

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

```
1  private static enum Type {
2      Zebra, Lion, Tiger
3  }
4
5  private static class Animal {
6      private final Type type;
7      private Animal(Type type) {
8          this.type = type;
9      }
10 }
11
12 private static enum Food {
13     Grass,
14     Meat
15 }
16
17 private static final Animal[] animals = new Animal[] { new Animal(Type.Zebra),
18                                                         new Animal(Type.Zebra),
19                                                         new Animal(Type.Zebra),
20                                                         new Animal(Type.Lion),
21                                                         new Animal(Type.Lion),
22                                                         new Animal(Type.Tiger),
23                                                         new Animal(Type.Tiger) };
24
25 private static final Map<Animal, Boolean> hungry = new HashMap<>();
26
```

```

29 public static void main(String[] args) {
30     // start all animals as hungry
31     for (Animal animal : animals) {
32         hungry.put(animal, true);
33     }
34
35     Random random = new Random();
36     // randomly choose one animal to feed and randomly choose type of food
37     Animal animal = animals[random.nextInt(animals.length)];
38     Food food = (random.nextInt(2) == 0 ? Food.Grass : Food.Meat);
39     feed(animal, food);
40 }
41
42
43 public static void feed(Animal animal, Food food) {
44     switch (animal.type) {
45         case Zebra:
46             eatNonMeat(animal, food);
47             break;
48         case Lion:
49             eatMeat(animal, food);
50             break;
51         case Tiger:
52             eatMeat(animal, food);
53             break;
54     }
55 }
56
57 private static void eatNonMeat(Animal animal, Food food) {
58     switch (food) {
59         case Grass:
60             hungry.put(animal, false);
61             break;
62         case Meat:
63             throw new IllegalArgumentException(String.format("Cannot feed a %s meat", animal.type.name()));
64     }
65 }
66
67 private static void eatMeat(Animal animal, Food food) {
68     switch (food) {
69         case Meat:
70             hungry.put(animal, false);
71             break;
72         case Grass:
73             throw new IllegalArgumentException(String.format("Cannot feed a %s grass", animal.type.name()));
74     }
75 }

```

Object Oriented Zoo

```
1 private static enum Food {
2     Grass, Meat
3 }
4
5 private abstract static class Animal {
6
7     protected boolean hungry = true;
8
9     public void eat(Food food) {
10         if (canEat(food)) {
11             hungry = false;
12         } else {
13             throw new IllegalArgumentException(String.format("Cannot feed a %s %s",
14                 getClass().getSimpleName(), food.name()));
15         }
16     }
17
18     protected abstract boolean canEat(Food food);
19 }
20 private static class Zebra extends Animal {
21     @Override protected boolean canEat(Food food) {
22         return (food == Food.Grass);
23     }
24 }
25 private static class Lion extends Animal {
26     @Override protected boolean canEat(Food food) {
27         return (food == Food.Meat);
28     }
29 }
30 private static class Tiger extends Animal {
31     @Override protected boolean canEat(Food food) {
32         return (food == Food.Meat);
33     }
34 }
```

```
36 private final Animal[] animals = new Animal[] { new Zebra(), new Zebra(), new Zebra(),
37                                                    new Lion(), new Lion(),
38                                                    new Tiger(), new Tiger() };
39
40 public static void main(String[] args) {
41     Zoo zoo = new Zoo();
42
43     Random random = new Random();
44     // randomly choose one animal to feed and randomly choose type of food
45     Animal animal = zoo.animals[random.nextInt(zoo.animals.length)];
46     Food food = (random.nextInt(2) == 0 ? Food.Grass : Food.Meat);
47     animal.eat(food);
48 }
```

OOP - Class v Object

- Classes are templates for objects.

```
1 public class Employee {  
2  
3     private final String name;  
4  
5     private final double salary;  
6  
7     public Employee(String name, double salary) {  
8         this.name = name;  
9         this.salary = salary;  
10    }  
11 }
```

```
1 public class Company {  
2  
3     private final String name;  
4  
5     private final List<Employee> employees;  
6  
7     public Company(String name, Employee ... employees) {  
8         this.name = name;  
9         this.employees = Arrays.asList(employees);  
10    }  
11  
12    public void addEmployee(String name, double salary) {  
13        Employee employee = new Employee(name, salary);  
14        this.employees.add(employee);  
15    }  
16 }
```

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

```
1  public class Company {  
2  
3      private final String name;  
4  
5      private final List<Employee> employees;  
6  
7      public Company(String name, Employee ... employees) {  
8          this.name = name;  
9          this.employees = Arrays.asList(employees);  
10     }  
11  
12     public void addEmployee(String name, double salary) {  
13         Employee employee = new Employee(name, salary);  
14         this.employees.add(employee);  
15     }  
16 }
```

Class Relationship

- Dependence - “uses-a”

```
1 public class Company {  
2     private final List<Employee> employees;  
3     ...  
}
```

