



**LOYOLA
UNIVERSITY
CHICAGO**

COMP 413: Intermediate Object-Oriented Programming

**Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science**

Week 6 Topics

- **Test 1 Discussion (note: 15.5 points)**
- Java Supplemental Topics – if time
- Project 3 UML 30-minute in-class Exercise
- Agile Development – if time
- Android Application Development – if time
 - Running some simple apps
 - Android framework & activity life cycle
 - More complex apps

Test 1 – Requirements

- Types of requirements
 - Functional vs. nonfunctional
 - **Functional** – what a program does
 - **Nonfunctional** (“-ilities”) – how a program behaves or performs, or its characteristics
 - Static (evolution) vs. dynamic (execution)
 - **Static** – does not require execution, can be evaluated / tested through reading, ...
 - **Dynamic** – observable or testable primarily at runtime

Test 1 – Requirements



- Functional requirements examples – dynamic (runtime)
 - The system will display a login screen when the key combination Ctrl-Alt-Del is pressed simultaneously
 - The program will combine an initial balance with a yearly earned interest amount to calculate the number of years required to reach a target balance

Test 1 – Requirements

- Nonfunctional requirements examples – both static and dynamic
 - Performance
 - Capacity
 - Reliability
 - Maintainability
 - Testability
 - Security
 - ...

Which is which?

Test 1 Problem 1a

- **1.5** For *each* of the following requirements, indicate whether it is 1) a functional, 2) a static (evolution) nonfunctional, or 3) a dynamic (execution) nonfunctional requirement.
 - The program can be changed easily to enhance its functionality:
static nonfunctional (2)
 - The program runs in linear time  in the number of input lines:
dynamic nonfunctional (3)
 - The program counts the number of its input lines:
functional (1)
 - The program can be tested easily:
static nonfunctional (2)
 - The program crashes in fewer than 5% of all invocations:
dynamic nonfunctional (3) 

Test 1 Problem 1b

- **0.5** Why can testability be considered the most important nonfunctional requirement?

Circle exactly one.

- Testability enables us to examine whether a program satisfies its functional requirements.
- Testability guarantees performance and reliability.
- Testability usually results in higher program portability.
- Testability requires pair programming.
- All of the above.

Test 1 Problem 1c

- **0.5** How are performance and maintainability typically related? *Circle exactly one.*
 - The more maintainable a program, the faster it will run.
 - The faster a program runs, the more maintainable it will be.
 - It may be necessary to sacrifice some maintainability in favor of performance.
 - Maintainability requires pair programming, while performance does not.
 - All of the above.

Test 1 – Primitives vs. Objects

- Value vs. reference semantics (primitive variables vs. object references)
 - Variables that refer to primitive values (ints, doubles, booleans, etc) have their contents copied when they are assigned to other variables; their values are completely independent after such a copy/assignment
 - Variables that refer to objects also have their contents copied when they are assigned to other variables, but then multiple variables may refer to the same object and alter it

Test 1 – Identity vs. Equality

- In Java, `==` is the identity operator
 - For primitive variables, it is true if their values or contents are equal, eg, if `x` and `y` contain 3, `x == y`
 - For reference variables, it is true only if they refer to the same object (so, an object is never `== null`)
- The `Object` class provides an inherited definition for the `equals()` (equality) method that matches the identity operator; however, the `equals()` method can be overridden
 - `Object`'s `equals()` says `x.equals(y)` iff `x == y`
 - We often override `equals()` to make equality be based on values of class instance variables

Test 1 Problem 2a

Remember that boolean values print as “true” or “false”.

- **0.5** What is the output of the following Java code? Briefly justify your answer. **(0.25 points each)**

```
float x = 1.3;  
float y = x;  
System.out.println(x == y); // answer here →  
// true (float values are the same – value identity)  
y = 2.8;  
System.out.println(x == y); // answer here →  
// false (independent float values, now different)
```

Test 1 Problem 2b

- Now consider the following Java class. Recall that *x instanceof C* tells you whether the *dynamic* type of *x* is *C* or a subtype of *C*.

```
class Coin {  
    private float value; // private instance variables  
    private float weight;  
    public void setValue(final float value) { this.value = value; } // “setter”  
    public float getValue() { return value; } // standard “getter” method  
    public float getWeight() { return weight; }  
    public Coin(final float value, final float weight) { // constructor  
        this.value = value;  
        this.weight = weight;  
    }  
    public boolean equals(final Object that) {  
        return (that instanceof Coin) && (this.getValue() ==  
            ((Coin) that).getValue());  
    }  
}
```

