ASTANA IT UNIVERSITY	**Cours:** Object-Oriented Programming **Deadline:** January 22, 2024 till (23:59)

Instructor: Toleu Altynbek

ASSIGNMENT 4

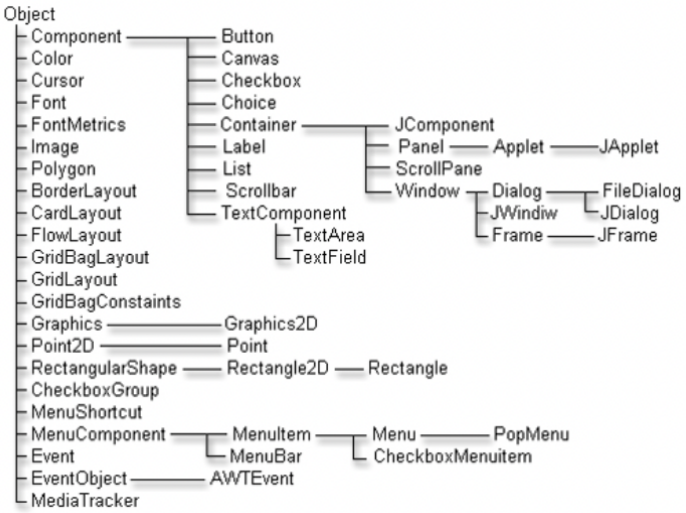
Lesson plan:

Inheritance
Encapsulation
Inheritance
Abstract class
Interfaces

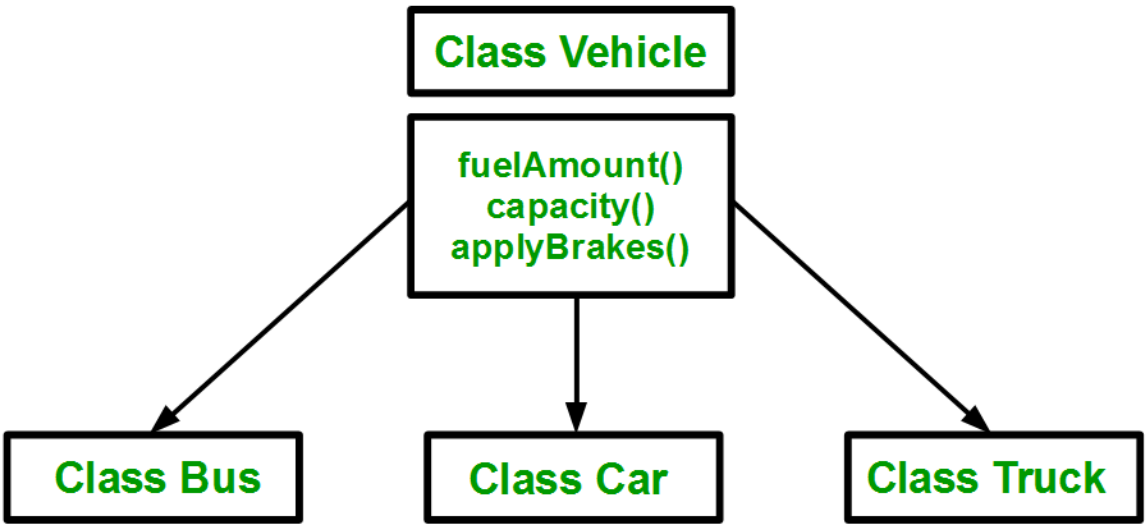
Inheritance, Encapsulation and Polymorphism

- Class hierarchies.
- Encapsulation and inheritance
- Polymorphism and method overriding.

