CS601: Principles of Software Development

Encapsulation. Inheritance.

Olga A. Karpenko

Parts of this presentations are based on the Java Solftware Solutions book by Lewis&Loftus.

Quiz 3

- One Exceptions

Announcements

- Code reviews
- Lab 1 Part 2 Due on Friday
  - May not use listFiles
- Friday: 2.15-3.20pm lecture in G12
  - Everybody can come to that section on Friday
- For viewing json files:
  - https://codebeautify.org/jsonviewer
- Office hours today:
  - 3:20-4, and 4:30-5:30

Top Principles of OO Design

- Abstraction
- Encapsulation
- Inheritance

Abstraction

- Show only "relevant" data and hide unnecessary details of an object

http://stackoverflow.com/questions/16014290/simple-way-to-understand-encapsulation-and-abstraction

Encapsulation

- "Bundling data and methods together"
- An object should be encapsulated:
  - Should protect & manage its own data
  - Changes to the state of the object should be done using object's methods
  - Other objects should not be able to reach in and change its state

Encapsulation: Example

```
public class LibraryCatalog {
  private ArrayList<Book> books = new ArrayList<Book>();

 // public methods that can interact with this data
 // but only how this class allows them
}
```

Encapsulation: Example

```
public class LibraryCatalog {
  private ArrayList<Book> books = new ArrayList<Book>();

  public void add(Book book) {
      // add the book to the ArrayList books
  }

  public boolean returnBook(String title) {
      // Return this book ..
  }
  public boolean checkoutBook(String title) {
      // Checkout the book ..
  }
}
```

Benefits of Encapsulation

- Other classes can treat this class as a black box, ignore implementation details
- Source code for this class can be written and maintained independently of other classes

Breaking Encapsulation

```
public ArrayList<Book> getBooks() {
  return books;
}
```

Breaking Encapsulation

```
public ArrayList<Book> getBooks() {
  return books;
}
```

Bad idea: if we return the ArrayList of books, the outside class can do anything with it

- delete all books, for instance, like that:

```
LibraryCatalog catalog = new LibraryCatalog();
catalog.add(new Book ("Harry Potter", "Rowling"));
catalog.add(new Book ("Hobbit", "Tolkien"));
List<Book> books = catalog.getBooks();
books.clear(); // removes all elements
```

The Right Approach

- Do ***not*** ask object for the info you need to do the work
- Ask the object (that has the information) to do the work for you

http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

Additional Reading

- Are Getters/Setters evil?
  - Expose implementation details

http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

Some Other OO Principles

- Single Responsibility Principle
  - A class should have only one "job"
- Open Closed Principle
  - open for extension, closed for modification
- Interface Segregation Principle
  - smaller, specific interfaces
  - ...

https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Lab 1

- Points will be docked off for breaking encapsulation
- Use the strictest possible access level as long as your classes can still function as needed

Class Relationships

- Types of relationships between classes:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B

Dependency

- One class relies on another in some way
  - Usually by invoking the methods of the other class
- Example:
  - In Die class, we used the class Math to generate random numbers

Dependency

- We don't want too many complex dependencies between classes
  - Nor do we want complex classes that don't depend on others
- Need to find the right balance

Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A bank account *has an* owner
  - A student *has* a name
  - A hotel *has an* Address

Inheritance

- Definition of one class can be based on existing class
  - is-a relationship
- Example:
  - A truck is a vehicle

Inheritance

```
class Vehicle {
  private int numDoors;
  private int numSeats;
  // other variables and methods
}

class Truck extends Vehicle {
  private boolean isPickup;
  // other methods and variables
}
```

Inheritance

- Another Example:
  - Class: Bank Account
  - Subclass: Savings Account

Inheritance

```
class Account {
   private double balance;
   // other variables and methods
}

class SavingsAccount extends Account {
  private double interestRate;
  // other variables and methods
}
```

Advantages

- Time saved in program development
- Reuse of proven & debugged code

Inheritance

- The existing class is called the *parent class*
  - *super class, base class*
- The derived class is called the *child class*
  - *subclass*

Inheritance

- The child inherits public and protected methods and data of the parent
- To tailor a derived class, the programmer can
  - add new variables or methods, or/and
  - modify the inherited ones

The protected Modifier

- Allows a member of a super class to be inherited into a subclass
- Provides more encapsulation than public visibility does
  - Intermediate level of protection
- Generally, should not be used for instance variables

Access levels

Modifier	Class	Package	Subclass	World
Public	Y	Y	Y	Y
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N

The super Reference

- Constructors are not inherited
- Use the parent's constructor to set up the "parent's part" of the object
  - Use super() to call parent’s constructor
  - In the first line of the child’s constructor

super Reference

```
public SavingsAccount(double bal, String
owner, double interestRate) {

    super(bal, owner);
    this.interestRate = interestRate;
}
```

