CPIT-252
Software Design Patterns

Overview of Object-Oriented Programming

Khalid Alharbi, Ph.D.

Overview of Object-Oriented Programming

An overview of fundamental concepts in Object-Oriented Programming, which is a prerequisite to understanding and applying design patterns.

Khalid Alharbi, Ph.D.

Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which contain data (fields, attributes or properties) and operations on these fields (methods).
- In class-based OOP, programs are composed of classes and objects are instances of these classes.
- In OOP, we think in terms of objects that do things.
- This contrasts with functional programming, where we think in terms of functions and their composition.

How functional decomposition is different?

- Before we discuss the benefits of OOP, let’s see how problems are solved in **functional decomposition**.
- Breaking down a program into smaller problems and then decomposing smaller problems into functional steps.
- This seems to be the natural approach to problem solving.
- However, it creates a program **centered around a “main program”**
    - This main program controls the details of the program’s data and operations.
- It also creates a program that **does not respond well to changes**.
    - A minor change requires several changes through the entire main program.

Why are there problems?

- Things always change, and software is always evolving.
- There's nothing we can do to prevent changes to a software system.
- Many software bugs are the result of changes to the code.
- The functional decomposition approach has design problems due to poor **modularity**, poor use of **abstraction** and poor **encapsulation**.
- The Object-Oriented approach aims to address these problems.
- The goal is to create designs that are reliable, resilient, easy to understand, maintainable and accommodate changes without breaking existing working code.

OOP: Seven Main Principles

Abstraction, Encapsulation, Inheritance, Polymorphism, Composition, Aggregation and Delegation.

Let’s answer the following questions:

- What is the difference between **Abstraction** and **Encapsulation**?
- What is the difference between **Inheritance** and **Composition**?
- What is the difference between **Composition** and **Aggregation**?
- What is **Polymorphism**? What is **Delegation**?
- Any takers?

Abstraction

- Abstraction refers to the set of concepts that address complexity by hiding unnecessary details from the user (**implementation hiding**).
  - Examples: The public methods of Java's Math class and the public methods in the String class.

```
    public static double sqrt(double a)
```

```
    public String substring(int beginIndex, int endIndex)
```

Encapsulation (I)

- **Encapsulation** is about grouping all data fields and operations on them in a single cohesive unit and hiding irrelevant implementation details of a class from other classes (**information hiding**).
- In Java, encapsulation is achieved through marking an instance variable as **“private”** and providing a method for retrieving and updating a particular variable (getters and setters).
- It’s about the bundling of data with the methods that operate on that data into a single component while restricting access to some of its internal data.

Encapsulation (II)

```
public class Product{

  private double price;
  private String name;

  public Product(String name, double price){
    this.name = name;
    this.price = price;
  }

  public double getPrice(){
    return this.price;
  }

  public void setPrice(double price){
    this.price = price;
  }
```

But why do we need all of this?

- Why do we need good abstraction? What makes poor abstraction bad?
- Why do we want to utilize encapsulation? What makes poor encapsulation bad?

Why Abstraction and Encapsulation?

- If you have **poor abstraction**, it means you have a class that is not focused on a single task.
  - You have attributes and methods that are unrelated to the purpose of the class. This is also known as **poor cohesion**.
  - This class becomes hard to maintain and hard to change and add features to.
- If you have poor **encapsulation**, it means you have leaky abstraction, where external objects can change the private instances of the class and use them in inappropriate ways.
  - Imagine you have a *Product* class with a public *weight* attribute, and you have several classes that touch this attribute directly changing it into a wrong value below the minimum product weight.

Why Abstraction and Encapsulation?

- Poor encapsulation implies poor abstraction and poor modularity.
- This makes adding things unclear and hard to do without breaking existing code.
- This leads to weak cohesion.
- We strive for good abstraction and good encapsulation.
- We want a method to do one operation and do it right.
- We want a class to deal with one thing and control the accessibility of everything in their objects.
- We want a package to contain a set of closely related classes.

Relationships: Inheritance

- Inheritance relationships are **“is-a”** relationship.
    - Student is a Person, Truck is a Vehicle, etc.
- Superclasses are more general than subclasses.
- Subclasses are more specific than superclasses.
- One class can extend another class.
- The subclass can add additional behavior or extend/override existing one.
- Inheritance allows us to build classes based on other classes and avoid duplicating code.
- The **extends** keyword indicates inheritance and the **super** keyword refers to superclass.

