OO ANALYSIS & DESIGN

CPSC 240 Lecture Notes

Abstract

This document contains material for Gusty's CPSC 240. Gusty took Dr. Karen Anewalt's CPSC 240 in Spring 2017. Gusty adopted many attributes from Karen's class. Karen's organization and material has been modified as needed to fit Gusty's teaching style.
This packet contains the course lecture notes

Gusty Cooper
ecooper@umw.edu

Module 1 – Java Language Review 1
Module 1 Overview 1
Oracle Java Tutorials 1
Useful references 1
Lectures 1
Labs 1
Writing Assignment 2
Class 1 – Java Basics 2
Learning Objectives 2
Assumptions 2
Software Development Process 2
Java Programs 2
Java Project Organization 2
Test Yourself 4
Class 2 – CPSC 240 Java Data Types 4
Learning Objectives Readings 4
Primitive Types, Reference Types, and Objects 4
Object References 6
Passing Parameters 7
Review 8
Class 3 – JavaDoc Documentation 9
Learning Objectives 9
JavaDoc Introduction 9
JavaDoc Standards 10
JavaDoc Tags 10
Generating JavaDoc HTML 10
Pass by Value and Pass by Reference 11

Module 2 – OOD Processes and Tools 13
Module 2 Overview 13
References 13
Software Tools for Creating UML 14
Lectures 15
Labs 15
Writing Assignment 15
Class 4 – OO Design and Object Discovery 15
Object-Oriented Design 15
Object Discovery 16
Unified Modeling Language (UML) 17
Class 5 – Use Cases and CRC Cards 18
Use Cases 18
Identifying Responsibilities & Relationships 21
CRC Cards – A Tool to Discover Classes, Responsibilities, and Collaborators 23
Review: 24
Class 7 – Class Diagrams 24
CPSC 240 Tips for Good Design 24
UML Class Diagrams and CASE Tools 26
Class 8 – SOLID Principles (T- 2/14) 31
Learning Objectives, Assignments, and Readings 31

References 31
Designing Classes 31
SOLID – The first letter – S 32

Module 3 – Java OO Features 35

Module 3 Overview 35
Class Member Accessibility 35
Inheritance 35
Abstract Classes 35
Generic Resources 35
Lectures 35
Labs 35

Class 10 – Inheritance 36
Learning Objectives 36
Inheritance Overview 36
Syntax for Inheritance 36
Class Member Accessibility 39

Class 11 - JavaFX 39
Learning Objectives, Assignments, and Readings 39
JavaFX 40
Scene Builder 40
Three Simple GUI Components 41

Class 12 – Interface Types and Polymorphism 41
Learning Objectives, Assignments, and Readings 41
Polymorphism 41
Collections Class with Comparable Interface Example 41
Collections Class with Comparator Interface Example 43
Highlights about interfaces 44

Class 13 – Unit Testing 45
Learning Objectives, Assignments, and Readings 45
Testing 45
Test Driven Development 45
Java Unit Testing 46
Pre-conditions and Post-conditions 47

Class 15 – Version Control 48
Learning Objectives, Assignments, and Readings 48
Version Control 48
Manual Version Control 48
Git - a shortened version of http://gitref.org 49
Git, Remote Repositories, and GitHub 50
GitHub and .gitignore 57
IntelliJ, Git, and GitHub 57

Module 4 – Design Patterns 59
Module 4 Overview 59
References 59
Lectures 59
Labs 60

Class 16 – Design Patterns and Strategy Pattern 60
Learning Objectives 60

Introduction to Design Patterns 60
Learning Objectives, Assignments, and Readings 60
Strategy Pattern from Wikipedia 61
Strategy Pattern Write-up From: https://sourcemaking.com/design_patterns/strategy 61
Where Have We Seen the Strategy Pattern? 64
Strategy Pattern Description from Our Textbook 64
Class 17 – Observer Pattern and Model-View-Controller Architecture 66
Learning Objectives 66
In-class Reflective Activity 66
Observer Pattern 66
Observer Design Pattern & Examples 69
Some real life examples of Observer Pattern 69
When to use the Observer Pattern: 69
UML Diagram for Observer Pattern 70
Components: 70
Class 18 – Abstract Classes and Template Pattern 71
Learning Objectives 71
Abstract classes 71
Template Method Pattern 72
Remember when we talked about protected? 73
Refactoring 74
Class 19 – Decorator Pattern (R - 3/30) 75
Learning Objectives, Assignments, and Readings 75
The Decorator Pattern 75

Module 5 – Parallelism 79
Module 5 Overview 79
Online Parallel Book 79
Parallelism 79
Concurrency 79
Lectures 79
Labs 79
Class 21, 22, and 23 Handout 79
Learning Objectives, Assignments, and Readings 79
Sequential Programming vs Parallel Programming 79
Parallelism vs Concurrency 80
Focus on Parallelism 81
Example Problem 82
Building a Better Algorithm 89
The Best Approach 91
Class 24, 25, and 26 Handout 93
Learning Objectives, Assignments, and Readings 93
Challenge of concurrency 94
ATM example 94
Trinkle Vending Machine example: 94
Summary of issues: 94
Understanding Issues 94
Example of Interleaving Issue 95
Using Mutual Exclusion to Prevent Bad Interleaving 96
Primitives for Creating a Critical Section 96

We still have 2 problems in the Code 97
Data Race 100
Data Race Rules 100
Guidelines 100
Deadlock 101
Deadlock Example 101
Deadlock Solutions 102

Module 1 – Java Language Review

Module 1 Overview

In module 1, we spend several classes reviewing Java and programming concepts that you have studied previously. This review allows everyone to re-condition their programming skills and get ready to expand them. The following subsections provide links that you can use for re-conditioning your programming skills.

Oracle Java Tutorials

Even though you have Java programming experience, the Oracle Java Tutorials contain illuminating information. For example, the Lesson on Object-oriented Programming discusses the basics of OOP as conducted in Java – Objects, Classes, Inheritance, Interfaces, and Packages.

https://docs.oracle.com/javase/tutorial/

If your Java programming needs refreshing, you should start with the trail *Learning the Java Language* (https://docs.oracle.com/javase/tutorial/java/index.html) and paying special attention to sub-trail *Java Basics*, which covers variables, arithmetic, Booleans, ifs, loops, etc. (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html).

Useful references

The Java API:
http://docs.oracle.com/javase/8/docs/api/

Java essentials syntax/usage sheet:
http://introcs.cs.princeton.edu/java/11cheatsheet/

How to write JavaDoc comments:
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

videos about JavaDocs in intelliJ:
https://www.youtube.com/watch?v=CAexSdMCuGg

Exceptions tutorial:
http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Lectures

- C1JavaBasics.docx
- C2JavaDataTypes.docx
- C3JavaDoc.docx

Labs

- Lab1JavaAPI.docx
- Lab2ClassesJavaDoc.docx

Writing Assignment

- Writing1IDEUsersGuide.docx

Class 1 – Java Basics

Learning Objectives

- Review the software development process.
- Review Java programming with classes and objects.
- Review Java programming with a main method that uses classes and objects.

Assumptions

You have some experience using Java from a previous course.
If this is NOT true, be sure to speak with Gusty after class today!

Software Development Process

Software is developed to solve a problem. All problems must be ________________ before they can be solved. This means there is a statement (or collection of statements) describing the problem to be solves.

Java Programs

Fact: Java was designed to support object-oriented (OO) programming.

Can you write a non-OO program in Java? That depends on what you consider non-OO. If you consider a non-OO program to be one without classes, then you cannot write a non-OO program in Java Every project that you create must include at least one ____________ and that class must include at least one _______________________.

However, you can write Java projects that largely ignore the principles of OO design, and you may have done that to a certain extent in your previous course. When you create a program with a `static main` and other supporting methods, you are not creating an OO design. In this course you learn how to create a good OO design and practice applying these ideas to projects – starting with some small items and working toward larger scale projects culminating in our group project.

For larger projects, embracing OO design results in a well-organized program that is more easily understood. In OO design, your goal is to divvy your project description into a series of customized ________________________________ that represent real world objects. These customized data types are called _________________ and the variables created of these types are called __________________________.

Java Project Organization

A project is created by coordinating the behavior of data types created with classes.
Each class that is used in the project resides in its own ______________.
Each class can contain both __________ members and _____________.

The name of the file must be __________________ as the name as the class (identical spelling & capitalization).

```
public class Greeter {
      //constructor
      public Greeter (String aName) {  name = aName; }

      //method
      public String sayHello( ) {  Return "Hello, " + name + "!"; }

      //data member
      private String name;
}
```

What's the purpose of a **constructor**?

What's the purpose of a **method**?

Why would a class include data?

Can you execute the class above alone? ________________! The class just defines a data type `Greeter`.

Where does execution of a Java program start?
Execution of a Java program starts with the class that contains the `main` method. To execute the method/s in the `Greeter` class, we need to create a class that contains a `main` method that will to instantiate an object (or objects) of type `Greeter` and make the calls to the methods of the Greeter class.

If we're creating a class strictly for the purposes of testing a second class, we often refer to the class as a _________________ class. Here's a tester class for the Greeter.

```
public class GreeterTester {
      public static void main (String[ ] args) {
            Greeter worldGreeter = new Greeter ("World");
            String greeting = worldGreeter.sayHello( );
            System.out.println(greeting);
      }
}
```

We can include a `main` method in our `Greeter` class, but this is awkward and not part of OO design.

The signature for the main method always looks like the one above:

```
      public static void main (String[ ] args)
```

You've undoubtedly encountered this signature in projects that you've written. Because many popular IDEs include this method header automatically, you may not have paid much attention to the details.
What does the reserved word `public` indicate?
What does the reserved word `static` indicate?
What does the reserved word `void` indicate?
What does the parameter identifier `args` represent?

Test Yourself

1. Can non-object oriented programs be written in Java?
2. A class is a way of defining a customized _________ __________.
3. True or false? You can name a Java file anything that you want without regard to the contents of the file.
4. A class created for the sole purpose of testing out methods/operation of another class is called a ______________ class.
5. Consider the `main` method definition - `public static void main (String[] args)` What does the word static indicate about the method?

   What is `String[] args`?

class, method, data types, classes, objects, file, data (or field), methods, the same, no, tester, no/yes, false, tester

Class 2 – CPSC 240 Java Data Types

Learning Objectives Readings

- Understand that Java objects are implicit pointers
- Understand the parameter passing possibilities in Java

Primitive Types, Reference Types, and Objects

Java allows programmers to declare variables of two classifications: *primitive types* (which do not have associated objects) and *reference types* (which have associated objects).

Primitive types are the types "built in" to the Java language.
`int, double, char, boolean, float, ..`

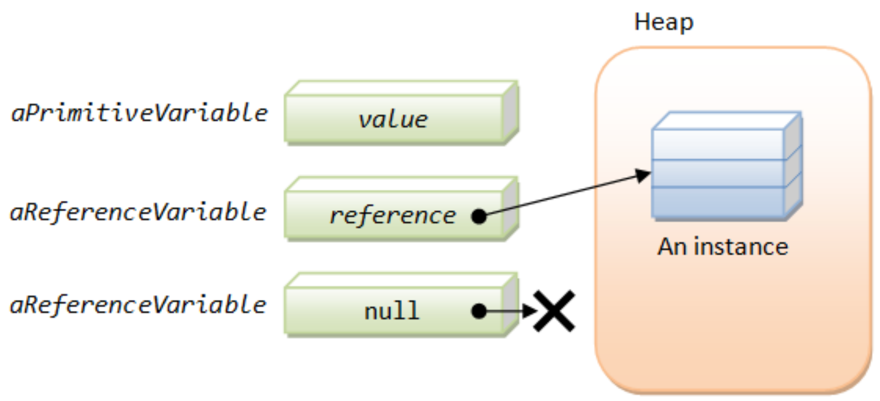
Reference types are defined in classes. Classes allow us to define more complex items.

- Classes can be ___________ (live in your own personal folder) or can be placed in a library and shared with other programmers (for example, in the ________________________).

- Variables whose type is defined by a class reference **objects** of the class.

- When you declare an object, you are __________________ the class.

You have probably used primitive and reference types in past projects, but you may not have thought about how they are handled differently in the language.   Let's think about that now.



Rules related to declaring variables using primitive types:
Example: `int sum;`

- All primitive types start with a lower-case letter (`int, boolean…`)
- Primitive types are NOT implemented with classes.
- Variables declared with a primitive type do not have to be instantiated.
- A fixed amount of memory is set aside for each variable whose type is a primitive, so there is a maximum/minimum value for each of these types (the maximum/minimum value that can be stored in that amount of memory space).

Example: `int sum;`
What does the memory allocation look like after this declaration?

Question: How could you find out the maximum or minimum value that a primitive numeric Java data type?

Rules related to declaring objects (variables created using a class):
- All classes included in the Java API start with a ____________________ (`Integer, ArrayList, ….).` Programmers also usually start their class names with a capital letter for consistency.

Question: Will your code compile if you write a class using a lower-case letter for the class name?   Example: `class greeter { …. }`

- When an object is declared, the computer sets aside space to store the address of the (yet to be determined) location of the actual object.

  Example: `ArrayList  myList;`
  What does the memory space associated with this variable look like?

- The space for an object is created when a class type is instantiated, which involves using the "new" operation.
  Example: `Integer myNumber = new Integer();`

- When the "new" operation executes, the space for the object is allocated and the address of that reserved space is associated with the object's identifier.

- When the "new" operation executes, the constructor for the class is called. If no arguments are provided, the ______________constructor is called.
  Example: `Integer myNumber = new Integer();`

- If one or more arguments are provided, the version of the constructor that requires that number of parameters is called.
  Example: `ArrayList  myList = new ArrayList(10);`

- Thus, when objects are used in Java, a special type of variable known as a ___________________ is created. This means that the value associated with the identifier is a memory address. It's the memory address of the actual location of the data.

Object References

Suppose that you declare an object in Java. The information associated with the identifier is a reference to an object (the address of the object) not the object itself.

In Java you're always using pointers to objects, but this fact is hidden from you. These are known as *implicit pointers.*

Implicit pointers are nice because they hide a lot of memory management tasks from the programmer, BUT they can also lead to odd errors if the programmer isn't cautious.

EX: Suppose we have a Greeter class[1]

```
public class Greeter {
   private String name;

   /**
    * Constructs a Greeter object for greeting people
    * @param aName name of person to be greeted
    */
   public Greeter(String aName) {
      name = aName;
   }
```

[1] Greeter code from Horstman book.

```
   /**
      Greet with a "Hello" message.
      @return "Hello" and name of greeted person.
   */
   public String sayHello() {
      return "Hello, " + name + "!";
   }
}
```

And we create a main program that declares objects of type Greeter:

```
public class GreeterTester {
   public static void main(String[] args) {
      Greeter worldGreeter = new Greeter("World");
      Greeter anotherGreeter = worldGreeter;
      anotherGreeter.setName(“Karen”);
      String greeting = worldGreeter.sayHello();
      System.out.println(greeting);
      System.out.println(anotherGreeter.sayHello();

   }
}
```

What is printed?

Why?
What if you want to make a copy? Use the copy constructor, which is implemented as the clone method (more about this later).

Tips:

Avoid using the = operator with objects because

Avoid using the == operator with objects because

Passing Parameters

The object that calls a method is called an *implicit parameter*. Sometimes the implicit parameter is referred to as the *calling object.*

The parameters in the parameter list (if any) are *explicit parameters*.

The values passed into a method are called *arguments*.

EX:
```
anotherGreeter.setName(“Dave”);
```

What is the implicit parameter?

What is the argument?

What is the explicit parameter?

If you need to refer to the implicit parameter (calling object), use the reserved word **this**. It’s a reference to the calling object.

```
class Greeter {
     ...
      public void setName (String name) {
            this.name = name;
      }

            //data member
          private String name;
     }
```

Review

1. This line of code instantiates an object of the Greeter class:
   `aGreeterObject.setName(“Joe”);`
    a. True or false?

2. Objects are reference variables in Java.

3. Write a call to the default constructor for the ArrayList class:

4. You should be cautious when you use the assignment and equality operators on objects in Java. True or False?

5. Which of the following are shown in this line of code: implicit parameter, explicit parameter, argument?
   `anObject.doSomethingAwesome(4, “Joe”);`

local, Java API, instantiating, Integer.MAX_VALUE (or Google), capital letters, yes, default, reference (or pointer), false, true, true

Class 3 – JavaDoc Documentation

Learning Objectives

- Understand the advantages of using Javadoc comments in Java
- Understand common Javadoc tags including `@param`, `@return`, and `@author`
- Understand when to use regular comments instead of Javadoc comments

JavaDoc Introduction

Java introduced a tool to produce HTML pages of documentation for your programs. This is super convenient because it lets you share the essential details of your program in a standard, easy to read way -and it doesn't take a lot of extra effort.

All you need to do is use "_______________" style comments in your program. Then use the Javadoc tool to generate the .html pages automatically from your Java source files.

Javadoc was used to document all standard Java packages in the ____________________ (Application Programming Interface). All are available on the web. Because developers are used to reading the API, they will find your documentation easy to __________________and __________________ because you'll be using the same format.

Writing Javadoc comments for your own classes allows you to

- ____________________ when creating documentation.
- easily _________________________ because the documentation is stored in the code itself.
- maintain documentation using the _____________________________________. This is easy for other developers to understand.
- __________________________ about your created data types via the web.

Rules:

- Javadoc comments are delimited by `/** .... */`
- Javadoc comments always _______________ the code that they describe.
- The first sentence of each Javadoc comment will be copied into your Javadoc .html files. You need to start the sentence with a __________________ and end it with a _________________________.

- You should provide a general description of your class as a ___________________________.

  Example:

```
/**
 * Class stores a name and creates a customized greeting.
 *
 * Additional descriptive text is included here.
 *
 * @author Kay Horstman
 * @version 1.0 6/6/2017
*/
```

```
public class Greeter {
```

- You should provide a separate description of each of your methods. For each method, you should also comment information about _______________ and ____________________________. To do this, use the `@param` and `@return` tags.
  Example:

```
   /**
    * Greet with a "Hello" message.
    * @return "Hello" and name of greeted person.
    */
   public String sayHello() {
      return "Hello, " + name + "!";
   }
```

Example:

```
   /**
    * Constructs a Greeter object for greeting people
    * @param aName name of person to be greeted
    */
   public Greeter(String aName) {
      name = aName;
   }
```

JavaDoc Standards

Java standards say that every class, every method, every parameter, and every return value should have a comment.

JavaDoc Tags

Some commonly used Javadoc tags are:

```
@param [parameter name] [parameter description]
@return [return argument description]
@throws [exception thrown name]
@exception [exception thrown] [exception description]

@since [version]
@deprecated [description]
@see [hyperlink]
```

In each of these tags, don't type the square brackets and instead fill in the indicated information.

Generating JavaDoc HTML

- To generate the Javadoc in Unix, type `javadoc *.java`
- The Javadoc tool will produce a .html file for each of your .java files, an index.html file, and a few summary files.