- See SavingsAccount.java

Example: Book and Dictionary

```
public class Book {
   private int pages;

   public Book (int numPages) {
      pages = numPages;
   }

   public void setPages (int numPages)
   {
      pages = numPages;
   }
   // more methods
}
```

Example: Book and Dictionary

```
public class Dictionary extends Book {
  private int definitions;

  public Dictionary (int numPages, int nDefinitions)
   {
      // todo: need to set pages variable
     // but it's private!


      definitions = nDefinitions;
   }
   // more code

}
```

Example: Book and Dictionary

```
public class Dictionary extends Book {
  private int definitions;

  public Dictionary (int numPages, int nDefinitions)
   {
      super(numPages);
      definitions = nDefinitions;
   }
   // more code

}
```

Example: USF Database

- Create a database of faculty, staff, and students at USF
- What classes should we have?

Example

- Faculty
  - name, id, officeNumber, courses, getters and setters, print()
- Staff
  - name, id, officeNumber, numVacationDays, getters and setters, print()
- Student
  - name, id, transcript, currentGPA, print()

- What's common between them?

Example

- Faculty
  - name, id, officeNumber, courses, getters and setters, print()
- Staff
  - name, id, officeNumber, numVacationDays, getters and setters, print()
- Student
  - name, id, transcript, currentGPA, print()

- Some getters and setters will be common too

Base (Parent) Class

- USFPerson
  - name, id, print()

Subclasses

- USFEmployee extends USFPerson
  - officeNumber
- USFStudent extends USFPerson
  - GPA, transcript

Subclasses of a Subclass

- USFFaculty extends USFEmployee
  - courses
- USFStaff extends USFEmployee
  - numVacationDays

USFPerson

```
public class USFPerson {
	private String name;
	private String id;

	public USFPerson(String name, String id) {
		this.name = name;
		this.id = id;
	}

	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
	  return name;
	}
	// other methods
}
```

http://web.cs.usfca.edu/basics/java-basics/inheritance

USFStudent

```
public class USFStudent extends USFPerson {
  private double currentGPA;

  public USFStudent(String name, String id, double gpa)
  {
      super(name, id);
      this.currentGPA = gpa;

  }
  public double getGpa() {
      return currentGPA;
  }
  // other methods
}
```

USFEmployee

```
public class USFEmployee extends USFPerson {
  private int officeNumber;

  public USFEmployee (String name, String id, int num)
  {
     super(name, id);
     this.officeNumber = num;

  }

  // other methods
}
```

Methods

- A subclass can override the inherited methods
- A subclass can add additional methods

Overriding methods

- Redefining a method using the same signature
  - Ex: we often override toString()
  - Use @Override

Example: Methods

- USFPerson

```
protected void print() {
  System.out.println("Name = " + name);
  System.out.println("ID = " + id);
}
```

- USFFaculty, USFStudent will inherit it
  - Can override it

Example: Overriding

- USFStudent

```
@Override
protected void print() {
  System.out.println("Name = " + getName());
  System.out.println("ID = " + getId());
  System.out.println("GPA =" + currentGPA);
  }
```

Example: Overriding

- USFStudent

```
@Override
protected void print() {
   super.print();
   System.out.println("GPA =" + currentGPA);
}
```

Example: Overriding

```
class USFEmployee extends USFPerson {
  private int officeNumber;

  @Override
  protected void print() {
    super.print();
    System.out.println("Office=" +
  officeNumber);
  }
}
```

Example: Overriding

```
class USFEmployee extends USFPerson {
  private int officeNumber;

  @Override
  protected void print() {
    System.out.print("Employee = " + getName());
    System.out.println("Office=" +
  officeNumber);
  }
}
```

- Don't have to call super.print()
- It's up to the child to decide how to override print()

Overriding

- A method in the parent class can be invoked explicitly using the super reference
- If a method is declared with the final modifier -> it cannot be overridden

Overloading vs. Overriding

- Overloading :
  - Methods in the same class
  - Methods have the same name, but different signatures
  - Lets you define a similar operation in different ways for different parameters

Example: Overloading

```
public class OverloadingExample {

	public void disp(char c) {
		System.out.println(c);
	}


	public void disp(char c, int num) {
		System.out.println(c + " " + num);
	}
}
```

Modified from the example in http://beginnersbook.com/2013/05/method-overloading/

Example: Overloading

```
public class OverloadingTest {
  public static void main(String[] args) {
    OverloadingExample obj = new
OverloadingExample();
    obj.disp('c');
    obj.disp('c', 5);

  }
}
```

Modified from the example in http://beginnersbook.com/2013/05/method-overloading/

Overloading vs. Overriding

- Overriding:
  - One method is in a parent class and one in a child class
  - Have the same signature, but different body
  - 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.