0.5 **TRUE** or FALSE (circle one)? The **equals** method does not care about the weight of the coin because coins are compared based on their face value



LOYOLA
UNIVERSITY
CHICAGO

Test 1 – Overriding Object equals()

- This is a typical overridden version of Object's equals:

```
@Override public boolean equals(final Object that) {  
    return (that instanceof Coin) &&  
        (this.getValue() == ((Coin) that).getValue());  
} // equality based on the values of the Coins
```

○ An equals method must obey these rules:

- Reflexive: if x is a non-null object, x.equals(x) is true (and x.equals(null) must be false)
- Symmetric: x.equals(y) == y.equals(x)
- Transitive: if x.equals(y) and y.equals(z) then x.equals(z) must be true (x.equals(y) && y.equals(z) == x.equals(z))
- Consistent: x.equals(y) must stay the same if the values that equals() uses to determine equality remain the same (in this case, the values of the Coins)

Test 1 Problem 2c

- 1 Given the above Coin class, what is the output of the following code?

```
final Coin n1 = new Coin(0.05, 5.0); // value, weight
final Coin n2 = new Coin(0.05, 5.1);
System.out.println(n1.equals(n2) + " " + (n1 == n2));
// answer here → what is the == operator comparing?
// true false (same value, different objects)
n2.setValue(0.10);
System.out.println(n1.equals(n2) + " " + (n1 == n2));
// answer here → what is the equals() method comparing?
// false false (different values and objects)
```

Test 1 Problem 2d

- 1 Given the above Coin class, what is the output of the following code?

```
final Coin n1 = new Coin(0.05, 5.0); // value, weight
final Coin n2 = n1;
System.out.println(n1.equals(n2) + " " + (n1 == n2));
// answer here → what is the == operator comparing?
// true true (same object)
n2.setValue(0.10);
System.out.println(n1.equals(n2) + " " + (n1 == n2));
// answer here → what is the equals() method comparing?
// true true (same object, updated)
```

Test 1 – Problem 2e

- Recall Coin's overridden equals() method:

```
@Override public boolean equals(final Object that) {  
    return (that instanceof Coin) &&  
        (this.getValue() == ((Coin) that).getValue());  
} // equality based on the values of the Coins
```

- 0.05** Given the above Coin class, what is the output of the following code?

```
final Coin n1 = new Coin(0.05, 5.0); // value, weight  
System.out.println(n1.equals("0.05"));  
// answer here →  
// false (object is String, not Coin)  
System.out.println(n1.equals(Double.valueOf(0.05)));  
// answer here →  
// false (object is Double (due to autoboxing), not Coin)
```


Test 1 – Other Object Methods

- Object provides 2 other methods (among many) often overridden by other classes:
 - hashCode(): if equals() is overridden,
x.equals(y) → x.hashCode() == y.hashCode()
 - hashCode() returns an int used in Maps, etc.
 - Overriding hashCode() is not needed for Test 1
 - toString(): toString() should be overridden to provide information about an object's state
 - toString() is invoked automatically when an object is printed or concatenated to a String

Test 1 – Interfaces

- A Java interface consists of 0 or more static variable declarations and/or abstract methods
 - An interface can extend one or more other interfaces; if it does then it inherits all of their methods
 - A concrete or abstract class can implement any number of interfaces, in addition to extending only one other class
- A Java interface provides a “contract” that all implementing classes must honor
 - The contract consists of all the abstract method declarations / signatures that the interface provides
 - Any concrete implementing class must fully define all of these methods, or else it must be declared abstract (note that this changes in Java 8)

Test 1 – Interface Examples

- A “marker” interface (no variables or methods)

```
public interface Cloneable { } // public is optional
```

- A typical interface

```
public interface Animal {  
    void speak(); String name(); // public is optional  
} // abstract is implicit and not required
```

- A generic interface

```
public interface Iterable<T> {  
    public Iterator<T> iterator();  
}
```

- A non-generic interface with a generic method

```
public interface Shape {  
    <Result> Result accept(Visitor<Result> v);  
}
```

