CS220 — Software Development II

J Maraist

September 11, 2017

Outline

Contents

1 Introduction 1
2 JUnit and test-driven development 3
3 Classes, objects, inheritance and notation 12

Course notes for CS220 are maintained in this file. They will be uploaded after classes, about once per week.

1 Introduction

The Software Design sequence

• Design algorithms
  – High-level problem-solving skills

• Implement algorithms as programs
  – Java - A modern programming language
  – Organize data and instructions
  – In both algorithms and programs, we must use low-level and precise logic
    * No ambiguity allowed
  – Debugging and testing

• Understand what programs will do
• Communicate technical information about your programs
• Learn how to operate as a technical professional
Software Design I

The six things a program can do

1. Get input
2. Give output
3. Do arithmetic
4. Update a stored value
5. Test a condition, and select an alternative
6. Repeat a group of actions

Four ways Java will help you organize your work

1. Grouping related data together
2. Defining sequences of operations
3. Associating data with operations relevant to the particular data
4. Naming these groups, sequences and associations for easy and repeated use

Software Design II

- A deeper look at inheritance and object-oriented design
- Exceptions
- Recursion
- Linear (list) data structures
- Describing and tracing the effects of programs
- Programming with files and directories
- Multi-dimensional arrays
- Debugging and testing
  - Knowing what "correct" means, and how to tell if your code meets that standard
  - This class will have a strong focus on test-driven development
- Skills-based class
- Many things we learn will build on what we’ve already studied
  - Including all of Software Design I
- Practice is essential
  - Expect to work on CS220 every day
Assignments
Each assignment is...

- Posted to the course website
- Submitted through AutoLab
  - [https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmaraist/](https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmaraist/)
  - Remember the campus VPN [vpn.uwlax.edu](http://vpn.uwlax.edu)
- You should have received an email with account information on Monday
  - Check your spam filter
  - If you registered on Friday or after, I may not have your info
  - *Email by 2pm if you still need one, and I’ll make additional accounts this afternoon*
- Tomorrow’s lab will step you through a first AutoLab submission

Textbooks
There is no required text for this class, but you may be happier with a reference of some sort and/or a source of practice exercises. Some options:

- The CS120 online book, *Programming in Java*, zyBooks
  - Can print sections/chapters as well
  - Subscribe:
    - *Sign up at zyBooks.com*
    - *Enter zyBook code: UWLAXCS220MaraistFall2017*
    - *Click Subscribe*
  - Several faculty recommend this book as a second reference for CS120
- O’Reilly has been reputable for reference books

2 JUnit and test-driven development

JUnit
- For specifying and running functional tests in Java
- A separate test for every method
  - Use Java annotations to mark the test
- The JUnit executable finds test methods, runs them, reports the results
- Eclipse will alert you to test failures
JUnit example test subject
From the JUnit wiki:

- A simple class

    ```java
    public class Calculator {
        public int evaluate(final String expression) {
            int sum = 0;
            for (String summand: expression.split("\+"))
                sum += Integer.valueOf(summand);
            return sum;
        }
    }
    ```

JUnit example test class

- Test Calculator with

    ```java
    import static org.junit.Assert.assertEquals;
    import org.junit.Test;
    public class CalculatorTest {
        @Test public void evaluatesExpression() {
            final Calculator calculator = new Calculator();
            final int sum = calculator.evaluate("1+2+3");
            assertEquals(6, sum);
        }
    }
    ```

- The @Test annotation: how JUnit finds tests
  - Documentation at junit.org/junit4/javadoc/latest/org/junit/Test.html.

- Method takes no parameters
- The class holding the tests has a zero-argument constructor
  - Which is implicitly present if we give no constructor at all
- Assertions
  - Methods provided by JUnit for asserting things which should be true
  - Javadoc API `org.junit.Assert`, documentation at junit.org/junit4/javadoc/latest/org/junit/Assert.html
  - Parameter order: description, expected value, then actual value
    - Always give the description!
  - For real-valued tests, an additional tolerance parameter `delta`
  - Pointer equality vs. equals
JUnit Assert JavaDoc

org.junit

Class Assert

java.lang.Object
  org.junit.Assert

public class Assert
  extends Object

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: Assert.assertEquals(...), however, they read better if they are referenced through static import:

import static org.junit.Assert.*;
...  
  assertEquals(...);

