Session 10 (of 24)

PGR112 Objektorientert programmering

Yuan Lin / yuan.lin@kristiania.no

Today’s Main Goal

- A Recap to Abstract & Interface
- Polymorphism
  - Compile-time polymorphism
  - Runtime polymorphism (dynamic method dispatch)
- Inner Class
- Upcasting & Downcasting
- Variable & Method hiding in Java

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
- Step 8: Abstract
- Step 9: Interface
- Now: Polymorphism
- …

A Recap to Abstract & Interface

- Q &A?

What is Polymorphism?

- Polymorphism is one of the core features of Object-oriented programming
- It allows you to define one interface and have multiple implementations
- So, what is the difference between interface and polymorphism?
  - Polymorphism is the abstract concept and interfaces are a way that implement that concept
  - All objects in Java are polymorphism, why? Because all Java classes extend the class object.
- Types of polymorphism
  - Compile-time polymorphism (static)
  - Runtime polymorphism (dynamic)
- Compile-time polymorphism is achieved by method overloading
- Runtime polymorphism is achieved by method overriding
- Overloading of static method in Java is compile time polymorphism

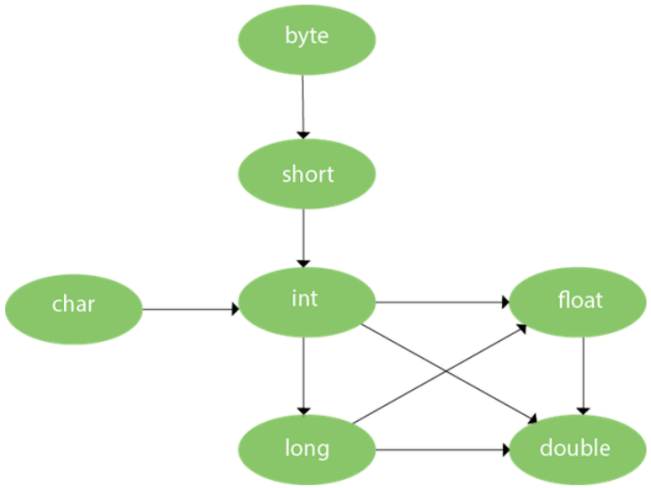
Compile-time polymorphism

- Compile-time polymorphism is method overloading
- What is method overloading?
  - By changing number of arguments
  - By changing the data type
- What about methods having the same type and number of parameters while parameters are in a different order?
  - It is not considered as overloading, and it is not a good coding practice.
- Java does not consider the return type difference as overloading
  - When you declare two methods with the same signature and different return types, the compiler throws error.

Compile-time polymorphism

- Constructor overloading is one type of compile-time polymorphism

```
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;
}
```



Src: javatpoint

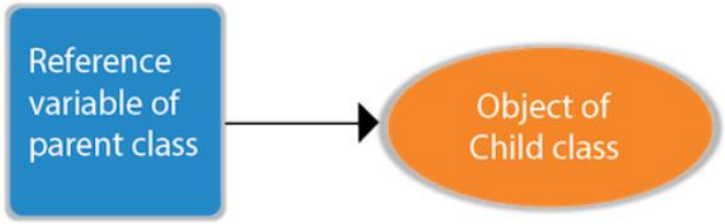
- Method overloading with type promotion
  - One type is promoted to another implicitly if no matching datatype is found
  - If there are matching types found, type promotion is not performed
  - If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

Runtime polymorphism

- Runtime polymorphism is a process in which a function call to the overridden method is resolved at Runtime.
- Runtime polymorphism is achieved by method overriding.
- What is method overriding?
  - Both methods implemented by the parent and child classes share the same name and parameters, while the implementations are different.
- What is method overriding related to @Override annotation?
  - @Override annotation can be used for both implementation of abstract method, or overridden regular method.
- What is late binding?
  - Late binding refers to the fact that which object on which method gets called is determined at runtime.

Runtime polymorphism - upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.
  - At compile time, the system only knows about the parent object during upcasting, so only those methods provided by parental class are visible.
  - JVM knows about the child class and executes the overridden method – this is called late binding.
- Compiler does not know if abstract method is implemented or not during upcasting. But there will be runtime error when we run the program.



