Session 16 (of 24)

PGR112 Objektorientert programmering
Yuan Lin / yuan.lin@kristiania.no

Today’s Main Goal

- OOP Repetition
- A Step-by-Step go through of BankLoanSolution
- Recipes for Clean Code

Today’s Main Goal

- OOP Repetition

Object Oriented Programming

- Object oriented programming is a methodology to design a program using classes and objects
- Major OOP concepts we have learned
  - Class
  - Object
  - Inheritance
  - Polymorphism
  - Encapsulation
  - Abstraction
- Other terms we have learned and used in OOP design
  - Aggregation
  - Low coupling & high cohesion
- We have also learned
  - Exception handling, HashSet, HashMap, Optional, equals, hashCode, Lambda..

Class

- A class is user defined prototype from which objects are created
- It represents the set of properties or methods that are common to all objects of one type.
- A class declaration can include
  - Modifiers: public, default access
  - Class name
  - Superclass: if any, preceded by keyword extends. In java, a class can only extend from one superclass. (Java does not support multi-inheritance)
  - Interfaces: if any, preceded by keyword implements. In java, a class can implement more than one interface. Multi-interfaces are comma-separated
  - Body: the class body is surrounded by {}
- Example

```
public class child implement parent implement interface1, interface2 {
    private String prop1;
    static void methods1() {/**method body..*/}
}
```

Object

- An object can be defined as an instance of a class.
- An object contains an address and takes up some space in memory.
- A constructor call (new) returns a reference to an object. A reference is information about the location of object data.
- Assigning a reference type variable copies the reference.
- Object reference is stored in stack, while object stores in heap.
- Setting an object to null makes the object unreachable and triggers garbage collection.
- The object class is the parent class of all the classes in java by default.

```
Person person0;
Person person1 = new Person();
Person person2 = new Person();
person1 = person2;
person1 = null;
```

Inheritance

- Inheritance allows one class to inherit the features (fields and methods) of another class.
  - Superclass (parent class) is the class who is inherited
  - Subclass (child class) is the class who inherits the other class
- Java does not support multiple inheritance.
- Inheritance achieves code reusability.
- Inheritance is used to achieve runtime polymorphism.
- During inheritance, only the object of subclass is created, not the superclass.

```
public class Child extends Parent{
    public static void main(String[] args) {
        Child child1 = new Child();
        Child child2 = new Parent(); //upcasting
    }
}
public class Child extends Parent1, Parent2{
    // this is wrong. Java does not support multi-inheritance
}
```

Polymorphism

- In Java, polymorphism is mainly divided into two types:
  - Compile-time polymorphism
  - Runtime polymorphism
- Compile-time polymorphism is achieved by function overloading
  - Functions are said to be overloaded if multiple functions are of the same name but different parameters (data type, number, order)
  - Functions are not said to be overloaded if only return type is different since JVM is not able to decide during compilation time.
- Runtime polymorphism is achieved by function overriding
  - Function overriding occurs when a method in the parent class is overridden by the child class.
  - A function call to the overridden method is resolved at Runtime.
  - Use @Override is a nice coding practice.
- Method hiding
  - When a subclass defines a static method with the same signature as a static method in the superclass.

Polymorphism

- Constructor overloading example – Function overloading
- A upcasting example – function overriding

```
public Computer(Processor processor, Memory memory) {
    this.processor = processor;
    this.memory = memory;
}
public Computer(Processor processor, Memory memory, SoundCard soundCard) {
    this.processor = processor;
    this.memory = memory;
    this.soundCard = soundCard;
}
```

```
/**
 * multi-level polymorphism
 * here MainecoonBabyCat does not override catActivity(), so
 * the method in its immediate parent class BabyCat is called
 */
Cat cat1, cat2, cat3;
cat1 = new Cat( id: 101);
cat2 = new BabyCat( id: 102);
cat3 = new MainecoonBabyCat( id: 103);
cat1.catActivity();
cat2.catActivity();
cat3.catActivity();
/**
 * data member is not overriden, but method does
 */
System.out.println("The weight of a MainecoonBabyCat is " + cat3.weight);
```

