CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 2

Instructor: Divya Chaudhary
d.chaudhary@northeastern.edu
Material Credits: Tamara Bonaci

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Review
  - Classes and objects
- Unit testing
- Inheritance and “is-a” relationship
  - Everything is an object
  - Equality (methods `equals`() and `hashCode`())
- Writing simple methods for classes
  - Methods for classes with containment
  - Methods that return objects
- Exceptions
- Documentation and Javadoc
- Visual representation and UML diagrams
- Codestyle

ADMINISTRIVIA

- HW1 due by 11:59pm PT on Monday, February 7th , 2022

- Please make sure to create a release for you HW1
  - How to create a release: https://northeastern.instructure.com/courses/103026/pages/intellij-how-tos

REVIEW

CS 5004, SPRING 2022 – LECTURE 2

OBJECTS AND CLASSES

- Object – an entity consisting of states and behavior
  - States stored in variables/fields
  - Behavior represented through methods

- Class – template/blueprint describing the states and the behavior that an object of that type supports

OBJECT-ORIENTED DESIGN: THE BEGINNING

- Classes – templates/blueprints describing the states and behavior that an object of that type supports

- Question: how do we design a class?

- Identify objects
- Identify properties
- Identify responsibilities

- Rule of thumb:
  - Nouns – objects and properties
  - Verbs – responsibilities (methods)

CLASSES AND VARIABLES IN JAVA

- Classes – templates/blueprints describing the states and behavior that an object of that type supports
- Classes contain:
  - Local variables – variables defined within any method, constructor or block
    - These variables are destroyed when the method has completed
  - Instance variables – variables within a class, but outside any method
    - Can be accessed from inside any method, constructor or blocks of a specific class
  - Class variables – variables declared within a class, outside of any method, with the keyword static

ACCESS-CONTROL MODIFIERS IN JAVA

- In Java, there exist four access levels:
  - Visible to the package (default, no modifier needed)
  - Visible to the class only (modifier **`private`**)
  - Visible to the world (modifier **`public`**)
  - Visible to the package and all subclasses (modifier **`protected`**)

UNIT TESTING

CS 5004, SPRING 2022 – LECTURE 2

UNIT TESTING

- Unit testing - search for errors in a subsystem in isolation
  - A "subsystem" typically means a specific class or object
  - The Java library JUnit helps us to easily perform unit testing
- Basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run
  - Each method looks for specific results, and either passes or fails
- JUnit provides "assert" commands to help us write tests
  - Idea - put assertion calls in your test methods to check things you expect to be true
  - If they are not, the test will fail

JUNIT SETUP AND TEAR DOWN

Methods to run before/after each test case method is called:

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

Methods to run once before/after the entire test class runs:

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

JUNIT TESTING - EXAMPLE

```java
/**
 * Simple class Person, that includes private instance variables firstName and lastName.
 * */
public class Person {

  private Name personsName;
  private String address;

  public Person(Name personsName, String address) {
    this.personsName = personsName;
    this.address = address;
  }

  public void setPersonsName(Name personsName) {
    this.personsName = personsName;
  }

  public void setAddress(String address) {
    this.address = address;
  }

  public Name getPersonsName() { return personsName; }

  public String getAddress() { return address; }
}
```

JUNIT TESTING - EXAMPLE

```java
import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class PersonTest {
  Person testPerson;
  String expectedName = "John Doe";
  String expectedEmail = "john_doe12345@gmail.com";
  String expectedAddres = "N/A";

  @Before
  public void setUp() throws Exception {
    testPerson = new Person( personsName: "John Doe",  address: "john_doe12345@gmail.com", "N/A");

  }

  @Test
  public void getName() {
    TestCase.assertEquals(this.expectedName, testPerson.getName());
```

INHERITANCE AND ”IS A” RELATIONSHIP

CS 5004, SPRING 2022 – LECTURE 2

INHERITANCE AND "IS A" RELATIONSHIP

- Inheritance - set of classes connected by an 'is-a' relationship
- 'Is-a' relationship - hierarchical connection where one category can be treated as a specialized version of another
  - Example 1:
    - Every student is a person
    - Every ALIGN student is a student
  - Example 2:
    - Every pepper is a vegetable
    - Every bell pepper is a pepper
    - Every banana pepper is a pepper