Inheritance Example (I)

```java
class Product {
  private int id;
  private float price;
  private String name;

 public Product(int id, float price, String name){
    this.id = id;
    this.price = price;
    this.name = name;
  }
 public double getSalePrice(double percentage){
    return this.price - ((percentage/100) * this.price);
 }
 public void addToShoppingCart(){
    System.out.println(this.name + " has been added to the shopping cart. ");
 }
}
```

Inheritance Example (II)

```
class FoodProduct extends Product{
  private LocalDate expirationDate;

  public FoodProduct(int id, float price, String name, LocalDate expirationDate){
    super(id, price, name);
    this.expirationDate = expirationDate;
  }
}

class ElectricProduct extends Product{
  private String voltage;

  public ElectricProduct(int id, float price, String name, String voltage){
    super(id, price, name);
    this.voltage = voltage;
  }
}
```

Polymorphism “Many Forms” (I)

- Polymorphism is about being able to refer to different subclasses of a superclass in the same way.
- Objects of different types can access the same way.
- We can refer to classes that are related to each other by inheritance in the same way.

Polymorphism “Many Forms” (II)

```
Product p1  = new FoodProduct(3452, 10.0, “Cheddar Cheese”, “2022-06-07”);
Product p2  = new ElectricProduct(4875, 30.0, “Extension cord”, “220v”);
System.out.Println(p1.getSalePrice(20.0));
System.out.Println(p1.addToShoppingCart());
System.out.Println(p2.getSalePrice(20.0));
System.out.Println(p2.addToShoppingCart());
```

Polymorphism “Many Forms” (III)

- Using the enhanced for statement (a.k.a. For-Each Loop)

```
Product p1  = new FoodProduct(3452, 10.0, “Cheddar Cheese”, “2022-06-07”);
Product p2  = new ElectricProduct(4875, 30.0, “Extension cord”, “220v”);
Product [] products = {p1, p2};
for(Product p: products){
   System.out.println(p.getSalePrice(20.0));
   System.out.println(p.addToShoppingCart());
}
```

Demo

Inheritance: The Good!

- Code organization
  - Code is broken and arranged in a related class and package.
- Code reuse
  - Why not copy and paste the code whenever you need it?
- Extension
  - Subclasses add new methods or customize and override inherited ones
- Readability
  - The code is readable due to the clear model structure of inheritance.
- But despite all these benefits, Inheritance comes with serious costs.

Inheritance: The bad!

- **Inheritance can be powerful. But..**
- **Inheritance is static** (i.e., defined at compile time).
  - We can't change the implementation inherited from super classes at runtime.
- **Inheritance breaks encapsulation** because it exposes the internal protected data of superclasses to subclasses.
  - Subclasses can access details of their parent classes.
- Inheritance creates **tight coupling** between superclasses and subclasses.
  - Changes in the superclass implementation will force subclasses to change.

Coupling

- Coupling is the degree of interdependence between classes, packages, or methods.
- **Tight coupling** in classes means classes are strongly connected to the point that they can’t be changed without breaking other parts in the system.
- With tight coupling, a small change in one method or attribute will result in ripple effects.
- Tight coupling will result in spending a long time debugging and understanding the relationships between parts of the system.

Association, composition, aggregation and delegation

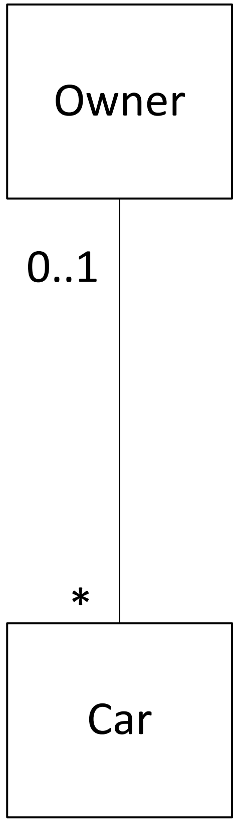
- What are the different types of associations?
- Why do we need aggregation and composition?
- Why do we want to favor composition over inheritance?
- What is delegation and how can it be implemented?

Relationships: Association

- Association is an “**has-a**” relationship
- A class has a reference to another object for another class.

