Session 8 (of 24)

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

Today’s Main Goal

- A Recap to Inheritance & Aggregation
  - HAS-A
  - IS-A
- Abstract
  - Abstract class
  - Abstract method
  - @Override annotation
- Static
- HashMap
- Practice Time

Status

- Step 1: IntelliJ, Git, Hello World
- Step 2: Variable, Method, Data type, Scope, Control statement
- Step 3: Class, Object, Scanner, Encapsulation
- Step 4: Package, Import, Modifiers, Overload, ArrayList, Enums
- Step 5: Scanner (cont.), Input validation, Debugger, Write test
- Step 6: Exception handling, File I/O, Logger, Date/Time
- Step 7: Inheritance, Aggregation, Super, Final
- Now: Abstract
- …

A Recap to Inheritance

- What is inheritance?
  - Inheritance means that you can create new classes built upon existing classes
  - Inheritance represents IS-A relationship.
- How to obtain inheritance?
  - Extends is the keyword
- We understood how sub-class can inherit from parent-class
  - Reuse of code

A Recap to Super

- Use of Super keyword
  - super can refer to immediate parent class object
  - super.methodname() can be used to invoke parent class method
  - super() can be used to invoke parent class parameterized or default constructor

```java
public abstract class Mammal extends Animal{

    public Mammal(int id) { super(id); }

    public void regulateBodyHeat() { System.out.println("Change in temp. Regulating body heat."); }
    public abstract void provideMilkForBaby();
}
```

```java
public class Cat extends Mammal{

    public Cat(int id) { super(id); }
    @Override
    public void provideMilkForBaby() { System.out.println("Providing milk through one of my 6-11 teats."); }
    @Override
    public void animalSound() { System.out.println("The cat says: miaaauuu"); }
    public void CatActivities(){
        super.regulateBodyHeat();
        animalSound();
    }
}
```

Is-a

- We should use inheritance with caution, only in context of “is-a”
- Inheritance provides a string link between two classes

```java
public class Cat extends Mammal{

    public Cat(int id) { super(id); }

    @Override
    public void provideMilkForBaby() { System.out.println("Providing milk through one of my 6-11 teats."); }

    @Override
    public void animalSound() { System.out.println("The cat says: miaaauuu"); }

    @Override
    public String toString() {
        return "Cat{" +
                "id=" + id +
                '}';
    }
}
```

A Recap Aggregation

- What is aggregation?
  - If a class have an entity reference, it is known as Aggregation.
  - Aggregation represents Has-a relationship.
- Why use aggregation?
  - Inheritance should be used only when the Is-a relationship is maintained throughout the lifetime of the objects involved
  - Otherwise aggregation is the best choice
- We also talked about composition, but will not emphasize it in this topic.

Has-a

- Code re-use can be best achieved when there is no Is-a relationship
- In Java, has-a simply means that an instance of one class has a reference to an instance of another class.
- Both classes are independent. They can exist without each other.

```java
public class Address {
    String street;
    int postal;
    String city;
    String state;
    String country;
    Address(String street, int postal, String city, String state, String country)
    {
        this.street=street;
        this.postal = postal;
        this.city =city;
        this.state = state;
        this.country = country;
    }
}
```

```java
public class College {
    String collegeName;
    //Creating HAS-A relationship with Address class
    Address collegeAddr;
    College(String name, Address addr){
        this.collegeName = name;
        this.collegeAddr = addr;
    }
}
```

So – What is Abstraction?

- The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context (John V. Guttag)

- What does this exactly mean?
  - We want to hide implementation details
  - We show only functionality to the user.

- We can perform abstraction using two mechanisms in Java
  - Abstract (this lecture) . Achieves 0-100% abstraction
  - Interface (next lecture) – achieves 100% abstraction

  Q: Inheritance has to have abstraction?