- Most IDEs have a menu option that will generate the Javadoc for you. Google to find out how to do this in your preferred IDE.
- Gusty will demo the process during class using IntelliJ (the free community edition)
    - `Tools > Generate JavaDoc…`

Note: For CPSC 240 you are not required to host the JavaDoc for your project publically. You can if you like and might find the cs.umw.edu server to be a good hosting location, or you might like to sign up for a Domain of One's Own, or host using a different hosting service.

Pass by Value and Pass by Reference

All explicit parameters are passed by value in Java. This means that a copy of the value of the parameter is passed in to the method. If the parameter is a primitive, any changes to the parameter's value are not "saved" by the method.

EX:

```
void update (int sum, int value) {
      sum = sum + value;
}
```

However, if the parameter is an object, a copy of the object's memory address will be passed into the method. When the object's identifier is referenced in the method, it points back to the original memory location. Thus, any changes to the object's referenced value will be saved when the method completes.

```
void update (ArrayList myList, int value) {
      myList.add(value);
}
```

JavaDoc, Java API, read, understand, program, create, JavaDoc comments, tell others, precede, verb, period, parameters, return value,

Module 2 – OOD Processes and Tools

Module 2 Overview

In this module, we study processes and tools for creating an OO design. We study the following techniques.

- Noun-Verb walk-through, which is an intuitive way to identify objects and methods in a software specification.
- SOLID – this is an acronym that stands for the following.
    - S – Single responsibility – a class should have a single responsibility
    - O – Open-closed principle – objects should be open for extension, but closed for modification.
    - L – Liskov substitution principle. If S is a subtype of T, the objects of type T in a program may be replaced with objects of type S without changing the correctness of that program.
    - I – Interface segregation principle – A class using an interface is a client of the interface. A client should not be forced to depend on methods it does not use. If an interface has lots of methods and a client uses a few, the client may have to be updated when the interface changes methods the client does not use. To avoid this, interfaces are split into smaller interfaces.
    - D – Dependency inversion principle – depend on abstractions, not concretions.

  We will study portions of SOLID.
- Unified Modeling Language (UML), which allows you to create a visual representation of a software system. UML has many drawings, and we will study several.
    - Use Case Diagram
    - Class Diagram
    - Sequence Diagram

References

The following subsections provide links to various supplemental material for this module.

General OO

Basics of OO terminology: https://docs.oracle.com/javase/tutorial/java/javaOO/

Why is it good to split a program into multiple classes:
http://programmers.stackexchange.com/questions/154228/why-is-it-good-to-split-a-program-into-multiple-classes

Video on Object Oriented Design (requires login to UMW library to view):
http://umw.kanopystreaming.com.ezproxy.umw.edu/video/classes-and-object-oriented-programming
(Note this is in Python, but the ideas all extend to Java)

Tutorial on OOD/OOP:
http://atomicobject.com/pages/OO+Programming

Overriding the equals method: https://www.sitepoint.com/implement-javas-equals-method-correctly/

SOLID

SOLID tutorial: http://code.tutsplus.com/series/the-solid-principles--cms-634
Solid http://www.kirkk.com/modularity/2009/12/solid-principles-of-class-design/
SOLID: http://zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/
SOLID: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
SOLID: http://howtodoinjava.com/2013/06/07/5-class-design-principles-solid-in-java/

edX UML Course
https://www.edx.org/course/uml-class-diagrams-software-engineering-kuleuvenx-umlx

UML Use Case

Tutorials
http://www.visual-paradigm.com/tutorials/writingeffectiveusecase.jsp

https://www.youtube.com/playlist?list=PL05DE2D2EDDEA8D68

http://csis.pace.edu/~marchese/CS389/L9/Use%20Case%20Diagrams.pdf

Videos
https://www.youtube.com/watch?v=tLJXJLfLCCM

https://www.youtube.com/watch?v=_JCsqdNf8bA

UML Class Diagram

Tutorials
http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf

Discussion on Types of Class Relationships
http://nirajrules.wordpress.com/2011/07/15/association-vs-dependency-vs-aggregation-vs-composition/

Software Tools for Creating UML

You need a tool to draw UML diagrams. You could use PowerPoint or Google Draw, but this would force you to create the various UML shapes. You want a tool that easily draws the shapes.

You can use any tool that you'd like to create the diagrams; however, I recommend https://www.draw.io. Draw.io is a web-based tool that you can use for free. Draw.io supports UML and other drawing symbol packages. Draw.io allows you to save your drawing a .xml file on your computer, which can be reopened in Draw.io. Draw.io also allows you to save your drawing as a PDF file on your computer, which can be viewed, printed, and submitted on Canvas.

We have Microsoft Visio and Rational Rose installed in B12. Both tools are commonly used in industry and would be nice skills to add to your resume. However, they are more complex than Draw.io, and I am not familiar with them.

Gliffy is another web-based tool – http://www.gliffy.com/uses/uml-software

Creately is another web-based tool – https://creately.com/

Lectures

- C4OOD&ObjectDiscovery.docx
- C5UseCases&CRCCards.docx
- C6ClassDiagrams.docx
- C7SOLID.docx

Labs

- Lab3UseCases.docx
- Lab4ClassDiagrams.docx
- Lab5DesignDecisions&Tradeoffs.docx

Writing Assignment

- Writing2BookReport.docx

Class 4 – OO Design and Object Discovery

Object-Oriented Design

Learning Objectives

- Understand the difference between a class and an object.
- Understand how to use the following terms related to object-oriented design: fields or data members, methods, encapsulation, state, behavior, and identity.

Object-Oriented Design

Object Oriented Design is a method of

Class is code, ADT is idea

Class = the implementation of an ADT ______________________

= data and methods, encapsulated _______________together

Object = instantiation of a class
= has a
state (______________________________________),

behavior (____________________________________),
and identity (a variable name)

The methods can be used to change the state of the object over time.

Object Discovery

Learning objectives:

- Understand how to use standard strategies to identify possible classes, responsibilities, and relationships in each project description.
- Understand how to use Use Cases and CRC cards in the early design process.

Identifying Classes

Rule of thumb for identifying classes in a project description
Look for the nouns. Not every noun will be a class, but some nouns will be classes.

UMW Room Reservation System Problem Description

UMW is in desperate need of scheduling software that would allow administrators, faculty, and student groups to reserve rooms for class meetings, meetings, and other functions on campus. The UMW community should be able to view a calendar that shows reserved rooms on any day. The calendar should note the times that each room is reserved and person who requested the reservation. Users should be able to search for available room based on room capacity (i.e. I need a room that will hold 30 or more people), building (i.e. I need a room in Trinkle Hall), date/time (i.e. I need a room on Tues Feb 8 from 3:00-5:00), AV equipment (i.e. I need a room with a projector), and seating (i.e. I need a room with tables not desks). The system should return a list of available room options and the user should be allowed to reserve a room from this list.

Write a list of possible classes that might be contained in the final system:

```
/**
 * Stores information about a user of a software system
 * @author CPSC 240 Instructor
 */
public class LoginAccount {

    Person p;
    String userName;
    String passWord;

    /**
     * Constructor initializes fields
     * @param person person with login account
     * @param userName user name
     * @param passWord pass word
     */
    public LoginAccount(Person p, String userName, String passWord) {
        this.p = p;
        this.userName = userName;
        this.passWord = passWord;
    }

    public String getUserName() { return userName; }

    public String getPassWord() { return passWord; }

    @Override
    public String toString() {
        return "LoginAccount{ " + userName + " " + passWord + " };

    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass())
            return false;
        else if (this == o)
            return true;
        else {
            LoginAccount la = (LoginAccount)o;
            return (userName.equals(la.userName) &&
                    passWord.equals(la.passWord);
        }
    }
}
```

Unified Modeling Language (UML)

UML is a tool with various shapes and arrows that is often hard to remember. If you used UML daily these shapes and arrows would become more intuitive. UML has two purposes.

1. Discovery – use UML to __________ the design solution of a problem. When discovering a design solution, the pretty, detailed diagrams are not necessary. Often, you can create your own short-hand version of pseudo UML on white boards and pieces of paper. One UML tool that we studied, the CRC cards, recommend small hand written index cards for this discovery process, which you eventually discard. Many of the artifacts creating during the discovery process are discarded.
2. Documentation – use UML to ____________ the design solution of a problem. One thing you will learn is the code has a long shelf life. A good example is LINUX, a popular operating system, which began in 1992. Many programming jobs fix bugs and add new capabilities to an existing, legacy code base. New programmers must

understand the existing code base to change it. UML can provide an overview, allowing programmers to understand the design solution. In this case, a pretty diagram can be helpful. I have always thought there are three things needed for maintenance programming.

a. A requirements specification. This is the written description of the system to be created, often captured in a requirements database tool.
b. High-level diagrams of the design solution. UML is a good tool for capturing these.
c. Code.

Abstract Data Type, grouped, value of fields, methods, discover, document

Class 5 – Use Cases and CRC Cards

Use Cases

Learning Objectives

- Understand that Use Case write ups are a way of documenting functional requirements and developing an understanding of a software project.
- Gain experience in documenting Use Cases using an example format.

Understanding System Use

Part of designing a system involves developing an understanding of how the system will be used.

____________________ requirements specify ways that end users will interact with a system. During the early phases of a software project, functional requirements are often documented using a UML tool called ____________________. A Use Case is a formal way of describing an activity that your software must handle.

Documenting use cases allows analysts/software engineers to think through a project in more detail and can help them to identify fundamental components that must be present in the final software solution.

There is no standard template for writing use cases. Each company uses its own standard. However, all use cases are capture the flow of a user interacting with the system.

Use Case Format Example:

Title: (has a verb)
Short description:
Main Flow:

1. sequence of steps in "standard" operation
2. more steps

Alternate Flow A:

1. Alternate sequence of steps for such things as user error.
2. More steps

Alternate Flow B: You may have several alternate flow

The main flow and alternate flow emphasize the role of the actor (usually an end user of the system) and the system itself.

Use Case Example – Password Update

Title: Change Password
Main Flow:

1. User indicates they want to change their password
2. System prompts for current password
3. User types in current password
4. System verifies password
5. System prompts for new password
6. User types in new password
7. System checks new password is acceptable (length, special chars, etc.)
8. System prompts user to retype password
9. User retypes new password
10. System confirms new passwords match

Alternate Flow A:

3. User types incorrect current password
4. System prompts password is incorrect and to try again. Use flow returns to standard step 3

Alternate Flow B:

7. System determines User has entered unacceptable password.
8. System prompts password is weak and to enter a stronger password. User flow returns to standard step 6

Alternate Flow C:

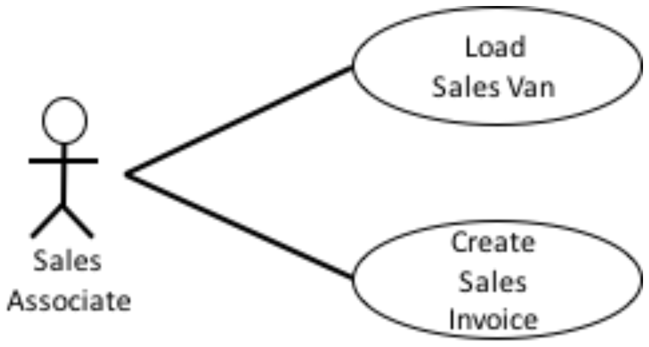
10. System determines two passwords do not match.
11. System prompts passwords do not match and to reenter. User flow returns to standard step 6.

Use Case Diagram

- Visual index to all use case write-ups for a system.
- Associate actors with actions, where an action is the use case title or a shortened version of it.

Use Case Diagram Example

The following is a single use case diagram that shows a Sales Associate has two actions – load sales van and create sales invoice.



Practice Writing Use Cases and Use Case Diagrams

Write two use cases for the room reservation system.

UMW Room Reservation System Problem Description

UMW is in desperate need of scheduling software that would allow administrators, faculty, and student groups to reserve rooms for class meetings, meetings, and other functions on campus. The UMW community should be able to view a calendar that shows reserved rooms on any day. The calendar should note the times that each room is reserved and person who requested the reservation. Users should be able to search for available room based on room capacity (i.e. I need a room that will hold 30 or more people), building (i.e. I need a room in Trinkle Hall), date/time (i.e. I need a room on Tues Feb 8 from 3:00-5:00), AV equipment (i.e. I need a room with a projector), and seating (i.e. I need a room with tables not desks). The system should return a list of available room options and the user should be allowed to reserve a room from this list.

Now that you have a better understanding of the project, chat with a friend and try to identify some potential classes in the system that we missed on our first pass.

Identifying Responsibilities & Relationships

This section describes an approach for identifying classes their responsibilities and the relationships among the classes. This is a key component to creating a good solution to a complex problem.

Learning Objectives

- Understand how to use standard strategies to identify possible classes, responsibilities, and relationships in each project description.
- Understand the noun-verb walkthrough technique.

Last time we talked about discovering objects and classes. A rule of thumb is to read through the project description and look for __________.

When you start identifying classes and objects, you need to start thinking about how to distribute ________________________________ among them.

How to identify responsibilities

Rule of thumb: look for _____________words in the project description

This will help you to identify actions that the system needs to perform. You can assign these actions to the classes.

Example: We talked about Room being a potential class in the project. The primary responsibility of this class is to store information about a single Room at UMW. The data stored would include the building name, room number, seating capacity, and hasProjector (Boolean value).

Note: Even though responsibilities correlate with actions, they don't directly translate into methods. Each responsibility might be implemented through ________________ methods.

Can you think of multiple methods required to implement the Room class's responsibility?

Look at one of the items you flagged as a potential class in the room registration example from last class.   Identify 1-3 potential responsibilities for that class.

Identifying relationships – Dependencies, Aggregation, Inheritance

Dependencies, aggregation, and inheritance are three main types of relationships of interest to programmers. They are described in the following subsections.

Dependencies

Class1 is dependent on class2 if class1 ______________________ objects class2 in any way

Example: Suppose that a method in the Room class returns a String object. Makes the Room class dependent on the String class. In other words, if the String class is modified in the future, you might also have to modify your class because your method might not work correctly with the updated String class.

When you're designing classes, it's good to have few dependencies.

This is called ____.

Having a design with few dependencies makes it easier to update your classes later.

Aggregation

Class1 ______________________________ objects of class2
Aggregation is a special type of dependency

This is a "_______________" relationship

Example: If you create a Patient class and inside the class you declare a data member of type Date. So, as a data member the Patient class **has a** Date object.

If the Date class changes, would you need to change the Patient class?

If the Patient class changes, would you need to change the Date class?

Inheritance

Class1 inherits ______________________________ from class2.

Class1 (the subclass) is a special case of the class2 (the superclass)

This is a "_____________" relationship

Example: Suppose you need to implement multiple types of users in your system: Staff and Students. You might implement a User class that stores information common to all users. Then you might inherit from the User class to create a specialized Staff class. You might also inherit from the User class to create a Student class.

If the User class changes, might you need to change the Staff class?

If the User class changes, might you need to change the Student class?

If the Student class changes, might you need to change the Staff class?

CRC Cards – A Tool to Discover Classes, Responsibilities, and Collaborators

Learning Objectives

- Given a project description, develop a list of potential classes
- Identify potential responsibilities for a given class
- Identify relationships that might be appropriate for a set of classes

CRC Cards

What are CRC cards? ____

On each index card describe ______________________________ and list its responsibilities and collaborators.

What are collaborators? ____________________________________

Classes that ____________________________________ in this class

Note: ____

The goal is to put down a bunch of ideas and then refine them. How might you refine them?

- If a class has too many responsibilities, ___________________________

Shoot for ______________________________ high-level responsibilities (which may turn into more than 3 methods at implementation).

- Eliminate classes with ____
- Remove responsibilities that do not ______________________________. Do not add extra responsibilities just because you can

CRC Cards in Early Design

How might you incorporate CRC cards into your early design?

Meet with a partner developer and examine the Project 1 Specification.

1. Do a noun/verb walk through
2. Create a candidate list for classes and operations
3. Develop user cases
4. Consider 1 use case
    - 1st person play protagonist & propose a responsible agent and method for the task
    - 2nd person play devil's advocate and ask for clarification or suggest an alternative
5. Put cards on table so classes are close to collaborators

As you talk through and review the classes, you can modify, discard, or create cards anytime.

Goal of CRC Cards

Remember, the goal of CRC cards is to discover classes, responsibilities, and relationships.

Review:

- Think of 3 questions that should be on next week's quiz related to the topics in the packet.

Functional, use cases, nouns, responsibilities, action, multiple, uses, loose coupling, instantiates, "has a", maybe, no, data and methods, "is a", note card (physical or electronic), classes, responsibilities, collaborators (relationships), one class, have dependencies, reduce its scope, 1 to 3, 0 responsibilities, add to final product,

Class 7 – Class Diagrams

CPSC 240 Tips for Good Design

Learning Objectives

- Understand Tips for Good Design

Tip 1: Modularize your code

A class represents an abstract __________________. Ideally the data type is a representation of ONE abstraction and has a ________________ responsibility.

Unless your project is incredibly simple, do not write all your code in a single class that contains only a main method.

Tip 2: Include a Javadoc header comment for every class

```
/**
   The Room class encapsulates data about a single room including the
   building name, room number, and seating capacity.
   @author Name Of Author
*/
```

Tip 3: Include a Javadoc method comment for every method

Provide a ______________ and use the @________ and @_________ tags in your comments

```
/** Greet with a "Hello" message.
    @return message with "Hello" and name of greeted person
*/
String public sayHello( ) { ... }
```

Tip 4: Don't use deprecated methods in the API. They are ____________________.

Tip 5: Include a constructor in your class

Why? This ensures that all objects have a ________________________ upon creation

Let's look at an example class:
Example:

```
            class Room {
                  int roomNumber;
                  String buildingName;
                  int capacity;

            }
```

What should happen in a constructor?

What types of things should not happen in a constructor?

Tip 6: Preserve encapsulation

What is encapsulation? Grouping data/methods together into a new data type
Why? This prevents your class from being misused

For someone else to use your class, they do not need to know the details of your code; they just need to know the public interface.

Example:

```
            class Room {
                int roomNumber;
                String buildingName;
                int capacity;

            }
```

You may think that allowing direct manipulation from `main` keeps your program simple.

```
        Room aRoom = new Room();
        aRoom.roomNumber = 24;
        aRoom.buildingName = “Trinkle”
        aRoom.capacity = 100;
```

Fine, but what happens when if the Room author changes the data type of roomNumber changes to a String?   Will our code in main still work?

Tip 7: Don’t use I/O in your class if you can avoid it.

NOTE: Including a method to assist with debugging is okay.
Why? Hardcoding the spacing, labels, and other output details __________ the reusability of your code.

Example:

```
class Room {
   int roomNumber;
   String buildingName;
   int capacity;

   public void printRoom () {
      System.out.println(“\n\n” + buildingName + “ “ + roomNumber + “\n”);
   }
}
```

What happens if you need to print the room info in another format? What if you reuse this code later in another context and you don’t display information in the same way?