Test 1 – Test-Driven Development

- In Test-Driven Development (TDD) we write tests before we write code, then write code to make the current tests pass
 - Writing tests first helps us to understand the requirements and know when to stop coding
- In Java the JUnit test system allows us to write tests and automatically execute them, and generates a pass/fail result log
 - JUnit provides methods like assertEquals()

Test 1 Problem 3

- Following a test-driven development (TDD) approach, consider first this JUnit test fragment:

```
final IntPair p = new DefaultIntPair(3, 5);  
final IntPair q = new DefaultIntPair(5, 3);
```

```
assertEquals(3, p.first());  
assertEquals(5, p.second());  
assertEquals("<3, 5>", p.toString());  
assertTrue(p.equals(p));  
assertFalse(p.equals(q));  
assertFalse(q.equals(p));  
assertFalse(p.equals(null));  
assertTrue(p.equals(q.reverse()));
```

```
// note: either DefaultIntPair extends class IntPair, or else  
// IntPair is an interface that DefaultIntPair implements
```

What do these test methods do?

Test 1 Problem 3

- Given this JUnit test fragment:

```
final IntPair p = new DefaultIntPair(3, 5);
final IntPair q = new DefaultIntPair(5, 3);

assertEquals(3, p.first());
assertEquals(5, p.second());
assertEquals("<3, 5>", p.toString());
assertTrue(p.equals(p));
assertFalse(p.equals(q));
assertFalse(q.equals(p));
assertFalse(p.equals(null));
assertTrue(p.equals(q.reverse()));
```

- Since *p* and *q* are *IntPairs*, what methods must *IntPair* define?

The JUnit tests require *IntPair* methods `first()`, `second()`, and `reverse()`

- What *Object* methods must *DefaultIntPair* override?

The JUnit tests require *Object* methods `toString()` and `equals(Object obj)`

The return type of `reverse()` must be ***IntPair*** in order for the last test to pass!

Test 1 Problem 3a

- **1.5** Your first job is to **reverse-engineer the IntPair interface** in a way that is consistent with the test shown above. Leave out any methods already provided by the class java.lang.Object.

Write the interface here:

```
public interface IntPair { // public not required
    int first(); // public abstract not required
    int second();
    IntPair reverse();
} // Object toString() and equals() methods must not appear
```

Test 1 Problem 3b, Part 1

- **3** Your next job is to **write the DefaultIntPair class** that implements the IntPair interface in a way that the test shown above passes. *I.e., implement all pair methods used in the test.* Keep in mind that pair instances, once created, are *immutable*.

```
public class DefaultIntPair implements IntPair {
    private int first, second;
    public DefaultIntPair(int first, int second) {
        this.first = first; this.second = second;
    } // note: interfaces cannot specify constructors
    // methods required by the IntPair interface
    public int first() { return first; }
    public int second() { return second; }
    public DefaultIntPair reverse() {
        return new DefaultIntPair(second, first);
    } // note: reverse can return any subtype of IntPair
    // continued on next slide
```


Test 1 Problem 3b, Part 2

- **3** Your next job is to **write the DefaultIntPair class** that implements the IntPair interface in a way that the test shown above passes. *I.e., implement all pair methods used in the test.*

```
// Overridden Object methods required by @Test method calls
@Override // optional
public String toString() {
    return "<" + first + ", " + second + ">";
}
@Override // optional
public boolean equals(final Object that) { // final optional
    return (that instanceof DefaultIntPair) &&
        (this.first() == ((DefaultIntPair) that).first()) &&
        (this.second() == ((DefaultIntPair) that).second());
} // typical equals method - uses both instance variables
}
}
```

<#>



LOYOLA
UNIVERSITY
CHICAGO

Test 1 – Generic Classes

- In Java a generic class is one that takes one or more “type parameters”, typically single capital letters, to be replaced with real type names when instantiating the class; a simple example:

```
public class Generic<T> { // T is the type variable or parameter
    private T inst; // instance variable
    public Generic(T value) { setInst(value); } // constructor
    public void setInst(T value) { inst = value; } // setter
    public T getInst() { return inst; } // getter
    @Override // override Object's toString()
    public String toString() { return "inst var: " + inst; }
}
// usage: Generic<Double> gd = new Generic<Double>(3.14159);
```

