

Example: Polymorphism in Action

- **In the example below, you would think that the call `human.eat()` in the feed method would call the `Human`'s `eat()` method**
  - But it doesn't!

- **Since `eat()` is an overridden method, the Java engine decides at runtime which `eat()` method to call based on the *type of* the parameter object**

Example

Program.java

```
public class Program {

    public static void main(String[] args)() {
        Human h = new Human();
        feed("Human", h);

        Caveman c = new Caveman();
        feed("Caveman", c);

        ModernPerson m = new ModernPerson();
        feed("Modern Person", m);
    }

    public static void feed(String label, Human human) {
       System.out.print(label + "--> ");
       human.eat();
    }
}
```

OUTPUT

```
Human--> Me hungry...
Caveman--> Me hunt lion and eat!
Modern Person--> Me going to McDonalds and eat!
```

Heterogeneous Collections

- **This polymorphic nature of overridden methods leads to some interesting software designs**
- **Now we can hold arrays and other collections using a base class reference**
  - However, the objects in the collection will behave appropriately for their type

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        Human h = new Human();
        people.add(h);

        Caveman c = new Caveman();
        people.add(c);

        ModernPerson m = new ModernPerson();
        people.add(m);

        // people is a collection of different types
        // of Human objects

        for(int i = 0; i < people.size(); i++) {
           people.get(i).eat();    // polymorphic behavior
        }
    }
}
```

Heterogeneous Collections *cont'd*

OUTPUT

```
Me hungry...
Me hunt lion and eat!
Me going to McDonalds and eat!
```

`instanceof` Operator

- **Java provides an `instanceof` operator to compare an object to a specified type**
  - You can use it to see if an object is an instance of a class, a subclass, or even a class that implements an interface (coming soon)
- **In the following example, we will ask each object if they are:**
  - `Human`
  - `Caveman`
  - `ModernPerson`
- **A `ModernPerson` object will answer yes to `ModernPerson` and `Human` because of their inheritance chain**

Example: `instanceof` Operator

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            if (people.get(i) instanceof Human) {
               System.out.print("Human--> ");
            }
            if (people.get(i) instanceof Caveman) {
                System.out.print("Caveman--> ");
            }
            if (people.get(i) instanceof ModernPerson) {
                System.out.print("ModernPerson--> ");
            }
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
Human--> Me hungry...
Human--> Caveman--> Me hunt lion and eat!
Human--> ModernPerson--> Me going to McDonalds and eat!
```

More About instanceof

- **If you wanted a single answer to an instanceof query, ask about the most specific types first and use and else if structure that would stop questioning once their was a match**

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            if (people.get(i) instanceof ModernPerson) {
                System.out.print("ModernPerson--> ");
            }
            else if (people.get(i) instanceof Caveman) {
                System.out.print("Caveman--> ");
            }
            else if (people.get(i) instanceof Human) {
               System.out.print("Human--> ");
            }
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
Human--> Me hungry...
Caveman--> Me hunt lion and eat!
ModernPerson--> Me going to McDonalds and eat!
```

getClass()

- **You can call the `getClass()` method on any object and it will return the class name**
  - If the class is defined in a package, it will return its fully qualified name (ex: "java.util.ArrayList")
  - If the class is not defined in a package, it will return the word "class" followed by the class name (ex: "class Human")

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            System.out.print(
               people.get(i).getClass() + "--> ");
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
class Human--> Me hungry...
class Caveman--> Me hunt lion and eat!
class ModernPerson--> Me going to McDonalds and eat!
```

`Object` : The Parent Class of all Classes

- **Why can you call `getClass()` on all objects?**
  - In Java, all classes implicitly inherit from a class called Object
- **This gives all Java classes a common ancestor**

Example

```
String s = "Hello";

ArrayList<Vehicle> myList = new ArrayList<Vehicle>();

ModernPerson me = new ModernPerson();

If (s instanceof Object)        // true
If (myList instanceof Object)   // true
If (me instanceof Object)       // true
```

- **`Object` defines some methods that are inherited by all child classes, including:**
  - `equals()`
  - `getClass()`
  - `toString()`
- **The presence of the `toString()` method is why you can display objects in a print statement**
  - The default `toString()` is automatically called
  - Programmers often override `toString()` for a custom implementation

`Object` : The Parent Class of all Classes *cont'd*

- **For a complete list, see:** **`https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`**

Exercises

EXERCISE 1

Create a Java application that manages a list of assets. WARNING: This is an academic example and is a hugely simplified version of anything real.

