COMP 313/413 Lecture Notes

Release 1.0

Konstantin Läufer

Aug 24, 2020

CONTENTS

Lecture notes for the mostly Scala and Android-based course COMP 313/413: Intermediate Object-Oriented Programming at Loyola University Chicago's Computer Science Department. This version of the course is normally taught by Konstantin Läufer.

Warning: These notes are still being written, so expect a few rough edges. But we're getting closer!

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction

In these lecture notes, we will study intermediate object-oriented development topics from various angles, including design principles and patterns, software architecture, and concurrency.

Please stay tuned for more content!

1.2 Basics of Object-Oriented Programming

The topics discussed in this chapter are considered the basics of object-oriented programming. Some of them are language-independent, others are specific to C#, Java, or Scala. These topics, except perhaps for the last two, are supposed to be covered by the CS 1/2 prerequisite chain for this course.

Todo: update all examples and references — meanwhile, please go here

1.2.1 Reference semantics vs. value semantics

- *Value semantics:* variables directly contain values.
- *Reference semantics:* variables contain addresses that refer (point) to objects.

Examples

- misc/IdentityAndEquality.java
- misc/Comparing.scala

References

- Java
- C# (see also *Effective C#* item 6)

1.2.2 Equality vs. identity

- *Equality:* are two (possibly) different objects equivalent?
- *Identity:* do two references refer to the same objects?
- How are equality and identity related?
- Reconciling value and reference semantics: identity of objects explained as equality of addresses.

Examples

- misc/IdentityAndEquality.java
- misc/Comparing.scala

References

- Java (see also *Effective Java* items 8, 9; see also here)
- C# (see also *Effective C#* items 6, 26)

1.2.3 Parametric polymorphism (generics)

Familiar from data structures course:

- without generics: `Stack` (of objects); loose typing
- without generics: `IntStack`, `StringStack`, `BookStack`; strict typing but lots of duplicate boilerplate code
- with generics: `Stack<Int>`, `Stack<String>`, `Stack<Book>`; strict typing without code duplication

Relatively easy to use, can be challenging to incorporate correctly in the design of one's abstractions.

Examples

- misc/Comparing.java
- misc/Comparing.scala

References

- Java (*Effective Java* chapter 5: items 23-29; see also this tutorial and here for advanced issues)
- C#
- Scala

1.2.4 Relationships among classes and interfaces

These are common relationships among concepts and are part of UML's class diagrams.

Class/interface-level relationships

These relationships are between classes and/or interfaces, so they *cannot* change at run time.

From strong to weak:

- *is-a*, *realizes-a*: generalization/specialization (subtyping)
- *uses-a*: dependency

Instance-level relationships

These relationships are between instances, so they *can* change at run time.

From strong to weak:

- *owns-a*: composition
- *part-of*: aggregation
- *has-a* or other specific relationship: association

Examples

- misc/Animals.java
- misc/Animals.scala
- Figure *A UML class diagram representing a taxonomy of vehicles.*

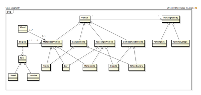

Fig. 1: A UML class diagram representing a taxonomy of vehicles.

1.2.5 Class-interface continuum

- *Concrete class* (C++, C#, Java, Scala): can be instantiated. All specified methods are fully implemented.
- *Abstract class* (C++, C#, Java, Scala): cannot be instantiated. Some or all of the specified methods are not implemented. A class cannot extend more than one abstract class.
- *Trait* (Scala only): cannot be instantiated directly. Some or all of the specified methods are not implemented. A class or trait can extend zero or more traits, and member lookup is automatically disambiguated based on trait order (see traits and mixins for details).
- *Interface* (Java, C# only): limit case of a fully abstract class for specification purposes only. None of the specified methods are implemented, and there are no instance variables.

Related to the single-responsibility and interface-segregation principles.

Examples

- misc/Animals.java
- misc/Animals.scala

References

- Java (see also *Effective Java* items 18, 19)
- C# (see also *Effective C#* items 22, 23)
- Scala

1.2.6 Subtyping vs. subclassing/inheritance

- Subtyping allows substituting a more specific object for a more general one, for example, when passed as an argument or assigned to a variable.
- Inheritance is a mechanism for a subclass to reuse state and behavior from a superclass.
  - inherit methods and fields
  - add fields
  - add or replace/refine methods
- Inheriting from a superclass enables weak syntactic subtyping. (In some languages, this relationship can be public or nonpublic.)
- The Liskov Substitution Principle (LSP) defines strong semantic (behavioral) subtyping.
- Implementing or extending an interface also enables syntactic subtyping (and semantic subtyping because interfaces have no behavior). Extending a trait also enables syntactic subtyping.

