Object-Oriented Programming in Java

Student Workbook 4

Version 2.1

Table of Contents

Module 1

Introduction to Objects

Section 1–1

Thinking in Objects

How we Organize the "Real World"

- **The "Real World" is made up of specialized roles, responsibilities, objects or ideas**
  - The goal of Object Oriented Programming is to model our software to represent the world around us
  - In OOP we treat all of these concepts as objects

Example: Objects at an Organization

- **Departments**
  - HR
  - Graphic Design
  - Software Engineering
- **Employees**
  - Director of Design
  - Director of Software Engineering
  - Product Owner
  - Developer
  - QA
  - DBA
- **Physical Tools**
  - Laptop
  - Monitor
  - Printer
  - Desk
  - Chair

Example: Objects at an Organization *cont'd*

- **Concepts**
  - Project
  - Report
  - Code Commit
  - Pull Request

Exercise: Objects at a Hotel

EXERCISE 1

Work as a group to figure out what objects you would need to run a successful hotel. Remember that objects can be a department, job title, person, physical object or even an idea or process.

Hint: look for nouns in the requirements.

Begin by discussing and understanding how a hotel operates. Ask your instructor if you need any clarification.

Use these scenarios as a guide:

1. Guests check in at the front desk with a clerk
2. When a guest checks in they are assigned a room, and given a room key
3. When guests check out they receive a receipt of their room charges
4. Guests can order room service from the restaurant and have it delivered
5. Meals at the restaurant can also be charged to the guests room
6. All guest requests should be made by calling the front office. (i.e. housekeeping, more towels, or maintenance issues such as a burnt out light bulb)
7. When a guest checks out housekeeping should be notified that a room is ready to be cleaned

You might come up with additional scenarios as you brainstorm.

Take time for each group to share their list with the class.

Section 1–2

Specialization and Responsibilities

What are Specialists?

- **A person who is really good at 1 thing**
  - Doctor vs. Heart surgeon
  - Mechanic vs. Airplane mechanic
  - Developer vs. Android Developer
- **Why do we specialize?**
  - A person who has expert knowledge and ability in a field
  - It is easier to assign responsibilities
  - Gives us a single point of reference
- **In Object Oriented Programming we call this Encapsulation**
  - Each object has a list of responsibilities
- **Only 2 Types of Responsibilities**
  - An object might be responsible to KNOW something (store data)
    - The brains
  - An object might be responsible to DO something (perform an action)
    - The muscle
  - Or… an object could do both

Example: Objects at an Organization

- **Responsibilities: Development Shop**
  - Know a project budget: Director Software Engineering
  - Decide how the software should work: Product Owner
  - Add functionality to the software: Developer
  - Hire more developers: Director Software Engineering
  - Make sure a new feature works as expected: QA
  - Write and maintain (own) the requirements: Product Owner
  - Understand the software requirements: Developer, QA
    - **NOTE**: This is NOT a duplication of responsibility, because how each team member uses the requirements will be different
    - Only one person OWNS the requirements, but others must understand them

Exercise: Responsibilities at a Hotel

EXERCISE 2

Work in the same group as before. Make a list of responsibilities that are required to run this hotel. Remember, there are only 2 types of responsibilities:

1. What information do you need to keep track of? (What do you need to **KNOW**?)
2. What tasks/actions need to be performed? (What do you need to **DO**?)

Use the list of objects that you came up with earlier and decide which object should be responsible for each responsibility in your list. Each responsibility should be assigned to only one object.

NOTE: You might create new objects, and you might decide that some of the object are unnecessary.

Use these scenarios as a guide:

1. Guests check in at the front desk with a clerk
2. When a guest checks in they are assigned a room, and given a room key
3. When guests check out they receive a receipt of their room charges
4. Guests can order room service from the restaurant and have it delivered
5. Meals at the restaurant can also be charged to the guests room
6. All guest requests should be made by calling the front office. (i.e. housekeeping, more towels, or maintenance issues such as a burnt out light bulb)
7. When a guest checks out housekeeping should be notified that a room is ready to be cleaned

Take time for each group to share their list with the class.

Section 1–3

CodeWars

CodeWars Kata

- **Area or Perimeter**
  - Given 3 numbers, return if the 3rd number is the area or the perimeter of the first 2 numbers
    - 1st is width
    - 2nd is height
    - 3rd is a calculation
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/5ab6538b379d20ad880000ab/java

Module 2

Java Classes : A Review

Section 2–1

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 Java, we can't have global functions or variables 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**

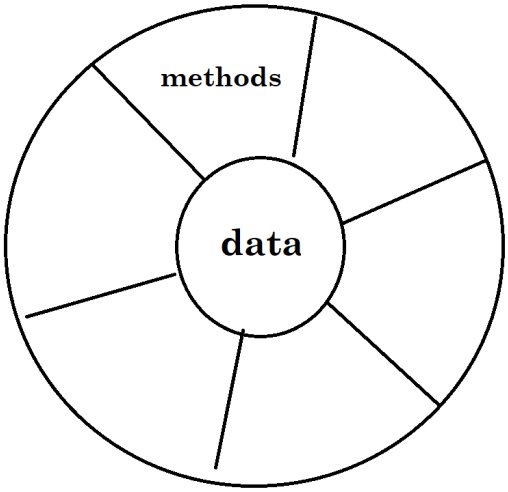
- **Objects on the other hand are instances of classes**
  - Classes are blueprints; objects are the things the blueprints build

- **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 is 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's 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);
```

- **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);
```

- **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());
```

Derived `getters`

- **Getters and Setters as special methods in Java, which give the outside world controlled access to the variables of a class**

- **Most getters in Java refer return the value of a backing variable**

  Example

```
public class Person {
  private String name;

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

- **But getters are not required to directly access a variable**
  - They can calculate the value that is to be returned

—

Example

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

  public String getFirstName(){
    return firstName;
  }

  public String getLastName(){
    return firstName;
  }

  // a calculated getter
  public String getFullName(){
    return firstName + " " + lastName;
  }
}
```

Derived `getters` *cont'd*

- **These getters are called derived getters**
  - You may also hear them referred to as derived properties in other programming languages
  - The purpose of the getter has not changed, it's only purpose is to return the value of some stored data in the class
  - Notice that `getFullName()` does not have a specific backing variable called `fullName`
    - It would be redundant to store the first and last names and then to also declare a fullName variable
- **In Java any method that begins with the words `get` or `set` should ONLY provide access to internal data**
  - These special methods SHOULD NOT perform any tasks other than to retrieve or store data

Boolean `getters`

- **Getters that access boolean variables generally do not begin with the word `get`**
  - This is not a hard and fast rule, but is a generally accepted Java best practice
  - They usually begin with the word `is()` since a boolean is intended to answer a true/false question

Example

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

  public String getName(){
    return name;
  }

  public int getAge(){
    return age;
  }

  public boolean isAdult(){
    return age > 18;
  }
}
```

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**
  - In this course we will use access modifiers for all members and functions

Exercise: Creating Classes

In the following exercises you will create classes for objects that you might use in a Hotel application. These classes will be similar to the objects that you discovered earlier, but there will likely be some differences.