Test 1 Problem 3c

- 2 Now generalize the IntPair interface by making it *generic in the item type of the pair*, allowing us to store two items not only of type int but of *arbitrary reference type as long as both items are of that same type*. Show the code (or explain what needs to be changed) for the resulting generic interface **Pair**.

```
public interface Pair<T> { // note: not Pair<T, U>; a single type
    T first(); // generic return types
    T second();
    Pair<T> reverse();
}
```

Test 1 Problem 3c Extended, Part 1

- **N/A** Your next job is to **write a generic DefaultPair class** that implements the Pair interface.

```
public class DefaultPair<T> implements Pair<T> {  
    private T first, second;  
    public DefaultPair(T first, T second) {  
        this.first = first; this.second = second;  
    } // note: interfaces cannot specify constructors  
    // methods required by the Pair interface  
    public T first() { return first; }  
    public T second() { return second; }  
    public DefaultPair<T> reverse() {  
        return new DefaultPair<T>(second, first);  
    } // note: reverse can return any subtype of Pair<T>  
    // continued on next slide
```

Test 1 Problem 3c Extended, Part 2

- **N/A** Your next job is to **write a generic DefaultPair class** that implements the Pair interface.

```
// Overridden Object methods required by @Test method calls
@Override // optional
public String toString() {
    return "<" + first + ", " + second + ">";
}
@Override // optional
public boolean equals(final Object that) { // final optional
    return (that instanceof DefaultPair<?>) &&
        (this.first().equals(((DefaultPair<?>) that).first())) &&
        (this.second().equals(((DefaultPair<?>) that).second()));
} // typical equals method - uses both instance variables
// note that we have to use the equals() methods of first
// and second, not ==, since first and second are objects
}
```

<#>



LOYOLA
UNIVERSITY
CHICAGO

Test 1 Problem 3d and 3e

- **1 Redefine p and q from the test fragment above using the new generic types Pair and DefaultPair** (the generic version of DefaultIntPair). *(Do not show the rest of the test code, just the first 2 modified lines!)*

```
final Pair<Integer> p = new DefaultPair<Integer>(3, 5);  
    // “final” optional; 3 and 5 are autoboxed to Integers  
final Pair<Integer> q = new DefaultPair<Integer>(5, 3);  
    // “final” optional; same with 5 and 3  
// type argument must be Integer to be consistent with @Tests
```

- **0.5 TRUE or FALSE (circle one)?** No changes to the rest of the test code above are required for applying the test to the new generic types Pair and DefaultPair.

Test 1 Problem 3e

- This JUnit test fragment works just as before, assuming that all other DefaultPair methods are implemented properly – equals() is the only tricky method to implement with generics because `that instanceof DefaultPair<T>` doesn't work due to “type erasure”; we must use `that instanceof DefaultPair<?>`:

```
final Pair<Integer> p = new DefaultPair<Integer>(3, 5);
final Pair<Integer> q = new DefaultPair<Integer>(5, 3);

assertEquals(3, p.first()); // 3 is autoboxed to Integer
assertEquals(5, p.second()); // so is 5
assertEquals("<3, 5>", p.toString()); // 3 and 5 are auto-unboxed
assertTrue(p.equals(p));
assertFalse(p.equals(q));
assertFalse(q.equals(p));
assertFalse(p.equals(null));
assertTrue(p.equals(q.reverse()));
// Note: <?> is a type “wild card”, meaning any type is OK
```

Test 1 Problem 3f

- **0.5** Why is it beneficial for Pair and its implementation class(es) to be generic? *Circle exactly one.*
 - Performance: the program will run faster.
 - Reliability: the program will crash less often.
 - Generality: Pair can be used whenever a pair of two values of the same type is required.
 - None of the above.
- Note: Whether a class is generic or not has no impact on its runtime performance – generic type information is removed from the generated Java class file (that's what “type erasure” means)

Test 1 Problem 3g

- **0.5** Why is it beneficial to write client code against the Pair interface instead of the DefaultPair class?
Circle exactly one.
 - Performance: the program will run faster.
 - Flexibility: we can switch from DefaultPair to some other implementation of Pair without changing the client code.
 - Conciseness: the resulting client code is much shorter.
 - None of the above.
- We often hide concrete classes behind interfaces because then all we need to know is that the provided concrete class implements all the interface's methods

