Object Oriented Java

OO Overview

- All about organization
- Like most things that happen to change programming (and the languages), it came from a need of programmers not computers.
- Prior to OOP (object oriented programming) programmers would typically think of algorithm/procedures to solve problems and then manipulate data to meet the algorithm/procedure.
  - This is typically good for core paradigms (like List, Tree, etc)
  - However, it breaks down for simpler mimics of real life (Bank application)

Procedural Zoo

```java
private static enum Type {
    Zebra, Lion, Tiger
}

private static class Animal {
    private final Type type;
    private Animal(Type type) {
        this.type = type;
    }
}

private static enum Food {
    Grass,
    Meat
}

private static final Animal[] animals = new Animal[] { new Animal(Type.Zebra),
                                                       new Animal(Type.Zebra),
                                                       new Animal(Type.Zebra),
                                                       new Animal(Type.Lion),
                                                       new Animal(Type.Lion),
                                                       new Animal(Type.Tiger),
                                                       new Animal(Type.Tiger) };

private static final Map<Animal, Boolean> hungry = new HashMap<>();
```

```java
public static void main(String[] args) {
    // start all animals as hungry
    for (Animal animal : animals) {
        hungry.put(animal, true);
    }

    Random random = new Random();
    // randomly choose one animal to feed and randomly choose type of food
    Animal animal = animals[random.nextInt(animals.length)];
    Food food = (random.nextInt(2) == 0 ? Food.Grass : Food.Meat);
    feed(animal, food);
}


public static void feed(Animal animal, Food food) {
    switch (animal.type) {
        case Zebra:
            eatNonMeat(animal, food);
            break;
        case Lion:
            eatMeat(animal, food);
            break;
        case Tiger:
            eatMeat(animal, food);
            break;
    }
}

private static void eatNonMeat(Animal animal, Food food) {
    switch (food) {
        case Grass:
            hungry.put(animal, false);
            break;
        case Meat:
            throw new IllegalArgumentException(String.format("Cannot feed a %s meat", animal.type.name()));
    }
}

private static void eatMeat(Animal animal, Food food) {
    switch (food) {
        case Meat:
            hungry.put(animal, false);
            break;
        case Grass:
            throw new IllegalArgumentException(String.format("Cannot feed a %s grass", animal.type.name()));
    }
}
```

Object Oriented Zoo

```java
private static enum Food {
    Grass, Meat
}

private abstract static class Animal {

    protected boolean hungry = true;

    public void eat(Food food) {
        if (canEat(food)) {
            hungry = false;
        } else {
            throw new IllegalArgumentException(String.format("Cannot feed a %s %s",
                        getClass().getSimpleName(), food.name()));
        }
    }

    protected abstract boolean canEat(Food food);
}
private static class Zebra extends Animal {
    @Override protected boolean canEat(Food food) {
        return (food == Food.Grass);
    }
}
private static class Lion extends Animal {
    @Override protected boolean canEat(Food food) {
        return (food == Food.Meat);
    }
}
private static class Tiger extends Animal {
    @Override protected boolean canEat(Food food) {
        return (food == Food.Meat);
    }
}
```

```java
private final Animal[] animals = new Animal[] { new Zebra(), new Zebra(), new Zebra(),
                                                new Lion(), new Lion(),
                                                new Tiger(), new Tiger() };

public static void main(String[] args) {
    Zoo zoo = new Zoo();

    Random random = new Random();
    // randomly choose one animal to feed and randomly choose type of food
    Animal animal = zoo.animals[random.nextInt(zoo.animals.length)];
    Food food = (random.nextInt(2) == 0 ? Food.Grass : Food.Meat);
    animal.eat(food);
}
```

OOP - Class v Object

- Classes are templates for objects.

```java
public class Employee {

    private final String name;

    private final double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

```java
public class Company {

    private final String name;

    private final Employee[] employees;

    public Company(String name, Employee[] employees) {
        this.name = name;
        this.employees = employees;
    }

    public Company addEmployee(String name, double salary) {
        Employee employee = new Employee(name, salary);
        Employee[] updated = new Employee[employees.length + 1];
        System.arraycopy(employees, 0, updated, 0, employees.length);
        updated[employees.length] = employee;
        return new Company(name, updated);
    }
}
```

OOP - Encapsulation

- Hides unnecessary complexity from the outside
  - How many of you know all the parts of a car, of an engine?
  - Yet, most all of you have likely driven a car
  - This is a form of encapsulation
- Principal tenant of OOP is encapsulation.
- Java gives you many tools to properly encapsulate your classes
  - Restricted variables / methods / classes
  - Patterns (getter/setter) to control access

Class v Method names

- Nouns = Class names
- Verbs = Method names