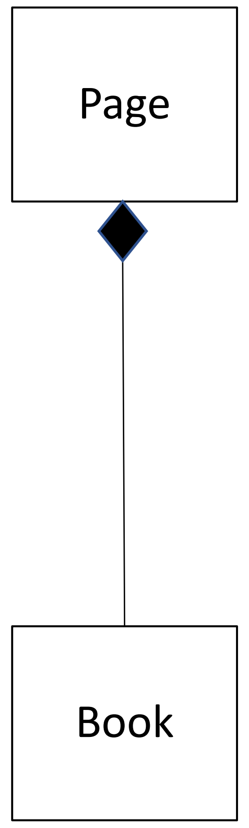
```
   public class Car{
         private Owner o;
   }
```

- In UML, multiplicity can indicate the number of instances involved in the relationship.
- Associations can also convey whole-part relationships (e.g., **Composition** and **Aggregation**).



Relationships: Composition (I)

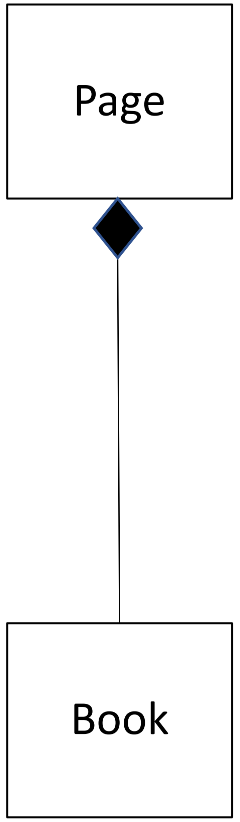
- Composition is a **“has-a**” or “**whole-part ”** special association relationship.
- Composition is a **strong** association with ownership.
  - An object is exclusively owned by another object.
  - If the composing object is deleted, all composed and associated objects are deleted too.
- In UML, composition is indicated with a solid/black diamond attached to the composing class.



Relationships: Composition (II)

- Composition in Java:

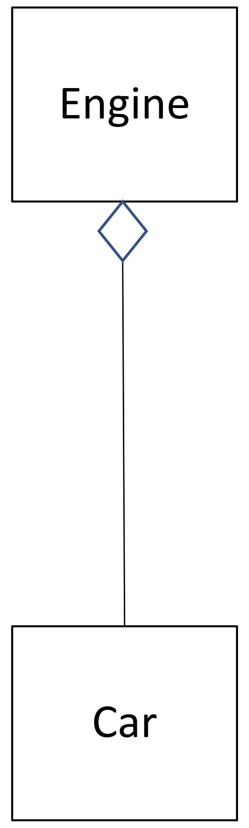
```
public class Book{
    private Page p;
    public Book(){
      this.p = new Page();
    }
}

public class Client{
    public static void main(String[]args){
      Book b = new Book();
      b = null;  // Both b and p are deleted.
    }
}
```



Demo

Relationships: Aggregation (I)

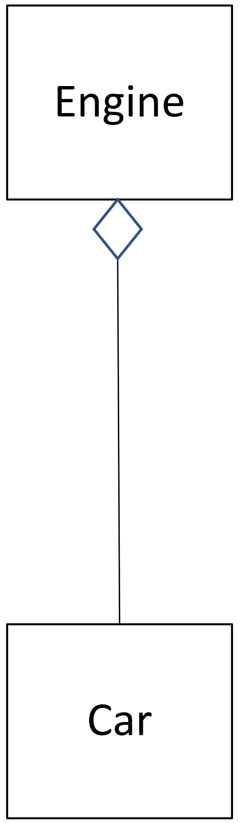
- Aggregation is a **“has-a**” or “**whole-part ”** special association relationship.
- Aggregation is a **weak** association with **no ownership**.
    - An object is not owned by the composing object.
    - If the composing object is deleted, all composed and associated objects are not affected.
- In UML, composition is indicated with a white diamond attached to the composing class.



Relationships: Aggregation (II)

- Aggregation in Java:

```
    public class Car{
        private Engine e;
        public Car(Engine e){
          this.e = e;
        }
    }

    public class Client{
        public static void main(String[]args){
          Engine e = new Engine();
          Car c = new Car(e);
          c = null;  // Only c is deleted; e is not deleted.
        }
    }
```



Demo

Relationships: Delegation (I)

When a method is called and needs to handle a message, there’re typically four ways to handle the request:

1. Handle the request by implementing the logic in the same method (**implementation**).
2. Leave it to the superclass to handle it via **inheritance**.
3. Pass the request to another object (**delegation**)
4. A combination of the three options.

Relationships: Delegation (II)

- Delegation is also often considered as a **“has-a”** relationship.
- A class has a reference to another object for another class (a.k.a. helper class) and sends a message to it when needed (invoke a method in the helper class).