JUnit Assert methods

<table>
<thead>
<tr>
<th>static void</th>
<th>assertEquals(String message, double expected, double actual, double delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Asserts that two doubles are equal to within a positive delta.</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(String message, float expected, float actual, float delta)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two floats are equal to within a positive delta.</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(String message, long expected, long actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two longs are equal.</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(String message, Object[] expecteds, Object[] actuals)</td>
</tr>
<tr>
<td></td>
<td>Deprecated, use assertArrayEquals</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(String message, Object expected, Object actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two objects are equal.</td>
</tr>
</tbody>
</table>

JUnit Assert methods

<table>
<thead>
<tr>
<th>static void</th>
<th>assertNull(Object object)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Asserts that an object is null.</td>
</tr>
<tr>
<td>static void</td>
<td>assertNull(String message, Object object)</td>
</tr>
<tr>
<td></td>
<td>Asserts that an object is null.</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(Object expected, Object actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two objects refer to the same object.</td>
</tr>
<tr>
<td>static void</td>
<td>assertEquals(String message, Object expected, Object actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two objects refer to the same object.</td>
</tr>
</tbody>
</table>

JUnit and exceptions
Can also specify tests which we expect to fail
• The Java API says that `ArrayList.get(0)` will throw an `IndexOutOfBoundsException` when the list is empty

• We verify this behavior with:

```java
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

• Documented on Test JavaDoc page junit.org/junit4/javadoc/latest/org/junit/Test.html

**ArrayList.get exceptions**

```
public E get(int index)

Returns the element at the specified position in this list.

**Specified by:**
get in interface List<E>

**Specified by:**
get in class AbstractList<E>

**Parameters:**
index - index of the element to return

**Returns:**
the element at the specified position in this list

**Throws:**
`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())
```

**Test annotation arguments**

From junit.org/junit4/javadoc/latest/org/junit/Test.html,
The Test annotation supports two optional parameters. The first, expected, declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails. For example, the following test succeeds:

```java
@Test(expected=IndexOutOfBoundsException.class) public void outOfBounds() {
    new ArrayList<>().get(1);
}
```

If the exception's message or one of its properties should be verified, the ExpectedException rule can be used. Further information about exception testing can be found at the JUnit Wiki.

The second optional parameter, timeout, causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds). The following test fails:

```java
@Test(timeout=100) public void infinity() {
    while(true);
}
```

Test-driven development
For class project we will adopt the discipline of test-driven development (TDD)

- When coding, repeat the following steps:
  1. Write a (failing) test case
  2. Get it to compile
  3. Get it to pass
  4. Simplify and remove duplication

- Once we have a test passing, we make sure that it keeps on passing

TDD Example

- Working on a financial application
- Need a class to make decisions as to whether certain commodities should be traded
  - A very mathematical question, so we’ll need to support various operations
- Specifically, we need to calculate the first statistical moment about a particular point
  - Don’t have a method for it yet
  - But our stats experts gave us a simple example for this test:

```java
@Test public void testFirstMoment() { final
    InstrumentCalculator calc = new InstrumentCalculator();
    calc.addElement(1.0); calc.addElement(2.0); assertEquals(
        "First moment about 2.0 for {1.0,2.0} within tolerance",
        -0.5, calc.firstMomentAbout(2.0), TOLERANCE); }
```
Making testFirstMoment compile

• This won’t even compile right now
  – We might already have a class InstrumentCalculator, but we haven’t written firstMomentAbout yet!
  – For this example, let’s assume we do have the class, with a method getElements()

• So we add a stub for the method, without trying to implement it
  – We could make it return an absurd value
    public class InstrumentCalculator { // ... keeping what’s already here
      public double firstMomentAbout(final double point) { return Double.NaN; }
    }
  – In our projects, we’ll usually throw an exception
    public double firstMomentAbout(final double point) { throw new UnsupportedOperationException(); }

• Whichever sort of stub we make, the test will now compile
  – But it will not yet pass

Making testFirstMoment pass

• The algorithm for calculating the first moment is standard — we look it up and implement it
  public double firstMomentAbout(final double point) { double numerator = 0.0; for(final double element : getElements()) {
    numerator += element - point; }
  return numerator / elements.size(); }

• Very often, a single test will correspond to much less code, or to only a small part of an algorithm

Write another failing test case

• The code we just added makes one test pass, but it’s not hard to conceive of cases which will fail
  • There’s a division in the algorithm: are we safe against division by zero?
    – And what should happen when we call firstMomentAbout() with an empty data set?
    – Let’s say that the specification calls for an InvalidBasisException

• Write another test for this case!
  @Test(expected = InvalidBasisException.class) public void testEmptyFirstMoment() { new InstrumentCalculator().firstMomentAbout(0.0); fail("Expected InvalidBasisException"); }

  – fail is another JUnit method, like the assert methods, but never succeeding
Making testEmptyFirstMoment compile

What do we need to do to make testEmptyFirstMoment compile?

- If InvalidBasisException is not already part of InstrumentCalculator's package, we must create it
- Otherwise it compiles

Making testEmptyFirstMoment pass

- We’ll need to throw an InvalidBasisException when there are zero elements
- So a revised firstMomentAbout:

```java
public double firstMomentAbout(final double point) { if 
    (getElements().isEmpty()) { throw new InvalidBasisException(); 
    }

double numerator = 0.0; for(final double element :
    getElements()) { numerator += element - point; } return 
    numerator / elements.size(); }
```

- Now the test passes!

Write a failing test case

- Our next task is to write a routine for the second statistical moment about a point.
- So we write a test for this case. Our stats experts again gave us a simple example:

```java
@Test public void testSecondMoment() { final InstrumentCalculator 
    calc = new InstrumentCalculator(); calc.addElement(1.0); 
    calc.addElement(2.0); assertEquals( "Second moment about 2.0 
    for {1.0,2.0} within tolerance", 0.5, 
    calc.secondMomentAbout(2.0), TOLERANCE); }
```

Making testSecondMoment compile

- The problem is again that we do not define the method we are now testing
- And again we make it compile by adding a vacuous definition of the method. We’ll just copy firstMomentAbout and change the name:

```java
public double secondMomentAbout(final double point) { if 
    (getElements().isEmpty()) { throw new InvalidBasisException(); 
    }

double numerator = 0.0; for(final double element :
    getElements()) { numerator += element - point; } return 
    numerator / elements.size(); }
```
Making testSecondMoment pass

- Unsurprisingly, the code for the first moment does not satisfy the second moment’s test!
- But the algorithm for the second moment is very similar, and we only need to make one change: from

  \[
  \text{numerator} += \text{element} - \text{point};
  \]

  to

  \[
  \text{numerator} += \text{Math.pow(element - point, 2.0)};
  \]

- And now it passes!

Remove duplication

- This time around there’s definitely duplication — we have two methods that are almost completely identical!
- In fact, the algorithm for any of the statistical moments has only the same variation that we see here
- So the best way to remove this duplication is with a more general private method `nthMomentAbout` which the others call

```java
public double nthMomentAbout(final double point, final double n) {
    if (getElements().isEmpty()) { throw new InvalidBasisException(); }
    double numerator = 0.0; for(final double element :
            getElements()) { numerator += Math.pow(element - point, n); }
    return numerator / elements.size(); }
```

```java
public double firstMomentAbout(final double point) { return
    nthMomentAbout(point, 1.0); }
```