```java
public class Company {

    private final String name;

    private final Employee[] employees;

    public Company(String name, Employee[] employees) {
        this.name = name;
        this.employees = employees;
    }

    public Company addEmployee(String name, double salary) {
        Employee employee = new Employee(name, salary);
        Employee[] updated = new Employee[employees.length + 1];
        System.arraycopy(employees, 0, updated, 0, employees.length);
        updated[employees.length] = employee;
        return new Company(name, updated);
    }
}
```

Class Relationship

- Dependence - “uses-a”

```
public class Company {
    private final Employee[] employees;
...
```

- Aggregation - “has-a”

```
public void addEmployee(String name, double salary) {
    Employee employee = new Employee(name, Math.round(salary));
    ...
}
```

- Inheritance - “is-a”

```
public class Company extends Organization {
...
```

Predefined Classes

- Use whenever possible…
- …except Calendar (or at least know the constraints of the API)
- Java 8 - new API LocalDate/LocalTime
  - Created via JSR-310
  - Informed from Joda Date/DateTime
  - If using Java 7 or lower, use Joda Date/DateTime

Immutable Objects

- Marking Class variables as final makes them immutable
- Prefer this when at all possible. Makes reasoning about object state easier and (as we'll see later) makes reasoning about concurrency much easier.
- Careful about immutable references to mutable objects (more about this later)

```java
public class Employee {

    private final String name;

    private final double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

Class Structure

1. Class signature
2. Static variables
3. Static methods
4. Instance variables
5. Instance constructors
6. Instance methods

Best Practices

- Minimize static methods
  - Hard to test / not overridable
- Prefer immutable instance variables
- Prefer fully encapsulated objects
  - private instance variables with getter methods

```java
// 1) class signature
public class Employee {

    // 2) static variables
    private static final double DEFAULT_SALARY = 50000d;

    // 3) static methods
    public static Employee construct(String name, double salary) {
        return new Employee(name, salary);
    }

    // 4) instance variables
    private final String name;

    private final double salary;

    // 5) constructors
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // 6) instance methods
    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }
}
```

Constructors

- Method invoked when instantiating the object (i.e., via the new keyword)
- If not specified there's a default no-args constructor
    - If you define one constructor the default no-args constructor is not created automatically for you. You can define yourself though
- Can have many constructors (as long as their signature is unique)
- Constructors can call other constructors

```java
private static final String DEFAULT_NAME = "Bob";
private static final double DEFAULT_SALARY = 0d;

// since we have other constructors, need to define the no-args constructor
public Employee() {
    // example of calling another constructor
    this(DEFAULT_NAME);
}

public Employee(String name) {
    // another example of calling another constructor
    this(name, DEFAULT_SALARY);
}

public Employee(String name, double salary) {
    this.name = name;
    this.salary = salary;
}
```

Naming conventions (cont)

- Contrary to **Core Java** recommendation, use the same name for variable assignment in constructors- the `this` and shadowing approach (TaSA)
  - Justification being that having multiple names referring to the same thing can be confusing- you're changing the canonical name simply because of a language construct.
  - Succinct and descriptive naming is important but hard to do right. Having to do this twice often just leads to not doing it right at all
    - The `aParameterName` construct mentioned in the textbook does not always work or becomes confusing; i.e., nouns which are not tangible- e.g., `aRate`
    - However, every parameter can be handled by the TaSA.
  - Biggest negative to the TaSA is solved with decent IDEs, however, you are (currently) not using an IDE. The negative is that the variable is shadowed.
    - However, shadowing can be caught by the compiler if your instance variables are `final` (score another win for immutability!)

