Session 2: OOP (2)

Overriding methods

- Method signature
- The **@Override** annotation is not mandatory
- The subclass can make the method “more public” but not “less public”
- The keyword **super** is a special reference to the instance’s parent

```
public class Animal {
    protected void move() {
        System.out.println("The animal is moving");
    }

    protected void moveTo(int x, int y) { }
}

public class Cat extends Animal {
    @Override
    public void move() {
        System.out.println("The cat " + name  + " is walking");
    }

    @Override
    protected void moveTo(int x, int y) {
        super.moveTo(x, y);
    }
}
```

Method signature

- Method signature:
  method name + parameter list

- Defining two methods with same name but different parameter list is called **method overloading**

- Not to be confused with method overriding

```
public class Animal {
    protected void move() {
        System.out.println("The animal is moving");
    }
}

public class Cat extends Animal {

    @Override
    public void move() {
        System.out.println("The cat " + name  + " is walking");
    }

    public void meow(int times) {
        for (int i = 0; i < times; i++) {
            meow();
        }
    }

    private void meow() {
        System.out.println("The cat " + name  + " is meowing");
    }
}
```

Exercise 1

- Overload a method that adds two values

1. Original method takes two integers, returns result as an integer
   int a + int b → int

2. Second method takes an integer and a double, returns result as an double
   int a + double b → double

3. Third method takes an integer and a double, returns result as an integer: ceiling(a + b)
   int a + double b → int

Exercise 2

- Verify if it is possible to reduce the access modifier of a method when extending it in a subclass. Explain your findings.

```
public class Animal {
    public void move() {
        System.out.println("The animal is moving");
    }
}

public class Cat extends Animal {

    @Override
    private void move() {
        System.out.println("The cat " + name  + " is walking");
    }
}
```

sourcemind

Return type is not part of method signature

Methods cannot be overloaded with different return types.

Interfaces

- Some animals run, swim, or fly
- Some animals run and swim, or swim and fly
- What is the issue with defining run(), swim(), and fly() methods in Animal superclass?

Interfaces

In the Java programming language, an interface is a reference type, similar to a class, that can contain **only** constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

Interfaces define the set of expected behavior but without any implementation.

```java
public interface FlyingAnimal {
    void fly();
}

public interface SwimingAnimal {
    void swim();
}

public class Animal {
    // ...
}

public class Cat extends Animal {
}
```

```java
public class Eagle extends Animal implements FlyingAnimal {
    @Override
    public void fly() {
    }
}

public class Duck extends Animal implements SwimmingAnimal,
                                             FlyingAnimal {
    @Override
    public void fly() {
    }

    @Override
    public void swim() {
    }
}
```

Interfaces as “contracts”

- Interfaces can also act as contracts
- A contract of an object is like a guarantee that is has a defined behavior
- For example, a set of birds can fly, so they have a fly() method. But they can be of different types of birds.
- In some cases, it does not matter. All that is needed is to know that you can invoke the fly() method.

```
FlyingAnimal[] birds = new FlyingAnimal[2];

birds[0] = new Eagle();
birds[1] = new Duck();

// Since every object in birds[] is a reference of type
FlyingAnimal
// We are sure that it has a fly() method
// But an eagle flies differently than a duck
for (int i = 0; i < 2; i++) {
  birds[i].fly();
}
```

Packages

- A package is a namespace that organizes a set of related classes and interfaces.
- Similar to organizing files in folders.
- Packages can have hierarchical structures, similar to folders
- Name of packages must match their physical location in filesystem
- Large Java applications are usually made of 100s or 1000s of classes, organized in packages

src/a/b/c/MyClass.java

```
package a.b.c;


class MyClass {
}
```

Class Access Modifiers

- **Public** classes and interfaces are accessible wherever the package is imported.
- **Non-public** classes and interfaces are accessible only inside the package that they belong to.

```
package database;

public interface Database {
    Cat find(String name);
}

---
class XmlStorage implements Database {
    @Override
    public Cat find(String name) {
        // ... implementation ...
    }
}
---
class JsonStorage implements Database {
    @Override
    public Cat find(String name) {
        // ... implementation ...
    }
}
```

Class Access Modifiers

Exercise 3

The App class is defined in database package.

Move App to a different package and run the program.

```
package database;

public class App {
    public static void main(String[] args) {

        Database database = null;

        if ("xml".equals(args[0])) {
            database = new XmlStorage();
        } else if ("json".equals(args[0])) {
            database = new JsonStorage();
        }

        Cat oliver = database.find("Oliver");
    }
}
```

sourcemind

Class Inheritance

- The name field is defined in Animal class
- Cat class also has name field by inheritance
- Not all animals walk but a cat walks
- A bird can have a name but walk() method does not define a bird’s behavior

```
public class Cat extends Animal implements Walkable {
    @Override
    public String getName() {
        return "Cat: " + name;
    }

    @Override
    public void walk() {
        System.out.println("The cat is walking");
    }
}

public class Animal {
    public String name;
    public String getName() {
        return name;
    }
}

public interface Walkable {
    void walk();
}
```