CLASS INHERITANCE

- Many programming languages (Java, C++, C#) provide a direct support for is-a relationship through class inheritance

- Class inheritance - new class extends existing class
  - Original/Extended class (also known as base class or super class)
  - New/Extending class (also known as derived class or subclass)

- Rules for derived classes (subclasses):
  - Derived class automatically inherits all NON-private instance variables and methods of the base class
  - Derived class can add additional methods and instance variables
  - Derived class can provide different versions of inherited methods

- Note: in Java, a class can extend only one class

CLASS INHERITANCE

Derived class automatically inherits all NON-private instance variables and methods of the base class

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

Parent class with protected fields

```
public class Student extends Person {
  private String studentID;

  public Student(String firstName, String lastName, String studentID) {
    super(firstName, lastName);
    this.studentID = studentID;
  }
```

Child class –direct access to the parent's protected fields

CLASS INHERITANCE

Derived class automatically inherits all NON-private instance variables and methods of the base class

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


Parent class with private fields

```
public class Client extends Person {
  private String clientID;

  public Client(String firstName, String lastName, String studentID) {
    super(firstName, lastName);
    this.clientID = studentID;
  }
```


Child class – no direct access to the parent's protected fields

EVERYTHING IS AN OBJECT IN JAVA

- `public class Object` – the root of the class hierarchy
  - Every class has `Object` as a superclass
  - All objects inherit public methods of `Object`

`protected Object clone()`	Creates and returns a copy of this object.
`Boolean equals (Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void finalize()`	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?> getClass()`	Returns the runtime class of this Object.
`int hashCode()`	Returns a hash code value for the object.
`void notify()`	Wakes up a single thread that is waiting on this object's monitor.
`String toString()`	Returns a string representation of the object.

EQUALITY

CS 5004, SPRING 2022– LECTURE 2

EQUALITY OF OBJECTS

How do we compare objects?

CLASS OBJECT – ROOT OF THE CLASS HIERARCHY

Constructor Summary

Constructors

Constructor and Description
`Object()`

Method Summary

Methods

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` Returns a hash code value for the object.
`void`	`notify()` Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()` Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()` Returns a string representation of the object.
`void`	`wait()` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
`void`	`wait(long timeout)` Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.
`void`	`wait(long timeout, int nanos)` Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

EXPECTED PROPERTIES OF EQUALITY

- Reflexive: `a.equals(a) == true`
  - Confusing if an object does not equal itself
- Symmetric: `a.equals(b)` ←→ `b.equals(a)`
  - Confusing if order-of-arguments matters
- Transitive: `a.equals(b) && b.equals(c)` → `a.equals(c)`
  - Confusing again to violate centuries of logical reasoning
    - A relation that is reflexive, transitive, and symmetric is an *equivalence relation*

SPECIFICATION FOR METHOD EQUALS()

- `public boolean equals(Object obj)` - indicates whether some other object is "equal to" this one.
- The equals method implements an equivalence relation:
  - It is reflexive: for any reference value `x`,`x.equals(x)`
  - It is symmetric: for any reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
  - It is transitive: for any reference values `x`,`y`, and `z`,if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is consistent: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return false

METHOD HASHCODE()

- Another method in `Object`- `public int hashCode()`
- Returns a hash code value for the object
- This method is supported for the benefit of hash tables such as those provided by `java.util.HashMap`."
- Contract (again essential for correct overriding):
  - Self-consistent:
- `o.hashCode() == o.hashCode()`
- -..so long as o doesn't change between the calls
  - Consistent with equality:
- `a.equals(b)` → `a.hashCode() == b.hashCode()`

IDEA: THINK OF HASHCODE() AS A PRE-FILTER

- If two objects are equal, they must have the same hash code
  - Up to implementers of methods `hashCode`() and `equals`() to achieve that
  - If you override **`equals`**(), you mist override **`hashCode`**() too

  - If two objects have the same hash code, the still may or may not be equal
    - "Usually not" leads to better performance
    - `hashCode`() in `Object` tries to (but may not) give every object a different hash code

    - Hash codes are usually cheaper to compute, so check first if you "usually expect not equal" – a pre-filter