It may be a better idea to include accessor methods that allow you to customize how the data is displayed in main.
Example:

```
        Room aRoom = new Room();
        System.out.println(“The room you reserved is: “ +
                           aRoom.getGuildingName() + “:” +
                           aRoom.getRoomNumber() + “\n\n”);
```

UML Class Diagrams and CASE Tools

Learning Objectives

- Understand Class Diagrams and CASE tools

- Understand standard UML notation for representing classes including data members and methods as well as relationships between classes.

Class Diagrams

Goal: ____

Often drawn using a Computer Aided Software Engineering (CASE) tool, for

example: ____

Class is depicted as a ____

Example:

You can include the class name, attribute information and method information.

You can include all of these items, some of these items, or no details.

If you want to include the data member details, use this format
visibility identifier: type

visibility

- + public
- - private
- # protected
- ~ package

Example:

```
-name: String
-age: int
```

If you want to include the method details, use this format
visibility methodName(identifier: Type): returnTypeIdentifier

Example:

```
+getName() : String
```

Sometimes you do not include all information in your class diagram, which allows your class diagram to focus on the higher-level organization of your program and to save space.

You can depict relationships

Dependency

Aggregation

Inheritance

And many others – see the text book for details

You can depict multiplicity

Any number (0 or more)
1 or more
zero or 1

exactly 1

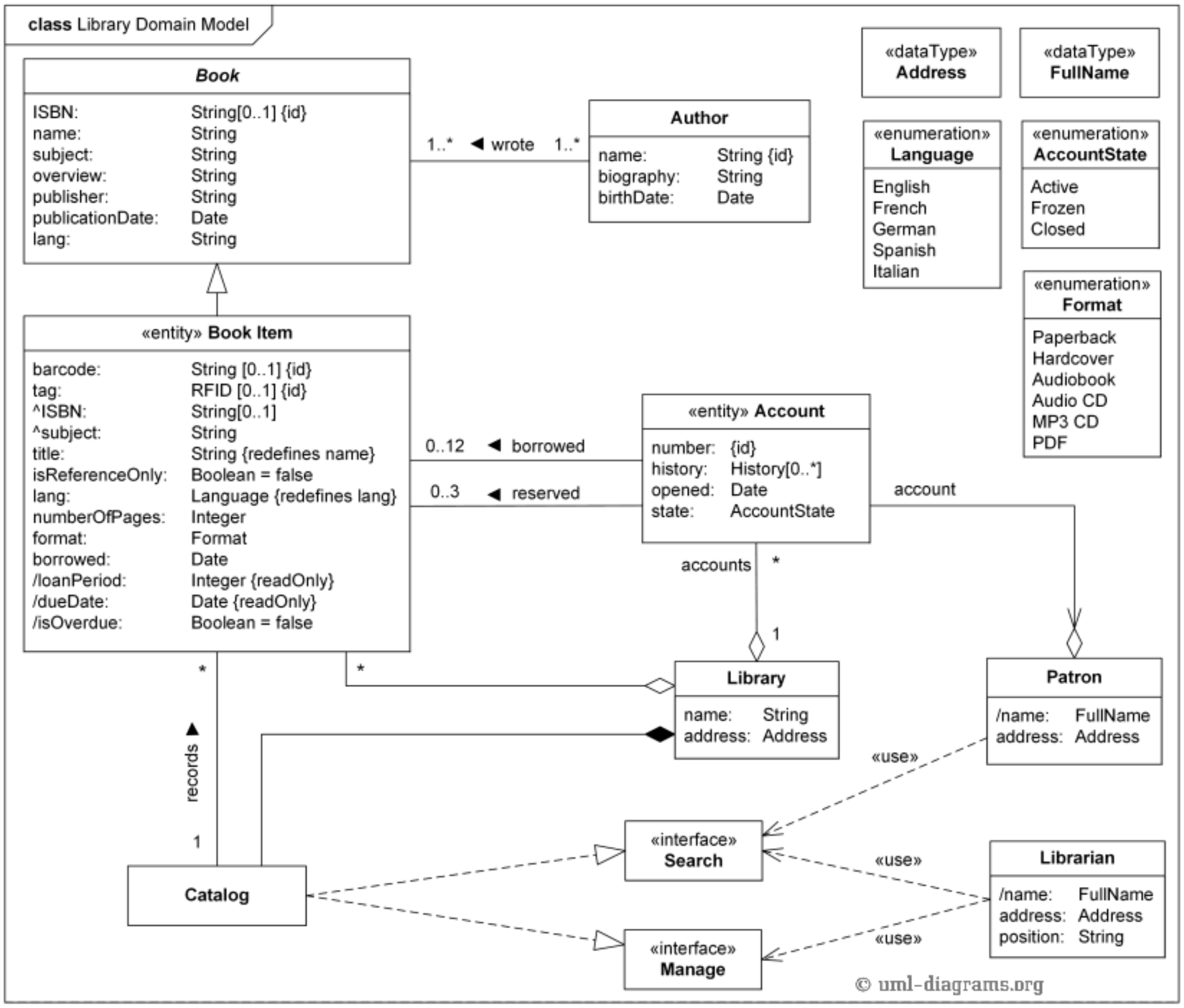
Example: This diagram shows that each message exists in exactly 1 message queue. And each message queue holds zero or more messages.

You can also depict responsibilities by writing them on the line for clarification

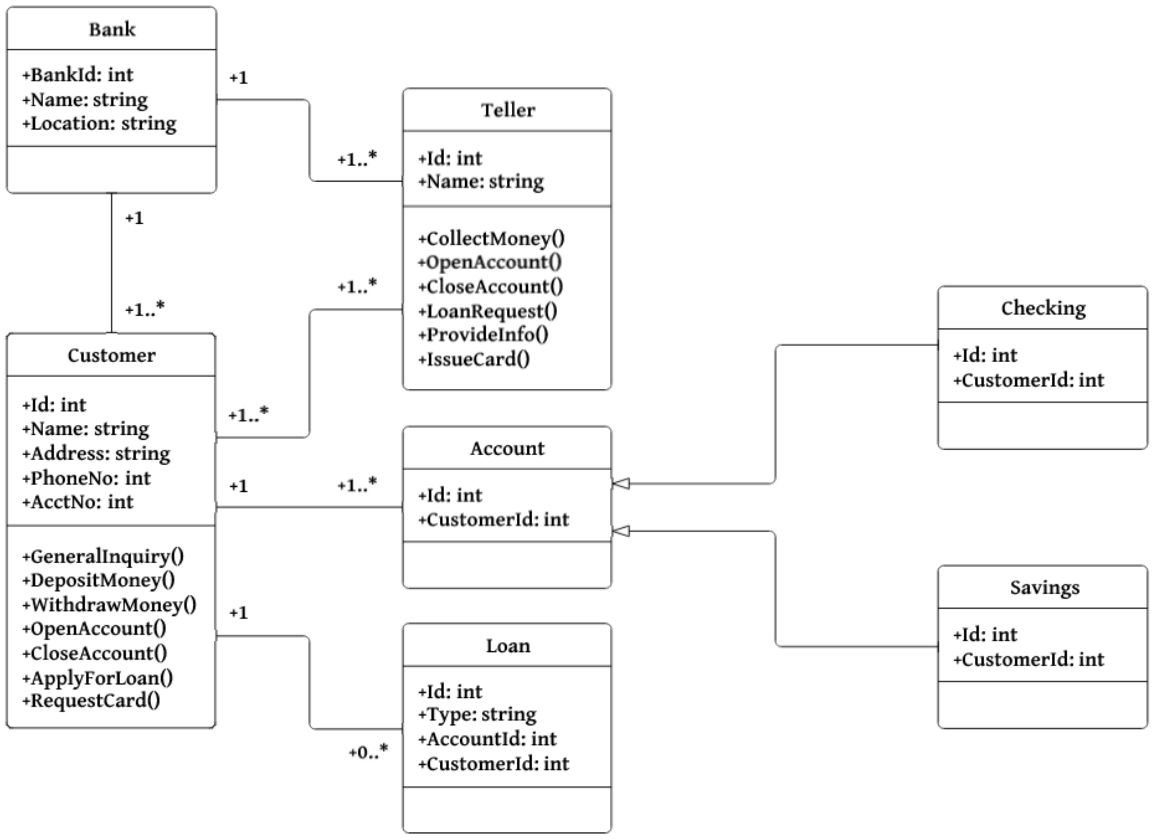
Example: A student registers for a course. A course has a student participant.

Class Diagram Examples

Example from: http://www.uml-diagrams.org/examples/class-example-library-domain.png



Example from: https://d2slcw3kip6qmk.cloudfront.net/marketing/pages/chart/what-is-a-class-diagram-in-UML/UML_class_diagram_example2-750x539.PNG



Data type, single, valid initial state, description, param, return, not supported, valid initial state, initialize fields, I/O, limits, visual overview of problem solution, Visual Studio, rectangle with three areas,

Class 8 – SOLID Principles (T- 2/14)

Learning Objectives, Assignments, and Readings

- Understand the Single Responsibility Principle
- Understand why this principle is important in software development
- Understand how to identify code that violates the Single Responsibility Principle
- Write code that doesn't violate the Single Responsibility Principle

References

https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Designing Classes

When writing classes we want to design for 5 things:

__________________ – all methods are related to a single abstraction

_________________ – supports all operations that are part of the abstraction

Note: You need not implement every possible operation for your class to be complete. You want it to completely implement the abstraction is its primary responsibility, but not more than that.

_________________ – make it easy for someone else to use your class, especially the most common operations required
_________________ – clear to programmers without generating confusion

Example:

- Select good identifiers for your methods
- Include Javadoc comments so that people know what to expect from your methods in terms of parameters and return values
- In comments, clearly state your methods preconditions and postconditions

_________________ – operations should be consistent in terms of names, parameters, return values, and behavior
Example:

SOLID – The first letter – S

SOLID – An acronym for the 5 most agreed upon OO Design Ideas

To be an effective object oriented programmer, this is probably the single best thing that you could read and wrap your head around. Google it!

Note: SOLID is object-oriented not Java specific, so you will find write-ups using lots of other OO languages. The principles still apply to Java even though the syntax will be different.

S = Single Responsibility Principal
One class should have one and only one responsibility.
You should write your class for only one purpose.

WHY? Having a single responsibility will allow your class to be modified in the future without needing to worry about dependencies and how the changes may affect another part of your code.

Your class might model a real-world object and bundle data about it.
Example:

```
public class Room {
      String buildingName;
      String roomNumber;
      int capacity;
```

```
            public Room() { capacity = 0; }
            public Room(String bN, String rN, int c) {
                  setBuildingName(bN);
                  setRoomNumber(rN);
                  setCapacity(c);
            }
            public String getBuildingName( ) { return  buildingName; }
            public String getRoomNumber( ) { return roomNumber; }
            public int getCapacity( ) { return capacity; }
            public void setBuildingName( String bN)  { buildingName = bN; }
            public void setRoomNumber (String rN) { roomNumber = rN; }
            public void setCapacity(int c) { capacity = c;}
      }
```

Example: Mailbox class (page 118 in OOD&P book)

```
public class Mailbox {
      …
      public void addMessage(Message aMessage) { … }
      public Message getCurrentMessage( ) { … }
      public Message removeCurrentMessage( ) {… }
      public void processCommand(String command) { … }
}
```

Do all the methods above relate to the same abstraction?

Example:
A typical example is a user management class. When you for instance create a new user you'll most likely send a welcome email. That's two reasons to change: To do something with the account management and to change the emailing procedure.

Example:

```
      class Book {
            String getTitle() { return "A Great Book"; }
            String getAuthor()  { return "John Doe"; }
            void turnPage( ) { //return a reference to the next page }
            void printCurrentPage( ) {
                  System.out.println(currentPageContent);
            }
      }
```

Think about what type of classes might call the methods in this object.
The Book class might be used by a Data Presentation Mechanism (a GUI).
It might also be used by a Book Management class (like a librarian).

Would they both want the information about the book to be presented in the same way?

A better implementation for the Book class

```
      class Book {
            String getTitle() { return "A Great Book"; }
            String getAuthor()  { return "John Doe"; }
            void turnPage( ) { //return a reference to the next page }
            Page getCurrentPageContent( ) {
                  return(currentPageContent);
            }
      }

      interface Printer {
            void printPage(Page p);
      }

      class PlainTextPrinter implements Printer {
            void printPage(Page p) {
                  System.out.println(content of p);
            }
      }

      class FancyPrinter implements Printer {
            void printPage(Page p) {
                  System.out.println("~~~~~~~~~~~~~~~~");
                  System.out.println(content of p);
                  System.out.println("~~~~~~~~~~~~~~~~");
            }
      }
```

We will cover other SOLID principles through the semester.

Cohesion, completeness, convenient, clarity, consistency

Module 3 – Java OO Features

Module 3 Overview

In Module 2 we studied ways to identify OO features in a problem statement. In Module 3 we study Java's features that support OO. We also discuss JavaFX, Unit Testing, and Version Control.

Class Member Accessibility

Modifier	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
`public`	yes	yes	yes	yes	yes
`protected`	yes	yes	yes	yes	no
no modifier	yes	yes	yes	no	no
`private`	yes	no	no	no	no

Inheritance

We discuss inheritance in class 10. Other resources for inheritance:
http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
http://www.homeandlearn.co.uk/java/java_inheritance.html

Abstract Classes

We discuss Abstract Classes in Class 18, which is in Module 6, Design Patterns.

Generic Resources

http://docs.oracle.com/javase/tutorial/java/generics/

Lectures

- C10Inheritance.docx
- C11JavaFX.docx
- C12InterfaceTypes&Polymorphism.docx
- C13UnitTesting.docx
- C15VersionControl.docx

Labs

- Lab6Inheritance.docx
- Lab7JavaFX.docx
- Lab8Interfaces&Polymorphism.docx
- Lab9GroupQuestionnaire.docx
- Lab10UnitTesting.docx
- Lab11VersionControl.docx

Class 10 – Inheritance

Learning Objectives

- Understand the concept of inheritance
- Understand how to identify parent/child relationships (sometimes called superclass/subclass relationships)
- Given a parent class, understand how to implement a child class with specified additions
- Understand how to identify places where inheritance can't be used

Inheritance Overview

The big picture concept behind inheritance is __________________________________.
You have an existing class. You need to make a new class that has all the characteristics of the existing class and has either_____________________________
and/or _______________________________associated with it.

EX: Which of the following would be the "base class" – the original class with the functionality/data that will be shared by all the classes?
User, AdministrativeUser

Employee, Manager

Student , UMWPerson, Faculty, Dean

Note: Another term for "base class" is "superclass." Use whichever term makes the most sense to you.
The terms for the class that extends the original are: "subclass", "child class" and "derived class." You can use these 3 terms interchangeably.

Syntax for Inheritance

```
public class Employee {
    private String name;
    private double salary;

    public Employee(String n) { name = n; }
    public void setSalary(double s) { salary = s; }
    public String getName() { return name; }
    public double getSalary() { return salary; }
}

public class Manager extends Employee {
    private double bonus;
    public Manager(String n, double b) { super(n); bonus = b; }
    public double getSalary() { ... } // overide Employee method
}
```

When inheriting from a superclass, you only declare the ________________________ between the subclass and superclass. The subclass automatically inherits ___________________________ and ____________________________ from the superclass.

How many data members does an object of type `Manager` have?

Which methods can be called by an object of type `Manager`?

Question: Is it possible to inherit just some selected data members and/or methods from the superclass?

Question: When implementing a subclass it possible to change the behavior of a method that is in the base class?

You can override a base class' method by writing a ______________________________ for that method ____in the superclass.

EX:

Remember the superclass is more _____________________. The subclass is more _________________.

Classes can be related through multiple levels of inheritance. This forms a ________________________.

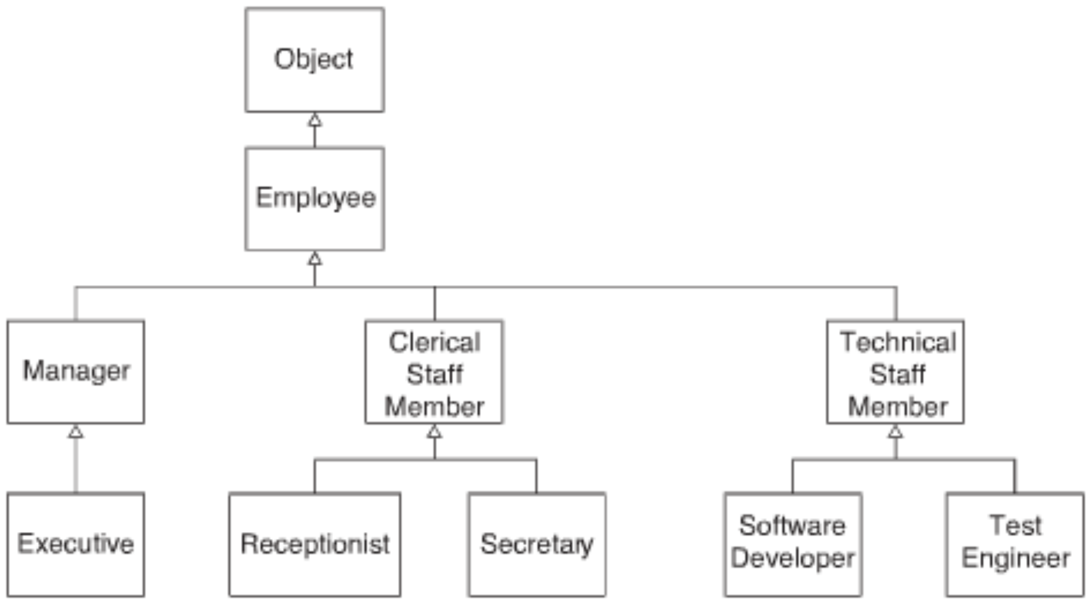


Figure 3

A Hierarchy of Employee Classes

Since a subclass "is a" superclass, you can substitute a subclass anywhere a superclass is expected. This is called the **Liskov ______________________________ principle**, where Liskov is the letter L of SOLID.

Example of Liskov:

```
Employee e;
…
e = new Manager("John Doe");
e.getName();  //calls method in the ______________class since the
____________________
e.getSalary(); //calls the ________________ version of the getSalary( ) method because
of polymorphism
```

Let's look at how we'd implement the `getSalary( )` method in the `Manager` class

```
public double getSalary( ) {
    return salary + bonus; //ERROR:
}
```

What about:

```
public double getSalary() {
    return getSalary()+ bonus; //ERROR:
}
```

Solution

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

Another option: use ______________________________ rather than `private` variables in your superclass.

The protected visibility level means that the class member is accessible within any code that extends the original class. But it's not accessible from outside the inheritance chain.

Suppose that the Clerical Staff Member class (from the hierarchy on the previous page) has a data member:

```
      protected String faxNumber;
```

Is this data member accessible in each of the following classes?
Clerical Staff Member?
Employee?
Secretary?
Executive?

Class Member Accessibility

Oracle describes controlling access to class members at the following site.
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

The following table summarizes access.

Modifier	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
public	yes	yes	yes	yes	yes
protected	yes	yes	yes	yes	no
no modifier	yes	yes	yes	no	no
private	yes	no	no	no	no

Reusability and extending code, additional data, new methods, User, Employee, UMWPerson, differences, all the data, all the methods, three, all in Manager and Employee, No, Yes, new implementation, signature must be the same, broad, specific, hierarchy, substitution, Employee, method does not exist locally, Manager, super.getSalary() + bonus, protected, yes, no, yes, no

Class 11 - JavaFX

Learning Objectives, Assignments, and Readings

- Understand the basics of GUI programming.