In this exercise you will only focus on the responsibilities related to the things that an object **knows.**

Create a new folder in the `pluralsight` directory named `workbook-4`. You will complete all of this weeks exercises in this folder.

EXERCISE 1

In IntelliJ create a new Java application named HotelOperations. You will create several classes and test their functionality. Use the `main()` method in the Main class to create and test your logic in the classes.

In each of the classes described below, you will create the class and add all necessary backing variables. Each class should have a constructor that sets the initial value of each backing variable.

NOTE: Remember to create derived getters where appropriate. Not all getters should have a backing variable. In those cases, use the other variables to calculate the response.

Room

The `Room` class is responsible for knowing everything related to a hotel room. A room is only available if it is clean and not currently occupied. Create the class with the following getters:

```
getNumberOfBeds()
getPrice()
isOccupied()
isDirty()
isAvailable()
```

Reservation

The `Reservation` class is responsible for storing information related to a guest stay.

Determine which backing variables you need to create and create a constructor to set initial values.

The room type can be either "king" or "double". If the room type is "king" the price per night is $139.00 if the room type is "double" the price per night is $124.00

If the stay is over a weekend, the price per night should increase by 10%

Create the class with the following getters and setters:

```
getRoomType()
setRoomType(String roomType)
getPrice()
getNumberOfNights()
setNumberOfNights(int numberOfNights)
isWeekend()
setIsWeekend(boolean isWeekend)
getReservationTotal()
```

Employee

The Employee class is used to store and calculate payroll information about and employee. It should manage the following information using private variables: `employeeId, name, department, payRate, hoursWorked`.

Include the following derived getters (you may also include others as necessary):
`getTotalPay`
`getRegularHours`
`getOvertimeHours`

Commit and push your code!

Section 2–2

Methods

Encapsulation Part 2

- **Information Hiding**
  - What data/information am I responsible for?
  - This is what getters and setters provide
- **Implementation Hiding**
  - Encapsulation is more than just hiding data
  - What tasks can I perform?
  - Do I have all of the data that I need to perform my tasks?
- **Cohesion**
  - How closely related is the data that I am responsible for?
  - How closely related are all the functions that I can do?
  - How closely are the functions related to all the data?
- **High vs Low Cohesion**
  - High cohesion means all members and functions are closely related
  - Low cohesion means that members and functions are unrelated
  - Our goal is high cohesion

Adding Methods

- **Methods are functions in a class**
- **Methods are often dependent on member attributes/variables**
- **They do more than just `get()` and `set()` data**
  - getters and setters are methods, but they are special methods reserved just to access internal data
- **Standard methods are used to perform additional actions or logic**

Example: Methods

Car.java

```
public class Car {
   // attributes
   private String make;
   private String model;
   private int speed;

   // constructor
   public Car(String make, String model) {
      this.make = make;
      this.model = model;
      this.speed = 0;
   }

   // getters and setters
   public String getMake() {
      return this.make;
   }

   public void setMake(String make) {
      this. make = make;
   }

   public String getModel() {
      return this.model;
   }

   public void setModel(String model) {
      this. model = model;
   }

   public int getSpeed() {
      return this.speed;
   }

   // no setter for speed

   // other methods
   public void accelerate(int changeInSpeed) {
      this.speed += changeInSpeed;
   }

   public void brake(int changeInSpeed) {
      if(changeInSpeed > this.speed){
         this.speed = 0;
      } else {
         this.speed -= changeInSpeed;
      }
   }
}
```

Example: Using the methods

- **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

```
String make = "Ford";
String model = "Mustang";

Car car1;
car1 = new Car(make, model);

System.out.println(car1.getSpeed());
car1.accelerate(10);
car1.accelerate(25);
System.out.println(car1.getSpeed());
car1.brake(10);
System.out.println(car1.getSpeed());
```

Exercise: Creating Classes

Continue working with the HotelOperations project that you created in the previous exercise.

EXERCISE 2

Room

Modify the `Room` class to add `checkIn(), checkout()` and `cleanroom()` methods to the room.

Once a room has been checked in, it should be occupied and marked as dirty.

When a guest checks out of a room it must first be cleaned by a housekeeper before another guest can check into the room.

Employee

Modify the `Employee` class to add a `punchIn` and `punchOut` methods. Each time the employee punches in, we track their start time.

When they punch out, we calculate how many hours they have worked and add that time to their hours worked.

To keep the math simple for now, each function will take a double as an input argument.

```
punchIn(double time)
punchOut(double time)
```

Example
input variables for time
```
10:00 am => 10.0
12:30 pm => 12.5
2:45  pm => 14.75
```

Bonus

Instead of using two methods (`punchIn` and `punchOut`) modify your code to only have a single method called `punchTimeCard`.

Commit and push your code!

Section 2–3

Method Overloading

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 an 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);
```

Why Overload Methods?

- **Methods can return specific information**
  - i.e. search for customer information by room number
- **Sometimes we don't have the required input (room number) so we have to search by different information**
  - i.e. search for customer by name
- **Without overloading we would have to create multiple different functions, with different names**

Example

```
searchForCustomerByRoomNumber(int room)
searchForCustomerByRoomAndDate(int room, LocalDate date)
searchForCustomerByName(String last, String first)
```

- **Thanks to overloading we can create a single function name that can handle all scenarios**

Example

```
searchForCustomer(int room)
searchForCustomer(int room, LocalDate date)
searchForCustomer(String last, String first)
```

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;
   }

   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());
   }
}
```

Exercise: Overloading Methods

EXERCISE 3

Continue working with the HotelOperations project.

Employee

Modify the Employee class and create overloaded methods for `punchIn()` and `punchOut()`. These overloaded methods should not accept any input parameters but should determine what the time and minutes are based on the current time.

HINT: Use Java's `LocalDateTime` object to get access to the current time, and to determine what hour and minute the employee clocked in/out.

After this exercise you should have these 4 methods:

```
punchIn(double time)
punchIn()
punchOut(double time)
punchOut()
```

Hotel

This class will allow you to create a simple hotel object and check if there are rooms available for the current day only. The hotel has 2 types of rooms a King Suite and a Basic Double room. Your `Hotel` class should track the `name`, `numberOfSuites`, `numberOfRooms`, `bookedSuites` and `bookedBasicRooms`. There should be no public setters for these variables.

Constructors
There should be 2 constructors. One sets only the number of suites and rooms. In this constructor the number of `bookedSuites` and `bookedBasicRooms` should default to 0. The other constructor should specify the number of `bookedSuites` and `bookedBasicRooms`.

```
Hotel(String name, int numberOfSuites, int numberOfRooms)
Hotel(String name, int numberOfSuites
                 , int numberOfRooms, int bookedSuites
                 , int bookedBasicRooms)
```

Methods
A user should be able to book one or more rooms (if they are available). The user will specify how many rooms they would like, and if it is a suite or a basic room.

```
public boolean bookRoom(int numberOfRooms, boolean isSuite)
```

The `bookRoom` method should determine if there are enough rooms available and update the booked inventory if they are. The method should return `true/false` based on whether the rooms were able to be booked.