Src: javatpoint

Runtime polymorphism

- Note that a method is overriden, not the data members. So the runtime polymorphism can not be achieved by data members.
- Runtime polymorphism can be achieved by multilevel inheritance.
- Guess the output of below code?

```
/**
 * 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);
```

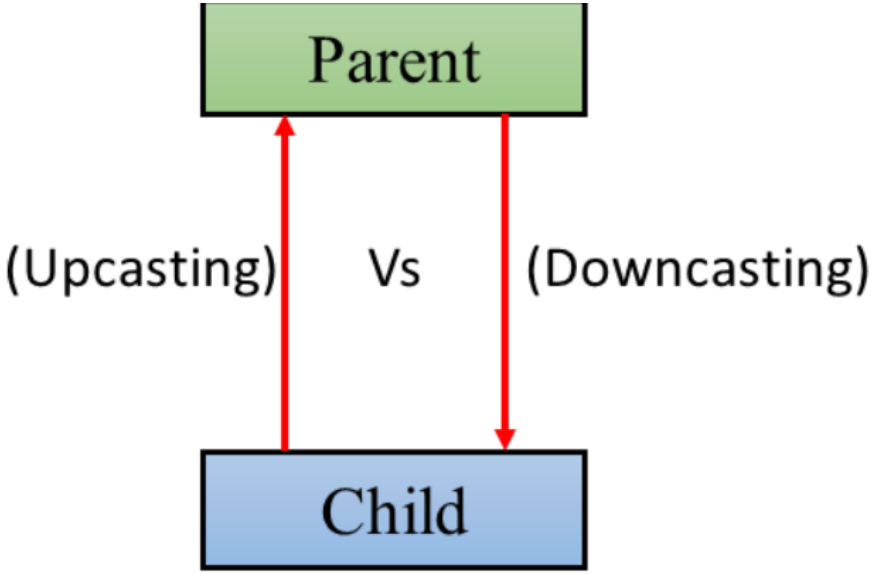
Runtime polymorphism

- Following are true:
  - A cat is a Cat
  - A cat is an Animal
  - A cat is an Object
- The expressions on the right are all valid.
- However, the animal instance has only access to methods provided by Animal class, and the object instance has only access to methods provided by Object class.

```
Cat cat1, cat2, cat3;
cat1 = new Cat( id: 101);
cat2 = new BabyCat( id: 102);
cat3 = new MainecoonBabyCat( id: 103);
Animal animal = cat3;
Object object = cat2;
cat1.catActivity();
cat2.catActivity();
cat3.catActivity();
```

Upcasting & downcasting

- We already introduced upcasting. Upcasting is done implicitly.
- So what is downcasting?
- Downcasting is that we forcefully cast a parent to a child.
- Downcasting must be done explicitly.
- First the compiler checks if downcasting is possible, if not, a ClassCastException is thrown.



Src: geeksforgeeks

Downcasting

- Why downcasting?
  - Due to upcasting, the object is allowed to access part of the parent class members, while not all…
  - Due to downcasting, a child object can acquire any methods of parent object and child object

```
/**
 * myPig can not call pigActivity, why?
 * Because at compile time, the system only knows about it is a Mammal object
 * Only those overriden methods defined in Mammal object are accessible
 */
//myPig.pigActivity();
/**
 * myPig instance is not able to access variable of Pig class since it has been
 * upcasted to Mammal class
 */
//System.out.println(myPig.numberOfLegs);
/**
 * now we are downcasting
 * newPig instance can access methods and variables provided by Pig class now
 */
Pig newPig = (Pig) myPig;
System.out.println(newPig.numberOfLegs);
newPig.pigActivity();
newPig.MammalActivities();
```

Downcasting

- How downcasting is possible? First upcasting, then downcasting

```
/**
 * it is possible to access sub class method but not super class method
 */
BabyCat babyCat = new MainecoonBabyCat( id: 200);
babyCat.printBabyCat();
/**
 * runtime error: class LectureCode.Session10.src.animals.BabyCat cannot be cast
 * to class LectureCode.Session10.src.animals.MainecoonBabyCat
 */
MainecoonBabyCat mainecoonBabyCat = (MainecoonBabyCat) new BabyCat( id: 100);
mainecoonBabyCat.printBabyCat();
mainecoonBabyCat.printMainecoonBabyCat();
/**
 * this is the right way to do downcasting.
 * You first do upcasting, then downcasting          *
 */
MainecoonBabyCat mainecoonBabyCat2 = (MainecoonBabyCat) babyCat;
mainecoonBabyCat2.printBabyCat();
mainecoonBabyCat2.printMainecoonBabyCat();
```

Method Hiding

- Method hiding works for static method.
- When a child class defines a static method with the same signature as a static method in the parent class, the child's method hides the one in parent class.
- Method hiding is not polymophic. It is determined at compile time.
- Method overriding is determined at runtime.

```
public abstract class Mammal extends Animal{

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

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

```
class Pig extends Mammal {
    public static void regulateBodyHeat(){
        System.out.println("Change in temp. " +
                "Regulating body heat of a pig.");
    }
    protected int numberOfLegs;
    public Pig(int id) { super(id); }