- Aggregation - “has-a”

```
1 public void addEmployee(String name, double salary) {  
2     Employee employee = new Employee(name, Math.round(salary));  
3     this.employees.add(employee);  
4 }
```

- Inheritance - “is-a”

```
1 public class Company extends Organization {  
2     ...  
}
```

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)

```
1  public class Employee {  
2  
3      private final String name;  
4  
5      private final double salary;  
6  
7      public Employee(String name, double salary) {  
8          this.name = name;  
9          this.salary = salary;  
10     }  
11 }
```

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

```
1 // 1) class signature
2 public class Employee {
3
4     // 2) static variables
5     private static final double DEFAULT_SALARY = 50000d;
6
7     // 3) static methods
8     public static Employee construct(String name, double salary) {
9         return new Employee(name, salary);
10    }
11
12    // 4) instance variables
13    private final String name;
14
15    private final double salary;
16
17    // 5) constructors
18    public Employee(String name, double salary) {
19        this.name = name;
20        this.salary = salary;
21    }
22
23    // 6) instance methods
24    public String getName() {
25        return name;
26    }
27
28    public double getSalary() {
29        return salary;
30    }
31 }
```

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

```
1  // since we have other constructors, need to define the no-args constructor
2  public Employee() {
3      // example of calling another constructor
4      this(null, 0d);
5  }
6
7  public Employee(String name) {
8      // another example of calling another constructor
9      this(name, 0d);
10 }
11
12 public Employee(String name, double salary) {
13     this.name = name;
14     this.salary = salary;
15 }
```

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

```
private String name;
```

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

```
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

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

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.

```
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$
```

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.

```
1 package edu.nyu.cs9053;  
2  
3 /**  
4  * User: blangel  
5  * Date: 8/17/14  
6  * Time: 5:57 PM  
7  */  
8 public class AccessibleEverywhere {  
9 }
```

```
1 package edu.nyu.cs9053;  
2  
3 /**  
4  * User: blangel  
5  * Date: 8/17/14  
6  * Time: 5:57 PM  
7  */  
8 class AccessibleWithinPackage {  
9 }
```

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)

```
1  // everyone (do not do)
2  public final String everyone;
3
4  // this class, this package and subclasses
5  protected final String almostEveryone;
6
7  // this class and this package
8  final String thisAndPackage;
9
10 // only this class
11 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)

```
1  // everyone (do not do)
2  public void everyone() { }
3
4  // this class, this package and subclasses
5  protected void almostEveryone() { }
6
7  // this class and this package
8  void thisAndPackage() { }
9
10 // only this class
11 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%)

```
1 public class Zebra {  
2  
3     private static final String SCIENTIFIC_NAME = "Equus quagga";  
4  
5     public static String getScientificName() {  
6         return SCIENTIFIC_NAME;  
7     }  
8  
9     private final String name;  
10  
11     public Zebra(String name) {  
12         this.name = name;  
13     }  
14  
15     public String getName() {  
16         return name;  
17     }  
18 }
```

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

```
1  public class ZebraTest {
2
3      @Test public void getName() {
4          String name = "foobar";
5          Zebra zebra = new Zebra(name);
6          assertEquals(name, zebra.getName());
7      }
8
9  }
```

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.


```
1 public class MethodInvocationExample {
2
3     public static void main(String[] args) {
4
5         MethodInvocationExample cbv = new MethodInvocationExample();
6         int left = 1;
7         int right = 2;
8         cbv.invoke(left, right);
9         // Call By Value
10        // left == 1
11        // right == 2
12
13        MethodInvocationExample cbr = new MethodInvocationExample();
14        cbr.invoke(left, right);
15        // Call By Reference
16        // left == 2
17        // right == 2
18    }
19
20    public void invoke(int left, int right) {
21        left = right;
22        // if right == 2, left == 2
23    }
24
25 }
```

```
1 public class MethodInvocationExample {
2
3     public static void main(String[] args) {
4
5         MethodInvocationExample cbv = new MethodInvocationExample();
6         Date date = new Date();
7         date.setTime(0L);
8         cbv.invoke(date);
9         // Call By Value - reference changed
10        // what does date.getTime() return? 0, 1 or 2?
11    }
12
13    public void invoke(Date date) {
14        date.setTime(1L);
15        date = new Date(2L);
16    }
17
18 }
```

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

```
1 public class MethodOverloadingExample {
2
3     public String compute(String foo, int bar) {
4         return null;
5     }
6
7     public String compute(String foo, double bar) {
8         return null;
9     }
10
11    public String compute() {
12        return null;
13    }
14
15    public String compute(int foo, double bar) {
16        return null;
17    }
18
19    // and so on...
20
21 }
```

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)

```
1  public class InitBlocks {  
2  
3      private static final String foo;  
4  
5      // Class initialization block!  
6      static {  
7          foo = "foo";  
8      }  
9  
10     private final String bar;  
11  
12     // Instance initialization block!  
13     {  
14         bar = "bar";  
15     }  
16  
17 }
```

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 assign

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!
- Do not need to import packages 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/Spring-2017/homework/week3>