The class should also include `getAvailableSuites` and `getAvailableRooms`. These are derived getters that SHOULD NOT have a private backing variable. Instead, these getters should calculate the response based on other member variables.

Commit and push your code!

Section 2–4

CodeWars

CodeWars Kata

- **12 Days of Christmas**
  - Your function accepts 2 strings. Each string is one of the lines in a verse of the song the 12 Days of Christmas.
  - The song is out of order, and this function is used to sort the lines into their correct order.
  - A sort function returns -1, 0 or 1 if the first line if less than, greater than or equal to the second line.

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/57dd3a06eb0537b899000a64/java

Module 3

Unit Testing

Section 3–1

Types of Software Testing

Introduction to Testing

- **Testing software has always been important**
- **Good software must be free of defects**
- **Software can be tested manually or it can be automated**
  - Manual testing involves having a testing engineer run the program through various scenarios
    - A testers goal is to try to find vulnerabilities and to break the software before end users find them
    - Manual testing is time consuming and tedious - any changes to the code requires all tests to be re-run
  - Test can be automated so that a large number of tests can be run automatically without needing to involve a test engineer
    - Automated tests are less time consuming and help free up time of a test engineer to place efforts elsewhere

Types of Testing

- **Exploratory Testing**
  - The name says it all - this type of testing is forging into the unknown - the tester tries to discover new ways that the software might be used
  - Exploratory testing is manual
- **Regression Testing**
  - Regression testing involves creating test scenarios for bugs that have previously been found and fixed
  - The goal of regression testing is to ensure that old bugs don't find their way back into the system
  - Since the issues being tested are well known regression testing is usually automated, but can also be performed manually
- **Automated Testing and the Testing Pyramid**
  - Unit testing
    - Tests that verify the methods of the objects in a system
  - Integration testing
    - Tests that verify the interaction between objects or sub-systems
  - End-to-End testing
    - Tests written to verify the full end-to-end experience of a user

The Testing Pyramid

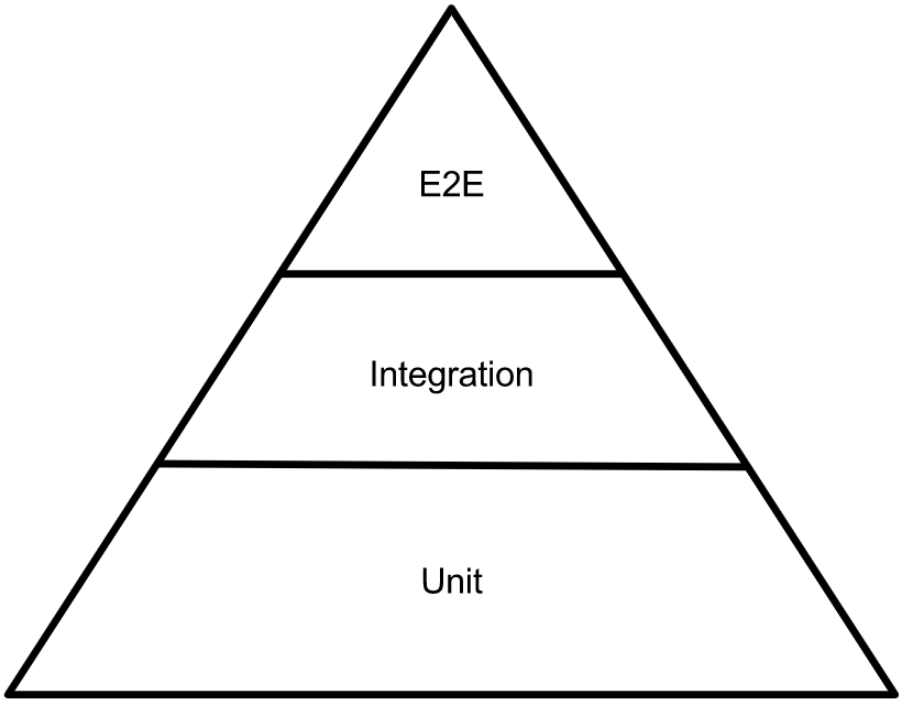
- **The testing pyramid is a common diagram used in software testing**

- **It is a representation of how many different types of tests are written**

  – Most of the automated test should be **Unit Tests**. They should be fast and verify that each component (class) works as designed

  – **Integration Tests** test the interaction between objects. These tests take longer to run than Unit Tests and there are fewer integration tests than unit tests.

  – **End-to-End** tests require significant setup effort and usually take a long time to execute. These are the fewest number of tests in the pyramid, but they may also take the longest overall to run.



JUnit Testing Framework

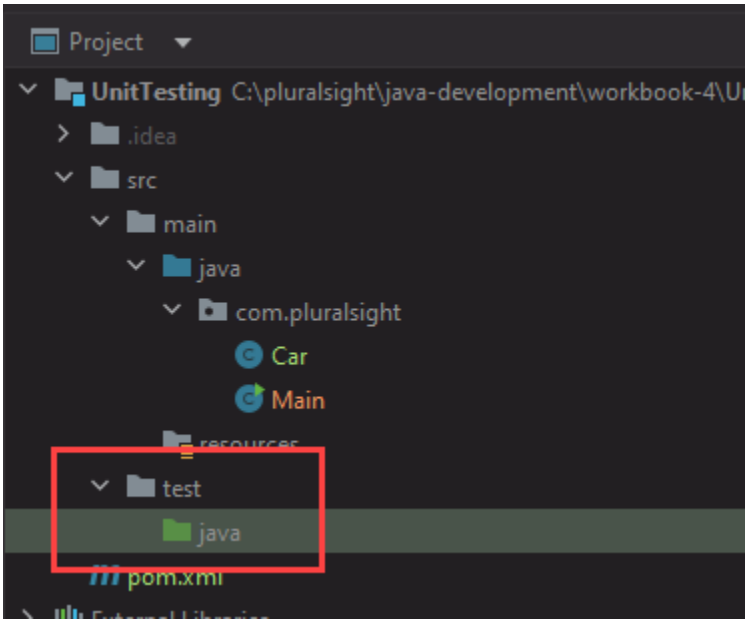
- **JUnit is a framework used to write unit tests in Java**
- **JUnit was first released in 2002 and is the predominant testing framework for Java development**
- **JUnit 5 - also named JUnit Jupiter - is the current release**
  - JUnit made several breaking changes when moving from JUnit 4 to JUnit 5
    - It is not a trivial process to upgrade existing tests to the JUnit 5 framework
  - JUnit Jupiter gives developers greater flexibility writing tests
    - Parameterized tests
    - Tagging tests - useful for filtering
    - Test Templates
- **JUnit provides a test runner**
  - Code cannot run on its own, it needs an application to manage its execution
  - A testing framework provides a test runner (application) which executes your code
    - Then reports if your code behaved the way that you expected it to behave