    @Override
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
```

Method Hiding

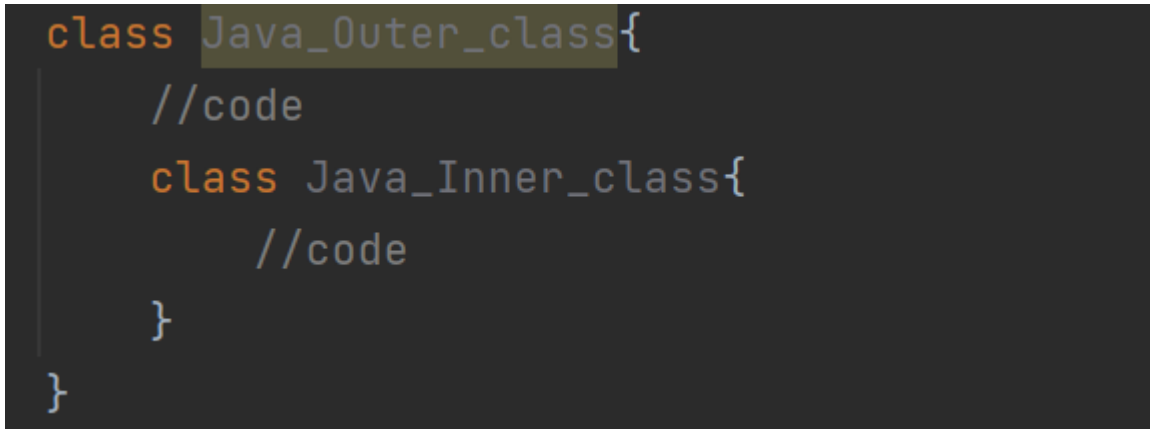
- What are the output respectively? And Why?
- Note that regulateBodyHeat is static method, MammalActivities is instance method.

```
Mammal myPig = new Pig( id: 1);
myPig.animalSound();
myPig.sleep();
myPig.provideMilkForBaby();
myPig.regulateBodyHeat();
myPig.MammalActivities();
```

```
Pig myPig = new Pig( id: 1);
myPig.animalSound();
myPig.sleep();
myPig.provideMilkForBaby();
myPig.regulateBodyHeat();
myPig.MammalActivities();
```

Inner Class

- What is Java Inner class?
  - It is also called nested class that is declared inside the class or interface.
  - The Inner class can access all the members of the outer class, including private data members and methods
- Why do we use Inner class?
  - We use it to logically group classes and interfaces.
  - Make the code more maintainable and readable.



Inner Class

- There are three types of inner classes
  - Member inner class
  - Anonymous inner class
  - Local inner class

Member Inner Class

- How to instantiate Member inner class in Java?

```java
class Java_Outer_class{
    private int data=30;
    class Java_Inner_class{
        void msg(){System.out.println("data is "+data);}
    }
}
public class Main {
    public static void main(String[] args) {
        Java_Outer_class outer = new Java_Outer_class();
        //syntax to instantiate inner class
        Java_Outer_class.Java_Inner_class inner = outer.new Java_Inner_class();
        inner.msg();
    }
}
```

Anonymous Inner Class

- A class that has no name is known as anonymous inner class.
- Java anonymous inner class can also use interface instead of abstract keyword.
- Question: how do we implement interface?

```java
/**
 * definiton of abstract anonymous inner class
 */
abstract class Person{
    abstract void eat();
}
class Java_Anonymous_Outer_class {
    public static void main() {
        //implementation of abstract anonymous inner class
        Person p=new Person(){
            void eat(){System.out.println("nice fruits");}
        };
        p.eat();
    }
}
public class Main {
    public static void main(String[] args) {
        Java_Anonymous_Outer_class.main();
    }
}
```

Local Inner Class

- Local inner class is created inside a method/block.
- This class belongs to the block they are defined within.
- This class have access to the attributes of the class enclosing it.

```
class Java_Local_Inner_Class {
    private int data = 30;//instance variable
    void display() {
        class Local {
            void msg() {
                System.out.println(data);
            }
        }
        Local l = new Local();
        l.msg();
    }
}
public class Main {
    public static void main(String[] args) {
        Java_Local_Inner_Class java_local_inner_class = new Java_Local_Inner_Class();
        java_local_inner_class.display();
    }
}
```

Before we end

- Goals for this session:
  - I know Compile time polymorphism
  - I know Runtime polymorphism
  - I know how to use upcasting
  - I know how to use downcasting
  - I know the concept of method hiding
  - I know how to use three types of inner classes

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.