# Notes

Final Exam Times

Project01

- **Project01** directory, not **Lab14**, **Project 1**, etc
- Sign up for 10 minute meeting at LINK in CourseInfo README

Today

- Quiz
- Time to work on and test Project01

CS112 –
Java
Programming

Spring 2024



# Class Design…

This is an example of BAD DESIGN. Banana Boat sunscreen, using propane as a propellant. Recalled after 1 day.

## Class Design Review

We have discussed

- Classes vs objects
- Solving problems with good class design
- Methods and constructors
- Method arguments and return types
- Method overloading
- Private and public members, encapsulation, get() and modify() methods
- Class variables and method variables, variable scope

**We are starting a 3 lecture series on technologies for advanced class design**

What is "Good Design"? Takes practice and experience. Easy to understand, generally useful and reusable

Class Data = STATE

Class Methods = transformations, operations, retrievals, conversions, filters, etc

## Identifying Classes and Objects

The core activity of object-oriented design is determining the classes and objects that will make up the solution

The classes may be part of a class library, reused from a previous project, or newly written

One way to identify potential classes is to identify the objects discussed in the requirements

Objects are generally nouns, and the services that an object provides are generally verbs

## Identifying Classes and Objects

A partial requirements document:

> The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final design

# Identifying Classes and Objects

A **class** usually represents a **group** (classification) of objects with the same behaviors

- Examples: `Coin`, `Student`, `Message`

A class represents the concept ("definition") of one such object

We are free to instantiate as many of each object as needed

## Identifying Classes and Objects

Sometimes it is challenging to decide whether something should be represented as a class

For example, should an employee's address be represented as a set of instance variables or as an `Address` object

A rule of thumb: it depends on the operations you want to do with the data: very little → variable, more → class

When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

## Identifying Classes and Objects

We want to define classes with the proper amount of detail

For example, it may be unnecessary to create separate classes for each type of appliance in a house

It may be sufficient to define a more general `Appliance` class with appropriate instance data

It all depends on the details of the problem being solved

# Advanced Topics in Class Design

Member classes

## Member Classes

We have seen >1 class per .java file

```
class Card { }
public class CardDeck {}
```

We can define a class within another class

```
class Student {
  class StudentRecord {
     StudentRecord(String name, int id, String login, String GitHubId) {…}
     void getName() {…}
  }
  // Other class Student methods
}
```

Why do this?

- The CONTAINED class is a member of class Student, so it can access class Student member variables and member functions
- ENCAPSULATES a limited part of Student's functionality, separate from the main class

Still just a class DECLARATION. If we want objects (within class Student) with the member class type, we must DEFINE them!

Usually private to containing class, but possible to make public. Could take that approach to bundle a bunch of related member classes into a containing wrapper class

Class Card could be a member class inside CardDeck…but it does not need access to other members of CardDeck, so no benefit to doing so.

# Advanced Topics in Class Design

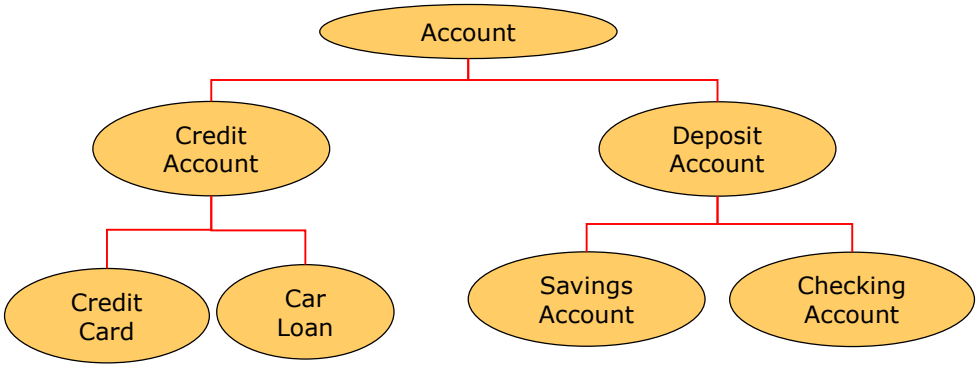
Member classes

Inheritance

## Inheritance

One class can be used to derive another via *inheritance*

Classes can be organized into hierarchies






"Child classes" are derived from "base classes", and add more specialized data and methods