Section 3–2

Setting Up Unit Tests

Adding Unit Tests

- In a Maven project structure, unit tests are separated from the application code by adding the tests to a parallel "test" folder

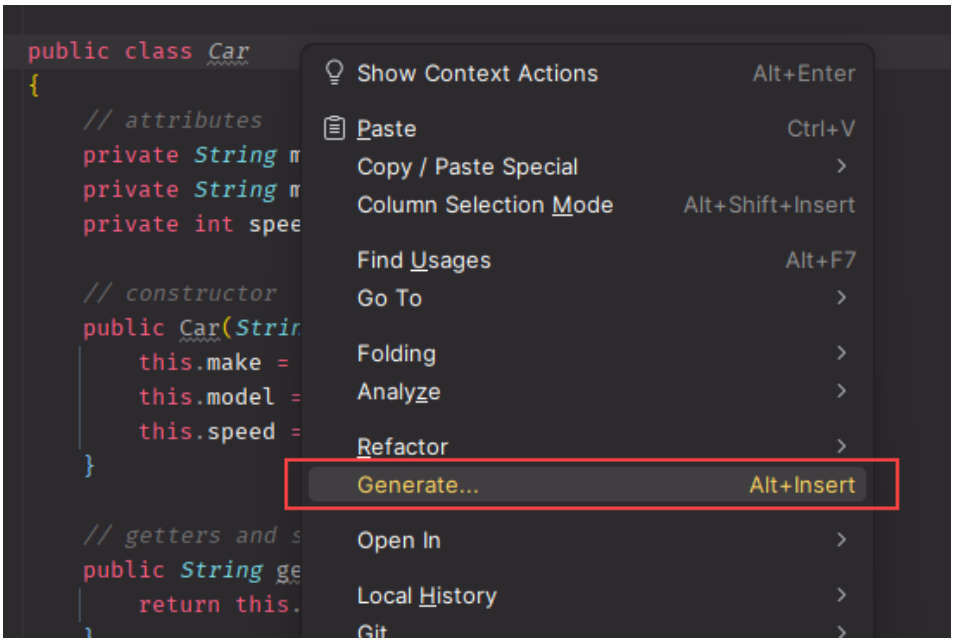


- It is common to create 1 unit test class for each class in your application

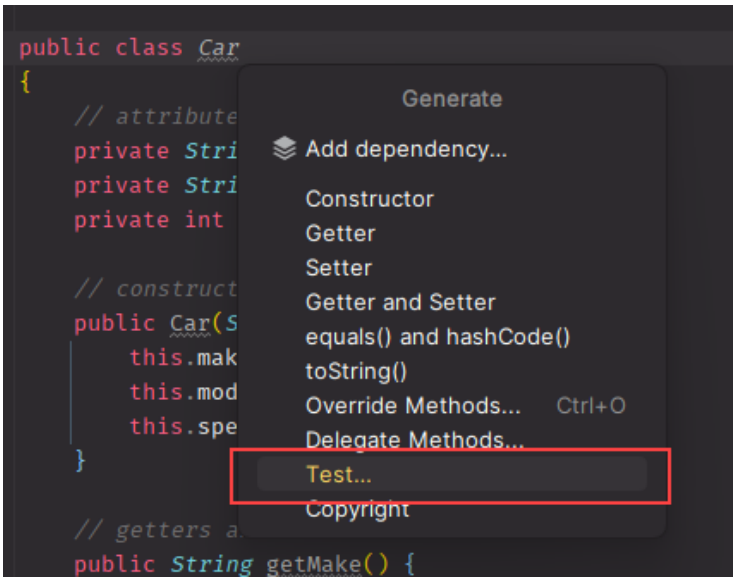
- IntelliJ makes it easy to create a test class

Adding Unit Tests *cont'd*

- **Open the class for which you want to create tests**
  - Right click anywhere in the file and choose `Generate...`

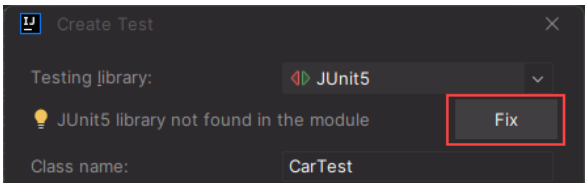


  - Then select `Test...`

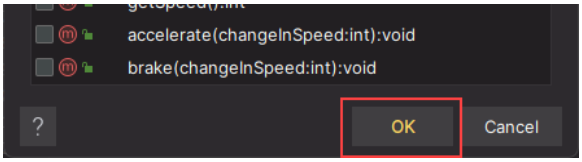


Adding Unit Tests *cont'd*

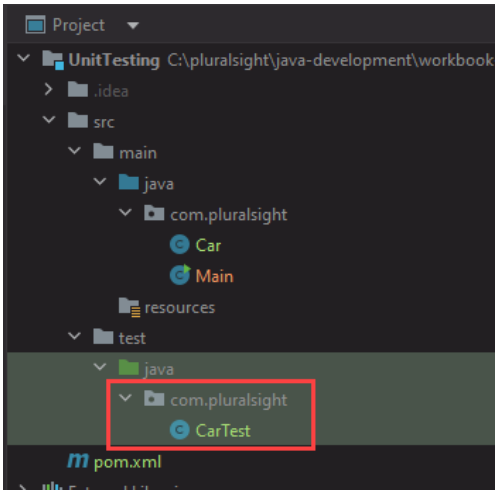
- Ensure that JUnit5 is the selected framework
  - If this is the first test you are adding to your project you will see a message that says JUnit5 library is not found in the module
  - Click the `Fix` button - this will add the required JUnit dependency to your project



- Then click `OK`



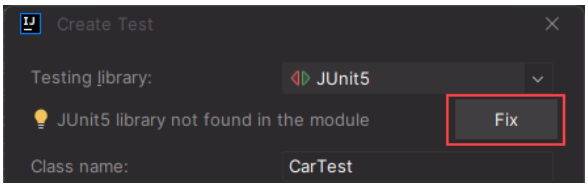
- **This added a new test class to your project**
  - Notice that it also created the same package structure that is found in your application



JUnit Dependencies

- **When you add a JUnit test class, you must also add the JUnit dependencies to your project**

- **If you clicked the `Fix` button those dependencies were added for you**



- **The dependency is added in your pom.xml file**
  - We will discuss the pom.xml file and how to manage dependencies in greater detail later - for now, just know that your tests will not compile without this change

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apac
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>UnitTesting</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>RELEASE</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

Writing Your First Test

- **A Unit Test is nothing more than a function**
  - It is only special because it is marked / labeled as a test function

Example

CarTest.java

```
@Test
public void acceleate_should_increaseCarSpeed() {
  // arrange

  // act

  // assert
}
```

- **The `@Test` annotation lets the JUnit test runner know that this is a function that needs to be executed when running all tests**