Reference vs object type

- Left side = reference type
- Right side = object type

- Even if the object is defined as a Cat, its reference is treated as Animal.
- To be able to use the methods defined in Cat class, you must “cast” its reference to the Cat type.
- Casting does not change the real type of the object.

```
Animal cat = new Cat();

cat.getName();  // Will this work?
cat.walk();     // Will this work?

Cat anotherCat = new Cat();
anotherCat.walk();

// Casting treats the reference as another type
Cat reallyCat = (Cat) cat;
reallyCat.walk();
```

super keyword

- When we need to override a superclass method, not completely but adding some functionality in the subclass.
- The super keyword is a reference to the immediate parent object.
- Similarly, the super() method invokes the immediate parent class construction.

super keyword

```
class Animal {
    protected int age;

    public Animal(int age) {
        this.age = age;
    }

    public void move() {
        System.out.println("Animal move...");
    }
}
```

```
class Cat extends Animal {
    private String name;

    public Cat(String name) {
        super(0);
        this.name = name;
    }

    @Override
    public void move() {
        super.move();
        System.out.println("Cat move...");
    }
}
```

Overriding vs. Hiding

```
class Animal {
    protected int age;

    public Animal(int age) {
        this.age = age;
    }

    public void move() {
        System.out.println("Animal move...");
    }

    public static String getCategory() {
        return "Generic Animal";
    }
}
```

```
class Cat extends Animal {
    private String name;

    public Cat(String name) {
        super(0);
        this.name = name;
    }

    @Override
    public void move() {
        super.move();
        System.out.println("Cat move...");
    }

    public static String getCategory() {
        return "Predator";
    }
}
```

Overriding vs. Hiding

- Redefining a static method in the subclass “hides” the parent method.
- You can access the method by specifying the **class reference** instead of the object re

```
// Superclass
Animal.getCategory();

// Subclass
Cat.getCategory();

// Not recommended
Animal animal = new Animal(0);
animal.getCategory();
```

instanceof operator

- Used to check the type of a object which the reference is referring to
- NOT the type of the reference
- Returns true or false

```
Cat cat = new Cat("Oliver");

if (cat instanceof Cat)
    System.out.println("instance of Cat");

if (cat instanceof Animal)
    System.out.println("instance of Animal");

if (cat instanceof Object)
    System.out.println("instance of Object");

if (cat instanceof Bird)
    System.out.println("instance of Bird");
```

Exercise 4

What is the output of the provided code?

```
Animal test = new Bird("Test");

if (test instanceof Cat)
    System.out.println("instance of Cat");

if (test instanceof Animal)
    System.out.println("instance of Animal");

if (test instanceof Object)
    System.out.println("instance of Object");

if (test instanceof Bird)
    System.out.println("instance of Bird");
```

Let’s practice:

Implement a Zoo class that holds N number of animals. Reuse some of the animal types or define new animal types and add instances of them in the array.

Implement the following methods in Zoo class:

```
int countFlyingAnimals() { ... }

boolean canSwim(Animal unknownAnimal) { ... }
```

instanceof and getClass()

- Animal a = new Cat()

  a is an instance of Cat, a is also an instance of Animal.

  a can be treated as a Cat, a can also be treated as an Animal.

- a.getClass() == Cat.class

  a is created as a Cat, so it’s class will always be constant.

getClass().isAssignableFrom(...)

- Animal a = new Cat()

- a.getClass().isAssignableFrom(Animal.class)

  Behaves similarly to instance of, however the comparison class can be provided during runtime.

  boolean anyAnimalExists(Class type) { ... }

Object type inference

- Java compiler can “guess” the type of reference when it is assigned to a newly created object or a primitive literal
- The new() operator determines the real type of the object
- When there is enough context information, the reference type can be skipped
- **This does NOT make Java a dynamically typed language**

```
// Inference of class types
var test1 = new Bird("Test 1");
var test2 = new Cat("Test 2");
var test3 = "Test 3";

// Inference of primitives
var test4 = 1;
test4 = test4 + 0.5;
```

OOP Relationships

- Association
  A student takes a course
- Aggregation
  A course has slides, homework, and exercises
- Composition
  A cat has paws, ears, …
- Generalization / Specialization
  A cat is a special type of animal
- Composition is stronger than aggregation and aggregation is stronger than association
- Generalization / Specialization is also called **Inheritance**.

Homework 4

Write a program that processes the checkout in a supermarket.

Model a ShoppingCart using an array of ProductItem that keeps the items and counts the final total price. When the total price is calculated, the payment should take place. Take note that payment is a general concepts and there are special payment types such as CashPayment and CreditCardPayment.

In your program, add some ProductItems in the store catalogue so that you can run the program with actual command-line input. You may use the Scanner class to accept and process input.

Checkout the next slides for more hints and details.

Homework 4

Sample input and output of the program:

```
Welcome to the checkout! Enter the product data to complete the checkout.

Enter the product number (q to quit): 1234
Enter the quantity: 1
Enter the product number (q to quit): 6783
Enter the quantity: 2
Enter the product number (q to quit): q
The total price is 100. Choose the payment type (1: cash, 2: credit card): 2
Enter the credit card number: 1111-2222-3333-4444
Payment accepted. Thank you!
```

Homework 4

Some of the classes that may exist in your program (complete the implementation).

```
class ProductItem {

        // product number, unit price, name, any other useful data

}

ShoppingCart {

        private ProductItem[] items; // Choose a maximum number of allowed items to simplify the program
        private int[] quantity; // the quantity of n-th item in the ProductItem[] array
        public void add(ProductItem item, int howMany) { ... }
        public double getTotal() { ... } //

}

interface Payment() {

        void accept(double amount);
        boolean verify();

} // Implement cash and credit-card payment methods
```