The base class `Asset` can be defined with the following:

Properties:	description : String
	dateAcquired : String
	originalCost : double
Methods:	constructor
	all getters / setters
	getValue() : double // returns original cost

You will build two derived classes: `House` and `Vehicle`.

The `House` class inherits from `Asset` can be defined with the following:

Properties:	address : String
	condition : int (1 -excellent, 2 -good, 3 -fair, 4 -poor)
	squareFoot : int
	lotSize : int
Methods:	constructor
	all getters / setters
	getValue() : double (override)
	// A house's value is determined as
	// $180.00 per square foot (excellent)
	// $130.00 per square foot (good)
	// $90.00 per square foot (fair)
	// $80.00 per square foot (poor)
	// PLUS 25 cents per square foot of lot size

The `Vehicle` class inherits from `Asset` can be defined with the following:

Properties:	makeModel : String
	year : int

odometer : int

Exercises *cont'd*

```
Methods:    constructor
            all getters / setters
            getValue() : double      (override)
              // A car's value is determined as
              // 0-3 years old  - 3% reduced value of cost per year
              // 4-6 years old  - 6% reduced value of cost per year
              // 7-10 years old - 8% reduced value of cost per year
              // over 10 years old - $1000.00
              // MINUS  reduce final value by 25% if over 100,000 miles
              //     unless makeModel contains word Honda or Toyota
```

This program will not have a user interface. In your `main()`, create an `ArrayList` of `Asset` objects.

Load it with your `Assets`. Include at least 2 houses (you have a vacation home!) and at least two vehicles. Use descriptions like "my house" or "Tom's truck" for the assets.

Now, loop thru the `Asset` collection displaying the description of each asset, the date you acquired it, how much you paid for it, and its value.

When that works, go back and include in the display either the address of the house or the year and make/model of the vehicle. You will need to use `instanceof` when you loop through the assets to detect the type of asset it is. Once you know it is a House or Vehicle, you will need to downcast it so that you can call the methods of the specific type.

For example:

```
String message = "";

if (myAssets.get(i) instanceof House) {
   House house = (House) myAssets.get(i);
   message = "House at " + house.getAddress();
}
else if (myAssets.get(i) instanceof Vehicle) {
   Vehicle vehcile = (Vehicle) myAssets.get(i);
   message = "Vehicle: " +
      vehicle.getYear() + " " + vehicle.getMakeModel();
}
```

`@Override` Annotation

- **Java 1.5 introduced the `@Override` annotation to tell the compiler that the method with the `@Override` MUST override an existing inherited method**
  - If the method doesn't exist, the compiler generates an error

Example

Human.java

```
public class Human {

    public void eat() {
        System.out.println("Me hungry...");
    }
}
```

Caveman.java

```
public class Caveman extends Human {

    @Override
    public void eat() {
        System.out.println("Me hunt lion and eat!");
    }
}
```

ModernPerson.java

```
public class ModernPerson extends Human {

    @Override                        // generates an error!
    public void dine() {
        System.out.println("Me going to McDonalds and eat!");
    }
}
```

- **You will see this a lot in our code next week**

Module 4

Abstraction

Section 4–1

Abstraction

Abstraction

- ***Data abstraction*** **is the process of exposing only essential information about an object and hiding implementation details**
  - In Java, this is achieved using the abstract classes and interfaces
- **An abstract class is a class that gathers attributes and methods together, but cannot be used to instantiate an object**
- **You know you are working with an abstract class when someone asks you "what type"?**
  - I'd like to open an account…
  - I'd like to buy an asset…
  - I'd like to take home a mammal…
  - I'd like to apply for a policy…

Abstract Class

- **An abstract class is defined with the keyword `abstract`**
  - That immediately means it can't be used to instantiate an object
  - But you can create child classes that inherit from it

Example

Asset.java

```
public abstract class Asset {
   protected String description;
   protected int yearAcquired;
   protected double originalCost;

   public Asset(String description, int yearAcquired,
                double originalCost {
      this.description = description;
      this.yearAcquired = yearAcquired;
      this.originalCost = originalCost;
   }

   public String getDescription() {
       return description;
   }

   public int getYearAcquired() {
       return yearAcquired;
   }

   public double getValue() {
       return originalCost;
   }
}
```

Elsewhere

```
// you cannot instantiate an abstract class

Asset myAsset = new Asset("Thingy", 2020, 125.00);  // ERROR
```