```java
public double secondMomentAbout(double point) { return
    nthMomentAbout(point, 2.0); }
```

- We already have tests in place, so we can be confident in this change

It’s OK to duplicate!

- This example seems draconian
  - Adding methods that we know are wrong!
  - Copying a method outright!
- But the point of TDD is that we are freed from worrying about more than one thing at a time
  - We might be setting up a test
– Or we might be writing code for a new feature, but *never at the same time as setting up a test*
– Or we might be refactoring away some horrible duplication, but *never at the same time as setting up a test or writing new code*
– *Do one thing at a time, and do it right*

• The tests we build up make later changes and additions much less risky

**How we’ll learn TDD**

• **How does a specification become a series of tests?**
  – Literally, we go sentence by sentence, translating the entire thing into a set of tests

• In the first two projects
  – I’ll give you the tests for each step
  – You’ll submit code making those tests — and only those tests — pass

• Over the semester, you’ll take responsibility for both the test and the primary code

**The various files**

• Three different roles, three different (sets of) classes and files
• All must compile together!

**Interface**

• Assignments usually specify one or more *Java interfaces*
  – Sec. 12.12 of the zyBook, or Ch. 8 of Schildt
• The methods I’ll expect your code to support
  – You should not edit the given interfaces!

**Implementation**

• Your job is to build these
• Just like what you’d write for CS120 projects

**Test classes**

• Makes sure that your interface methods perform as specified
• For early projects, given for each step
  – Run by AutoLab, available for you to run yourself
  – See the *Testing this assignment* paragraphs
• For later projects, you will write the tests yourself
3 Classes, objects, inheritance and notation

Real-life objects
This object is commonplace, and yet complicated

- It possesses some state
  - Including its current location, gear, current speed
- It has some behaviors
  - Like moving, accelerating, braking
- It interacts with other objects
  - Like the road, other cars, trees, people
- It is made of other objects
  - Like the engine, seats, tires, radio

Most of us can use it as a black box

- Don’t need to understand how it works
- Just need to know what we can do with it

Software objects

Definition
A software object is an entity in a program that possesses state (attributes) and behavior (actions).

- May interact with other objects
- May be composed of other objects
- Can be treated as a black box

Definition
Object-oriented programming (OOP) is a programming paradigm that uses the concept of objects to model entities. A program consists of interactions between some number of objects.
Object instances & classes
For our programs:

- We do want to be able to use many objects (which may be similar or different).
- We don’t want to have to rewrite lots of code.

In Java, we can achieve this by:

- Providing a template (blueprints) for all objects of the same type (group or kind) in a class, which specifies
  - attributes (data members), or properties of an object, and
  - behavior (actions) that an object can do
- Creating separate instances (objects) of that class to use in our programs

Class diagrams
Definition
A class diagram describes a class and how it can be used properly.

- Sketch of attributes and behaviors for objects of that type
- No details about how it works

### Class Name

<table>
<thead>
<tr>
<th>attributes(instance variables)</th>
<th>operations(methods)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td></td>
</tr>
<tr>
<td>- String makeModel - int</td>
<td></td>
</tr>
<tr>
<td>mileage</td>
<td></td>
</tr>
<tr>
<td>«constructor» + Car(String,</td>
<td></td>
</tr>
<tr>
<td>int)[0.5em] «update»</td>
<td></td>
</tr>
<tr>
<td>+ void setMakeModel(String)</td>
<td></td>
</tr>
<tr>
<td>+ void setMileage(int)[0.5em]</td>
<td></td>
</tr>
<tr>
<td>«query» + String</td>
<td></td>
</tr>
<tr>
<td>getMakeModel() + int</td>
<td></td>
</tr>
<tr>
<td>getMileage()</td>
<td></td>
</tr>
</tbody>
</table>

Class diagrams - attributes

- The Car class has two simple attributes — used to store information about an individual car.
- Each different Car object (instance) in a program has its own attributes (instance variables).
- In Java, attributes correspond to fields
Class diagrams - private attributes

- Both of the Car class instance variables have private access visibility.
  - Information only accessible from within the class
- In Java, private attributes correspond to private fields
- How can such private variables be read or written?

Class diagrams - instance methods

- **Update** methods (Mutators)
  - Change something about the state of an object
  - Often void methods: do an action but don’t give output
- **Query** methods (Accessors)
  - Tell us something about the state of the object
  - Methods which are not void: give information back

Class diagrams - public methods

- These methods all have public access
- Can be used outside of the class (from other code)
- Allows programmer to control how objects get modified and what information is revealed

What do the minus signs mean?

- Both of the Car class instance variables have private access visibility.

What does the plus sign mean?

- These methods all have public access
Class diagrams - constructors

- Used to create an object
- Has the same name as the Class
- Has no (explicit) return type
- Almost always public (Why?)

The constructor here takes two parameters as input.

Implementing the class

Creating and using objects in a program

- Creation:

  // GENERAL
  final TYPE objectRef = new CONSTRUCTORNAME(params);

  // CONCRETE
  final Car myCivic = new Car("Honda Civic", 214118);

- Ask the object to perform an action:

  // GENERAL
  objectRef.methodName(params);

  // CONCRETE
  myCivic.getMakeModel();
  myCivic.setMileage(myCivic.getMileage() + 1);
Object-oriented design

Now that we have our Car class, we can build more complex classes that use cars.

• Let’s create a CarLot that tracks cars (such as for a used car dealership)

<table>
<thead>
<tr>
<th>CarLot</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Car car1 - Car car2</td>
<td>- String makeModel - int mileage</td>
</tr>
<tr>
<td>«constructor» + CarLot() [0.5em] «query» + void printCars()</td>
<td>«constructor»</td>
</tr>
<tr>
<td></td>
<td>+ Car(String, int) [0.5em] «update»</td>
</tr>
<tr>
<td></td>
<td>+ void setMakeModel(String)</td>
</tr>
<tr>
<td></td>
<td>+ void setMileage(int) [0.5em]</td>
</tr>
<tr>
<td></td>
<td>+ String getMakeModel() + int getMileage()</td>
</tr>
</tbody>
</table>

• The CarLot is an aggregate class, made up of other objects

• Relationships between classes can be signified on UML
  – Diamond arrow is a "used by" relationship

Running a Java program

A CarLot will create Car objects, but what creates the CarLot itself? The solution is to use static methods (and variables).

• Content marked static is independent of any object instance

• Usually associated with the class itself

class CarLot {
  private Car car1;
  private Car car2;

  public static void main(String[] args) {
    CarLot myCarLot = new CarLot();
  }

  public CarLot() {
    car1 = new Car("Honda Civic", 214118);
    ...
  }

The main method

In Java, the main method has special significance

• Provides a point of entry for starting a program
  – Must be public and static

• Any class can have a main method

• Must have proper signature (including String array param)
• In OO paradigm, main typically creates a top-level object and invokes a method which then takes over