- Understand how to build an FXML-based JavaFX application, including the Model-View-Controller aspects.
    - Understand the view portion is in a .fxml file.
    - Understand the controller portion is in its own .java file.
    - Understand the business portion is in its own collection of .java files.
- Understand how to use SceneBuilder to drag and drop GUI components on a Window.
    - Understand how to name variables and handler methods in SceneBuilder.
    - Understand how to use SceneBuilder to edit a .fxml file.
    - Understand how to use SceneBuilder to extract code for the controller

JavaFX

JavaFX is Java's third generation Graphical User Interface system. Java created Abstract Windowing Toolkit (AWT) its first GUI library in 1995. AWT GUI components are rendered and controlled by the GUI components of the underlying OS. Java created Swing its second GUI library in 1997. Swing is built on top of AWT, and building a Swing GUI application often uses features from both AWT and Swing. In 2008, Java introduced JavaFX and it has several releases. JavaFX is intended to replace Swing, but both will remain for the foreseeable future[2]. JavaFX and Swing are similar, but we will use JavaFX as part of our class, labs, and projects. We will not become expert GUI developers. We will use GUI components just enough to have a basic user interface to our applications.

Scene Builder

Scene Builder is a drag-n-drop GUI builder for creating .fxml files. A .fxml file is the following.

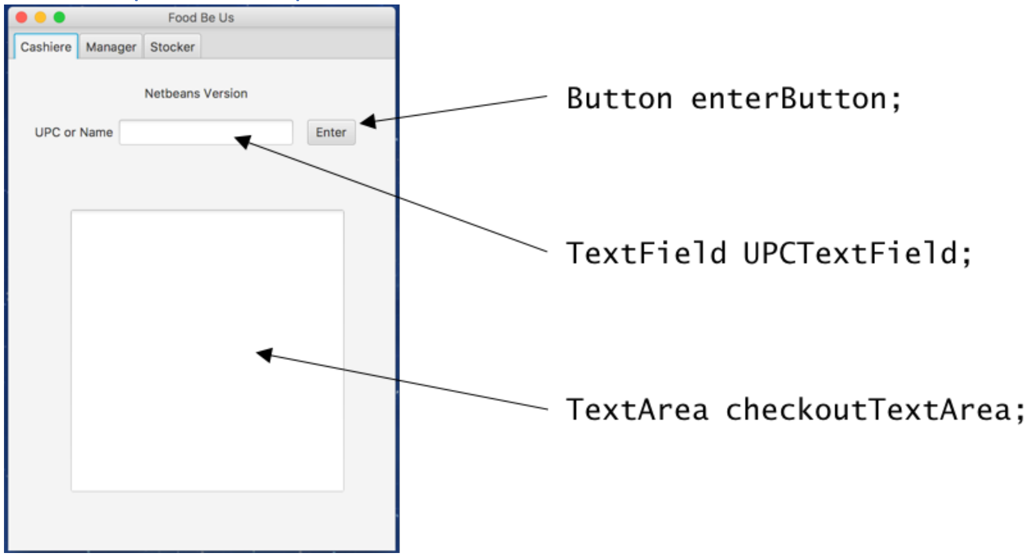
- A text file that looks like XML.
- describes the GUI components on a JavaFX program.
- Is loaded by FXMLLoader when a JavaFX program begins.

Scene Builder is free. It is maintained by Gluon. Scene Builder can be downloaded at the following link.

http://gluonhq.com/products/scene-builder/

[2] See http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html

Three Simple GUI Components



This section describes three simple GUI components and how we can use them in our Java programs. Our hands-on JavaFX lab demonstrates using these concepts. Each GUI component allows us to do the following.

- Connect a method to the component that is called on specified events. The sample diagram connects `handleEnterButton(MouseEvent event)`, to `enterButton`. The method is called when a mouse clicks in button.
- Get text from TextField and TextArea components. Getting text is typically done when a user clicks on a button and hits the enter key. In the sample diagram, the method `handleEnterButton` calls `UPCTextField.getText()` to get what user has typed.
- After processing the typed information, the resulting output is placed in the `TextArea` by calling `checkoutTextArea.setText()`. You can also call `UPCTextField.setText()`.

Class 12 – Interface Types and Polymorphism

Learning Objectives, Assignments, and Readings

- Understand the concept of polymorphism
- Understand implications of polymorphism for object oriented code
- Explain how interfaces are useful in designing large programs

Polymorphism

Polymorphism is the ability to select the appropriate method for an object based on context.

Today we explore this concept and why it is helpful in programs.

Collections Class with Comparable Interface Example

Consider the `Collections` class in the Java library. The `Collections` class contains a static method called `sort()` which can sort an array list

```
      Collections.sort(list);
```

The objects in the list can be of any type if the type implements the `Comparable` interface type.

This allows the program to be sure that _____________________________________

EX:
```
public interface Comparable<T> {
      int compareTo(T  other);
}
```

So, what is an interface type anyway?

Interface: ____

It's like a stub that provides some information about a type, but allows you (the interface user) the flexibility of making implementation choices.

An interface allows you to create a _______________________ for a datatype. You define the methods that must be included, but don't include __________________ or __________________________________.

Obviously, you cannot use this sort of thing on its own, but you can use it as a template to design a class that fits with some existing code.

You can write multiple classes that can implement the same interface – each implementing its operations in its own way. This helps to make code _____________________________.

The `Comparable` interface is a generic type. When you define the interface, specify the type to be used.

EX:
```
Comparable<Student> aStudent;
Comparable<Student> anotherStudent;
//then you can say
 if (aStudent.compareTo(anotherStudent) > 0)
        //swap position of aStudent and anotherStudent in list
```

Just FYI, `compareTo()` is set up to return a negative number if the calling object should come before the parameter object, zero if they are equal, and a positive number if the calling object should come after the parameter object. (This is a standard programming convention for this type of comparison method.)

Having this type of flexible interface associated with the task of sorting is convenient because sorting always involves a comparison like the one shown above. If your data type implements the Comparable interface, then it will play well with the sorting processes that are already written in the Java library. What does it look like when you write a class that implements an interface?

```
public class Student implements Comparable<Student> {
    private String name;
    private int ID;
    public Student (String aName, int anID) {
        name = aName;
        ID = anID;
    }
    public String getName( ) { return name; }
    public int getID( ) { return ID; }
    public int compareTo(Student other) {
        return ID - otherID;
    }
}
```

How to use

```
import java.util.*;
public class StudentSortTester {
    public static void main (String [] args) {
        ArrayList<Student> students = new ArrayList<Student>();
        students.add(new Student (“Seinfeld”, 456789));
        students.add(new Student(“Costanza”, 123888));
        students.add(new Student(“Kramer”, 378931));
        Collections.sort(students);

        for (Student s : students)
            System.out.println(s.getName() + “ ”+ s.getID());
    }
}
```

Great but what if you want to change to sort by name instead of ID? You could change the `compareTo()` method implementation – but who wants to change their code all the time?

Collections Class with Comparator Interface Example

Another option is to use a different `sort()` method that does a different type of comparison. Implement the Comparator interface type, which requires one method

```
public interface Comparator {
      int compare(T first, T second);
}
```

Again, it will return a negative number, zero, or positive number

If `thingToCompare` is an object of a class that implements the Comparator interface type, then

```
Collections.sort(list, thingToCompare);
```

sorts the objects in the list according to the sort order that thingToCompare defines.

Example of Comparator Interface:
```
public class StudentComparatorByName implements Comparator<Student> {
      public int compare(Student s1, Student s2) {
            return s1.getName( ).compareTo(s2.getName());
      }
}
```

Then in `main()`, call

```
Comparator<Student> studentToComp = new StudentComparatorByName( );
Collections.sort(students, studentToComp);
```

Bonus: now you can use any type of object in the list (not just ones that implement the Comparable type).   Why might that be handy?

You could even write the comparator method so that it behaves differently based on a parameter.

```
public class StudentComparator implements Comparator<Student> {
public int StudentComparator(bool ascending) {
            if (ascending) direction =1;
            else direction = -1;
      }
      public int compare(Student s1, Student 2s) {
            return direction = s1.getName( ).compareTo(s2.getName( ))
      }
private int direction;
}
```

Then change how you call to change the sort direction:

```
//sort Z to A
      Comparator<Student> reverseComp = new StudentComparator(false);
//sort A to Z
      Comparator<Student> comp = new StudentComparator(true);
```

Highlights about interfaces

- Interfaces never have instance variables.   Interfaces can have named constants (declared as static final and provided with a value at declaration).
- A class can implement as many interfaces as you want it to.
  EX: `public class MarsIcon implements Icon, Shape { ….`
- An interface can extend another interface

  ```
  public interface MoveableIcon extends Icon {
      void translate(int x, int y);
  }
  ```

Classes that implement `MoveableIcon` will have to implement `translate()` and all the methods defined by the Icon interface.

Compare two elements, guarantees your class includes methods with signature defined in interface, set of expectations, method implantation, variable declarations, compatible, create your own sorting order

Class 13 – Unit Testing

Learning Objectives, Assignments, and Readings

- Understand testing and various testing types.
- Understand unit testing.
- Understand testing frameworks.
- Understand JUnit.

Testing

Testing software is just as important as implementing it. Recall the software development process has an implementation phase and a testing phase. Also recall that in an agile software development process there is an oscillation between implementation and testing. The various types of testing that occurs are described in the following subsections.

Unit Testing

Unit testing tests an individual unit of code. In Java, you perform unit testing on a _________ __________, by constructing objects and calling all methods in the class. Often a method requires several __________ ___________ to test it. This class focused on unit testing.

Integration Testing

Integration testing tests the entire software system. All classes that comprise the system are built into an executable image, which is tested. Integration testing is typically performed from a ___________ perspective.

Code Coverage Testing

Code coverage testing runs test cases and collects _____________ about the various lines of code that have been executed. Code coverage testing can include unit and integration tests.

Performance Testing

Performance testing measure the performance of software. Performance testing includes such efforts as determining ___________ ____________ of the algorithm, how much memory code uses, and how much file I/O the code performs. The results from performance testing can be used to provide system requirements for running the software.

Test Driven Development

Test driven development is a ______________ ________________ _____________ where you oscillate between programming a class and programming the unit test cases for the class. IntelliJ and NetBeans support this oscillation between your classes and JUnit classes.

Java Unit Testing

JUnit

JUnit is a testing framework for performing unit testing on Java classes. JUnit is open source host on GitHub – https://github.com/junit-team/junit5. The testing framework consists of the following.

- JUnit Package – A package of classes that can be used to construct tests.
- Annotations – A collection of annotations used to annotate methods. The most used annotation is ________________ that identifies a method as a test case.

```
@Test
void getSalePrice() {
   BikePart bp1 = new BikePart("saddle1","1234567890","45.49", "40.00",
                                        false);
   assertEquals("40.00", bp1.getSalePrice());
}
```

- Assertion Methods – A collection of assertion methods that can be used to assert ___________ ___________ that are compared to the ____________ _____________.
- Test Runner – A mechanism to run the tests. IntelliJ, NetBeans, and Eclipse have built in test runners. You can also run JUnit from the command line.

IntelliJ JUnit Testing

The following video shows the basics of using JUnit with IntelliJ.
https://www.youtube.com/watch?v=Bld3644bIAo

Netbeans JUnit Testing

The following video shows the basics of using JUnit with NetBeans.
https://www.youtube.com/watch?v=Q0ue-T0Z6Zs

JUnit Annotations

- `@test` – Identifies a test method.
- `@test (expected = Exception.class)` – Fails if the method does not throw the named exception.
- `@test (timeout = 100)` – Fails if the method takes longer than 100ms.
- `@BeforeClass` – Static methods are executed one time before that start of tests.
- `@AfterClass` – Static methods are executed one time after all tests.
- `@Before` – Method is executed before each test case. For example, you may open a file and read some information.
- `@After` – Method is executed after each test case. For example, you may close a file.

```
@Before
public static void setUp(){}
@BeforeClass
public static void setUpClass() throws Exception {}
@AfterClass
public static void tearDownClass() throws Exception {}
```

JUnit Assertions

In each of the following assertions, message is an optional message displayed when the assertion fails.

- `assertEquals([message,] expected, actual)` – check expected and actual are same value.
- `assertEquals([message,] expected, actual, tolerance)` – check expected float/double and actual float/double are same to a tolerance of number of decimal digits.
- `assertFalse([message,] boolean)` – check boolean expression if false.
- `assertTrue([message,] boolean)` – check boolean expression is true.
- `assertNull([message,] object)` – compares object to null
- `assertNotNull([message,], object)` – compares object to not null
- `assertSame([message,] expected, actual)` – checks expected and actual refer to same object
- `assertNotSame([message,] expected, actual)` – checks expected and actual refer to differen objects

Pre-conditions and Post-conditions

Pre-conditions and post-conditions are assertions, that are sort of related to the JUnit assertions; however, pre-conditions and post-conditions use the Java `assert` statement. Pre-conditions tell what is expected of parameters when a method is called. Post-conditions check what must true when the method is finish. In both cases, when the `assert` statement fails, it throws an `AssertionError`. The Java `assert` statement can be turned-off at compile time. This means they will not be included in the resulting code. For this reason, Java does not recommend using assertions on parameters of public methods. For public methods that check parameters, Java recommends that you explicitly check parameters and throw your own exceptions

A pre-condition states what must be _____________ of the parameters when a method is called for the method to operate correctly. If the precondition is not true when the method is called, then your code makes _____________________________ about the outcome.

A post-condition states what must be _________________ after the method ends, assuming the preconditions were met when the method is called.

EX:

```
private int size = board.length;
/**
   Update the position of the player to a new row/column.
   @param row  The new row location of the player
   @param column  The new column location of the player
   Precondition: The row and column params are valid
   Postcondition: Player position updated to valid row and column



*/
void updateLocation(int row, int column) {
      assert ((row < size) && (column < size) &&
              (row >=0)    && (column >=0));
      playerPositionRow = row;
      playerPositionColumn = column;
}
```

Syntax:

```
      assert (condition)
```

If the condition is true, execution continues.
If the condition is false, an `AssertionError` is thrown and execution stops.

Assertions can be turned on/off ___________________________.
Assertions are primarily helpful during debugging.

single class, test cases, user, statistics, execution speed, software development process, @test, expected values, actual values, true, no guarantee, true

Class 15 – Version Control

Learning Objectives, Assignments, and Readings

- Understand version control.
- Understand Git.
- Understand GitHub.

Version Control

Version control is keeping track of various versions of your software. Everyone has updated their phone apps to newer versions. When Facebook, or any app, fixes bugs and/or adds new features to their software, they create a new version. You can create versions during your development. Suppose you are developing a Bicycle Warehouse program, and your first goal is to simply load the main warehouse with bicycle parts from a warehouse delivery file. You could implement these capabilities and create a version before adding new features. A version control tool allows you to select the code from any of its saved versions. Being able to revert to a previous version of your software is a convenient feature. The following are a couple of examples.

- If you are adding new capabilities and accidently delete good lines of code, you use the version control tool to retrieve the previous version that had those lines of code.
- If you are testing and discover a bug that was not present earlier in your development, you can examine previous versions to see when you introduced the bug.

Manual Version Control

Suppose you have a software project in a folder – `MyBicycleWareHouse`. This folder may have subfolders that represent packages, documentation, and input/output files. Suppose you have just finished developing and testing Version 1.0. You could manually perform version control by copying `MyBicycleWareHouse` and all subfolders to a version 1.0 folder. For example, using a Linux recursive copy command.

```
$ cp –R MyBicycleWareHouse MyBicycleWareHouseVersion1.0
```

After this command, the folder `MyBicycleWareHouseVersion1.0` contains version 1.0 of your software project. You can now develop Version 2.0 in the original `MyBicycleWareHouse` folder.

Git - a shortened version of http://gitref.org

Git is a version control tool that stores snapshots of our project, where a snapshot is like copying the entire project; however, the project is not really copied. Instead, Git records a manifest of what your project files look like at the point the snapshot is taken. The manifest is stored in the .git folder. The following discussion uses Linux commands to demonstrate the concepts; however, you will perform them using your IDE. For example, instead of typing `git commit` you may select a button to perform the commit, which will open an edit window for you message.

Git – Creating Git Repository

Suppose Suzy has a folder, `gitPractice`, in which she has a few files – `README`, `BikePart.java`, `BikePartTester.java`. You create the Git repository using the `git init` command.

```
$ pwd
gitPractice
$ ls
BikePart.java     BikePartTester.java     README
$ git init
$ git config –global user.name “Suzy”
$ git config –global user.email “suzy@email.com”
$ ls –a
.                 .git               BikePartTester.java
..                BikePart.java      README
```

If you examine the status of your Git repository at this point, you will see you have three files that are not being tracked.

```
$ git status –s
?? BikePart.java
?? BikePartTester.java
?? README
```

You must add the files to your Git repository for Git to begin tracking them.

Git – Adding and Committing Files to Git Repository

A Git repository has a two-stage process for placing files in it. First files are added to the repository using `git add` and then files are committed to the repository using `git commit`. When you commit files to the repository, you include a message describing the commit. The example commit shown uses the `-m 'message here'`, but `git commit` allows entering a message (comment) via an editor when the `-m` is omitted.

```
$ git add README BikePart.java BikePartTester.java
```

Alternatively, you can add all files in a folder with the following.

```
$ git add .
```

You can see the status of your repository with `git status -s`.

```
$ git status -s
A BikePart.java
A BikePartTester.java
A README
```

You can also examine status without the –s.

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   BikePart.java
      new file:   BikePartTester.java
      new file:   README
```

To complete the second stage, you perform the following command.

```
$ git commit -m 'Demonstration of Git'
[master (root-commit) d4e0a98] Demonstration of Git
 3 files changed, 11 insertions(+)
 create mode 100644 BikePart.java
 create mode 100644 BikePartTester.java
 create mode 100644 README
```

Git, Remote Repositories, and GitHub

We want to store our local Git repository on a remote Git repository, and GitHub[3] (https://github.com) is where we will store it. GitHub provides free storage of public repositories, which means they are open for anyone to view. If you were a company, you would pay for a private repository.

In this scenario, there are two teammates who will use one remote GitHub repository to work on a common project. Both teammates create an GitHub accounts, and one of the teammates creates

[3] There are other alternatives such a BitBucket, but we will use GitHub.

a repository in which to store the project code. The a GitHub repository that parallels the local Git repositorys. The teammates use `git push` and `git pull` commands to move data between local repositories and the Github repository.

Teammate Suzy created the initial code as described in the previous sections. Suzy's local Git repository is in `gitPractice` folder on her computer. Suzy's GitHub account is suzy, and she created a BikePart repository. Suzy's BikePart repository URL is https://github.com/suzy/BikePart.git. Suzy connects her GitHub repository to her local repository as follows.