  A : The answer is no. But abstraction is a good coding practice

Abstract

- Abstract reserved keyword can be applied to
  - Abstract class
  - Abstract method

Some rules for abstract classes

- An abstract class must be declared with the abstract keyword
- Abstract class can have abstract and non-abstract method (method with body)
- Abstract class can not be instantiated
- Abstract class can have constructors and static methods
  - Why static method is allowed? Because static method can be called directly without an instance.
- Abstract class can have final methods which forces subclasses not to change the body of the method.

```
abstract class Shape{
    private int Id;
    private boolean filled;
    protected Color color;
    static double weightOfLine;
    /** abstract class can also have constructor
     * @param Id
     */
    public Shape(int Id) {
        this.Id = Id;
        System.out.println("A shape is created, and its id is " + Id);
    }
    /** abstract method
     */
    abstract void draw();
    /** non-abstract method (method with body)
     */
    public void colorMe(){
        this.color = Color.Green;
    }
    /** final method, not to be overriden
     */
    final void fillMe() {
        this.filled = true;
        System.out.println("I am filled");
    }
    static void setLineWeight(double weightOfLine) {
       Shape. weightOfLine = weightOfLine;
        System.out.println("The weight of line is " + weightOfLine);
    }
}
```

Some rules for abstract classes

- Do you notice how Super keyword is used here?
- And do you notice how Static is used here?

```
class Rectangle extends Shape{
    public Rectangle(int Id) {
        super(Id);
    }
    /**
     * implementation of the abstract method
     */
    void draw(){
        System.out.println("drawing rectangle");
    }

    /**
     * override non-abstract method
     */
    @Override
    public void colorMe(){
        this.color = Color.Red;
        System.out.println("my color is "+this.color);
    }
    public void myOperations() {
        this.draw();
        super.fillMe();
        colorMe();
        Shape.setLineWeight(1.5);
    }
}
```

Abstract methods

- Abstract methos is a method declared as abstract and has no implementations
- Abstract methods can only be declared in abstract class.
- The child class thus must ensure that the abstract methods are implemented. But do they HAVE to do it?
- No ☺
- If the child class does not implement it, the class must be abstract.

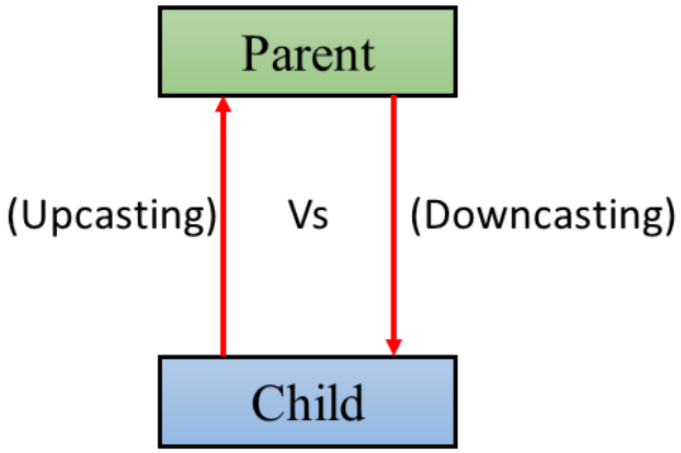
```
abstract class Rectangle extends Shape{
    public Rectangle(int Id) {
        super(Id);
    }
    /**
     * override non-abstract method
     */
    @Override
    public void colorMe(){
        this.color = Color.Red;
        System.out.println("my color is "+this.color);
    }
    public void myOperations() {
        this.draw();
        super.fillMe();
        colorMe();
        setLineWeight(1.5);
    }
}
class Square extends Rectangle{
    public Square(int Id) {
        super(Id);
    }
    @Override
    void draw() {
        System.out.println("drawing square");
    }
}
```

Abstract methods

- Note that both apply.
- This is called upcasting. Upcasting is the typecasting of a child object to a parent.

  //upcasting

  Parent o = new Child();



From geeksforgeeks

```java
public class Main {
    public static void main(String[] args) {
        Square square = new Square( id: 1);
        square.setLineWeight(2.0);
        square.myOperations();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Rectangle square = new Square( id: 1);
        square.setLineWeight(2.0);
        square.myOperations();
    }
}
```

@Override Annotation

- The @Override annotation indicated that the child class method is overwriting its base class method (with method body).
- All it does is validate that a superclass method with the same signature exists.
- Abstract method doesn’t need @Override annotation.
- Do we HAVE TO use @Override annotation?
- No ☺ It is not necessary, but highly recommended. Why?
  - If the annotated method doe not actually override anything, the compiler issues a warning
  - It can help to make the source code more readable.