## Inheritance

A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones

One benefit of inheritance is *software reuse*

By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

## Deriving Subclasses

In Java, we use the reserved word `extends` to establish an inheritance relationship

```
public class Car extends Vehicle
{
   // class contents
}
```

```
//********************************************************************
//  Book.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//********************************************************************

public class Book
{
   protected int pages = 1500;

   //-----------------------------------------------------------------
   //  Pages mutator.
   //-----------------------------------------------------------------
   public void setPages(int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  Pages accessor.
   //-----------------------------------------------------------------
   public int getPages()
   {
      return pages;
   }
}
```

```
//********************************************************************
//  Dictionary.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

public class Dictionary extends Book
{
   private int definitions = 52500;

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public double computeRatio()
   {
      return (double) definitions/pages;
   }
```

continue

continue

```
   //---------------------------------------------------------------
   //  Definitions mutator.
   //---------------------------------------------------------------
   public void setDefinitions(int numDefinitions)
   {
      definitions = numDefinitions;
   }

   //---------------------------------------------------------------
   //  Definitions accessor.
   //---------------------------------------------------------------
   public int getDefinitions()
   {
      return definitions;
   }
}
```

```
//********************************************************************
//  Words.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an inherited method.
//********************************************************************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary webster = new Dictionary();

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

```
//***************
//  Words.java
//
//  Demonstrates
//***************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary webster = new Dictionary();

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

**Output**

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```



What methods and variables belong to base class? Which to derived class?
EXAMPLE: class Animal, class Cat, class Dog

# Advanced Topics in Class Design

Member classes

Inheritance

Overriding Inherited Methods

## Overriding Methods

A child class can *override* the definition of an inherited method in favor of its own

The new method must have the same signature as the parent's method, but can have a different body

The type of the object executing the method determines which version of the method is invoked



Parent class object or child class object

## Overriding

A method from the parent class can be invoked explicitly in the child class using the `super` reference

If a method is declared with the `final` modifier, it cannot be overridden by a child

The concept of overriding can be applied to data and is called *shadowing variables*

Shadowing variables should be avoided because it tends to cause confusing code

```
//********************************************************************
//  Thought.java       Author: Lewis/Loftus
//
//  Represents a stray thought. Used as the parent of a derived
//  class to demonstrate the use of an overridden method.
//********************************************************************

public class Thought
{
   //-----------------------------------------------------------------
   //  Prints a message.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("I feel like I'm diagonally parked in a " +
                         "parallel universe.");

      System.out.println();
   }
}
```

```
//********************************************************************
//  Advice.java       Author: Lewis/Loftus
//
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//********************************************************************

public class Advice extends Thought
{
   //-----------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("Warning: Dates in calendar are closer " +
                         "than they appear.");

      System.out.println();

      super.message();  // explicitly invokes the parent's version
   }
}
```

```
//********************************************************************
//  Messages.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an overridden method.
//********************************************************************

public class Messages
{
   //-----------------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();  // overridden
   }
}
```

## Output

```
I feel like I'm diagonally parked in a parallel universe.

Warning: Dates in calendar are closer than they appear.

I feel like I'm diagonally parked in a parallel universe.
```

```
//-----------------------------------------------------------------
//  Creates two objects and invokes the message method in each.
//-----------------------------------------------------------------
public static void main(String[] args)
{
   Thought parked = new Thought();
   Advice dates = new Advice();

   parked.message();

   dates.message();  // overridden
}
}
```

## Overloading vs. Overriding

**Overloading** is multiple methods with the same name in the same class, but with different signatures

**Overriding** is two methods, one in a parent class and one in a child class, that have the same signature

Overloading lets you define a similar operation in different ways for different parameters

Overriding lets you define a similar operation in different ways for different object types

## Quick Check

### True or False?

| | |
|---|---|
| A child class may define a method with the same name as a method in the parent. | True |
| A child class cannot override a `final` method of the parent class. | True |
| It is considered poor design when a child class overrides a method from the parent. | False |
| A child class may define a variable with the same name as a variable in the parent. | True, but shouldn't |

## Notes

Project01

- Clear what to turn in Wednesday?

Let's do some examples of Inheritance…

Don't overuse inheritance
Examples:  Shape (position, charToPrint), Rectangle(w, l), Square(sideLen)