- **The naming conventions of a test is different than regular functions**
  - The name of a test should read more like a sentence which describes the scenario that you are trying to test
  - See these popular naming convention guidelines
    * https://dzone.com/articles/7-popular-unit-test-naming

Parts of a Unit Test

- **Each unit test consists of 3 main sections**
  - Arrange
    - This is where you set up all variables that you will need to execute the test
  - Act
    - This is the **one method call** that you are testing in this test scenario
  - Assert
    - This is where you write code to verify that your method under test performed as expected

Example

CarTest.java

```
@Test
public void acceleate_should_increaseCarSpeed() {
  // arrange

  // act

  // assert

}
```

- **It is easy as a developer to get lazy and combine sections - but this generally leads to sloppy development**
  - Unit tests should be very easy to read in order to understand what is being tested
  - It is best to get into a habit of taking time to make your tests clean

Arrange

- **As the name suggests, the arrange section is where you set up your test**

- **Declare all variables that you will need to use throughout your test**
  - Declaring all variables helps you avoid the use of "magic numbers" or "magic strings" within your tests
  - It also helps clarify the meaning and intent of your test
  - Variables:
    - **`Car:`** `this is the object under test`
    - **`speedChange:`** `the input parameter to test`
    - **`expectedSpeed:`** `the expected speed after the test`

Example

`CarTest.java`

```
@Test
public void acceleate_should_increaseCarSpeed() {

  // arrange
  Car car = new Car("Ford", "F150 Raptor");
  int speedChange = 15;
  int expectedSpeed = 15;

  // act
  car.accelerate(speedChange);

  // assert
  int actualSpeed = car.getSpeed();
  assertEquals(expectedSpeed, actualSpeed);
}
```

Act

- **The Act section is the section used to call the function that is being tested in this scenario**
  - You may call more than one function on a class during the test but each scenario should only truly be testing **ONE METHOD**
  - Other method calls should only be used to set up the test, or verify that the test succeeded

Example

CarTest.java

```
@Test
public void acceleate_should_increaseCarSpeed() {
  // arrange
  Car car = new Car("Ford", "F150 Raptor");
  int speedChange = 15;
  int expectedSpeed = 15;

  // act
  car.accelerate(speedChange);

  // assert
  int actualSpeed = car.getSpeed();
  assertEquals(expectedSpeed, actualSpeed);

}
```

Assert

- **The Assert section is intended to verify that your test completed successfully**
  - At this point of the test the object method will have been executed
  - It is your responsibility to verify that the method performed it's task as expected
    - If the method returned a value, verify that the value is correct
    - If the method did not return a value, verify that the internal state of the object was modified correctly

Example

CarTest.java

```
@Test
public void acceleate_should_increaseCarSpeed() {
  // arrange
  Car car = new Car("Ford", "F150 Raptor");
  int speedChange = 15;
  int expectedSpeed = 15;

  // act
  car.accelerate(speedChange);

  // assert
  int actualSpeed = car.getSpeed();
  assertEquals(expectedSpeed, actualSpeed);
}
```

Writing Assertions

- **The `org.junit.jupiter.api` package has an `Assertions` object**

- **The `Assertions` object contains many methods that are used to validate your test**
  - **`assertEquals`**`(expected, actual)` - compares the value of 2 input arguments for equality
  - **`assertNotEquals`**`(expected, actual)` - compares the value of 2 input arguments for equality and fails if they are equal
  - **`assertArrayEquals`**`(expectedArray, actualArray)` - compares the size and values of all elements in 2 arrays
  - **`assertTrue`**`(booleanValue)` - checks if an input value is true
  - **`assertFalse`**`(booleanValue)` - checks if an input value is false

Writing Tests

- **Writing tests is easy, but writing GOOD tests can be tricky**
- **Before writing your test, make sure that you understand what scenario you are testing and why you are testing it**
  - You may test multiple scenarios of the same method in a class
    - Don't try to fit them all into the same test
    - Each scenario should have its own test, so that the purpose of each test is clear
- **Make sure that your test does not deviate from the purpose of the scenario**
- **An application that is thoroughly tested will have many more lines of test code than application code**
  - Be careful in using lines of code as a measurement of quality
  - You should write enough unit tests to be confident in the quality of your functional code

Example: Unit Tests

Example

CarTest.java

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CarTest{

  @Test
  public void acceleate_should_increaseCarSpeed() {
    // arrange
    Car car = new Car("Ford", "F150 Raptor");
    int speedChange = 15;
    int expectedSpeed = 15;

    // act
    car.accelerate(speedChange);

    // assert
    int actualSpeed = car.getSpeed();
    assertEquals(expectedSpeed, actualSpeed);
  }

  @Test
  public void brake_should_decreaseCarSpeed() {
    // arrange
    Car car = new Car("Ford", "F150 Raptor");
    int speedUpBy = 15;
    int slowDownBy = 10;
    int expectedSpeed = 5;
    car.accelerate(speedUpBy);

    // act
    car.brake(slowDownBy);

    // assert
    int actualSpeed = car.getSpeed();
    assertEquals(expectedSpeed, actualSpeed);
  }
```

```
  @Test
  public void brake_should_stopCar_whenChangeIsGreater() {
    // arrange
    Car car = new Car("Ford", "F150 Raptor");
    int speedUpBy = 15;
    int slowDownBy = 20;
    int expectedSpeed = 0;
    car.accelerate(speedUpBy);

    // act
    car.brake(slowDownBy);

    // assert
    int actualSpeed = car.getSpeed();
    assertEquals(expectedValue, actualValue);
  }
}
```

Exercise: Creating Unit Tests

In this exercise you will focus on writing unit tests using JUnit. Classes are perfect for unit testing because the provide a natural organization for creating (and testing) small units of code.

EXERCISE 1

Open the HotelOperations project that you created earlier.

You will create unit tests for your Hotel application. Consider what actions each of the methods completes and think about how you would use an assertion to verify each action.

RoomTest

Create a `RoomTest` unit test class to test the methods of the `Room` class. Write tests for the following functions:

```
checkIn() // room should be occupied and dirty
checkout()
cleanroom()
```

How many tests should you write?
Should you be able to check in a room, if it is already occupied?
What if it is dirty?
Should housekeeping be able to clean the room if it is occupied?

EmployeeTest

Create and `EmployeeTest` class to test the `punchIn` and `punchOut` methods of the `Employee` class. Verify that when an employee punches out, their time is calculated correctly.