Encapsulation

- A Java class is the example of encapsulation.
- Technically in encapsulation, the variables of a class is hidden from any other class and can be accessed only through methods if its own class.
- Encapsulation is achieved by Getters/Setters/Access modifiers
  - Declare all variables as private
  - Provide public Getters/Setters for access from outside of the class.
- Advantages of encapsulation
  - Data hiding; Read-only or write-only access flexibility, Reusability; Easy testing

```
private String ssn;

public String getSsn() { return ssn; }

public void setSsn(String ssn) { this.ssn = ssn; }
```

Abstraction

- In Java, abstraction is achieved by interface and abstract classes.
- Feature of abstract classes – keyword: extends
  - An abstract class is a class that is declared with an abstract keyword.
  - An abstract method is a method that is declared without implementation.
  - An abstract class can have both abstract and concrete methods (methods that have function body).
  - An abstract method must be overriden in the subclass, or either make the subclass abstract.
  - An abstract class can not be directly instantiated with new.
- Features of interfaces – keyword: implements
  - Interfaces can only have abstract methods and variables.
  - If a class that implements an interface does not implement abstract methods, it must be declared as abstract.
  - A class can implement multiple interfaces.

Abstraction

- Abstract class example
  - The abstract class contains both abstract and concrete methods.
  - Class that extends abstract class must override the abstract method (or declare itself as abstract class)

- Interface example
  - The interface contains only abstract methods.
  - Class that implements interface must override the abstract method (or declare itself as abstract class)

```java
public abstract class Loan implements java.io.Serializable {
    double annualInterestRate;
    int numberOfYears;
    double loanAmount;
    abstract void getAnnualInterestRate();
    public void calculateLoanPayment() {
        double monthlyInterestRate = annualInterestRate / 1200;
        double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
                (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
        double totalPayment = monthlyPayment * numberOfYears * 12;
    }
}
public class CarLoan extends Loan {
    private String carModel;

    @Override
    void getAnnualInterestRate() { annualInterestRate = 0.4; }
}
```

```java
public interface ILoanProxy {
    void registerLoan(Loan loan) throws Exception;
    void printAllLoans() throws Exception;
}
public class LoanProxy implements ILoanProxy {
    @Override
    public void registerLoan(Loan loan) throws Exception {
        /**method body*/
    }
    @Override
    public void printAllLoans() throws Exception {
        /**method body*/
    }
}
```

Aggregation

- Aggregation represents the relationship where one object contains other objects.
- It represents the weak relationship between objects.
- It is also terms as a has-a relationship (while inheritance represents the is-a relationship)

```
public abstract class Loan implements java.io.Serializable {
    double loanAmount;
    CustomerInfo customer;
    public Loan(double loanAmount,CustomerInfo customer) {
        this.loanAmount = loanAmount;
        this.customer = customer;
    }
}
public class CustomerInfo {
    private String ssn;
    public String getSsn() { return ssn; }
    public void setSsn(String ssn) { this.ssn = ssn; }
}
```

Low Coupling High Cohesive

- Coupling refers to the level one class knows details / has dependency to another class.
  - Private, protected, public modifiers to display visibility level of a class
  - We can use interfaces for the weaker coupling since no concrete implementation is provided.
- Cohesion refers to the level of a component which performs a single well-defined task.
  - High cohesive means a well-defined task is contained in a single package/unit.
  - Weak cohesive means a module might contain many unrelated tasks.
- Below example shows low coupling high cohesive since Loan related tasks are located inside ILoanProxy interface while, customer related tasks are located inside ICustomerProxy.

```
public interface ILoanProxy {
    Loan getLoan(Integer loanId);
    void registerLoan(Loan loan) throws Exception;
    void printAllLoans() throws Exception;
    void printLoanbySsn(String ssn) throws Exception;
}
public interface ICustomerProxy {
    CustomerInfo getCustomer(String ssn);
    boolean isValidSsn(String personalIdentifyNumber) throws Exception;
    void registerCustomer(CustomerInfo customerInfo) throws Exception;
}
```