Naming Convention (cont)

BUG NOT CAUGHT BY COMPILER ->
The assignment to name is shadowed

```java
private String name;

public Employee(String name, double salary) {
    name = name;
    this.salary = salary;
}
```

```java
private final String name;

public Employee(String name, double salary) {
    name = name;
    this.salary = salary;
}
```

<- BUG CAUGHT BY COMPILER
The assignment to name is shadowed but because the instance variable name is final and not assigned to the compiler complains

Interlude - Packages

- Group similar classes by a namespace
- Commonly reverse domain notation (i.e., “com.google.xxxxx”)
- No package means the default package - NEVER do this.
- Classes should be organized in packages. Think of them like folder structures.
- In fact, classes should be placed in nested directories matching their package structure. The period in the package denotes a new directory.

```java
package edu.nyu.cs9053;

/**
 * User: blangel
 * Date: 8/17/14
 * Time: 5:53 PM
 */
public class WithinPackage {

}
```

```
edu.nyu.cs9053
    WithinPackage
```

```
blangel@lenoir$ pwd
/Users/blangel/projects/NYU-CS9053/Template/src/main/java/edu/nyu/cs9053
blangel@lenoir$ ls
WithinPackage.java
blangel@lenoir$
```

Encapsulation Constructs

- Access Privileges
    - `public`
    - *no modifier* (referred to as “default” or “package-private”)
    - `protected`
    - `private`
- Available for placement on
    - Class (except protected / private unless nested [see chapter 5])
    - Field
    - Method

Class - public v (package)

- The public keyword means the class is accessible to everyone everywhere
- The (default or package-private) means the class is only accessible from other classes within the same package.

```java
package edu.nyu.cs9053;

/**
 * User: blangel
 * Date: 8/17/14
 * Time: 5:57 PM
 */
public class AccessibleEverywhere {
}
```

```java
package edu.nyu.cs9053;

/**
 * User: blangel
 * Date: 8/17/14
 * Time: 5:57 PM
 */
class AccessibleWithinPackage {
}
```

Field - `public` v (default) v `protected` v `private`

- The public keyword (like Class) means accessible to everyone everywhere
- The protected keyword means accessible to the Class itself, everyone within the same package and any subclass
- The (default or package-private) means accessible to the Class itself and everyone within the same package
- The private keyword means accessible only to the Class itself (not subclasses)

```java
// everyone (do not do)
public final String everyone;

// this class, this package and subclasses
protected final String almostEveryone;

// this class and this package
final String thisAndPackage;

// only this class
private final String restricted;
```

Method - `public` v (default) v `protected` v `private`

- The public keyword means accessible to everyone everywhere
- The protected keyword means accessible to the Class itself, everyone within the same package and any subclass
- The (default or package-private) means accessible to the Class itself and everyone within the same package
- The private keyword means accessible only to the Class itself (not subclasses)

```java
// everyone (do not do)
public void everyone() { }

// this class, this package and subclasses
protected void almostEveryone() { }

// this class and this package
void thisAndPackage() { }

// only this class
private void restricted() { }
```

OO & Procedural Coexisting

- Java allows both procedural (as we saw last lecture) and OO to coexist.
- Definitely skewed towards OO but there are language constructs to allow for methods to be created agnostic of an Object

- To mark a field or method as Class level (instead of instance, or Object, level) use the keyword `static`
  - Have already seen this with the Math class
- Almost always you'll be making objects and instance fields/methods (probably 90%)

```java
public class Zebra {

    private static final String SCIENTIFIC_NAME = "Equus quagga";

    public static String getScientificName() {
        return SCIENTIFIC_NAME;
    }

    private final String name;

    public Zebra(String name) {
        this.name = name;
    }

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

Interlude - Deviation from Textbook

- Do NOT use main method for testing
    - Clutters your actual application code.
    - Not scalable- some classes may have 20 testable methods, so you'll double the size of your class by having 20 additional test methods (invoked from main)
    - Test code should not end up inside your application
- Instead, use a testing framework like `JUnit`
    - Treat the test code as a separate unit/application which depends upon your application (more about this later).
        - Until then, at least isolate your testing to a separate Class

```java
public class ZebraTest {

    @Test public void getName() {
        String name = "foobar";
        Zebra zebra = new Zebra(name);
        assertEquals(name, zebra.getName());
    }

}
```

Method Invocation - CBV or CBR

- CBV = call by value
  - parameters to method are copied to new value
  - method cannot change reference (but can change values associated with the reference!)
- CBR = call by reference
  - parameters are sent by reference
  - allows method to change callee’s reference
- Java is CBV
  - Be extremely careful though, as even those it’s CBV the underlying reference’s data can be changed.