Abstract Class *cont'd*

- **Typically, you create one or more classes that extend the abstract class**

Example

House.java

```
public class House extends Asset {
   private String address;
   private int squareFeet;
   private int lotSize;

   public House(String description, String address,
        int squareFeet, int lotSize,
        int yearAcquired, double originalCost {

      super(description, yearAcquired, originalCost);

      this.address = address;
      this.squareFeet = squareFeet;
      this.lotSize = lotSize;
   }

   public String getAddress() {
       return address;
   }

   public int getYearAcquired() {
       return squareFeet;
   }

   public int getLotSize() {
       return lotSize;
   }

   public double getValue() {
       return (180 * squareFeet) + (0.25 * lotSize);
   }
}
```

Abstract Class *cont'd*

Elsewhere

```
// you can instantiate a class that inherits from an
// abstract class

House myHouse = new House("Ranch House", "402 Stevens",
    2000, 43560, 2020, 220000);
```

- **You can also use variables defined as an abstract class type to reference objects from its derived types**

Example

```
// Assume House and Jewelry extend the Asset class

Asset[] myAssets = new Asset[3];

myAssets[0] = new House("Ranch House", "402 Stevens",
    2000, 43560, 2020, 125.00);

myAssets[1] = new House("Rental", "3329 Duchess",
    1600, 5445, 1995, 53000);

myAssets[2] = new Jewelry("Ring", "Diamond", 1.5, 1979, 1200);

// Loop thru the assets and get the value of each

double myNetWorth = 0;
for(int i = 0; i < myAssets.length; i++) {
   myNetWorth += myAssets[i].getValue();
}
```

Abstract Methods

- **Abstract methods are methods in an abstract class that are placed there so that all derived type have to provide an implementation of the method**

Example

Asset.java

```
public abstract class Asset {
   protected String description;
   protected int yearAcquired;
   protected double originalCost;

   public Asset(String description, int yearAcquired,
                double originalCost {
      this.description = description;
      this.yearAcquired = yearAcquired;
      this.originalCost = originalCost;
   }

   public String getDescription() {
       return description;
   }

   public int getYearAcquired() {
       return yearAcquired;
   }

   // now child classes *MUST* override this method
   public abstract double getValue();
}
```

Example

House.java

```
public class House extends Asset {
   private String address;
   private int squareFeet;
   private int lotSize;
```

Abstract Methods *cont'd*

```
    public House(String description, String address,
         int squareFeet, int lotSize,
         int yearAcquired, double originalCost {

       super(description, yearAcquired, originalCost);

       this.address = address;
       this.squareFeet = squareFeet;
       this.lotSize = lotSize;
    }

    public String getAddress() {
        return address;
    }

    public int getYearAcquired() {
        return squareFeet;
    }

    public int getLotSize() {
        return lotSize;
    }

    public double getValue() {
        return (180 * squareFeet) + (0.25 * lotSize);
    }
}
```

Exercises

EXERCISE 1

Continue working in the Asset example from the previous module.

Modify your `Asser` class to be an abstract class. Then make the `getValue()` method an abstract method.

Retest your app.

Try to create an instance of an `Asset`. What happens?

Module 5

Interface-Based Programming

Section 5–1

Interfaces

Interfaces

- **Interfaces in Java are collections of abstract methods**
  - They are used to define a set of features that a class ***must*** implement if it implements the interface
- **Interfaces do *not* have data members**

Example

IMovable.java

```
public interface IMovable {

   Point move(int xUnits, int yUnits);
   void goHome();
}
```

- **Interfaces can't be instantiated**
- **Instead, interfaces are implemented by classes**

Implementing an Interface

- **A class implements the interface using the `implements` keyword**

Example

```
public class Turtle implements IMovable {
   private String name;
   private Point currentLocation;

   public Turtle(String name) {
      this.name = name;
      this.currentLocation = new Point(25, 25);
      this.power = 100;
   }
   // getters and setters not shown
   public Point move(int xUnits, int yUnits) {

      // the turtle moves the number of units specified in
      // the direction specified

      currentLocation.setX(currentLocation.getX() + xUnits);
      currentLocation.setY(currentLocation.getY() + yUnits);

      return currentLocation;
   }

   public void goHome() {
      this.currentLocation = new Point(25, 25);
   }
}
```

- **If a class says it implements an interface, but the compiler doesn't find an implementation for the interface methods, it generates an error**

- **Each class that implements the interface can choose their own way to implement the interface's behavior**