```
punchIn(double time)
punchOut(double time)
```

Commit and push your code!

Section 3–3

CodeWars

CodeWars Kata

- **Array plus Array**
  - Calculate the sum of 2 integer arrays.
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/5a2be17aee1aaefe2a000151/java

Module 4

Static Methods and Variables

Section 4–1

Introduction to Static

What is `static`?

- **There are 2 types of variables and methods in Java**
  - Instance variables and static variables
  - Instance methods and static methods
- **So far we have only been dealing with instance variables and methods**
  - Instance variables and methods are defined as part of a class
  - Each time a new instance of the class is created a new instance of the variables and methods are also created
    - I.e. Each class has its own copy of the variables/methods
- **Static members are treated differently**
  - A static variable or method is created immediately when a class is loaded into memory
  - The application will only have a single copy of that variable or method
    - It is created as a shared resource among all instances of that class
- **Static variables should never store instance specific information**

Instance Members

- **The class definition is a template for what variables and methods will be created when a new object is instantiated**

Example

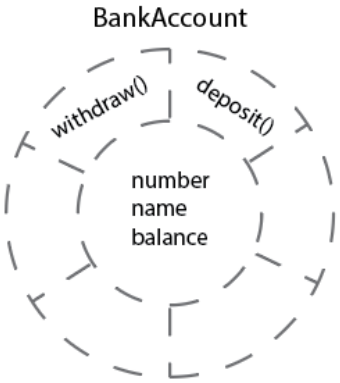
```
public class BankAccount {

  private String number;
  private String name;
  private double balance;

  public BankAccount(String number, String name,
                     double balance) {
    this.number = number;
    this.name = name;
    this.balance = balance;
  }

  public double deposit(double amount) {
    this.balance += amount;
  }

  public double withdraw(double amount) {
    this.balance -= amount;
  }

}
```

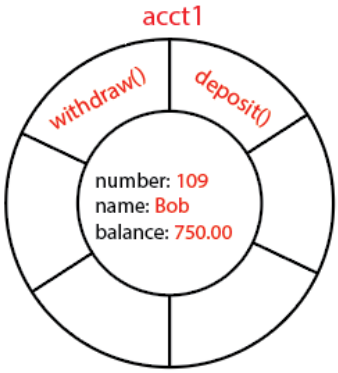
Example

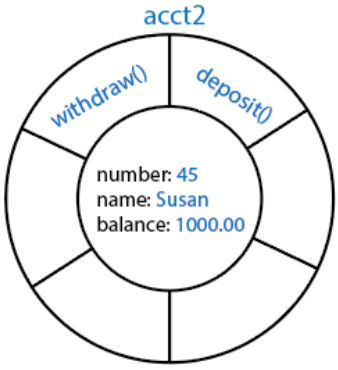
```
BankAccount acct1 = new BankAccount("109","Bob", 745.00);
BankAccount acct2 = new BankAccount("45","Susan", 1000.00);
```

NOTE: The BankAccount type is just a template. Methods and variables are not created until the `new` keyword creates a new instance.

The `withdraw()` and `deposit()` methods in `acct1` can only access and change the variables inside `acct1`. They do not have access to the variables in `acct2`.







Static Members

- **Static variables and methods are created only once**
  - Each instance DOES NOT get its own copy of the data

Example

```
public class BankAccount {

  private static double interestRate;

  private String number;
  private String name;
  private double balance;

  public BankAccount(String number, String name,
                     double balance) {
    this.number = number;
    this.name = name;
    this.balance = balance;
  }

  public static double getInterestRate() {
    return interestRate;
  }

  public static void setInterestRate(double interestRate) {
    BankAccount.interestRate = interestRate;
  }

  public void deposit(double amount) {
    this.balance += amount;
  }

  public void withdraw(double amount) {
    this.balance -= amount;
  }

}
```

Example

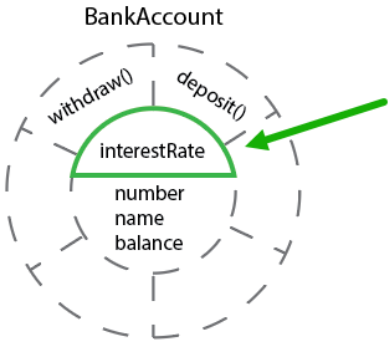
```
BankAccount.setInterestRate(0.045);
```

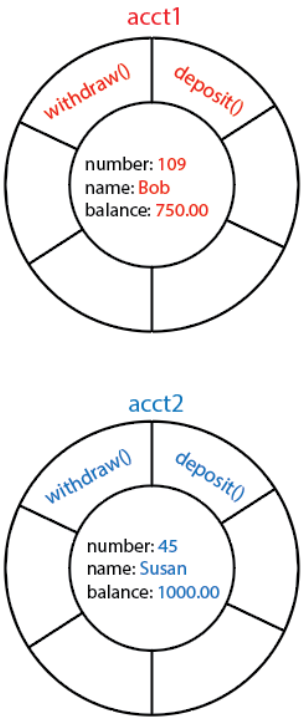
NOTE: Static methods are available without creating a BankAccount object.

```
BankAccount acct1 = new BankAccount("109","Bob", 745.00);
BankAccount acct2 = new BankAccount("45","Susan", 1000.00);
```

NOTE: Making a change to the `interestRate` will change the value for **ALL** instances of BankAccount.

Notice that `acct1` and `acct2` DO NOT have a copy of `interestRate`.





Static Methods in Java

- **Launching a Java program**
  - In order to launch a Java application you must have a method with the following signature
  - It is the entry point into the application

Example

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

- **There are other examples of static methods all throughout Java libraries**
  - You can spot a static method because the function is tied to the type name instead of a variable name

- **Static methods are used as factory or helper functions to accomplish common tasks**
  - The are standalone functions that are not dependent on instance variables

Example

```
String name = "Michael";
String.format("Hello %s", name);

int age = Integer.parseInt("35");
double price = Double.parseDouble("15.95");
```

Static Classes

- **Java does not support global functions - all functions must be defined in a class**
  - But sometimes we need a library of reusable function
  - Static classes are similar to global functions, they give us a place to organize related functions
- **Static classes are made up of ONLY static methods and variables**
  - This allows us to create a library of variables and methods without having to create or maintain multiple copies
  - It is not possible to create an instance of a static class
- **The Java `Math` class is an example of a static class**

Example: Math Class

Example

NOTE: all methods and variables in this class are static. All methods return a double

```
public class Math {

  // the constructor is private
  // so that an instance cannot be created
  private Math(){}

  public static final double PI = 3.1415926535...;

  // other static constants

  public static double sqrt(double a){…}

  public static double ceil(double a){…}

  public static double floor(double a){…}

  public static double round(double a){…}

  public static double pow(double a, double b){…}

  //additional static methods

}
```

Example

```
double num1 = Math.pow(5,2) // 25.0
double num2 = Math.sqrt(25) // 5.0
double num3 = Math.floor(12.8) //12.0
double num4 = Math.round(12.8) //13.0

// area of a circle : area = πr²
double radius = 5;
double area = Math.PI * Math.pow(radius, 2);
```

Exercise: Static Classes

Complete your work in the `workbook-4` folder.

EXERCISE 1

Create a new Java application named StaticClasses. The purpose of this exercise is to create a class that only has static methods.

In essence the class will be a name formatting factory. This class acts as a library of functions that are used to standardize the way a name should be formatted.

In our scenario a name can have the following components (`FirstName` and `LastName` are required, all other parts are optional):

```
Prefix:      Mr., Mrs., Miss., Dr. etc
FirstName
MiddleName
LastName
Suffix:      PhD, Jr, III etc
```

The standard formats that we require for all names must match this pattern:

```
LastName, Prefix FirstName MiddleName, Suffix
Johnson, Dr. Mel B, PhD
Johnson, Mel B, PhD
Johnson, Mel
```

NameFormatter

Create a new class called `NameFormatter`. Ensure that the constructor is private. Write the following static methods to return a formatted name in the standard format.