```
$ git remote add origin https://github.com/suzy/BikePart
$ git remote -v
origin      https://github.com/suzy/BikePart (fetch)
origin      https://github.com/suzy/BikePart (push)
```

In this example, `origin` is the name of the remote repository that is used by git commands. Origin is a typical Git name for the remote repository. We could have named the remote repository something else as follows.

```
$ git remote add ongithub https://github.com/suzy/BikePart
```

Now we can push and fetch from the local Git repository to the GitHub Git repository. If I attempt to push at this point, I get an error because my GitHub repository has a README file that is not in my local repository.

```
$ git push origin master
```

At this point, your GitHub repository has the files that were in your local Git repository. Your teammates can now clone the GitHub repository onto their computers. They can create a folder cd into the folder, use the `git clone` command, which creates a local Git repository with BikePart.java and BikePartTester.java.

```
$ mkdir gitPractice2
$ cd gitPractice2
$ git clone https://github.com/suzy/BikePart .
$ ls
BikePart.java           BikePartTester.java     README
```

You can see that this remote is pointing to the same GitHub repository as previous.

```
$ git remote -v
origin      https://github.com/gustycooper/BikePart (fetch)
origin      https://github.com/gustycooper/BikePart (push)
```

The teammate can make changes and push them to GitHub. I edit BikePart.java and perform the git status.

```
$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   BikePart.java

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a
[master ee28c5f] Updated BikePart.java from a teammate account.
 1 file changed, 1 insertion(+)
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 397 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/gustycooper/BikePart
   4ae79dd..ee28c5f  master -> master
```

Now I can return to the original `gitPractice` folder and perform a `git pull origin master`, which will update the files to match those on GitHub. Notice `BikePart.java` has changed.

```
$ cd ../gitPractice
git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/gustycooper/BikePart
 * branch            master     -> FETCH_HEAD
   4ae79dd..ee28c5f  master     -> origin/master
Updating 4ae79dd..ee28c5f
Fast-forward
 BikePart.java | 1 +
 1 file changed, 1 insertion(+)
```

Now, suppose the teammate working in `gitPractice` adds a new file, `LoginAccount.java`.

```
$ git add LoginAccount.java
$ git commit -m ‘Added LoginAccount to project’
$ git push origin master
```

The teammate working in gitPractice2 can perform a git pull to get the latest version of files on the GitHub repository.

```
$ cd ../gitPractice2
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
```

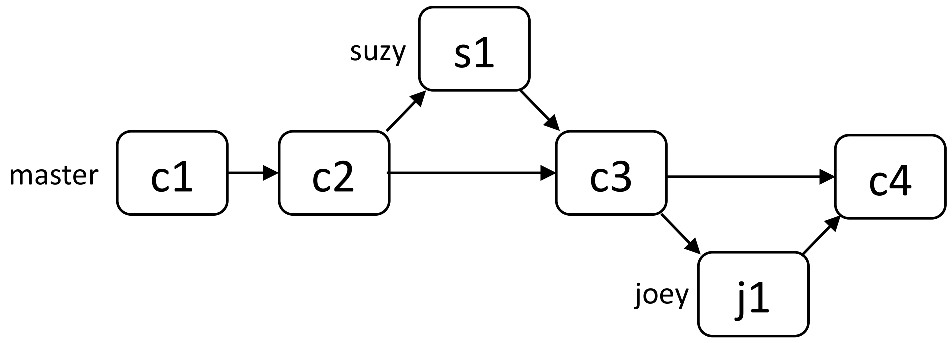
```
Unpacking objects: 100% (3/3), done.
From https://github.com/gustycooper/BikePart
 * branch            master     -> FETCH_HEAD
   ee28c5f..b940545  master     -> origin/master
Updating ee28c5f..b940545
Fast-forward
 LoginAccount.java | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 LoginAccount.java
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
```

Git Use and Teammate Coordination

The previous sections describe a way that teammates can use Git and GitHub to develop a project. This is a simple way, but it requires coordination between teammates. The approach works best when teammates do not update the same files concurrently. The team would divvy work such that teammates are working on different files. They could message each other when their files are ready for testing, allowing teammates to perform a git pull to get the latest files.

Git Branches

In this scenario, Suzy (in gitPractice) creates a branch names suzy; and Joey (in gitPractice2) creates a branch named joey. First Suzy edits BikePart.java on her branch, and then Joey edits BikePartTester.java in his branch. Suzy and Joey push their branches to GitHub, which results in three branches – master, suzy, and joey. At GitHub we perform one pull request to merge suzy branch into master and a second pull request to merge joey into master. We also delete suzy and joey from GitHub. GitHub's master has changes from both Suzy and Joey. Suzy and Joey then pull master into their local Git repositories and continue working. The following figure demonstrates this concept.



Suzy's Local Git Branch - suzy

```
$ git branch suzy
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ git branch
* master
  suzy
$ git checkout suzy
Switched to branch 'suzy'
```

```
$ git status
On branch suzy
nothing to commit, working tree clean
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ vim BikePart.java
$ git status
On branch suzy
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   BikePart.java

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a
[suzy 2b9f71c] Changed BikePart.java on suzy branch.
 1 file changed, 1 insertion(+)
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ git status
On branch suzy
nothing to commit, working tree clean
```

Pushing Suzy's Local Git Branch to GitHub

```
$ git push origin suzy:suzy
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 354 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/gustycooper/BikePart
 * [new branch]      suzy -> suzy
```

Joey's Git Branch

At this point, the GitHub repository has two branches, master and suzy. The following is Joey changing BikePartx.java.

```
$ cd ../gitPractice2
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ git branch joey
$ gits status
-bash: gits: command not found
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
$ git checkout joey
M     BikePartTester.java
Switched to branch 'joey'
$ vim BikePartTester.java
$ git status
```

```
On branch joey
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   BikePartTester.java

no changes added to commit (use "git add" and/or "git commit -a")
Gustys-iMac:gitPractice2 gusty$ git commit -a
[joey 11b7c62] Changed BikePartTester.java on joey branch.
 1 file changed, 1 insertion(+)
$ ls
BikePart.java		BikePartTester.java	LoginAccount.java README
$ git status
On branch joey
nothing to commit, working tree clean
```

Pushing Joey's Local Git Branch to GitHub

```
$ git push origin joey:joey
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 333 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/gustycooper/BikePart
 * [new branch]      joey -> joey
```

Merging Branches on GitHub

At this point, the GitHub repository has two branches, master, suzy, and joey. You can select the suzy branch on GitHub and select create pull request. This should say suzy and master can be merged without conflicts. Once the pull request is done, you will have Pull requests 1 on the top navigation bar of your GitHub repository. You can select the joey branch on GitHub and select create pull request. This should say joey and master can be merged without conflicts. Once the pull request is done, you will have Pull requests 2 on the top navigation bar of your GitHub repository.

Select the Pull request 2 option on the top navigation bar, which shows a page with describing the two branches. Select the "Changed Bikepart.java on suzy branch." Perform the pull request, which will merge the suzy branch onto the master branch. After successful merge, select Delete Branch to delete the suzy branch. Repeat this with joey branch.

At this point the GitHub repository is a single branch – master – that has the updates that Suzy and Joey made. NOTE: Suzy and Joey worked on different files. If they had edited the same file, GitHub can still perform the merge if their edits did not conflict with each. If the edits conflicted, you would have to decide which of the edits to keep/discard.

Suzy Updating Her Local Git Repository

At this point, Suzy and Joey want to get the latest version of master into their local repositories. Suzy does this by the following.

```
$ cd ../gitPractice
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ git status
On branch suzy
nothing to commit, working tree clean
$ git checkout master
Switched to branch 'master'
$ git branch
* master
  suzy
$ git pull origin master
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 3), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/gustycooper/BikePart
 * branch            master     -> FETCH_HEAD
   b940545..71e41f9  master     -> origin/master
Updating b940545..71e41f9
Fast-forward
 BikePart.java       | 1 +
 BikePartTester.java | 1 +
 2 files changed, 2 insertions(+)
$ git branch -d suzy
Deleted branch suzy (was 2b9f71c).
```

Suzy Updating Her Local Git Repository

Joey gets the latest version of the GitHub master into his local repository as follows.

```
$ cd ../gitPractice2
$ ls
BikePart.java           BikePartTester.java     LoginAccount.java README
$ git status
On branch joey
nothing to commit, working tree clean
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git branch
  joey
* master
$ git pull origin master
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 2), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/gustycooper/BikePart
 * branch            master     -> FETCH_HEAD
   b940545..71e41f9  master     -> origin/master
Updating b940545..71e41f9
Fast-forward
 BikePart.java       | 1 +
 BikePartTester.java | 1 +
 2 files changed, 2 insertions(+)
$ git branch -d joey
Deleted branch joey (was 11b7c62).
```

GitHub and .gitignore

Often you have more files in your local git repository than you want to store on GitHub. The `.gitignore` file is used to define which files are **NOT** pushed from your local repository to GitHub. Your IDE has pattern `.gitignore` files. The following is some of a `.gitignore` that I have for a Java project developed in IntelliJ.

```
# Package Files #
*.jar
*.war
*.ear
*.zip
*.tar.gz
# .gitignore
.gitignore
# IntelliJ folders
.idea/
out/
```

IntelliJ, Git, and GitHub

Using Git Integration with IntelliJ
https://www.jetbrains.com/help/idea/2017.1/using-git-integration.html#d465818e14
Setting up a local repository
https://www.jetbrains.com/help/idea/2017.1/setting-up-a-local-git-repository.html
Adding files to a local repository.
https://www.jetbrains.com/help/idea/2017.1/adding-files-to-a-local-git-repository.html
Committing changes to a local repository.
https://www.jetbrains.com/help/idea/2017.1/committing-changes-to-a-local-git-repository.html
Registering you GitHub account with IntelliJ
https://www.jetbrains.com/help/idea/2017.1/registering-github-account-in-intellij-idea.html
NOTE: I used the recommended Token approach.
Managing remotes – GitHub
https://www.jetbrains.com/help/idea/2017.1/managing-remotes.html
Push Dialog
https://www.jetbrains.com/help/idea/2017.1/push-dialog-mercurial-git.html

Module 4 – Design Patterns

Module 4 Overview

Design patterns are an important concept for programmers. A design pattern provides a best practice solution to well-known problems. We study an overview of design patterns along with a detailed analysis of several design patterns such as the following.

- Strategy Pattern
- Observer Pattern
- Model View Controller Pattern
- Template Pattern
- Decorator Pattern

References

The following are references for topics related to working in groups and design patterns.

Google's Unconscious Bias Video
https://www.youtube.com/watch?v=_KfKmGb_bT4

Tips for working in a group
https://www.cs.cmu.edu/~pausch/Randy/tipoForGroups.html

Pair Programming
https://www.youtube.com/watch?v=YhV4TaZaB84

Interview w/UMW Alum, Lucy Bain – two parts
https://www.youtube.com/watch?v=cl1PTUQvcX0
https://www.youtube.com/watch?v=bM0qnbNbIuM

Atlassian Pair Programming (Lucy Bain)
https://www.youtube.com/watch?v=fQ-x-T34z9w

Abstract Classes and Interfaces
https://www.youtube.com/watch?v=AU07jJc_qMQ

Why/When to Use Abstract Classes
https://www.youtube.com/watch?v=yyU3bXyc_oU

Template Method Pattern
https://www.youtube.com/watch?v=aR1B8MlwbRI

Lectures

- C16DesignPatterns&StrategyPattern.docx
- C17ObserverPatternMVC.docx
- C18AbstractClasses&TemplatePattern.docx
- C19DecoratorPattern.docx

Labs

- Lab12ObserverPattern.docx
- Lab13TemplatePattern.docx
- Lab14IteratorPattern.docx

Class 16 – Design Patterns and Strategy Pattern

Learning Objectives

- Understand design patterns
- Understand why patterns are useful tools for programmers
- Understand the standard format for design patterns

Introduction to Design Patterns

What is a design pattern?

____

They were originally document by a Gamma, Helm, Johnson, & Vlissides in their book Design Patterns, which is commonly referred to as the "_______________________" in CPSC circles.

The original book contains __________ patterns.

Why should you know design patterns?

They've been proven over time to be efficient and effective for solving

____

It gives you a ____________________________________ that you can use to talk about coding problems & solutions

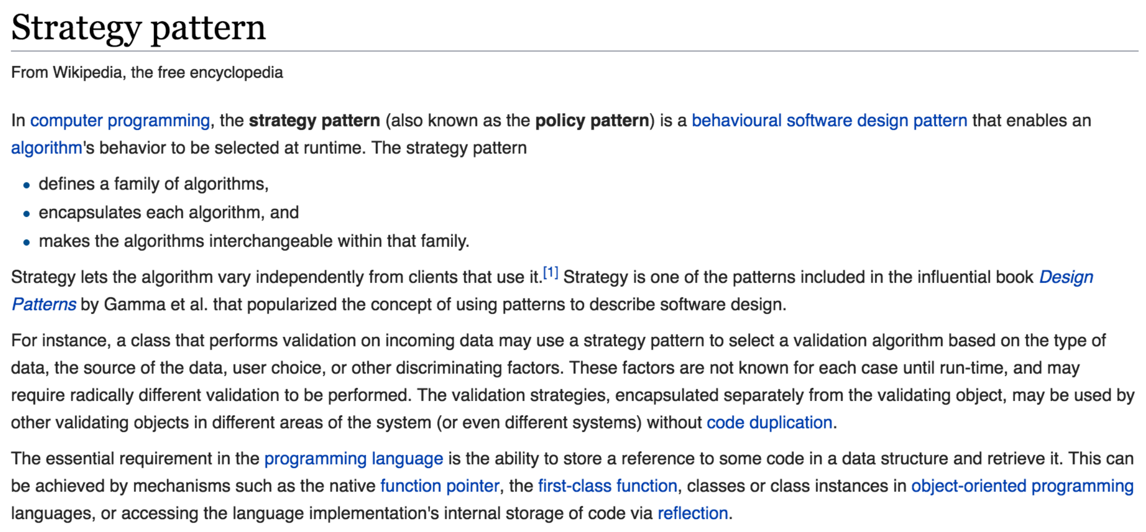
We're going to talk about a subset of these common design patterns:

- Strategy pattern
- Template Method Pattern
- Iterator pattern
- Observer pattern
- Decorator pattern
- Singleton pattern

Learning Objectives, Assignments, and Readings

- Understand what a design pattern write up looks like
- Understand the essentials of the Strategy Pattern
- Understand and recognize the Strategy Pattern in project descriptions

Strategy Pattern from Wikipedia



Strategy pattern

From Wikipedia, the free encyclopedia

In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a behavioural software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern

- defines a family of algorithms,
- encapsulates each algorithm, and
- makes the algorithms interchangeable within that family.

Strategy lets the algorithm vary independently from clients that use it.[1] Strategy is one of the patterns included in the influential book *Design Patterns* by Gamma et al. that popularized the concept of using patterns to describe software design.

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

The essential requirement in the programming language is the ability to store a reference to some code in a data structure and retrieve it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.

Strategy Pattern Write-up From: https://sourcemaking.com/design_patterns/strategy

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

Problem

One of the dominant strategies of object-oriented design is the "open-closed principle".

SIDENOTE: In object-oriented programming, the **open**/**closed principle** states "software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification"; that is, such an entity can allow its behavior to be extended without modifying its source code.

Figure 1 demonstrates how this is routinely achieved:

- encapsulate interface details in a base class, and bury implementation details in derived classes.
- Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.

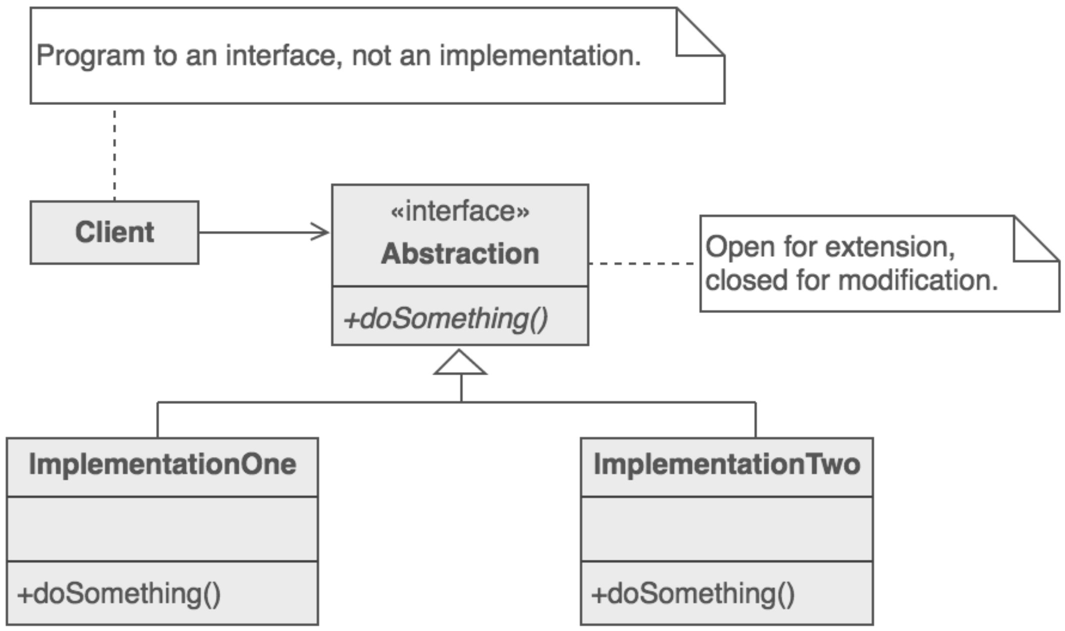


Figure 1 Class Diagram for a Typical Strategy Pattern Implementation

A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in Figure 1 is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

A more popular characterization of this "abstract coupling" principle is "Program to an interface, not an implementation".

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

Structure

The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.

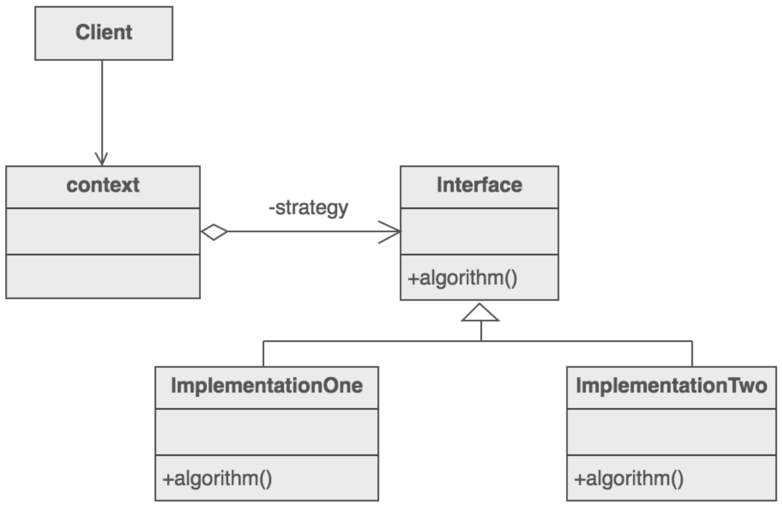


Figure 2 Class Diagram for an Alternate Implementation of the Strategy Pattern

Example

A Strategy defines a set of algorithms that can be used interchangeably.
Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on trade-offs between cost, convenience, and time.

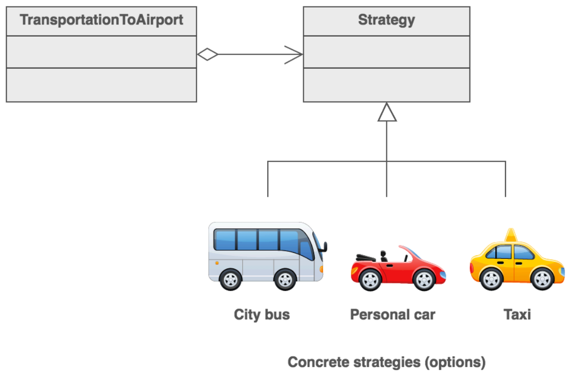
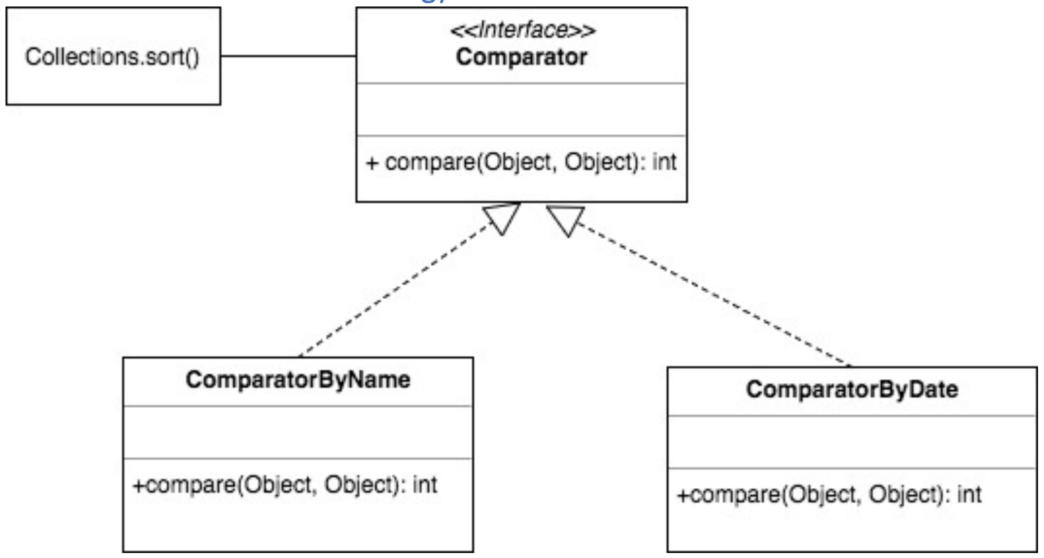


Figure 3 Example of the Strategy Pattern in Action

Check List

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.

Where Have We Seen the Strategy Pattern?


Collections.sort()
<<Interface>>
Comparator
+ compare(Object, Object): int
ComparatorByName
+compare(Object, Object): int
ComparatorByDate
+compare(Object, Object): int

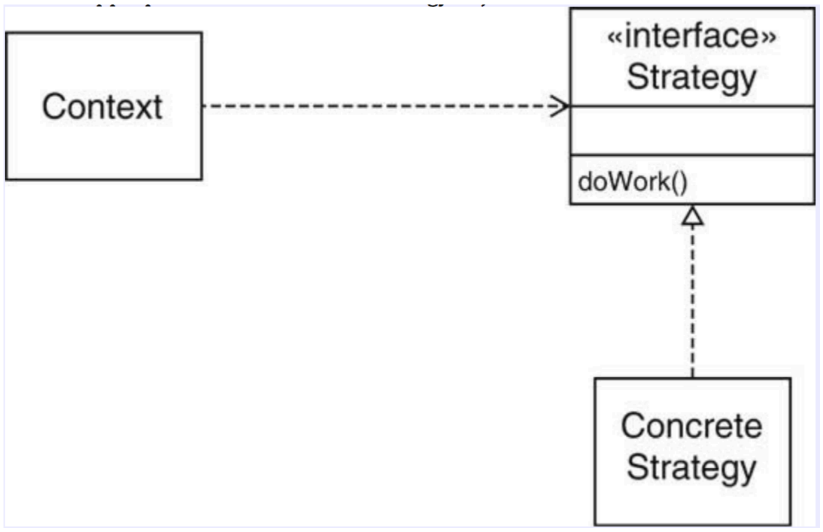
Strategy Pattern Description from Our Textbook

Pattern: Strategy
Context:

1. A class (which we'll call the context class) can benefit from different variants of an algorithm.
2. Clients of the context class sometimes want to supply custom version fo the algorithm.

Solution:

1. Define an interface type that is an abstraction for the algorithm. We'll call this interface type the strategy.
2. Concrete strategy classes implement the strategy interface type. Each strategy class implements a version of the algorithm.
3. The client supplies a concrete strategy object to the context class.
4. Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object.



In Chapter 4, you encountered a different manifestation of the STRATEGY pattern. Recall how you can pass a `Comparator` object to the `Collections.sort` method to specify how elements should be compared.