Test 1 Problem 3h

- **0.5** Why would it make sense for the Pair interface to extend the List interface? *Circle exactly one.*
 - Performance: the program will run faster.
 - Compatibility: existing methods on List will work on Pair instances.
 - Conciseness: the resulting client code is much shorter.
 - None of the above.
- If Pair extends List then any class that implements Pair must also implement all of the List interface methods

Week 6 Topics

- Test 1 Discussion
- **Java Supplemental Topics – if time**
- Project 3 UML 30-minute in-class Exercise
- Agile Development – if time
- Android Application Development – if time
 - Running some simple apps
 - Android framework & activity life cycle
 - More complex apps

Week 6 Topics

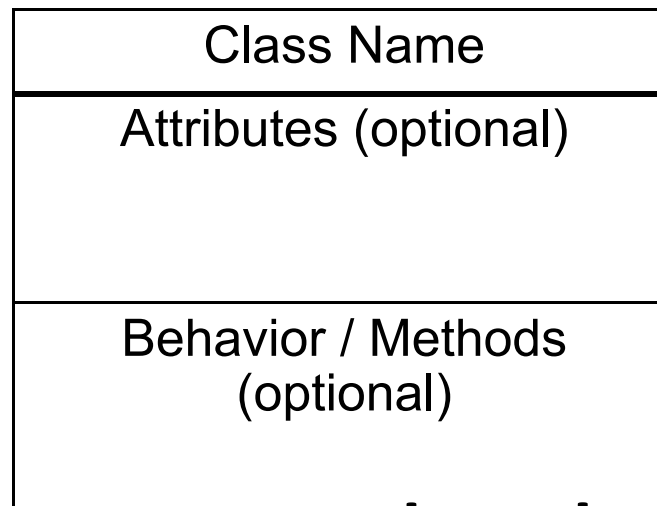
- Test 1 Discussion
- Java Supplemental Topics – if time
- **Project 3 UML 30-minute in-class Exercise**
- Agile Development – if time
- Android Application Development – if time
 - Running some simple apps
 - Android framework & activity life cycle
 - More complex apps

Project 3 Review

- You'll work in Groups based on the 2-person Project 3 teams to implement the Visitor, Composite, and Decorator design patterns in order to draw diagrams on an Android Canvas (Group info is on slide 46) – your Groups will create Project 3 UML Class Diagrams
 - The initial, base implementation does not run in Android
- I'll briefly look at the following areas in Android Studio:
 - The Visitor<Result> generic interface
 - TODOs plus Outline and Stroke (decorators!), and Polygon accepting classes; Point + Location (a decorator); other “model” classes; Draw, Size, and Bounding Box Visitor classes
 - The Android Canvas and Paint classes: online documentation
 - Unit tests run using Gradle; the Fixtures class; Mockito

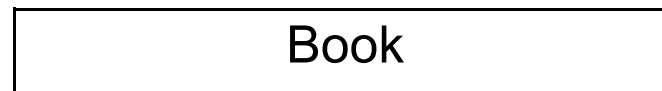
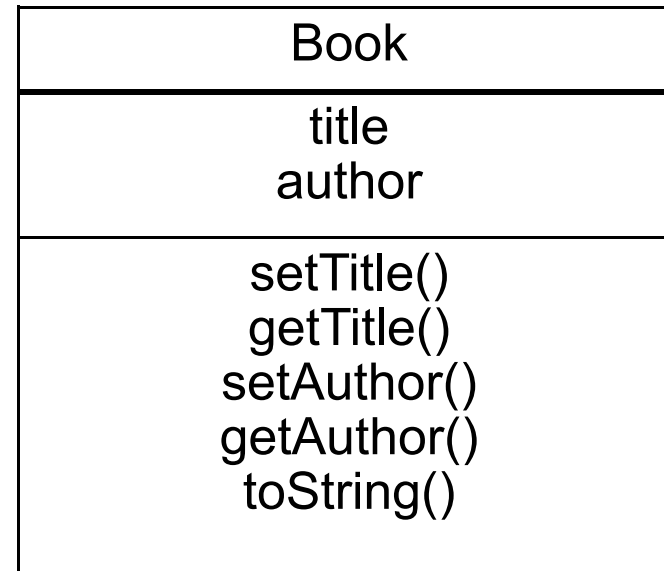
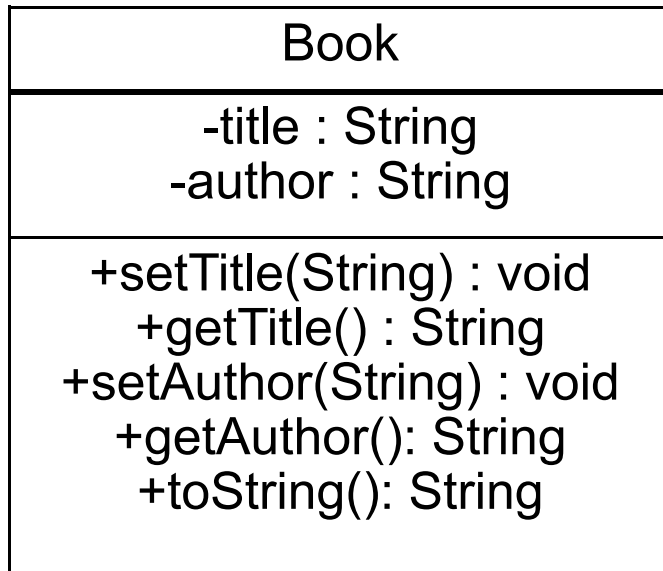
The Unified Modeling Language UML