```
public static String format(String firstName,
                            String lastName)

public static String format(String prefix,
                            String firstName,
                            String middleName,
                            String lastName,
                            String suffix)

public static String format(String fullName)
```

HINT: Begin with the easiest methods first.

You should assume that an incoming `fullName` will always come formatted in this order:

```
Prefix FirstName MiddleName LastName, Suffix
Dr. Mel B Johnson, PhD
Mel B Johnson, PhD
Mel Johnson
```

Create unit tests to test the methods with a few different variations of names.

Commit and push your code!

Section 4–2

CodeWars

CodeWars Kata

- **Roman Numeral Encoder**
  - Convert decimal numbers into Roman Numerals
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/51b62bf6a9c58071c600001b/java

Module 5

Class Interactions

Section 5–1

Defining Class Relationships

Class Relationships

- **In OOP the goal is to make classes independent of each other**
  - It is impossible to make them completely independent, otherwise each class would have to be its own class
  - So the real goal is to minimize how many dependencies each class has
- **Cohesion refers to how closely variables and methods WITHIN a class are related to each other**
  - Remember, the goal in OOP is to have high cohesion within a class
- **Coupling refers to how closely classes are related to each other**
  - Tight coupling means that classes are highly dependent on each other
  - Loose coupling means that classes have minimal dependency on each other
  - The goal in OOP is that classes should be **loosely coupled**
- **Before classes can interact they have to know about each other**
  - This is usually accomplished through a Has-A relationship

Defining a Has-A Relationship

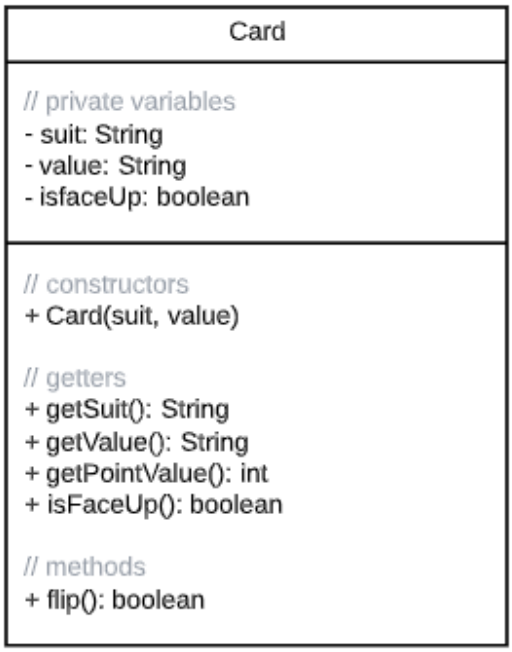
- **We will learn about class relationships by developing a card game**

- **Most card games have the same basic structure - these are the classes that we will need**
  - Card
  - Hand
  - Deck

- **We could add many more classes, but for our purposes we will stick with these three classes**

The Card Class

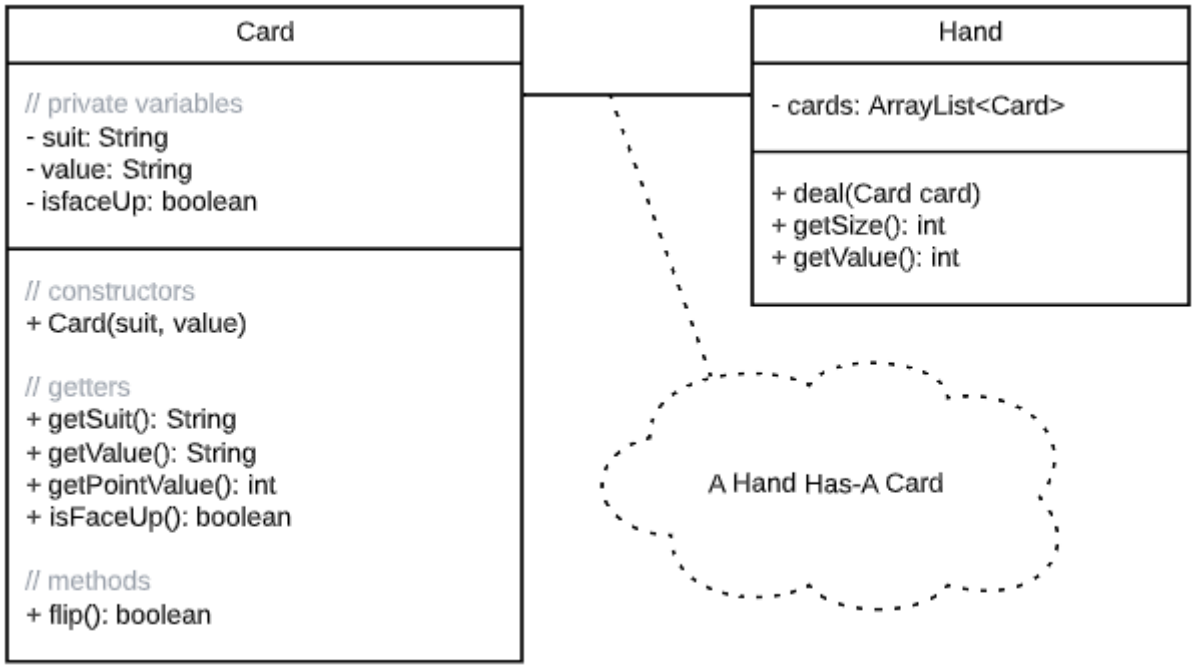
- **A Card will be the primary object within this application**
  - A **-** indicates that the field or method is private
  - A **+** indicates that the field or method is public



```
public class Card {
  private String suit;
  private String value;
  private boolean isFaceUp;

  public Card(String suit, String value) {
    this.suit = suit;
    this.value = value;
    this.isFaceUp = false;
  }

  public String getSuit(){
    // only return the suit if the card is face up
    if(isFaceUp){
      Return suit;
    } else {
      Return "#";
    }
  }

  public String getValue(){
    // only return the value if the card is face up
    if(isFaceUp){
      // this is the string value of the card
      // i.e. A, K, Q, J, 10, 9 ...
      return value;
    } else {
      return "#";
    }
  }

  public int getPointValue(){
    // only return the value if the card is face up
    if(isFaceUp){
      // determine point value and return it
      // A = 11
      // K, Q, J = 10
      // all numeric cards are equal to their face value
    } else {
      return "#";
    }