```
Comparator comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

The comparator object encapsulates the comparison algorithm. By varying the comparator, you can sort by different criteria. Here is the mapping from the pattern names to the actual names:

Name in Design Pattern	**Actual Name**
`Context`	`Collections`
`Strategy`	`Comparator`
`ConcreteStrategy`	A class that implements the `Comparator` interface type
`doWork()`	`compare()`

Best practice for solving problems, gang of four (GOF), 23, common recurring programming challenges, shared language, related, discoverable, NOT, part of API,

Class 17 – Observer Pattern and Model-View-Controller Architecture

Learning Objectives

- Understand the concept of a Model-View-Controller Architecture. Explain the role of the model, the view, and the controller.
- Understand the benefits of the MVC architecture
- Understand the Observer Pattern and consider examples of how it might be applied
- Perform in-class reflective activity.

In-class Reflective Activity

For the upcoming group project, write your reflections on the following two considerations.

- What are you concerned about regarding your ability to contribute to your group?
- What makes you a great group member?

Observer Pattern

The observer pattern is the idea behind the **Model-View-Controller (MVC) Architecture**

- The ______________________________ holds the information in a data structure

  (array, tree, queue, ….). This is just the ________________. It does not have a visual

  appearance. The model knows nothing of how it __________________________.

- Other objects, ______________________, draw the visible parts of the data, in a format specific to the view. (For example, you might display data as a graph or bar chart or table).

  The views are interested in __________________________ of the model. The views don't know about the controllers.

- Each view also has a _______________________, an object that processes user

  interaction (____________________________________, etc).

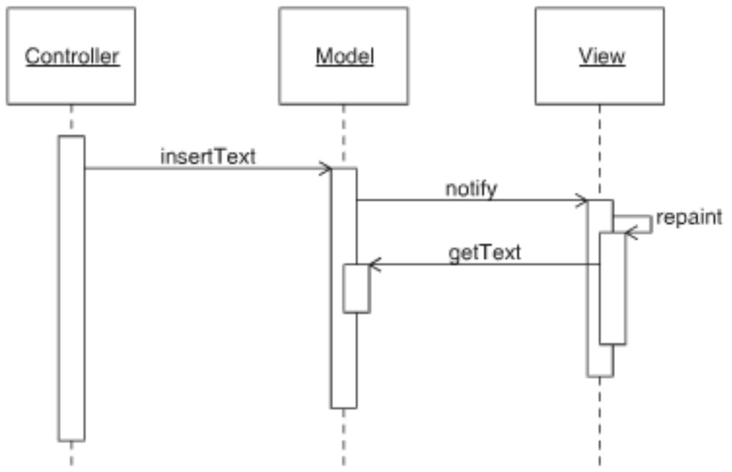
- The model knows about the number of views. Model knows nothing in detail about the observers except that it should notify them whenever the model data changes.

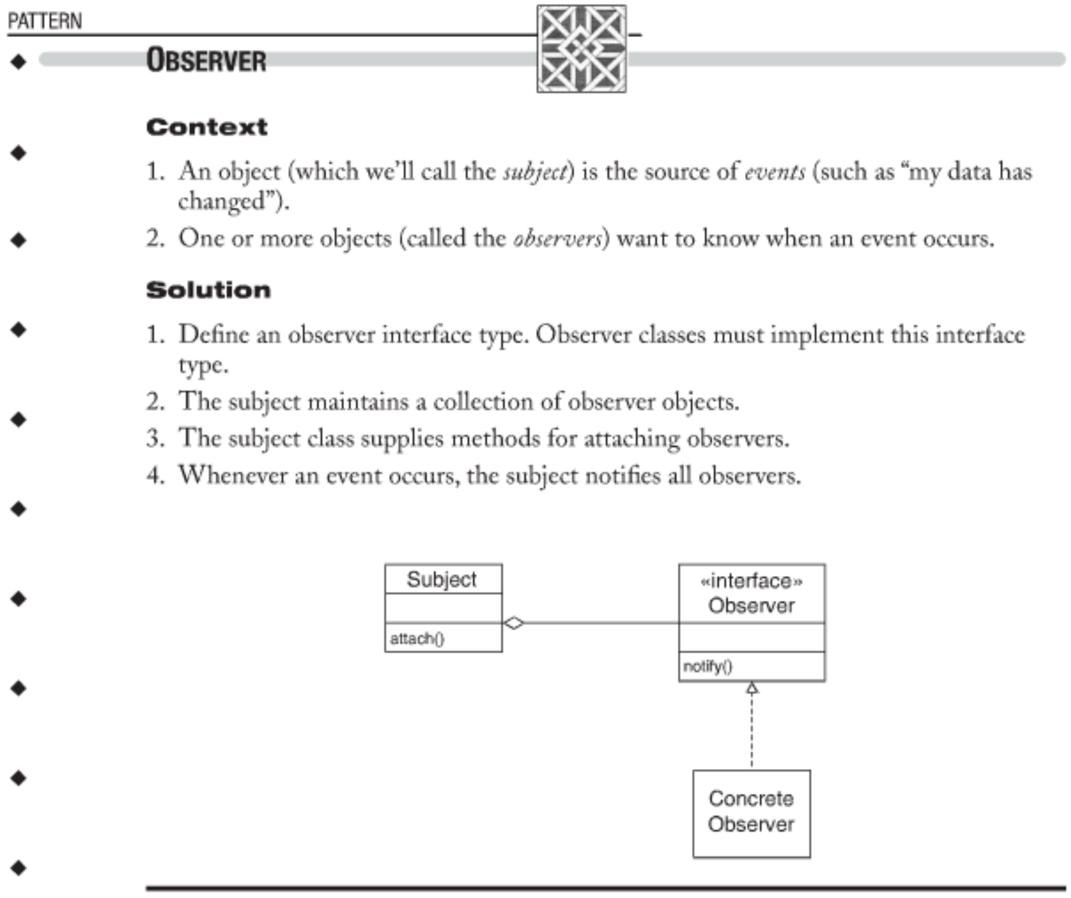
Why is this a great thing?

This makes it easy to
____.

It also makes it easy to support multiple
____.

EX: Suppose you want to add text to your user interface





PATTERN

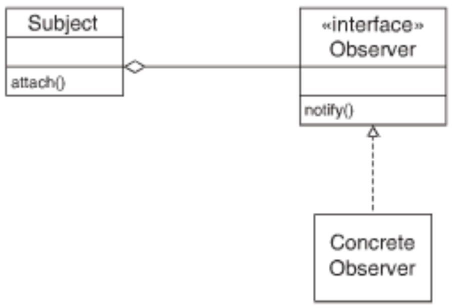
OBSERVER

Context

1. An object (which we'll call the *subject*) is the source of *events* (such as "my data has changed").
2. One or more objects (called the *observers*) want to know when an event occurs.

Solution

1. Define an observer interface type. Observer classes must implement this interface type.
2. The subject maintains a collection of observer objects.
3. The subject class supplies methods for attaching observers.
4. Whenever an event occurs, the subject notifies all observers.



This pattern is used frequently in graphics/user interface design.   EX: ActionListener class in the Java API.

Observer Design Pattern & Examples

from http://www.java2blog.com/2013/02/observer-design-pattern-in-java.html

As the name suggests it is used for observing some objects. Observers watch for any change in state or property of subject. Suppose you are interested in particular object and want to get notified when its state changes. Then you observe that object and when any state or property change happens to that object, it notifies you.

As described by GoF:
"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically".

You can think of observer design pattern in two ways

- **Subject-Observer relationship**: The object, which is being observed, is referred to as the Subject and the classes, which observe the subject, are called Observers.

- **Publisher-Subscriber relationship**: A publisher is one who publishes data and notifies the list of subscribers who have subscribed for the same to that publisher. A simple example is a Newspaper. Whenever a new edition is published by the publisher, it will be circulated among subscribers whom have subscribed to publisher.

The observers will not monitor the object for state since observers will be notified for every state change of subject, until they stop observing subject. So the subject (or publisher) follows the Hollywood principle: "Don't call us, we will call you".

Some real life examples of Observer Pattern

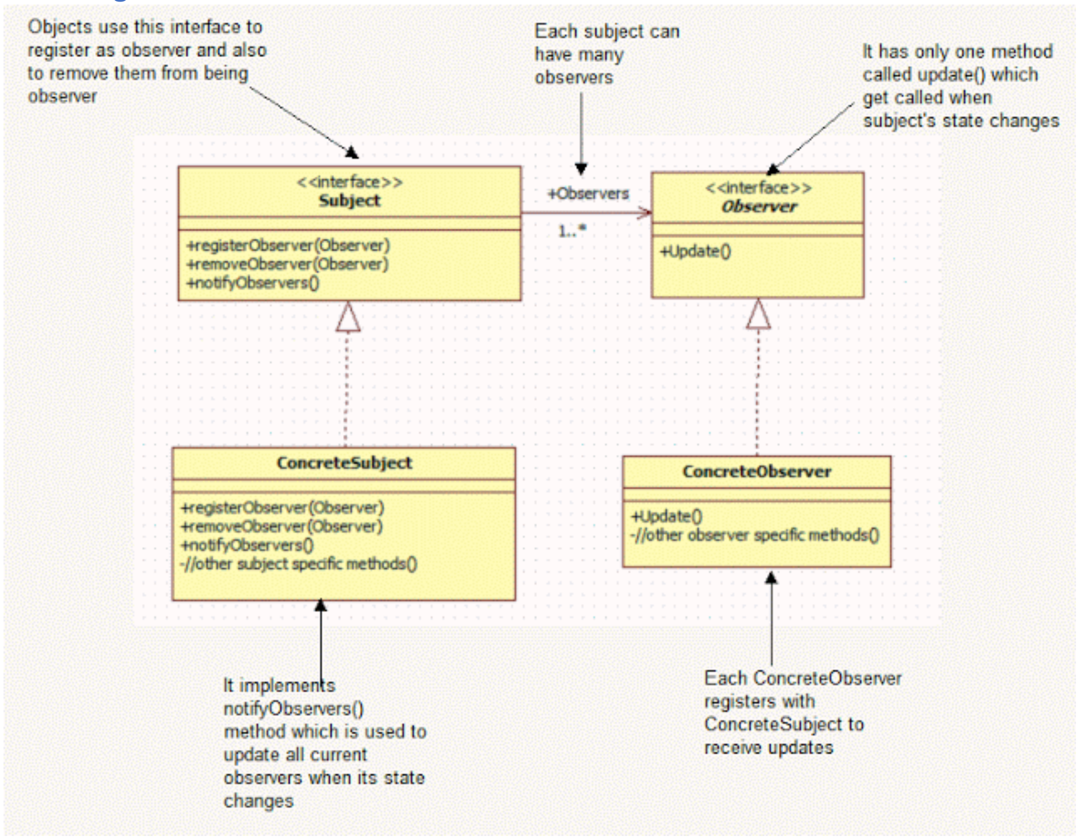
Suppose you're shopping online. Some sites are setup so that when you search for a product and it is unavailable, you see an option to "Notify me when product is available." If you click this option and provide your email address, you subscribe that product. When the state of product changes (i.e. it becomes available), you will get notification email. In this case, the Product is the subject and you are an observer.

Think about Facebook. If you subscribe to someone (by putting them on your friend or close friend list), then whenever they post new updates, you will be notified.

When to use the Observer Pattern:

- When one object changes its state, then all other dependent objects must automatically change their state to maintain consistency.
- When the subject doesn't know about the number of observers it has.
- When an object should be able to notify other objects without knowing who those objects are.

UML Diagram for Observer Pattern


Objects use this interface to register as observer and also to remove them from being observer
Each subject can have many observers
It has only one method called update() which get called when subject's state changes
<<interface>> Subject
+registerObserver(Observer)
+removeObserver(Observer)
+notifyObservers()
+Observers
1..*
<<interface>> Observer
+Update()
ConcreteSubject
+registerObserver(Observer)
+removeObserver(Observer)
+notifyObservers()
-//other subject specific methods()
ConcreteObserver
+Update()
-//other observer specific methods()
It implements notifyObservers() method which is used to update all current observers when its state changes
Each ConcreteObserver registers with ConcreteSubject to receive updates

Components:

Subject

- Knows its observers
- Has any number of observer
- Provides an interface to attach and detaching observer object at run time

Observer

- Provides an update interface to receive signal from subject

ConcreteSubject

- Stores state of interest to ConcreteObserver objects.
- Sends notification to it's observer

ConcreteObserver

- Maintains reference to a ConcreteSubject object
- Maintains observer state consistent with subjects.
- Implements update operation

Model, data and business algorithms, will be deployed, the view, getting updates, controller, menus-dialogs-mouse movement, update the three pieces, views,

Class 18 – Abstract Classes and Template Pattern

The topics today are a little disjoint. They wrap up our coverage of relationships between classes and how to build extendable code

Learning Objectives

- Understand the concept of an abstract class.
- Understand similarities and differences between abstract classes and interface types
- Understand the purpose of the Template pattern. Identify times when you might use this in a program.
- Understand the side effects of making data and/or methods protected rather than private
- Understand the term refactoring

Abstract classes

Abstract classes are a sort of blend between an ______________________ and a ________________________________.

Abstract classes are classes that can contain ____________________________________ like a regular class.

However, unlike a regular class not all of the methods contain an _________________________________.
Some of them are just headers (like the methods in an interface).

RULE: You cannot ____ an abstract class. Why?

You can inherit from an abstract class and when doing so, must ____.

It's also legal to have variables whose type is an abstract class.

Example:
`SelectableShape` is an abstract class. `HouseShape()` is a class that extends the `SelectableShape` class. It's legal to say:

```
	SelectableShape shape = new HouseShape();
	//legal because SelectableShape methods are defined in HouseShape
```

Why would we use an abstract class?

They are convenient placeholders for factoring out
_______________________________________.

So why bother with interfaces? Why not just have abstract classes?
A class can only extend _______________ abstract class, but it can implement
______________ interface types.

In the API, an interface type and an abstract class are often supplied in pairs.