```java
public class MethodInvocationExample {

    public static void main(String[] args) {

        MethodInvocationExample cbv = new MethodInvocationExample();
        int left = 1;
        int right = 2;
        cbv.invoke(left, right);
        // Call By Value
        // left == 1
        // right == 2

        MethodInvocationExample cbr = new MethodInvocationExample();
        cbr.invoke(left, right);
        // Call By Reference
        // left == 2
        // right == 2
    }

    public void invoke(int left, int right) {
        left = right;
        // if right == 2, left == 2
    }

}
```

```java
public class MethodInvocationExample {

    public static void main(String[] args) {

        MethodInvocationExample cbv = new MethodInvocationExample();
        Date date = new Date();
        date.setTime(0L);
        cbv.invoke(date);
        // Call By Value - reference changed
        // what does date.getTime() return? 0, 1 or 2?
    }

    public void invoke(Date date) {
        date.setTime(1L);
        date = new Date(2L);
    }

}
```

Method Overloading

- Methods can have the same name provided their signature is different
  - Method signature is composed of four things
    - Return type
    - Name
    - Number of parameters
    - Parameter types
  - For overloading, the name can be the same provided the number of parameters and/or the parameter types are different
    - The return type must be the same (for you but not for the compiler, more about this later, called covariant return types)
- Same rules apply for constructor overloading

```java
public class MethodOverloadingExample {

    public String compute(String foo, int bar) {
        return null;
    }

    public String compute(String foo, double bar) {
        return null;
    }

    public String compute() {
        return null;
    }

    public String compute(int foo, double bar) {
        return null;
    }

    // and so on...

}
```

Initialization!

- Refers to how class, instance and local field values are initially set.
- Besides constructors and explicit assignment there is another language construct which can be used to initialize class and instance field values (but not local) - initialization blocks!
  - Initialization blocks are blocks of code (code surrounded by braces) which are run once.
    - Class initialization blocks are run once at initial class load
    - Object initialization blocks are run once just prior to the constructor methods being called

Initialization! (cont)

```java
public class InitBlocks {

    private static final String foo;

    // Class initialization block!
    static {
        foo = "foo";
    }

    private final String bar;

    // Instance initialization block!
    {
        bar = "bar";
    }

}
```

Instance Field Initialization

- There are four possible ways to initialize an instance field value
  - Via explicit field initialization
  - Via initialization blocks
  - Via constructor
  - Via default initialization
    - If none of the proceeding initializations happen the a default value is assigı

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Class Field Initialization

- There are three possible ways to initialize a class field value
  - Via explicit field initialization
  - Via initialization blocks
  - Via default initialization
    - If none of the proceeding initializations happen the a default value is assigned

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Local Variable Initialization

- Only one way to initialize a local variable (i.e., a variable local to a method)
    - Via explicit initialization
        - Explicit initialization can be delayed
- There is no default value assigned; if you do not initialize a local variable the compiler will complain

Imports

- Shortcut way of referencing other classes from outside your package
    - Not necessary to import classes without your same package
- Never necessary to import classes within `java.lang`
- Can always fully reference a class
    - e.g.; `java.util.List`

Classpath

- Needed for compiling and invocation -> directly related to the imports
    - Reference via `-cp` or `-classpath` flag
- Best way to learn is via usage - practice!
- No need to add packages to the classpath starting with `java.xxxx`
- Classloading is the act of resolving Class objects at runtime. Not in scope for this class

Javadoc

- Behind source-code, your best friend in terms of learning how others' code works.
  - Google search online to find.
- Writing your own good java-documentation takes time, practice and lots of reading of others' - just like writing good java code.
  - General Rules
    - Always Javadoc your Classes and all of your public methods.
    - Always Javadoc any method (even private) if it is sufficiently complicated
    - Be terse, descriptive and informative
      - Do not do -> `@param bank a bank`

Read Chapter 5

All sections except 5.3 & 5.7 will be covered in next lecture

- You can skip sections 5.3 & 5.7

Homework 3

https://github.com/NYU-CS9053/Fall-2019-II/homework/week3