Relationships: Delegation (III)

- Delegation in Java:

```
public class ShoppingList{
    private String item;
    private List<String> items;
    private Notification notification;

    public ShoppingList(){
       items = new LinkedList<String>();
       this.notification = new Notification();
    }
    public void add(String item){
      this.items.add(item);
    }
   public void sendNotification(){
      this.notification.post(this.items);
    }
}
```

Here:

ShoppingList **Delegates** all the work to both the Java's Linkedlist class and a Notification class.

Demo

Use Composition instead of subclassing (I)

“Favor Composition over Inheritance”, this is a fundamental object-oriented design principle.

Use Composition instead of subclassing (II)

In Effective Java by Joshua Bloch, Item 16: Favor composition over inheritance:

"[Subclassing] is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software."

" Instead of extending an existing class, give your new class a private field that references an instance of the existing class. … The resulting class will be rock solid, with no dependencies on the implementation details of the existing class "

Object-Oriented Programming in Java

An overview of fundamental Object-Oriented concepts in the Java programming language.

Khalid Alharbi, Ph.D.

Functions vs. Methods

- Both **functions and methods** refer to a series of statements inside a block of code. They may take inputs (called parameters) and return some value to a caller.
- The difference is where they're declared:
  - A **method** is defined **inside** of a class.
  - A **function** is defined **outside** of a class.
- In Java, C#, there're only methods.
- In C, Erlang there're only functions.
- In C++, Python, there're both functions and methods.

Parameters vs Arguments

- Parameters: Variables that are part of the method's signature or declaration.

```
int add(int x, int y){ // x and y are parameters.
     return x + y;
}
```

- Arguments: values or variables passed when invoking a method or a function.

```
int x = 5;
add(x, 10); // x and 10 are arguments.
```

Class Variables vs Instant Variables

Instant Variables	Class Variables
Allocated when an object is created using the keyword “new” and destroyed when the object is destroyed.	Allocated when the program starts and destroyed when the program exits.
Declared as normal inside a class but outside a method or a block.	Declared using the keyword “static” inside a class but outside a method or block.
Accessed directly by calling the variable name.	Accessed by calling ClassName.VariableName
Hold values that are unique to each object.	Hold values that are shared by all objects or instances of the class.

```
public class Student {
  private String name;    // instant variable
  private int id;         // instant variable
  private static count;   // class variable
  public Student(String name){
    this.name = name;
    this.id = ++count;
  }
}
```

Example: Using only instance variables

```java
class Product {
  private int id;
  private float price;
  private String name;
  private int totalProducts;

  public Product(String name, float price){
    this.name = name;
    this.price = price;
    this.id = ++totalProducts;
  }
  public String toString(){
    return "Name:" + this.name +"\nPrice"+ this.price + "\nId: " + this.id;
  }
  public int getTotalProducts(){
    return this.totalProducts;
  }
}
```

Example: Using only instance variables (cont.)

```
class App{
  public static void main(String[]args){
    Product p1 = new Product("Chair", 150.00f);
    Product p2 = new Product("Desk", 100.00f);
    System.out.println(p1);
    System.out.println(p2);
    System.out.println("Total products: " + p2.getTotalProducts());
  }
}
```

```
Name:Chair
Price150.0
Id: 1
Name:Desk
Price100.0
Id: 1
Total products: 1
```

Demo

What went wrong?

Example: Using class and instance variables

```
class Product {
  private int id;
  private float price;
  private String name;
  private static int totalProducts; // class variable

  public Product(String name, float price){
    this.name = name;
    this.price = price;
    this.id = ++totalProducts;
  }
  public String toString(){
    return "Name:" + this.name +"\nPrice"+ this.price + "\nId: " + this.id;
  }
  public int getTotalProducts(){
    // class variables must be qualified by type name (class name).
    return Product.totalProducts;
  }
}
```

Example: Using only instance variables (cont.)

```
class App{
  public static void main(String[]args){
    Product p1 = new Product("Chair", 150.00f);
    Product p2 = new Product("Desk", 100.00f);
    System.out.println(p1);
    System.out.println(p2);
    System.out.println("Total products: " + p2.getTotalProducts());
  }
}
```

```
Name:Chair
Price150.0
Id: 1
Name:Desk
Price100.0
Id: 2
Total products: 2
```

Demo

What is not so right?