Template Method Pattern

Idea: Supply an algorithm for __________________________, provided that the sequence of steps does not

depend on ____________________________

How it works:

a _________________________ defines a method that calls primitive operations that a _________________ needs

to supply. Each subclass can supply the primitive operations, as is most appropriate for it. The

template method contains the knowledge of _______________________________________ the primitive operations into a more complex operation.

You should use this pattern if:

- You have a set of subclass methods that are almost ______________ in implementation
- You can express the differences between the methods as another method. If you can, then

  move the common code into a _________________. Call the method for the variant part.

You should use this pattern if:

- You have a set of subclass methods that are almost ______________ in implementation
- You can express the differences between the methods as another method. If you can, then

  move the common code into a _________________. Call the method for the variant part.

PATTERN

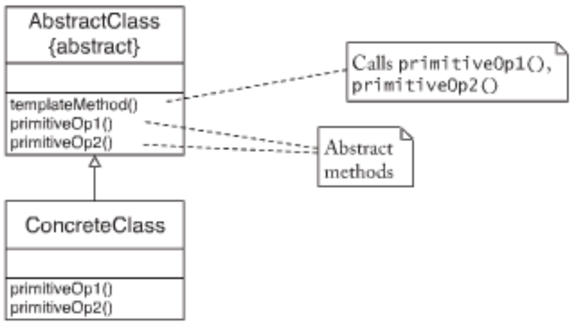
Template Method

Context

1. An algorithm is applicable for multiple types.
2. The algorithm can be broken down into *primitive operations*. The primitive operations can be different for each type.
3. The order of the *primitive operations* in the algorithm doesn't depend on the type.

Solution

1. Define an abstract superclass that has a method for the algorithm and abstract methods for the primitive operations.
2. Implement the algorithm to call the primitive operations in the appropriate order.
3. Do not define the primitive operations in the superclass or define them to have appropriate default behavior.
4. Each subclass defines the primitive operations but not the algorithm.



You should use this pattern if:

- You have a set of subclass methods that are almost ________________ in implementation
- You can express the differences between the methods as another method. If you can, then

  move the common code into a ____________________. Call the method for the variant part.

Remember when we talked about protected?

Our book recommends against using protected methods and data because

It's impossible to predict

How can you be sure that they all need access to the superclass' data and methods?

All classes in a __________________________also get access to protected data. You might as well make it public.

Recommended:
A class can supply a public interface for all clients and a protected interface for subclasses. Why?

Refactoring

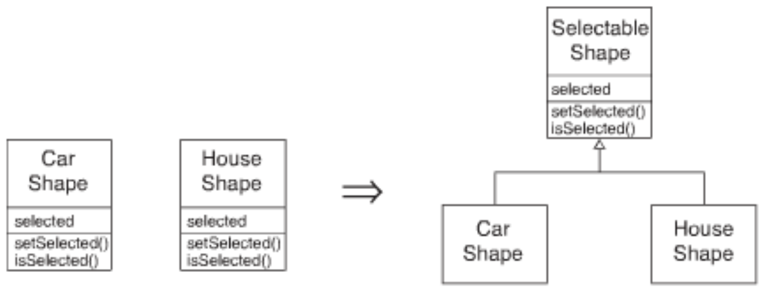
Refactoring means ___________________________ code in a ______________________________ way.

Martin Fowler (author of our UML book) created a list of rules to use when refactoring code.

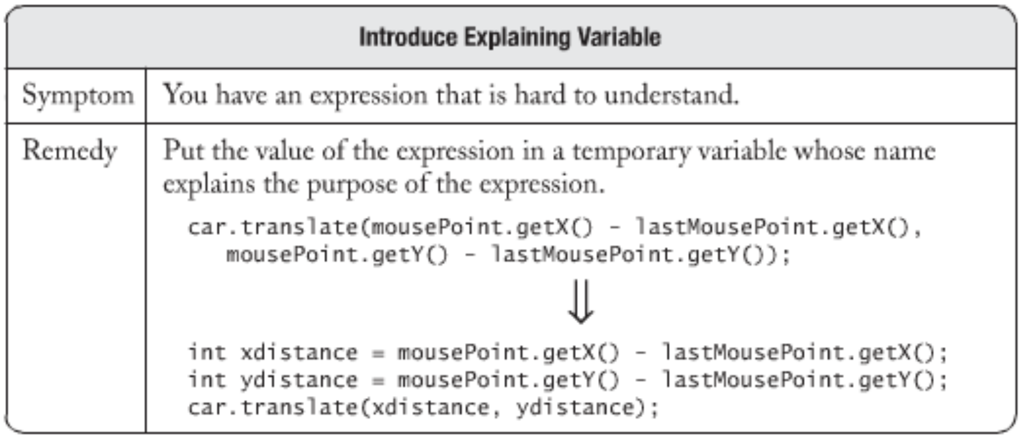
Obviously, code has to be retested after refactoring to make sure that you haven't broken anything.

Examples of rules for refactoring

1. Extract superclasses

Extract Superclass	
Symptom	You have two classes with similar features.
Remedy	Create a superclass and move the common features to the superclass. 

1. Introduce Explaining Variables



Introduce Explaining Variable	
Symptom	You have an expression that is hard to understand.
Remedy	Put the value of the expression in a temporary variable whose name explains the purpose of the expression.

```
car.translate(mousePoint.getX() - lastMousePoint.getX(),
   mousePoint.getY() - lastMousePoint.getY());
```

⇓

```
int xdistance = mousePoint.getX() - lastMousePoint.getX();
int ydistance = mousePoint.getY() - lastMousePoint.getY();
car.translate(xdistance, ydistance);
```

How are the refactoring rules different from the design patterns?

Refactoring tells you____.

Design patterns tell you how to produce
____ (so you hopefully won't need to refactor).

Interface, regular class, data-full methods-method signatures, implementation, instantiate, could call methods that do not exist, implement method signatures without code, similarities, one, multiple/many, multi-step process – create outline of steps that applies to all types, the data implementation, abstract class, subclass, combining, identical, abstract class, subclass

Class 19 – Decorator Pattern (R - 3/30)

Learning Objectives, Assignments, and Readings

- Take Quiz 8.
- Be able to explain the concept of a Decorator Pattern.
- Talk about the Decorator Pattern and consider examples of how it might be applied

The Decorator Pattern

The Decorator Pattern applies wherever a class ________________________________

of another class while ____

In other words:

The key to this pattern is that the decorated component is completely

____

EX:
Suppose there's more info than will fit in your screen; you need a way to maneuver around the screen. How is this handled in your average window?

The scroll bar adds ____ to the underlying

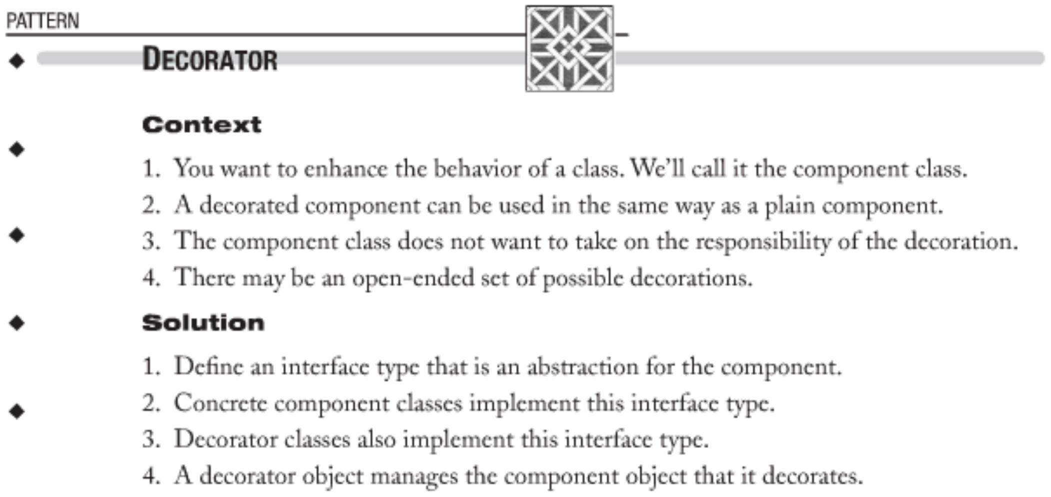
window and is therefore called a ________________________________

Note the window does nothing to acquire a scrollbar. The scrollbar is merely layered onto it.

Advantages:

1. ____
 (it would be a pain of scrollbars had to be written independently for every component - with different variables, options, etc.

2. Your component class can't ____ ways that it may need to be decorated in the future.


PATTERN

DECORATOR

Context

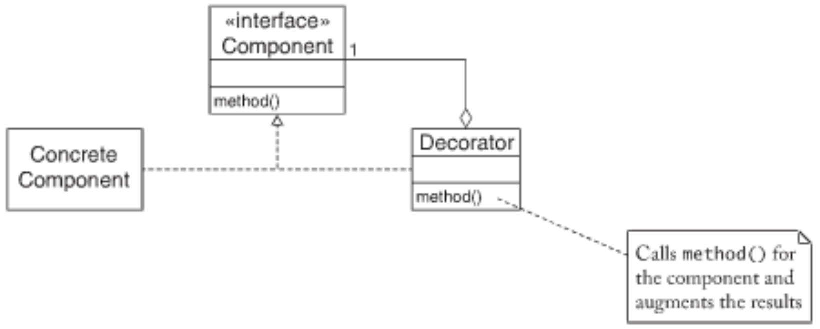
1. You want to enhance the behavior of a class. We'll call it the component class.
2. A decorated component can be used in the same way as a plain component.
3. The component class does not want to take on the responsibility of the decoration.
4. There may be an open-ended set of possible decorations.

Solution

1. Define an interface type that is an abstraction for the component.
2. Concrete component classes implement this interface type.
3. Decorator classes also implement this interface type.
4. A decorator object manages the component object that it decorates.

-
-
-
-
-
-

5. When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.



Name in Design Pattern	Actual Name
`Component`	`Reader`
`ConcreteComponent`	`FileReader`
`Decorator`	`BufferedReader`
`method()`	The `read` method. Calling `read` on a buffered reader invokes `read` on the component reader if the buffer is empty.

How to figure out whether a pattern applies to a situation

Go through the ____________________________ and_________________________ parts of the pattern description and make sure that _____________% of the statements apply.

, , Enhance behavior of class without changing original, passive, functionality, decorator, makes code reusable – maintainable, predict, context, solution, 100%

Module 5 – Parallelism

Module 5 Overview

In this module, we study parallelism and concurrency. Each of our class periods have time dedicate to group work. Use you group work time wisely.

Online Parallel Book

http://homes.cs.washington.edu/~djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf

Parallelism

Described in Classes 21 through 23.

Concurrency

Described in Classes 24 through 26.

Other resources

Video:(Requires login to UMW library to view)
http://umw.kanopystreaming.com.ezproxy.umw.edu/video/parallel-computing-here

Lectures

- C21to23Parallelism.docx
- C24to26Concurrency.docx

Labs

- Lab15Parallelism.docx.

Class 21, 22, and 23 Handout

Classes 21, 22, and 23 used the material in this section.

Learning Objectives, Assignments, and Readings

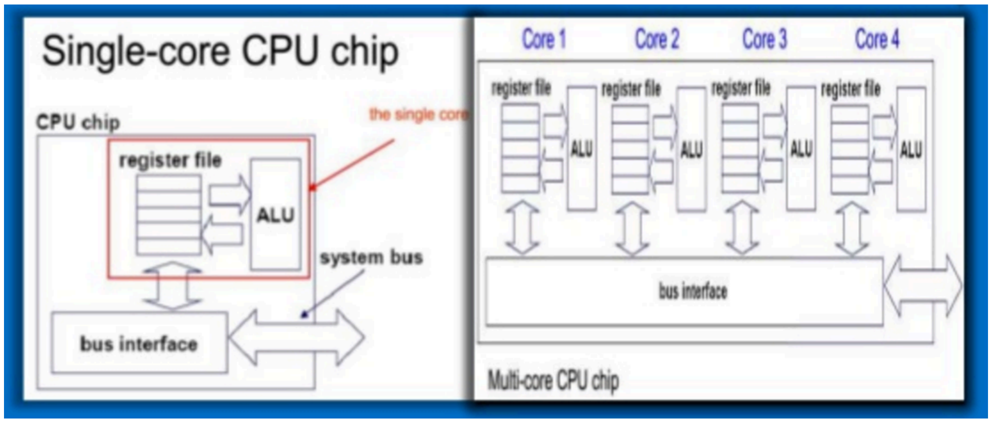
- Half of the class dedicated to the group programming project
- Understand the difference between sequential and parallel processing
- Understand the difference between a parallelism and concurrency
- Learn some Java library methods for simple threading

Sequential Programming vs Parallel Programming

When you run a sequential algorithm, how are statements processed?

When you run a parallel algorithm, how are the statements executed differently than in a sequential algorithm?

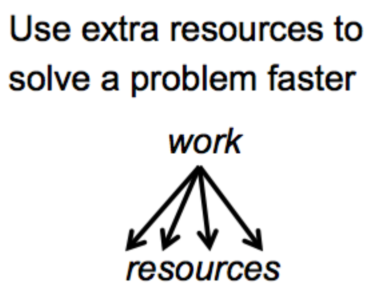
In the computing industry, we're seeing a move away from sequential processing and toward parallelism. Why is there a move toward parallelism?



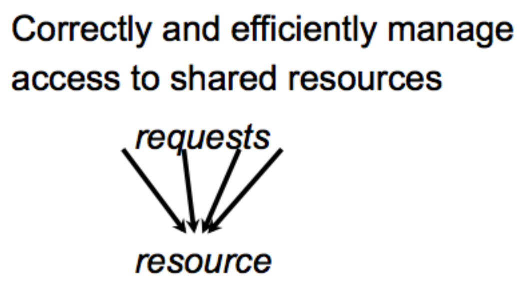
Parallelism vs Concurrency

There are no industry standards for these terms yet. The online book that we're using makes a clear distinction between these two very different programming challenges.

____:



____:



There's a subtle difference between parallelism and concurrency. In your own words, restate the cooking related analogy for parallelism. What issues will be important?

Now restate the cooking related analogy for concurrency. What issues will be important?

Focus on Parallelism

Define parallelism:

In all of our discussions, we're going to focus on one approach to parallel algorithms called

________________________________.

We'll implement:

- A set of *threads*, each with its own program counter & call stack
    - No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
    - To *communicate* by write data somewhere another thread reads

There are other possible approaches to parallelism including message-passing, dataflow, and data parallelism. If you get really excited about parallelism, talk to Dr Finlayson. Part of his research involves creating a new programming language that makes parallelism easy even for beginners!

As we create parallel programs, we'll need to be able to do three new things that we don't have to worry about in sequential programs:

1. Have an instruction that allows us to create and run ____________________________________

   These are called _________________________

2. Have a way to share _______________________

   In Java, we can allow threads to reference the same objects

3. Have an instruction that allows us to ___________________________________

   Think of this as a way to force a thread to wait for another thread to finish

Let's look at the basics provided in the Java API: java.lang.Thread

To get a new thread running:

1. Define a subclass **C** of **java.lang.Thread**, overriding **run**
2. Create an object of class **C**
3. Call that object's **start** method
   - **start** sets off a new thread, using **run** as its "main"

What if we instead called the **run** method of **C**?

Example Problem

Let's look at a simple problem with a simple sequential solution and then think about how to apply parallelism to improve the solution.

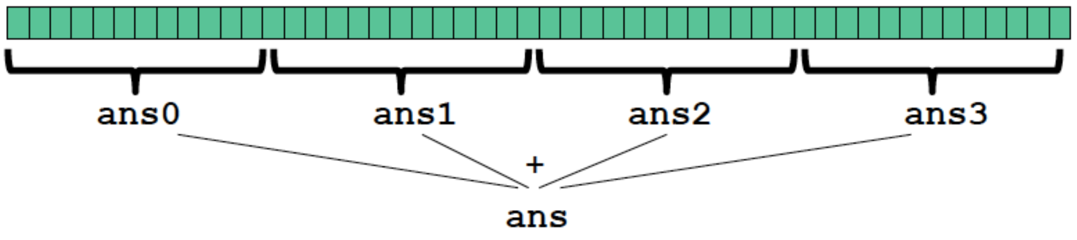
Problem: You have an array of integers and want to calculate the sum of these integers.

Sequential Solution

```
int sum(int[] arr) {
  in tans = 0;
  for (int i = 0; i < arr.length; i++)
    ans += arr[i];
  return ans;
}
```

First stab at a parallel algorithm (assume there are 4 processors):

- Use the first processor to sum the first 1/4 of the array and store the result somewhere.
- Use the second processor to sum the second 1/4 of the array and store the result somewhere.
- Use the third processor to sum the third 1/4 of the array and store the result somewhere.
- Use the fourth processor to sum the fourth 1/4 of the array and store the result somewhere.
- Add the 4 stored results and return that as the answer.



Parallel algorithm in pseudocode:

```
int sum(int[] arr) {
  results = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; ++i) { // parallel iterations
     results[i] = sumRange(arr,(i*len)/4,((i+1)*len)/4);
  }
  return results[0] + results[1] + results[2] + results[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; ++j)
      result += arr[j];
   return result;
}
```

Draw a 4 item array. Convince yourself that the algorithm works. If the array only contains 4 items, do you think you'll get a speedup over a conventional sequential algorithm?

Note the **FORALL** loop, which is like a **for** loop, but represents the fact that each iteration can be done in parallel. You're promising that all the iterations can be

done ______________________________ without _____________________________with each other.

There is no FORALL statement in Java. Instead, we'll need to:

- Define a subclass **C** of **java.lang.Thread**, overriding **run**
- Create 4 *thread objects* of class C, each given a portion of the work
- Call **start()** on each thread object to actually *run* it in parallel
- *Wait* for threads to finish using **join()**
- Add together their 4 answers for the *final result*

We'll be building a solution on the next few pages. Realize that the first attempts will have errors that we'll correct to learn how to carry out the steps correctly.

Incorrect Attempt #1:
Let's see if we can find all the bulleted steps listed above in the solution. If we can't that's a problem.

```
class SumThread extends java.lang.Thread {
   int lo; // fields for communicating inputs
   int hi;
   int[] arr;
   int ans = 0; // for communicating result
   SumThread(int[] a, int l, int h) {
     lo=l; hi=h; arr=a;
   }

   public void run() { // overriding, must have this type
    for(int i=lo; i<hi; i++)
      ans += arr[i];
   }
}
```

What step does this part of the code implement?

Why can't we pass data into the run( ) method as a parameter?

```
class C {
  static int sum(int[] arr) {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) {
      ts[i] = new SumThread(arr,(i*len)/4,((i+1)*len)/4);
    }
    for(int i=0; i < 4; i++) {
      ans += ts[i].ans;
    }
    return ans;
  }

}
```

What steps are shown above?

What steps are missing?