- Information about OO classes can be represented by UML class diagrams linked by association lines:



- Attributes, parameters, and methods are often preceded by visibility indicators (+, -, #, ~) and followed their data type names, like *name : String*

Example UML Class Diagrams

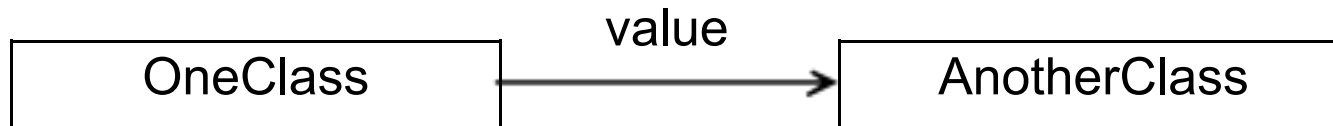


UML Class Diagrams: Relationships

- **Relationships between classes are shown by lines and arrows between class diagrams**
 - Navigability, multiplicity, dependency, aggregation, and composition
 - Inheritance and interfaces
- **Attributes can show multiplicity (so can relationships)**
 - $n \rightarrow$ exactly n of these
 - $* \rightarrow$ 0 or more
 - $m..n \rightarrow$ between m and n
- **Keywords, notes, and comments can also appear**

UML Class Diagrams: Relationships (1)

- One class has attribute *value* with another class type:



- Class multiplicity relationship (**OneClass** collection):

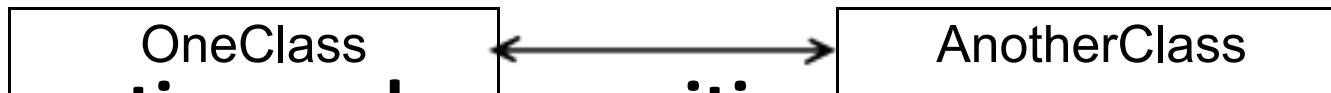


UML Class Diagrams: Relationships (2)

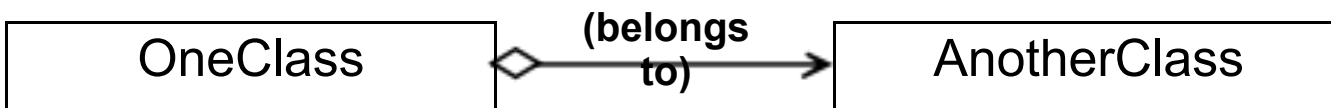
- One class depends on another (dependency):



- Simple association and bi-directional association:



- Aggregation and composition:

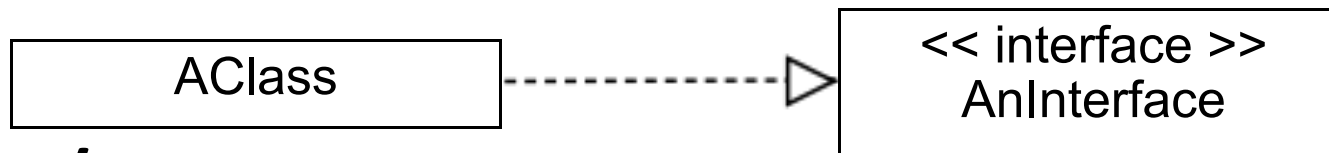


UML Class Diagrams: Relationships (3)

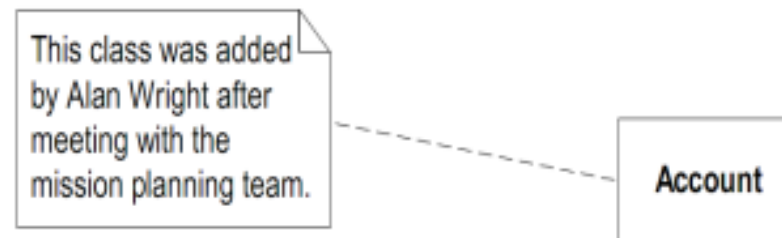
- One class inherits from another (inheritance):



- A class implements an interface:



- Notes/comments:



Project 3 In-Class Exercise Deliverables

1. A preliminary UML class diagram showing your analysis model. The diagram should show any relationships among classes (in some cases, there is more than one relationship between two classes). You can draw this diagram by hand (recommended) or by tool. The classes in the diagram should also include the following elements where applicable:

- operations (methods) and attributes (instance variables)

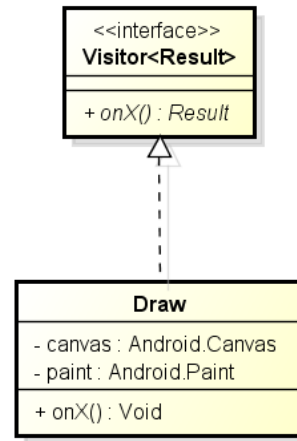
Submit your diagram on Sakai (photo, etc) by 10/8

2. A brief post (200-300 words) by 10/8 on Sakai describing the analysis and modeling process you went through, including:
 - Your rationale for identifying/choosing those classes.
 - For each design pattern you choose to apply, a brief explanation of the problem this pattern solves in the context of the API.
 - Any other comments, reflections, questions, doubts, etc.

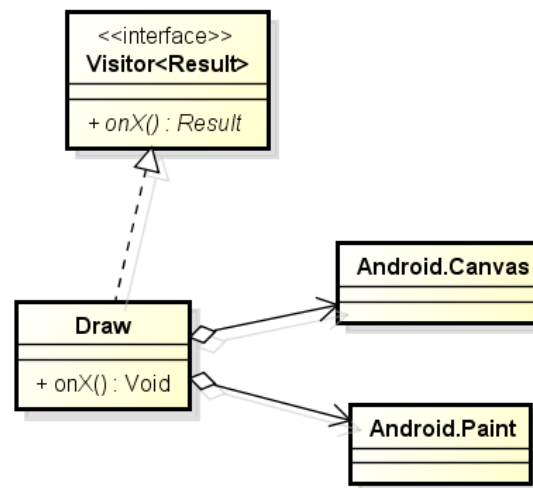
Project 3 UML Class Diagrams

Example: two ways to show a “has-a” relationship

1. With instance variables:



2. With aggregation :



<#>



LOYOLA
UNIVERSITY
CHICAGO

Project 3 In-Class Exercise Groups

Name	Team	Group
Killham, Eric John	01	01
Tapia, Rene	01	01
Cicale, Julia	02	01
Rodriguez Orjuela, Jose Luis	02	01
Mir, Sarfaraz Ali Khan	03	01
Nowreen, Syeda Tashnuva	03	01
Soliz Rodriguez, Percy Gabriel	04	02
Misra, Anadi	05	02
Pacheco, Andrea	05	02
Mehta, Shipra Ashutosh	06	02
Sindhu, Pinky	06	02
Al Khofi, Sundas Abdullatif A	07	03
Liu, Siyuan	07	03
Meghrajani, Aman Maheshkumar	08	03
Goel, Neha	08	03
Oakey, Anthony William	09	03

<#>



**LOYOLA
UNIVERSITY
CHICAGO**