Implementing an Interface *cont'd*

Example

```
public class Robot implements IMovable {
   private String name;
   private Point currentLocation;
   private int power;

   public Robot(String name) {
      this.name = name;
      this.currentLocation = new Point(0, 0);
      this.power = 100;
   }
   // getters and setters not shown

   public Point move(int xUnits, int yUnits) {

      // the robot can only move the number of units
      // if it has the appropriate power

      int biggestUnit = (xUnits >= yUnits) ? xUnits : yUnits;

      if (power >= biggestUnit) {
         currentLocation.setX(currentLocation.getX() + xUnits);
         currentLocation.setY(currentLocation.getY() + yUnits);

        power -= biggestUnit;
      }

      return currentLocation;
   }

   public void goHome() {
      this.currentLocation = new Point(0, 0);
   }

}
```

Why Interfaces?

- **There are several reasons that you might decide to use interfaces in your application**

- **1 - They can enforce uniformity amongst a set of classes where inheritance doesn't make sense**
  - For example, in Java, the `Collection` interface is implemented by several collection classes

Example

```
// NOTE:  This example does NOT include all of the code in
//        the Collection interface

public interface Collection<T> extends Iterable<T> {

    int size();
    boolean isEmpty();

    boolean add(T e);
    boolean remove(Object o);
    void clear();

    boolean contains(Object o);
    boolean  containsAll(Collection<?> c);

    Object[] toArray();

    boolean equals(Object o);
    int hashCode();

    // Not all code shown
}
```

- **Classes that implement `Collection` include `ArrayList`, `LinkedList`, and `PriorityQueue`**

Why Interfaces? *cont'd*

- **2 - They allow use to define features implemented by one or more methods and then pick and choose which features make sense for our classes**

Example

IMovable.java

```
public interface IMovable {

    Point move(int xUnits, int yUnits);
    void goHome();
}
```

IDrawable.java

```
public interface IDrawable {

    void draw();
}
```

ICleaner.java

```
public interface ICleaner{

    void clean();
}
```

- **Java doesn't support multiple inheritance of classes, but you can implement many interfaces**

Example

RobotVacuum.java

```
public class RobotVacuum implements IMovable, ICleaner {
    // code here
}
```

Why Interfaces? *cont'd*

Robot.java

```
public class Robot implements IMovable {

    // code here
}
```

EtchASketch.java

```
public class EtchASketch implements IDrawable, ICleaner {

    // code here
}
```

Interfaces and Heterogeneous Collections

- **When you have heterogeneous collections, you can query to see of an object implements an interface**
  - If it does, cast the object to the instance type and then use the methods of the interface

Example

```
ArrayList<Object> things = new ArrayList<Object>();

things.add(new Person());
things.add(new Robot());
things.add(new EtchASketch());
things.add(new Car());
things.add(new RobotVacuum());

for(int i = 0; i < things.size(); i++) {

   if (things[i] instanceof ICleaner) {
      ICleaner cleaner = (ICleaner) things[i];
      cleaner.clean();
   }

}
```

Interfaces and Default Methods

- **Java 8 introduced the ability to define default methods in interfaces**
  - A default method defines an implementation that can be overridden in the class if need be
  - But for legacy purposes, classes that already implemented the interface will inherit the default implementation

Example

```
public interface IDrawable {

   void draw();

   default void print() {
      System.out.println("This object can draw things.");
   }
}
```

- **This feature was created so that collections could work with Java 8's lambda expressions**
  - Java 8 added a `forEach` method to `List` and `Collection` interfaces

Multiple Defaults

- **There is a possibility that a class is implementing two interfaces with same default methods**

- **This would introduce ambiguity**

Example

IMovable.java

```
public interface IMovable {

   void move(int xUnits, int yUnits);
   void goHome();

   default void print() {
      System.out.println("I can move!");
   }
}
```

ICleaner.java

```
public interface ICleaner {

   void clean();

   default void print() {
      System.out.println("I can clean!");
   }
}
```

- **To solve this ambiguity, you would need to override the default methods and then call whichever inherited default(s) you wanted**

Multiple Defaults *cont'd*

Example

```
public class RobotVacuum implements IMovable, ICleaner {

   public void move(int xUnits, int yUnits) {
     // code not shown
   }

   public void goHome() {
     // code not shown
   }


   public void clean() {
     // code not shown
   }


   public void print() {
      System.out.println("I am a robot vacuum!");
      IMovable.super.print();
      ICleaner.super.print();
   }
}
```

Exercises

EXERCISE 1

Rather than define an interface in this application, you will implement a Java interface in a custom class.