Exception handling

- Types of Java exceptions
  - Checked Exception: exceptions that are checked at compile time
  - UnChecked Exception: exceptions that are checked at run time
- Checked exceptions must be handled (try-catch) or specified by method signature (using throws)
- Unchecked exceptions are not forced by compiler to either handle or specify.
- Try-Catch-Finally block is used to handle exceptions.
- We can specify our own customized exceptions, which extends from Exception class.
- Throw & Throws: throw is used to throw an exception explicitly; throws is used in method signature to declare an exception might occur from inside the method.

```java
public void runExample1() {
    try {
        String s="OOP course";
        int i = Integer.parseInt(s);
        System.out.println(i);
    } catch (NumberFormatException e) {
        System.out.println(e);
    } finally {
        System.out.println("final block is " +
                "executed anyhow...");
    }
    System.out.println("rest of the code...");
}
```

```java
class InvalidAmountException extends Exception {
    public InvalidAmountException(String message) { super(message); }
}
class InsufficientFundException extends Exception {
    public InsufficientFundException(String message) { super(message); }
}

class StudentBank{
    private double balance;
    public void deposite(double amount) throws InvalidAmountException {
        if(amount <= 0) {
            throw new InvalidAmountException(amount + "is not valid");
        }
        balance+= amount;
    }
    public void withdraw(double amount) throws InsufficientFundException {
        if(balance < amount) {
            throw new InsufficientFundException("Insufficient fund");
        }
        balance-= amount;
    }
    public void showBalance() { System.out.println("Your current balance is "+balance); }
}
```

ArrayList, HashSet, HashMap

- Collection includes List and Set, where ArrayList is List, HashSet is Set.
- ArrayList is a resizable array. You can access an element by using index.
  - It stores elements in order.
  - You can store duplicated elements.
- HashSet is a collection of items. You can access items by iterative methods:
  - Use enhanced for loop
  - Use Iterable forEach loop
  - Use Iterator<T> interface
- Different from Collection, HashMap uses (key, value) pair to store elements. You can access element by key.
- HashSet does not allow duplicate values. HashMap allows duplicate values, while only unique keys.
- Neither HashSet nor HashMap maintains order.

HashSet, HashMap

- HashSet and HashMap

```
Animal myPig = new Pig( id: 1); // Create a Pig object
Cat cat = new Cat( id: 2);
HashSet<Animal> animals=new HashSet();
/**
 * hashset can contain duplicate elements, but during iterating,
 * duplicated elements will be ignored
 */
animals.add(myPig);
animals.add(myPig);
animals.add(cat);
/**
 * hashset can contain null element
 */
animals.add(null);
Iterator<Animal> i=animals.iterator();
/**
 * The elements iterate in an unordered collection
 */
while(i.hasNext())
{
    System.out.println(i.next());
}
```

```
/**
 * HashMap implements iterable interface
 */
HashMap<Integer, Animal> animals = new HashMap<>();
animals.put(myPig.id, myPig);
animals.put(cat.id, cat);
System.out.println("Printing animals using enhanced for loop:");
for (Animal animal :
        animals.values()) {
    animal.animalSound();
    animal.sleep();
    System.out.println(animal);
}
System.out.println("Printing animals using forEach loop:");
animals.values().forEach(
        (animal -> {
            animal.animalSound();
            animal.sleep();
            System.out.println(animal);
        })
);
System.out.println("Printing animals using Iterator:");
Iterator<Integer> it = animals.keySet().iterator();
while(it.hasNext())
{
    int key=(int)it.next();
    animals.get(key).animalSound();
    animals.get(key).sleep();
    System.out.println(animals.get(key).toString());
}
```

Today’s Main Goal

- A Step-by-Step go through BankLoanSolution

A Step-by-Step go through BankLoanSolution

- What OOP concepts have we used for BankLoanSoluton?
  - Encapsulation
  - Inheritance
  - Abstraction (abstract & Interface)
  - Polymorphism (overriding, overloading, upcast, downcast)
  - Aggregation
  - Low coupling high cohesion
- We also used
  - HashSet, Lambda, etc…

Today’s Main Goal

- Recipes for Clean Code

Recipes for Clean Code

- Naming Convention
  - Meaningful names
  - Camel case for variable and functions, Nouns and Pascal case for class name
  - SNAKE UPPER CASE for constants
- Low coupling, high cohesion
  - Avoid large functions
  - Single responsibility principle
- Code reusability
- Avoid deep nesting
- Comments