Example: Using class and instance variables [Hot Fix]

```
class Product {
  private int id;
  private float price;
  private String name;
  private static int totalProducts; // class variable
  public Product(String name, float price){
    this.name = name;
    this.price = price;
    this.id = ++totalProducts;
  }
  public String toString(){
    return "Name:" + this.name +"\nPrice"+ this.price + "\nId: " + this.id;
  }
  // getTotalProducts should be a class method
  public static int getTotalProducts(){

    // class variables must be qualified by type name (class name).
    return Product.totalProducts;
  }
```

Example: Using only instance variables [Hot Fix](cont.)

```
class App{
  public static void main(String[]args){
    Product p1 = new Product("Chair", 150.00f);
    Product p2 = new Product("Desk", 100.00f);
    System.out.println(p1);
    System.out.println(p2);
    System.out.println("Total products: " + Product.getTotalProducts());
  }
}
```

```
Name:Chair
Price150.0
Id: 1
Name:Desk
Price100.0
Id: 2
Total products: 2
```

Demo

Abstract class vs Interface

Abstract Class	**Interface**
A class with one or more abstract method (methods that are declared with no implementation).	A placeholder for a collection of methods with no implementation.
Can include non-abstract methods with complete implementation.	Can’t have concrete implementation of the method.
An abstract class can have properties, abstract methods, and non-abstract methods.	An interface only include the method’s signature or declaration with no implementation.
A class can extend an abstract class and implements its abstract method.	A class can implement multiple interfaces but only inherits or extends one super class.
Example: Food is an abstract concept for things we can eat. We can’t create an instance of Food itself but we can create an instance of Pizza, Soup, or Rice, which are types of Food.	Example: Skills (Athlete, Poet, Writer, Chef, etc.). We can’t create an instance of a Skill but we can create an instance of Person, who may acquire multiple skills but remains a Person.

Abstract class example (Declaration)

```java
abstract class Food{
   /* TODO: variable declarations: name, quantity, weight,
    * weightUnit, servingSize, servingUnit, calories, etc. */
   // TODO: Constructor

   // All food has calories that can be calculated differently.
   public abstract float getTotalCalories();
   // All food can be eaten in different ways.
   public abstract String eat();
   // All food has the same calories printed info
   public String getCaloriesInfo(){
       return "Name: " + this.name + "\n" + "Quantity: " + this.quantity +
       "\n" +  "Weight: " + this.weight + " " + this.weightUnit + "\n" +
      "Serving Size: " + this.servingSize + " " + this.servingUnit + "\n" +
      "Calories:" + this.calories + "\n";
}
```

Abstract class example (Usage)

```
class Orange extends Food{
   private static final float CALORIES_PER_GRAM = .37f;
   private String type;

   public Orange(float weight, String weightUnit, String type) {
     super(weight, weightUnit);
     super.calories = CALORIES_PER_GRAM;
     this.type = type;
   }
   // All food has calories that can be calculated differently.
   public float getTotalCalories(){
     return super.calories * super.weight;
   }
   // All food can be eaten in different ways.
   public String eat(){
      return ”Peel an orange by hand or use a knife to cut it into wedges.”
   };
}
```

Interface example (Declaration)

```
interface Athlete{
   // An athlete may compete in different sports.
   void compete(String competition);
   // An athlete may train or practice in different ways.
   void practice();
   // An athlete may develop different performance levels.
   void developPerformanceLevels(int level);
}

interface Artist{
   // An artist may paint.
   void paint();
   // An artist may sell their arts.
   float sell();
}
```

Interface example (Usage)

```java
class Talented implements Athlete, Artist{
   private int age;
   private String name;

   public Talented(int age, String name) {
     this.age = age;
     this.name = name;
   }
   // can compete, practice, developPerformanceLevels, paint, and sell
   @Override
   public void compete(String competition){
     System.out.println("I'm competing in " + competition);
   }
   @Override
   public float sell(){
      return 499.99f;
   };
   // TODO: Override the remining abstract methods
}
```

The final keyword

- The **final** keyword can be added to a variable, method or class.
- A final variable is a constant whose value is initialized once and cannot be changed.

```
static final double PI = 3.14159265358979
```

- A final method cannot be overridden by subclasses.

```
final void generateReport(){}
```

- A final class classes cannot be subclassed and inherited from.

```
final class Math{}
```

Wrap up and what's next

- Fundamental concepts in Object-Oriented Programming.
  - Abstraction, encapsulation, inheritance, polymorphism, composition, aggregation and delegation.
- The benefits and disadvantages of inheritance.
- Tight coupling and poor cohesion.
- Favor composition over inheritance.
- Abstract classes and interfaces.
- Class variables and instance variables.
- **Next:** Introduction to design patterns and creational design patterns