Examples

- misc/Animals.java
- misc/Animals.scala

References

- Java (see also *Effective Java* item 17; see also these pitfalls)
- C# (see also *Effective C#* item 22)

1.2.7 Subtype polymorphism: static vs. dynamic type

- *Static type:* declared type of a variable.
- *Dynamic type:* actual type of the object to which the variable refers.
- *Dynamic method binding:* `x.f(a1, a2, ...)`. Two steps:
    1. Verify whether receiver x supports method f based on static type.
    2. Search for version of f to be invoked starting from dynamic type and proceeding upward until found.
- How are static and dynamic type of a variable related?
- If step 1 succeeds, will step 2 always succeed as well?
- *Casting:* treat an object as if it had a different static type. Three different situations: - *downcast* - *upcast* - *crosscast*
- Overloading versus overriding. - `@Override`/`override` correctness in Java and Scala

Examples

- misc/MethodBinding.java
- misc/InterfaceCast.java
- misc/Super.java
- misc/Super2.java
- misc/MethodBinding.scala
- misc/InterfaceCast.scala

References

- Java (see also *Effective Java* item 52)
- C# (see also *Effective C#* item 3)

1.2.8 Being a good descendant of java.lang.Object/System.Object

Classes are usually required to provide the following methods (these specific ones are for Java):

- `toString` (for displaying instances in a meaningful way)
- `equals` (if an instance can be in an equivalence class that include other instances)
- `hashCode` (ditto)
- `compareTo` (if instances are ordered)
- `clone` (if instances are mutable)
- `finalize` (if instances need to release resources)

Also related to the Liskov substitution principle.

Examples

- misc/IdentityAndEquality.java
- misc/Comparing.java
- misc/Comparing.scala

References

- Java (see also *Effective Java* items 8 through 12; see also here for equals, below for clone; detailed Javadoc is here)
- C# (see also *Effective C#* items 5, 9, 10, 27)

1.2.9 Clone in the context of the Composite pattern

In general, cloning allows you to make a copy of an object. The clone method in Java is similar to the copy constructor in C++, but it is an ordinary method, unlike the copy constructor. Once you have the original object and its clone, then you can modify each one independently. Accordingly, cloning is necessary only if the objects are mutable.

Cloning models the real-life situation where you build a prototype of something, say a car or a piece of furniture, and once you like it, you clone it as many times as you want. These things are composites, and the need to be cloned deeply (recursively).

As another example, imagine a parking garage with a list of cars that have access to it. To build another garage to handle the growing demand, you can clone the garage and the customer access list. But the (physical) cars should not get cloned. That's because the garage is not composed of the cars.

As we can see, the conceptual distinction between aggregation and composition has significant consequences for the implementation of the relationship. True, both relationships are represented as references in Java. However, composites usually require a deep clone (if cloning is supported) where each parent is responsible for cloning its own state and recursively cloning its children.

As mentioned above, you don't need to support cloning at all if your objects are immutable because you wouldn't be able to distinguish the original from the clone anyway.

References

- Java (see also *Effective Java* item 11; see also here for more detail)
- C# (see also *Effective C#* items 14, 27)

1.2.10 Packages/namespaces

- Mechanism for grouping related or collaborating classes (cf. default package-level member access).
- In Java, implemented as mapping from fully qualified class names to file system. In Scala, this is much looser.

Examples

- misc/Outer.java

References

- Java (see also here)
- C#

1.2.11 Member access

- public
- protected
- default (package)
- private

Related to the information hiding and open-closed principles.

References

- Java (see also *Effective Java* items 13, 14, 15; see also here)
- C# (see also *Effective C# item 1)

1.3 Overview of a Lightweight Development Process

A successful development process usually comprises these minimal elements:

- automated regression testing
- refactoring
- continuous integration

1.4 Summary

In these notes, we studied intermediate object-oriented development topics from various angles, including design principles and patterns, software architecture, and concurrency.

Todo: Key takeaways: derive from stated learning outcomes.

1.5 Appendix: Course Syllabi

These are the official course syllabi for the most recent section(s) of this course.