A child class can override the constructor of the parent class.

A child class cannot override a final method of the parent class.

It is considered poor design when a child class overrides a method from the parent.

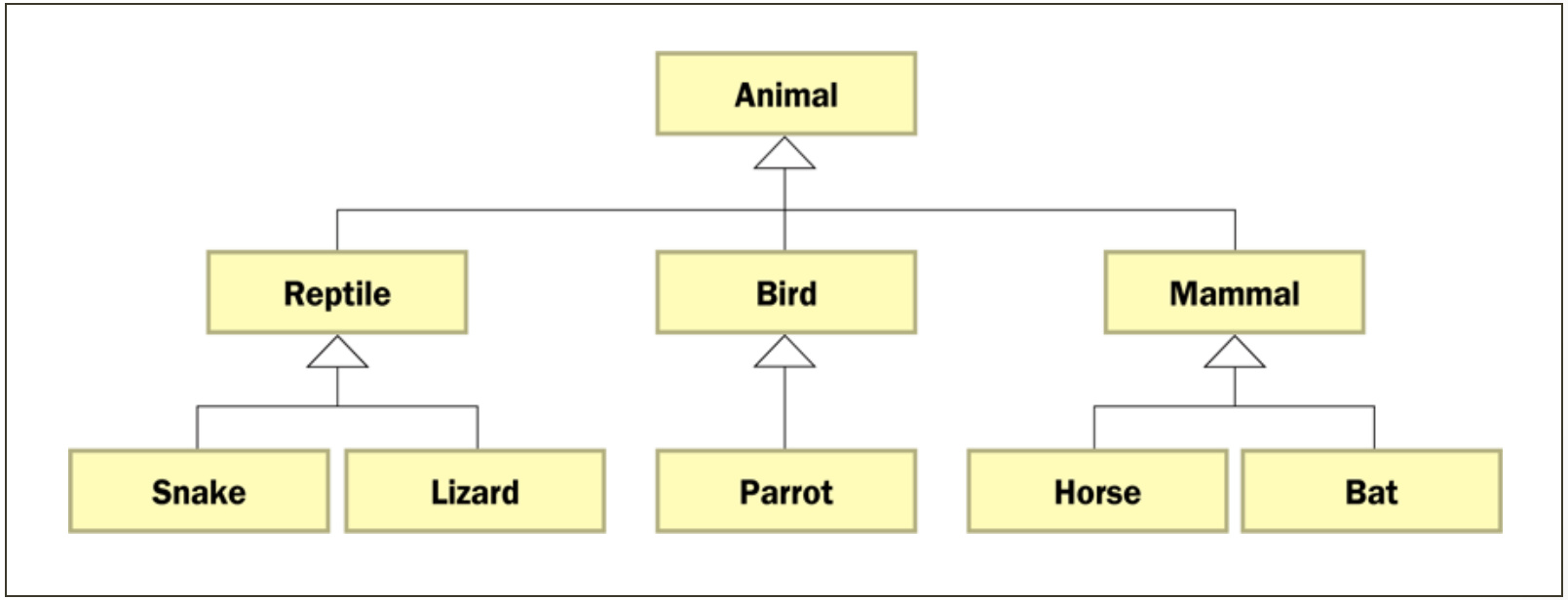
A child class may define a variable with the same name as a variable in the parent.

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 can override the constructor of the parent class	False
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

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes

The Object Class

- In the java.lang package
- All classes are derived from the Object class
- The ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods
  - inherited by all classes
- Ex: `toString` method
  - When define the toString() for the class, we are actually overriding an inherited definition

The Object Class

- The equals method
  - by default returns true if two references are the same
  - We can override it in any class

Visibility Revisited

- Private members of the parent cannot be referenced by name in the child class
  - they exist in a child object and can be referenced indirectly

Example: Book and Dictionary

```
public class Book {
    private int pages;

    public Book (int numPages) {
        pages = numPages;
    }

    public void setPages (int numPages){
        pages = numPages;
    }

    public int getPages() {
      return numPages;
    }
```

Example: Book and Dictionary

```
public class Dictionary extends Book {
  private int definitions;

  public Dictionary (int numPages, int nDefinitions)
   {
      super(numPages);
      definitions = nDefinitions;
   }

  public void printInfo() {
     System.out.println(getPages());
     System.out.println(definitions);
  }
}
```

Inheritance Design Issues

- Every derivation should be an is-a relationship
- Design classes to be reusable and flexible
- Find common characteristics of classes and push them high in the class hierarchy
- Override methods as needed to change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

Multiple Inheritance

- Multiple inheritance allows a class to be derived from two or more classes
  - inheriting the members of all parents
- Java does ***not*** support multiple inheritance
  - Each class has only one direct parent