The interface we are interested in is Comparable. It has one method:

```
int compareTo(T o);
```

You can read the documentation on it here:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`

The Comparable interface is implemented by user-defined Java classes when you want to be able to sort collections contain that type of object.

See the following for hints:
`https://beginnersbook.com/2017/08/comparable-interface-in-java-with-example/`

In this exercise, you will build a Java application that defines a class named `Person`. A `Person` object has a `firstName`, a `lastName`, and an `age`.

In the `main()`, you will build an `ArrayList` of `Person` objects similar to that below:

```
ArrayList<Person> myFamily - new ArrayList<Person>();
myFamily.add( new Person("Dana", "Wyatt", 63) );
myFamily.add( new Person("Zachary", "Westly", 31) );
myFamily.add( new Person("Elisha", "Aslan", 14) );
myFamily.add( new Person("Ian", "Auston", 16) );
myFamily.add( new Person("Zephaniah", "Hughes", 9) );
myFamily.add( new Person("Ezra", "Aiden", 17) );
```

You should sort the list by last name and display the results. To sort, your `Person` class will have to implement the `compareTo` method defined in `Comparable`.

BONUS: If two people have the same last name, sort by first name.

Module 6

Generics

Section 6–1

Generics

Generics

- **Generics were added to Java in JDK 5.0**
  - They are a major part of many popular programming languages, including C++, C# and Swift

- **Generic can be applied to classes or methods**

- **When applied to a class, generics allow you to generalized the data type of member variable(s)**
  - They type of the variable(s) are usually specified using a letter like **`T`** and then replaced with an actual data type like `Integer`, `String` or a user-defined data type when the `object` is instantiated

- **When applied to a method, generics allow you to generalized the data type of parameters and return types**

- **IMPORTANT NOTE: Java *<u>doesn't</u>* support using <u>*primitive types*</u> with generics**
  - This means you should use one of Java's wrapper's for primitive types
  - For example: `Integer` for `int`, `Double` for `double`, `Character` for `char`, `Boolean` for `bool`, etc

Generic Classes

- **Creating a generic class is a matter of:**
  - Placing the generic parameter(s) in < > after the name of the class
  - Using the generic parameter(s) everywhere you expect a specific parameter once an object is instantiated

Example

Pair.java

```
class Pair<T>  {

  // A pair contains two objects
  private T leftThing;
  private T rightThing;

  Test(T leftThing, T rightThing) {
    this.leftThing = leftThing;
    this.rightThing = rightThing;
  }

  public T getLeftThing()  {
    return this.leftThing;
  }

  public T getRightThing()  {
    return this.rightThing;
  }
}
```

- **You can then instantiate objects by proving the generic parameter in < > after the name of the class**

Example

```
Pair<Integer> twoNums = new Pair<Integer>(63, 65);
Pair<String> twoWords = new Pair<String>("Me", "You");
```

Generic Classes *cont'd*

Example

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

        // Create an instance of a Pair<T> where T is a String
        Pair<Integer> twoNums = new Pair<Integer>(63, 65);
        System.out.println(twoNums.getLeftThing() +
           " - " + twoNums.getRightThing());

        // Create an instance of a Pair<T> where T is an
        Pair<String> twoWords = new Pair<String>("Me", "You");
        System.out.println(twoWords.getLeftThing() +
           " n " + twoWords.getRightThing());
    }
}
```

OUTPUT

```
63 - 65
Me n You
```

Generic Methods

- **Generic methods use generic parameters and return types**
- **You can use generic methods in generic classes, as you saw in the previous examples**
- **You can also use generic methods in standard Java classes**
  - If you use the generic parameter as a return type, you must surround it with < >

Example

Labeler.java

```
public class Labeler  {

   // A generic method that displays a label and a value
   static <T> void displayWithLabel(String label, T value)  {
      System.out.println(label +  ": " + value);
   }
}
```

MainApp.java

```
public class MainApp {

   public static void main (String[] args) {
      Labeler labeler = new Labeler();

      // Calling generic method with String argument
      labeler.displayWithLabel("Name", "Dana");

      // Calling generic method with Integer argument
      labeler.displayWithLabel("Age", 63);
   }
}
```

OUTPUT

```
Name: Dana
Age: 32
```

Exercises

There are no exercises in this module. But the discussion should help you understand the grammar you've been using to create `ArrayLists` in Java.