1.5.1 Course: COMP 313/413: Intermediate Object-Oriented Development

- Prerequisite: Comp 271
- Official course description: Comp 313 | Comp 413

1.5.2 Sections: COMP 313-002/004 and COMP 413-003/004 Fall 2020

- *General format:* This is an *online "flipped"* class.
  - About 90 minutes of weekly class time will consist of prerecorded videos; I will provide more details on these shortly.
  - The remaining 60 minutes will consist of two synchronous, interactive Zoom sessions, of which you are expected to attend at least one; I will provide links to these shortly in MS Teams (see below).
- *Class time (fall 2020):* You are expected to attend at least one of these synchronous, interactive sessions.
  - main synchronous session: Thu 19:00-20:00 on Zoom
  - alternate synchronous session: Thu 13:00-14:00 on Zoom
- *Communication:* All communication regarding this class takes place in Zoom (verbal) and MS Teams (written). Most will be in the team-level channel specific to this term. For individual or group-level concerns, you may use direct individual or group messages in MS Teams; my user ID is **klaufer@luc.edu**. *(Please DO NOT use email!)*
- *Instructor:* Konstantin Läufer
- *Office hour:* Wed 13:30-14:45 and Fri 10:30-11:45 on Zoom (appointment recommended)
- *TA:* Sean Higgins
- *Office hour:* TBD
- *Texts:*

  Head First Design Patterns (required) *by Kathy Sierra, Bert Bates, Elisabeth Robson, Eric Freeman* Publisher: O'Reilly Media, Inc. Release Date: October 2004 ISBN: 9780596007126 access free on Safari

  Effective Java, 3rd Edition (recommended) *by Joshua Bloch* Publisher: Addison-Wesley Professional Release Date: December 2017 access free on Safari
- *Additional resources:* appendix-resources

- Grading (tentative):
  - 40% quizzes & tests
  - 40% projects & presentations *(Percentage effort on each group project will be measured by an end-of-term questionnaire. Group project grades and/or final course grades may be adjusted to account for significant discrepancies in effort among group members.)*
  - 15% in-class activities
  - 5% participation (in-class and online, including announcements of and reports from relevant professional events, GitHub issues and PRs for course examples, etc.)
- Ground rules
- Sakai site for this section (gradebook)
- MS Team *(mandatory subscription and participation)*
- Important dates (tentative) for take-home quizzes and tests:
  - Week 4 - Wed 16 September: quiz 1
  - Week 7 - Wed 7 October: test 1
  - Week 10 - Wed 28 October: quiz 2
  - Week 13 - Wed 19 November: test 2
  - Week 16 (finals week) - Wed 9 December: test 3
- *Recording of Zoom class meetings:* In this class software will be used to record live class discussions. As a student in this class, your participation in live class discussions will be recorded. These recordings will be made available only to students enrolled in the class, to assist those who cannot attend the live session or to serve as a resource for those who would like to review content that was presented. All recordings will become unavailable to students in the class when the course has concluded. The use of all video recordings will be in keeping with the University Privacy Statement shown below.
- *Privacy Statement:* Assuring privacy among faculty and students engaged in online and face-to-face instructional activities helps promote open and robust conversations and mitigates concerns that comments made within the context of the class will be shared beyond the classroom. As such, recordings of instructional activities occurring in online or face-to-face classes may be used solely for internal class purposes by the faculty member and students registered for the course, and only during the period in which the course is offered. Students will be informed of such recordings by a statement in the syllabus for the course in which they will be recorded. Instructors who wish to make subsequent use of recordings that include student activity may do so only with informed written consent of the students involved or if all student activity is removed from the recording. Recordings including student activity that have been initiated by the instructor may be retained by the instructor only for individual use.
- *Bloom's Taxonomy:* To be used in study guides for quizzes and tests. The letters refer to the levels of learning from the cognitive domain of Bloom's taxonomy:
  - K: know the term
  - C: comprehend the concept
  - A: apply the technique

1.5.3 Detailed Course Outline

Todo: add outline

1.5.4 Key Resources

Todo: add resources

1.6 Appendix: TODO

Todo: update all examples and references — meanwhile, please go here

(The original entry is located in /home/laufer/Dropbox/Home/Work/writing/lucoodevcourse/source/basicoop.rst, line 10.)

Todo: add sample projects, activities, tests, and (tiered) master list for presentations

(The original entry is located in /home/laufer/Dropbox/Home/Work/writing/lucoodevcourse/source/index.rst, line 36.)

Todo: Key takeaways: derive from stated learning outcomes.

(The original entry is located in /home/laufer/Dropbox/Home/Work/writing/lucoodevcourse/source/summary.rst, line 8.)

Todo: add outline

(The original entry is located in /home/laufer/Dropbox/Home/Work/writing/lucoodevcourse/source/syllabus.rst, line 22.)

Todo: add resources

(The original entry is located in /home/laufer/Dropbox/Home/Work/writing/lucoodevcourse/source/syllabus.rst, line 29.)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

Todo: add sample projects, activities, tests, and (tiered) master list for presentations