



# Session 8 (of 24)

PGR112 Objektorientert programmering

Yuan Lin / [yuan.lin@kristiania.no](mailto: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**
- ...

# 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

# Is-a

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

```
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 +  
            '}';  
    }  
}
```

# 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.

```
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;  
    }  
}
```

```
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
- When you think about OOP design in real life
  - Sub classes might have different functions thus need different implementations. Those functions can be defined in parent class as abstract methods
  - The common functions that sub classes share can be defined as regular methods or even final.

## 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
- Abstract class can have final methods which forces subclasses not to change the body of the method.

# Abstract methods

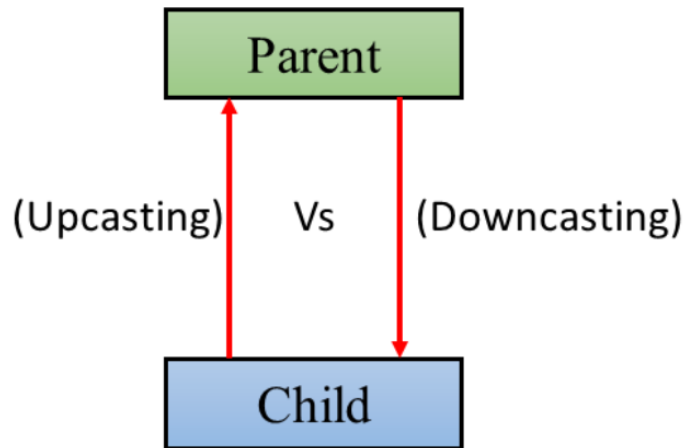
- Abstract method is a method declared as abstract and has no implementations
- Abstract methods can only be declared in abstract class.
- The child class thus must ensure that the abstract methods are implemented.
- But do they HAVE to do it? No 😊
- If the child class does not implement it, the class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

# Abstract

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

//upcasting

Parent o = new Child();



From geeksforgeeks

```
Shape circle = new Circle( Id: 1, radius: 2.0);
circle.setLineOfWeight(1.5);
System.out.println("The color is "+circle.getColor());
System.out.println("The area is " + circle.getArea());
System.out.println("The Perimeter is " + circle.getPerimeter());

Shape rectangle = new Rectangle( Id: 2, length: 4.0, width: 3.0);
rectangle.setColor(Color.Red);
System.out.println("The color is "+rectangle.getColor());
System.out.println("The area is " + rectangle.getArea());
System.out.println("The Perimeter is " + rectangle.getPerimeter());
System.out.println("The line of weight is " + rectangle.getLineOfWeight());
```

# @Override Annotation

- The @Override annotation indicates 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 does 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 Circle is created, whose id is %d, and is a subclass of %s", getId(), super.toString());
}
```

```
A Circle is created, whose id is 1, and is a subclass of A shape is created, and its id is 1
```

# 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

```
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();

}
```

```
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: miaaaauuu"); }
    public void CatActivities(){
        super.regulateBodyHeat();
        animalSound();
    }

}
```

# 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.

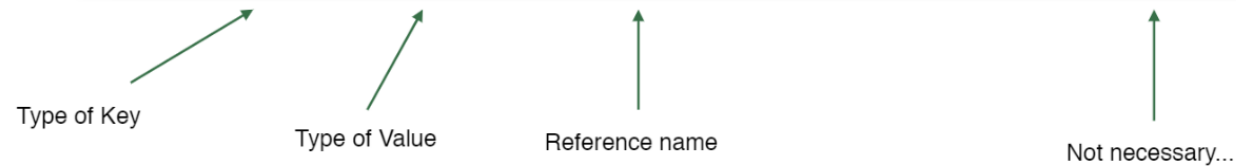
# HashMap

- 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.