  }

  public boolean isFaceUp(){
    return isFaceUp;
  }

  public void flip(){
    isFaceUp = !isFaceUp;
  }
}
```

The Hand Class

- **In a card game, a hand is dependent on a card**
  - A hand cannot do much without any cards
- **We refer to this as a Has-A relationship**
  - A `Hand` has-a `Card` (or possibly many)
  - The line between the classes indicates that there is an association or dependency between the 2 classes



- **The value of the `Hand` is determined by the value of each card in the `Hand`**
  - A `Hand` has a dependency on a `Card`
  - But a `Card` DOES NOT have a dependency on a `Hand`

The Hand Class *cont'd*

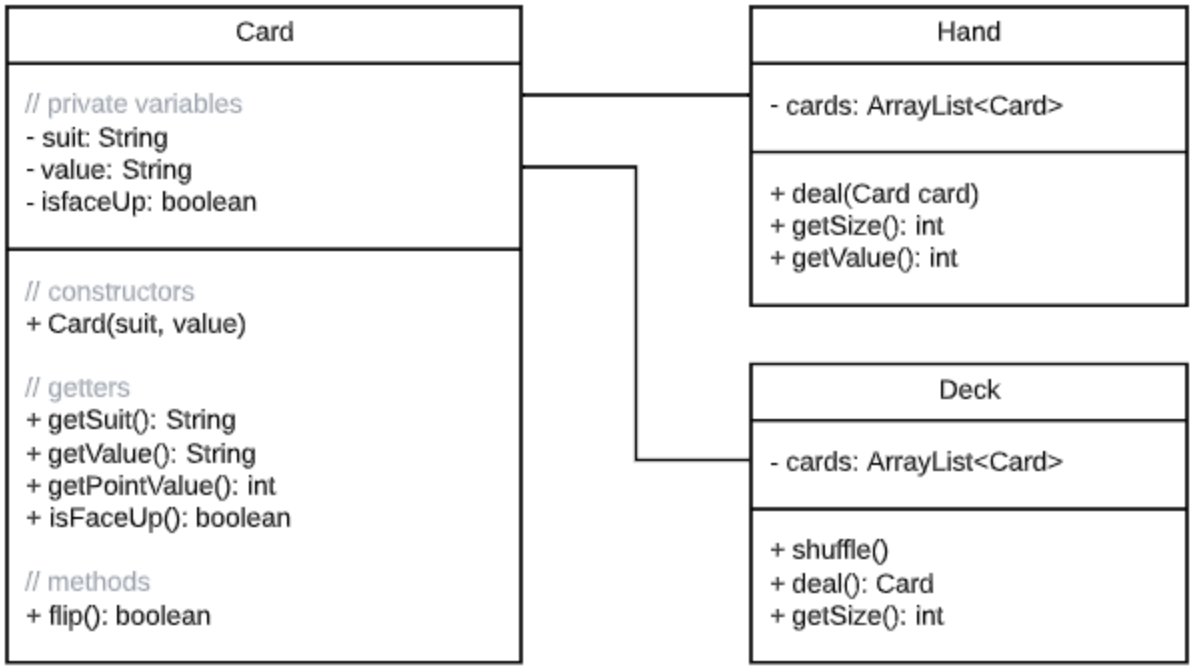
Example

NOTE: The Hand class has stored a list of cards, and then accesses each card as needed to determine the total value of the hand

```
public class Hand {
  private ArrayList<Card> cards;

  public Hand(){
    cards = new ArrayList<>();
  }

  // A Card is dealt to the Hand and the Hand is responsible
  // to store the card
  public void Deal(Card card){
    cards.add(card);
  }

  public int getSize(){
    return cards.size();
  }

  // The Hand uses the methods of each card to determine
  // the value of each card - and adds up all values
  public int getValue(){
    int value = 0;

    for(Card card: cards){
      card.flip(); // turn the card over to see the value
      value += card.getPointValue();
      card.flip(); // hide the card again
    }

    return value;
  }
}
```

The Deck Class

- **Similar to the `Hand`, a `Deck` has a dependency on the `Card` class**
  - A `Deck` has one or more `Cards`
- **Other responsibilities of the Deck**
  - A Deck should create all cards in its constructor
  - Shuffle all cards
  - Deal cards one at a time



- **Notice that there is no dependency between `Deck` and `Hand`**
  - A `Hand` does not need to know about the `Deck`
  - And the `Deck` does not need to know about the `Hand`

The Deck Class *cont'd*

Example

```
import java.util.Collections;

public class Deck {
  private ArrayList<Card> cards;

  public Deck(){
    cards = new ArrayList<>();
    String[] suits = {"Hearts","Spades","Diamonds","Clubs"};
    String[] values = {"2","3","4","5","6","7","8",
                       "9","10","J","Q","K","A"};

    // these loops create all the cards in the deck
    // and add them to the ArrayList
    for(String suit: suits){
      for(String value: values){
        Card card = new Card(suit, value);
        cards.add(card);
      }
    }
  }

  public void shuffle(){
    Collections.shuffle(cards);
  }

  public deal(){
    // deal the top card (if there are any cards left
    if(cards.size() > 0){
      Card card = cards.remove(0);
      return card;
    } else {
      return null;
    }
  }

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

Communication Between Classes

- **By separating the data and logic into 3 separate classes, your main application does not need to keep track of all the information**

- **The application only needs to know where it can find the information and which objects can do the work**
  - Notice in the example below, that the main application does not have any logic related to the calculation of card or hand values
  - The only purpose of the application is to manage the flow of the game

Example

```
public class MainApp {

  public static void main(String[] args) {

    Deck deck = new Deck();

    Hand hand1 = new Hand();

    // deal 5 cards
    for(int I = 0; I < 5; i++) {
      // get a card from the deck
      Card card = deck.deal();
      // deal that card to the hand
      hand1.deal(card)
    }

    int handValue = hand1.getValue();
    System.out.println("This hand is worth " + handValue);
  }
}
```

Exercise

EXERCISE 1

Create a new Java Maven project named BlackJack.

Use the class definitions that we have discussed in this module to create classes for a card game. You will need a deck of cards, and you will need to be able to deal cards to each players hand.

The examples we have covered do not track the player names. Add any variables and/or classes that you think are necessary to meet these requirements.

Start the game by prompting the user for the name of each player.

Create a deck and shuffle the cards, then deal 2 cards to each hand. Display each player's hand and determine which player is closest to 21 without going over. Declare that player as the winner.

Scoring:
All number cards are worth their numeric value
Face cards are worth 10 points
Ace is worth 11 points

Optional Bonus

Prompt the user for the number of players that will be playing the game. Then create that many Hands.

If you have time. Take turns and allow each player to choose if they want to Hit to take another card, or to Stay.

Ace is worth 11 points by default. Add logic to count Ace as 1 point if 11 would cause the hand to bust.

Section 5–2

CodeWars

CodeWars Kata

- **Keep Hydrated**
  - Calculate how many full liters of water you should have consumed based on the number of hours you have exercised

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/582cb0224e56e068d800003c/java