@Override toString()

- When we do System.out.println(), we get something like LectureCode.Session8.src.AbstractExample.Square@7cc355be
- The output is classname@hashcode
- Since all classes in Java inherit from the Object class, and the object class has some basic methods such as toString().
- We can override the toString() method

```
@Override
public String toString() {
    return String.format("A Square with perimeter= %f, color = %s, fill = %b, lineWeight = %f " +
            "is created, which is a subclass of %s", perimeter, color, filled, weightOfLine, super.toString());
}
```

```
A Square with perimeter= 1,500000, color = Red, fill = true, lineWeight = 2,000000 is created, which is a subclass of A Rectangle
 with color = Red, fill = true, lineWeight = 2,000000 is created
```

Static

- Static is applicable for Blocks, Variables, Methods, Classes
- When a member is declared static, it can be accessed before any objects of its class are created, or without reference to any object.
- Static variables:
  - All instances of the class share the same static variable. Static variable can be accessed by non-static method.
  - In Java, static variables can only be created at class level. Not allowed in local methods.
- Static methods:
  - They can only directly call other static methods
  - They can only directly access static variables
  - They cannot refer to **this** – why?
- Java supports static classm – which we will introduce in later lectures.

Static

- Abstract class can have static methods
  - Why static method is allowed? Because static method can be called directly without an instance.
- Can Static method be abstract in Java?
- No ☹ Why?
  - When a method is described as abstract by using the abstract type modifier, it becomes responsibility of the subclass to implement it
  - When a method is described as static, it makes it clear that this static method cannot be overridden by any subclass. Because static members are compile-time elements, while overriding them will make it runtime elements.
  - Thus Abstract static method is not allowed in Java.

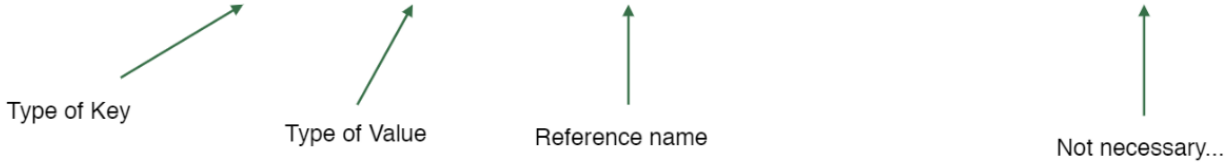
HashMap

- Using ArrayList has some downsides
- If we want to retrieve one specific object (without knowing the index), then it’s a bit cumbersome.
- Let’s look at an alternative: HashMap…

HashMap

Create HashMap

```
HashMap<String, String> capitalCities = new HashMap<String, String>();
```



Use HashMap

```
capitalCities.put("England", "London");
```

```
capitalCities.get("England");
```

```
capitalCities.remove("England");
```

```
capitalCities.clear();
```

```
public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
    for (Map.Entry<String, String> entry :
            capitalCities.entrySet()) {
        System.out.println(entry.getKey());
        System.out.println(entry.getValue());
    }
}
```

Some rules about HashMap

- Java HashMap allows us to store key and value pair.
- Java HashMap Is great to use if we want to find an object based on a key.
- Java HashMap contains only unique keys.
- If you try to insert the duplicate key, it will replace the element of the corresponding key.
- Java HashMap may have one null key and multiple null values.
- Java HashMap maintains no order
- We can put objects of defined types and sub-types.
- HashMap class is found in java.util package.

Iterating over objects in a HashMap

- Iterate over HashMap can be done in many different ways.
- We can use value() method
- We can use entrySet() method

```
HashMap<Integer, Animal> animals = new HashMap<>();
animals.put(myPig.id, myPig);
animals.put(cat.id, cat);
System.out.println("Printing animals:");
for (Animal animal :
        animals.values()) {
    animal.animalSound();
    animal.sleep();
    System.out.println(animal.toString());
}
```

```
for (Map.Entry<Integer, Animal> animal :
        animals.entrySet()){
    animal.getValue().animalSound();
    animal.getValue().sleep();
    System.out.println(animal.getValue().toString());
}
```

Iterating over objects in a HashMap

- We can also use Iterator interface

```
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());
}
```

Before we end

- Goals for this session:
  - I can use abstract classes, and I understand what that means
  - I can use abstract methods, and I understand what that means
  - I understand what aggregation entails
  - I know Static
  - I understand why we use @Override annotation
  - I know how to use try-with-resources
  - I know how to use a HashMap

Remember, do not just read code, play with it.
Good luck with the tasks!

Remember, there is help available all week, use Mattermost or GitHub.