We need to call ______________ on each thread

We need to *Wait* for threads to finish using **join()**

Incorrect Attempt #2:

From here out, we'll abbreviate the SumThread class by hiding the details of the run() method. Everything about it was correct in the first example, so we won't need to make any changes to it.

```
class SumThread extends java.lang.Thread {
   int lo; // fields for communicating inputs
   int hi;
   int[] arr;
   int ans = 0; // for communicating result
   SumThread(int[] a, int l, int h) {
     lo=l; hi=h; arr=a;
   }
   public void run() { // overriding, must have this type
     for(int i=lo; i<hi; i++)
       ans += arr[i];
   }
}
class C {
  static int sum(int[] arr) throws java.lang.InterruptedException {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) {
      ts[i] = new SumThread(arr,(i*len)/4,((i+1)*len)/4);
      ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) {
      ans += ts[i].ans;
    }
    return ans;
   }
}
```

What steps are shown in the sum( ) method above?

- The sum() method creates 4 instances of the SumThread class
- SumThread is a subclass of java.lang.Thread
- Notice the instruction in red. We call start() on each instance of SumThread which will create our parallel threads.
- Each instance gets a unique subset of the array indices to work on

(note: low bound is included and high bound is excluded for each SumThread so there's no overlap)

- Each SumThread object has an **ans** field. This is a shared memory location for communicating the thread's answer back to the main thread.
- Then the main thread sums the 4 ans fields to calculate the final sum
- But the problem is ____. The main thread doesn't wait for the helper threads to complete before calculating the sum.

Solution: we need to delay the second for loop. We don't want it to execute until all of the helper threads have terminated.

Attempt 3: Correct in spirit but not the best solution.
We need a new instruction to tell the computer to wait for threads to finish. The new instruction is called join().

```
class SumThread extends java.lang.Thread {
   int lo; // fields for communicating inputs
   int hi;
   int[] arr;
   int ans = 0; // for communicating result
   SumThread(int[] a, int l, int h) {
     lo=l; hi=h; arr=a;
   }
   public void run() { // overriding, must have this type
     for(int i=lo; i<hi; i++)
       ans += arr[i];
   }
}
class C {
  static int sum(int[] arr) throws java.lang.InterruptedException {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) {
      ts[i] = new SumThread(arr,(i*len)/4,((i+1)*len)/4);
      ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) {
      ans += ts[i].ans;
      ts[i].joint(); // wait for helper to finish
    }
    return ans;
  }
}
```

Note the new line in red.

Now the main thread will block until the first thread terminates. Then until the second thread terminates, etc.

Note that to create the equivalent of the FORALL statement, we've used 2 loops. The first creates all of the threads and starts them. Then the second loop blocks and waits for them all to finish.

This is called ____ parallelism.

Describe what each of the following methods in the java.lang.Thread library does:

Start()

Run()

Join()

Building a Better Algorithm

But we said the previous algorithm was "correct in spirit", so what's wrong with it? It gives us the correct answer. It uses parallelism.

Can you think of any limitations to our solution?

Hint: What if our computer had 8 cores? What if our computer only had 3 cores available? How would this affect the execution of the solution that we wrote?

Big picture issues:

1. We should ____________________________ the number of threads.

```
static int sum(int[] arr, int numThreads) throws
java.lang.InterruptedException {
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[numThreads];
  for(int i=0; i < numThreads; i++) {
    ts[i]=new SumThread(arr,(i*len)/numThreads,((i+1)*len)/numThreads);
    ts[i].start();
  }
  for(int i=0; i < numThreads; i++) {
    ts[i].join();
    ans += ts[i].ans;
  }
  return ans
}
```

Summarize the differences between this version and the “Attempt #3 Correct in Spirit” version.

2. We should use only the processors ________________________________

Some processors might be used by other programs.

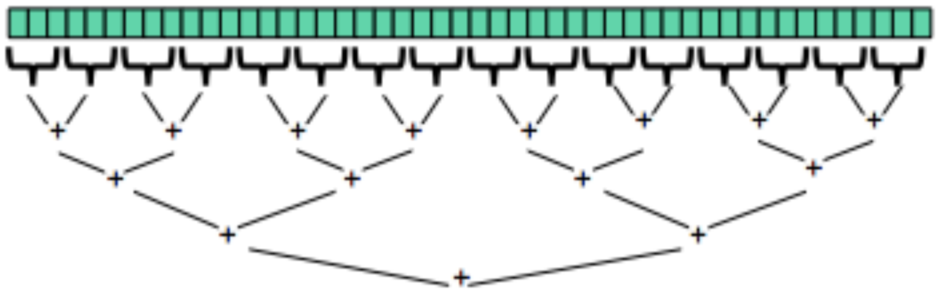
3. It’s possible that the chunks divvied up to the various processors take ______________________ amounts of time.

What’s meant by a load imbalance?

The Best Approach

We're going to need to change our algorithm a bit.

Describe the divide and conquer strategy for the problem (the counterintuitive strategy to subdivide the problem into tiny chunks.)



Here's the code:

```
class SumThread extends java.lang.Thread {
  int lo; // fields for communicating inputs
  int hi;
  int[] arr;
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { arr=a; lo=l; hi=h; }
  public void run() {
    if (hi - lo == 1) {
     ans = arr[lo];
    } else {
      SumThread left  = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right = new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.start();
      left.join();
      right.join();
      ans = left.ans + right.ans;

    }
  }
}

int sum(int[] arr) {
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

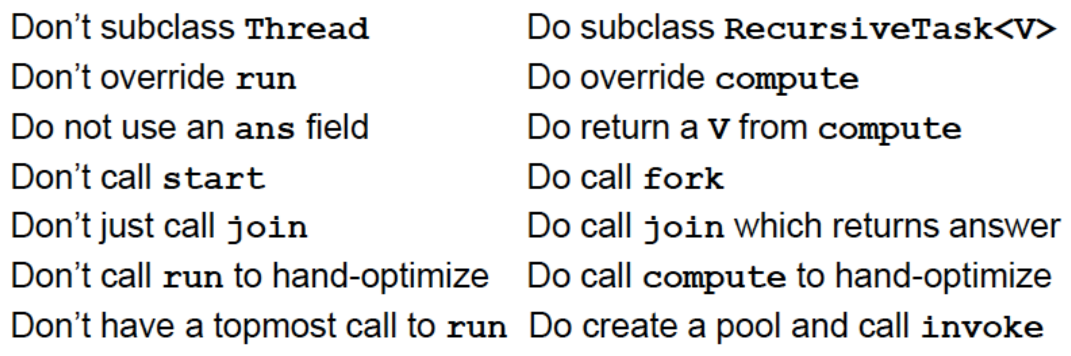
But how to we pick the right chunk size for each processor? If we pick chunks of size 1 (each processor processes 1 element), then this solution's final step to combine the sums is identical to the sequential algorithm. That's not a speedup at all!

We've compensated for this by assigning a CUTOFF above. If the chunk size is less than this value, it uses a sequential algorithm. For larger chunks, it uses the parallel algorithm.

This will give the correct solution but….. creating all those threads isn't free. It takes time. There's a lot of ________________________ involved in creating Java threads in this way. We'll need a different library to maximize our speedup.

Another reason to use a different library is that the java.lang.Threads library requires you to do a lot of the optimization by hand. That's a lot of work and it's hard to get the cutoff selected correctly.

There's a ForkJoin Framework specifically for divide and conquer parallel algorithms. This is going to solve all our problems.



Don't subclass `Thread`	Do subclass `RecursiveTask<V>`
Don't override `run`	Do override `compute`
Do not use an `ans` field	Do return a `V` from `compute`
Don't call `start`	Do call `fork`
Don't just call `join`	Do call `join` which returns answer
Don't call `run` to hand-optimize	Do call `compute` to hand-optimize
Don't have a topmost call to `run`	Do create a pool and call `invoke`

Final code using fork-join framework

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
class SumArray extends RecursiveTask<Integer> {
   static int SEQUENTIAL_THRESHOLD = 1000;
   int lo;
   int hi;
   int[] arr;
   SumArray(int[] a, int l, int h) { lo=l; hi=h; arr=a; }
   public Integer compute() {
      if(hi - lo <= SEQUENTIAL_THRESHOLD) {
         int ans = 0;
         for(int i=lo; i < hi; ++i)
            ans += arr[i];
         return ans;
      } else {
          SumArray left  = new SumArray(arr,lo,(hi+lo)/2);
          SumArray right = new SumArray(arr,(hi+lo)/2,hi);
          left.fork();
          int rightAns = right.compute();
          int leftAns = left.join();
          return leftAns + rightAns;
      }
   }
}

class Main {
  static int sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new
SumArray(array,0,array.length));

//Alternative
     static final ForkJoinPool fjPool = new ForkJoinPool();
     return fjPool.invoke(new SumArry(array,0,array.length)
  }
}
```

Class 24, 25, and 26 Handout

Classes 24, 25, and 26 used the material in this section.

Learning Objectives, Assignments, and Readings

- Understand common challenges with concurrent programs
- Understand how mutual exclusion prevents bad interleaving

- List and describe the 3 operations included in a lock
- Know the 3 ways that data can be protected within a program
- Understand the tradeoffs between performance and critical sections
- Define the term deadlock and describe how it can occur

Challenge of concurrency
The challenge of concurrency is to control access by multiple threads to a shared resource (or resources).

ATM example
Two people share a checking account. It has a balance of $100.

At 9:00, person 1 logs in and checks the balance. At 9:01, person 1 requests to withdraw $100. This is approved at 9:03 & person 1 has the money in hand.

At 9:00, person 2 logs in and checks the balance. It's $100. At 9:02, person 2 requests to withdraw $100. The bank approves and at 9:03, person 2 has the money in hand.

Is this operation good business?

Trinkle Vending Machine example:
Not writing this one down to protect my information sources. Trust me, you'll remember it.

Summary of issues:
The typical model we'll consider has multiple threads predominantly operating

___________________________________. For example, perhaps we create a different thread to handle each customer's bank requests.

Typically, these threads won't need to access the same account and can proceed

without ________________________________ and without ______________________________. Occasionally, the threads will access the same account and will need to coordinate before performing particular activities.

Warning: These types of programs are really hard to debug. You (the programmer) can't necessarily control the order in which events occur. It's hard (sometimes) to reproduce buggy behavior. And if you can't reproduce it, that makes it hard to find problems.

Understanding Issues
Consider the bank example. Here's a typical sequential program to handle the typical bank operations.

```
class BankAccount {
      private int balance = 0;

      int getBalance() { return balance; }
      void setBalance(int x)  {balance = x; }
      void withdraw(int amount) {
            int b = getBalance();
            if (amount > b)
                  throw new  WithdrawTooLargeException();
            setBalance(b – amount);
      }
      //other operations like deposit, …
}
```

If two threads are operating and one calls

```
      x.withdraw(100);
```

and the other calls

```
      y.withdraw(100);
```

Will it cause a bank error if….

- x & y are different objects referring to distinct bank accounts?

- x & y refer to the same account, but one request is guaranteed to finish it's call before the other call begins?

- x & y refer to the same account and the two calls happen at the same time?

In this last case, we say the calls ______________________________________.
Interleaving calls can cause problems.

Example of Interleaving Issue

```
      Thread 1                              Thread 2
      ---------------------                 -----------------------
      int b = getBalance();

                                            int b = getBalance();
                                            if (amount > b)
                                                  throw new …;
                                            setBalance(b-amount);

      if (amount > b)
            throw new …;
      setBalance(b – amount);
```

What's the result?

Is this what should have happened?

Using Mutual Exclusion to Prevent Bad Interleaving

To prevent interleaving, we need to use a technique called ___________________________.

Mutual exclusion means we'll only allow one thread to access a particular chunk of code at a time.

This allows us to create a _______________________________________ : a sequence of operations for which interleaving will not be allowed.

To use mutual exclusion we have to use synchronization primitives defined in the programming languages. This is something that you can ______________________ write from scratch.

Primitives for Creating a Critical Section

We're going to use a primitive called a ___________________________________
It's an Abstract Data Type (ADT) with 3 operations:

> ___________________ : creates a new lock that is "not held" initially
>
> __________________ : takes a lock and blocks until it is currently "not held" (which may be immediately). It sets the lock to "held" and returns.
>
> _________________ : takes a lock and sets it to "not held"

The lock works like a "do not disturb" sign. We can call the acquire( ) method to turn on the "do not disturb()" sign. The acquire operations doesn't return until the caller is the thread that most recently hung the sign.

The lock ADT will always make sure that the data structure "does the right thing" no matter how many different threads simultaneously perform acquire operations and/or releases. For example, if 4 threads simultaneously execute an acquire()

operation for a "not held" lock, one will _______________ and return immediately while the other three block. When the winner calls "release()", …

In pseudocode, this might look like:

```
class BankAccount {
      private int balance = 0;
      private Lock lk = new Lock();
      …

      void withdraw(int amount) {
            lk.acquire(); /*may block*/
            int b = getBalance();
            if (amount > b)
                  throw new WithdrawTooLargeException();
            setBalance(b-amount);
            lk.release();
      }
      //deposit would also acquire/release the lock lk
}
```

Now different threads can operate on different accounts at the same time.

And only one thread can operate on a specific account at a specific time (thanks to the lock).

We still have 2 problems in the Code

1. What happens when an exception is thrown? That thread never

________________________________ the lock. So no other thread can ever have access to that account. This isn't good.

2. We need re-entrant locks. In other words, if a particular thread holds the lock at a particular time and needs to execute code that acquires the lock, the computer should be able to figure out tat this same thread already holds the lock and continue on rather that stopping the thread in it's tracks.

Additional details about how our 3 lock methods work:

`new()` creates a new lock with no current holder and a count of _____________________

`acquire( )` blocks if there is a current holder ____.
Otherwise if the current holder is the thread calling it, don't block and increment the counter.
Otherwise there is no current holder, so set the current holder to the calling thread.

`release()` only releases the lock (sets the current holder to "none") if the count is 0. Otherwise it ______________________________

How this works in Java syntax

```
      synchronized (expression) {
            statements
      }
```

This looks very like a while loop, but it's not a loop.

How it works:

1. The expression is evaluated. It must produce (a reference to) an object – not null or a number. This object is treated as a lock. In Java, every object is a lock that any thread can acquire or release. This decision is a little weird, but it's convenient in an object-oriented language.

2. The synchronized statement acquires the lock. ie The object that is the result of step 1. This may block until the lock is available. Locks are re-entrant in Java, so the statement will not block if the executing thread already holds it.

3. After the lock is successfully acquired, the statements are executed.

4. When control leaves the statements, the lock is released. This happens either when the final } is reached OR when the program "jumps out of the statement" via an exception, a return, a break, or a continue statement.

Note that the lock is released at the ending }
This is true even if an exception only causes a part of the "body" not to complete

EX:

```
      class BankAccount {
            private int balance = 0;
            private Object lk = new Object();
            int getBalance() {
                  synchronized (lk) {
                        return balance;
                  }
            }
            void setBalance(int x) {
                  synchronized (lk) {
```

```
                balance = x;
            }
        }
        void withdraw(int amount) {
            synchronized (lk) {
                int b = getBalance();
                if (amount > b)
                    throw new WithdrawTooLargeException();
                setBalance(b-amount);
            }
        }
        //deposit and other operations would also use synchronized(lk)
    }
```

Or we can save a step and NOT make a new object just for the lock. Instead we can use the calling object like this

EX:

```
    class BankAccount {
        private int balance = 0;
        int getBalance() {
            synchronized (this) {
                return balance;
            }
        }
        void setBalance(int x) {
            synchronized (this) {
                balance = x;
            }
        }
        void withdraw(int amount) {
            synchronized (this) {
                int b = getBalance();
                if (amount > b)
                    throw new WithdrawTooLargeException();
                setBalance(b-amount);
            }
        }
        //deposit and other operations would also use synchronized(lk)
    }
```

And finally, when we want to lock the entire body of a method (as shown above), Java includes a syntax shortcut to reduce typing. The final result would look like this:

EX:

```
    class BankAccount {
        private int balance = 0;
        synchronized int getBalance() {
            return balance;
        }
        synchronized void setBalance(int x) {
            balance = x;
        }
        synchronized void withdraw(int amount) {
              int b = getBalance();
              if (amount > b)
                    throw new WithdrawTooLargeException();
              setBalance(b-amount);
        }
        //deposit and operations would also use synchronized(lk)
  }
```

Data Race

Data race – _______________________________________

Example: one thread might read an object field at the same moment that another thread writes to the same field

Example: one thread might write an object field at the same moment that another thread reads the same object field

It's never an error if two threads just read an object field simultaneously. Why?

Data Race Rules

Every memory location should either be

1. Thread-local:

2. Immutable:

3. Synchronized:

Bottom line: Avoid sharing objects unless those objects are specifically being used to enable shared-memory communication among threads.

Guidelines

- **Avoid Data Races**: Use locks to ensure that two threads never simultaneously

  ____.

- **Use consistent locking:** For each location that needs synchronization, identify a lock that is always held when accessing that location. This will prevent data races, but not necessarily bad interleavings.

- **Start with coarse-grained locking** and move to finer-grained locking only if

____. Ie. use fewer locks to guard more objects (use one lock to guard an array of accounts). Only move to a separate lock for every single bank account if performance improvements are needed.

- **Make critical sections large enough for correctness but** __________________________. Don't perform I/O or expensive computations w/in critical sections.

- **Think in terms of what operations need to be atomic (__________________________).** Determine a locking strategy after you know what these critical sections are.

- **Don't implement your own concurrent data structures.** Use the carefully tuned ones written by experts and provided in standard libraries.

- **Avoid deadlock.**

Deadlock

What is deadlock? If a collection of threads are blocked forever, all waiting for

another thread in the collection to do something (_______________________________________) then we say the threads are deadlocked.

Deadlock Example

```
Class BankAccount {
      …
      synchronized void withdraw(int amt) {… }
      synchronized void deposit (int amt) {…{
      synchronized void transferFo(int amt, BankAccount a) {
            this.withdraw(amt);
            a.deposit(amt);
      }
}
```

Looks good, right?

- There are no data races because only methods that directly access fields are synchronized– like withdraw and deposit - and these acquire the correct lock.

- transferTo seems atomic. For another thread to see the intermediate state where the withdrawal has occurred but not the deposit, would require operating on the withdraw n-from account. And since transferTo is synchronized, this can't happen.

But this deadlock situation can happen if we're unlucky in the ordering.

EX: One thread is transferring from account A to B, while the other is transferring from B to A

Thread 1: x.transferTo(1, y)	Thread 2: y.transferTo(1.x)
acquire lock from x	
withdraw 1 from x	
	acquire lock from y
	withdraw 1 from y
	block on lock for x
block on lock for y	

The first thread holds the lock for A and is blocked on the lock for B. The second thread holds the lock for B and is blocked on the lock for A. So both are waiting for a lock to be released by a thread that is blocked.

_________________________________.

Deadlock Solutions

Deadlock Solution Option 1

Option 1: Don't synchronize the transferTo() method. Because withdraw and deposit will remain synchronized the only downside is that someone might "see" the state where the withdraw has occurred but not the deposit. That might be okay.

Deadlock Solution Option 1

Option 2: Use a coarser grained locking. Ie one lock for __________________________

In both options you avoid deadlock because we've ensured that no thread ever holds

_______________________________________.

If it's essential that the critical section acquire two locks, then make a required order that all threads have to use. Ie. If each thread HAS to acquire lock A before it acquires lock B. Then we can never have deadlock.