```java
public class CarLot {
    public static void main(String[] args) {
        CarLot myCarLot = new CarLot();
        myCarLot.manage();
    }
    public void manage() {
        // Most of program functionality goes here
    }
}
```

Adding more complexity
Suppose the car dealership also sells commercial vehicles (trucks/vans) which have varying carrying capacities (e.g., 1/5/10 tons).

One solution: Create a separate class

```
public class Car {
    String makeModel;
    int mileage;

    Car(String makeModel, int mileage) {
        this.makeModel = makeModel;
        this.mileage = mileage;
    }
    void setMakeModel(String makeModel) {
        this.makeModel = makeModel;
    }
    int getMakeModel() {
        return this.makeModel;
    }
    void setMileage(int mileage) {
        this.mileage = mileage;
    }
    int getMileage() {
        return this.mileage;
    }
}
```

```
public class Truck {
    String makeModel;
    int mileage;
    int capacity;

    Truck(String makeModel, int mileage, int capacity) {
        this.makeModel = makeModel;
        this.mileage = mileage;
        this.capacity = capacity;
    }
    void setMakeModel(String makeModel) {
        this.makeModel = makeModel;
    }
    int getMakeModel() {
        return this.makeModel;
    }
    void setMileage(int mileage) {
        this.mileage = mileage;
    }
    int getMileage() {
        return this.mileage;
    }
    void setCapacity(int capacity) {
        this.capacity = capacity;
    }
    int getCapacity() {
        return this.capacity;
    }
}
```
Finding a better solution
Our first solution has some problems:

• Large amount of duplication
• Harder to write general-purpose code

Certain types of objects have things in common

• Cars/trucks/motorcycles
• Savings/checking/investment accounts

We should adjust our model to exploit these commonalities.

• Done via inheritance in Java

Inheritance
Inheritance is when one class (the subclass or child class) is based on another class (the super class or parent class), which the child class extends or modifies in some way.

• Superclass (or parent) contains similarities
• Subclass (or child) extends the parent
  – Inherits methods and variables from the parent
  – Can add more methods and variables or modify existing ones

Allows us to make our code simpler and more useful!

Inheritance in UML and Java
Inheritance can be represented in UML with arrows from children to parents

• Each child is a more specific kind of parent object
• Called an is-a relationship

class Vehicle {
  // data and methods
}
class Car extends Vehicle {
  // more data and methods
}
class Truck extends Vehicle {
  // more data and methods
}
Class diagrams for inheritance

The `Vehicle` superclass

The # indicates *protected* access

- Like `public`: accessible within any child subclass (and further descendants)
- Like `private`: not accessible from other classes

Variables and methods in `Vehicle` are *inherited* by all descendants

Implementing the `Vehicle` class

Other than the `protected` variables and the change of names, the code is *identical* to the `Car` class.

- Make sure to include comments in your code!

The `Car` subclass

Every method and variable from `Vehicle` is *inherited* by `Car`

- *Except* for the constructor

A child’s constructor is responsible for calling the parent constructor.
• Called via super

    super();

• Must be the first action in the child constructor!
  – Call can be explicit or implicit (no arguments)

• Can also reference variables and methods in parent class

    super.methodName();
    super.variableIdentifier;

(Re-)Implementing the Car class

The Car constructor simply takes its inputs and "passes them up" to the Vehicle constructor via the super call. The Vehicle constructor is responsible for assigning the values to the instance variables.

```java
public class Car extends Vehicle {
    /**
     * Constructor for Car
     * @param makeModel The make and model of the Car
     * @param mileage The miles on the car
     */
    public Car(String makeModel, int mileage) {
        super(makeModel, mileage);
    }
}
```

Implementing the Truck class

- Truck has an extra instance variable
- Constructor runs super() to set common variables, then stores capacity itself
- Has some additional unique methods

Inheritance structures

Objects can be part of an inheritance hierarchy, with multiple levels of ancestors and descendants.

• In Java, everything is descended from the Object class
Object

Vehicle

Car

Truck

Vehicle is a descendant of Object

Vehicle is an ancestor of Car and Truck