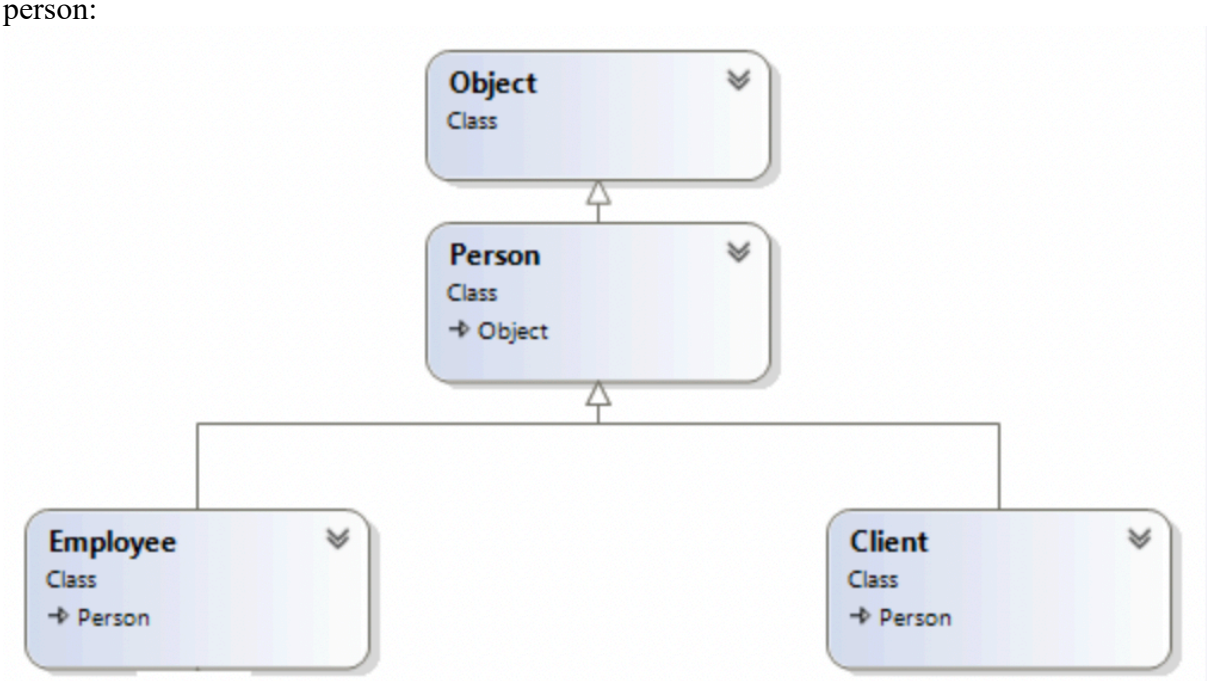
A class hierarchy is a structure in object-oriented programming that organizes classes into a hierarchical structure based on inheritance relationships. In such a hierarchy, classes can inherit properties and methods from other classes, promoting code reuse and program organization. Typically, classes in a hierarchy are organized so that more general classes are higher up in the hierarchy, and more specialized classes inherit their properties and methods.



Inheritance



One of the key aspects of object-oriented programming is inheritance. Using inheritance, you can expand the functionality of existing classes by adding new functionality or changing old ones. For example, there is the following Person class, which describes an individual person:



```
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
```

And perhaps later we will want to add another class that describes an employee of the enterprise - the Employee class. Since this class implements the same functionality as the Person class, since an employee is also a person, it would be rational to make the Employee class a derivative (heir, subclass) of the Person class, which, in turn, is called a base class, parent or superclass:

```
class Employee extends Person{
    public Employee(String name){
        super(name);    // если базовый класс определяет конструктор
                        //  то производный класс должен его вызвать
    }
}
```

To declare one class as an inheritor of another, you must use the **extends** keyword after the name of the inheritor class, followed by the name of the base class. The Employee class is based on Person, and therefore the Employee class inherits all the same fields and methods that the Person class has.

If constructors are defined in the base class, then in the constructor of the derived class you must call one of the constructors of the base class using the **super** keyword. For example, the Person class has a constructor that takes one parameter. Therefore, in the Employee class, in the constructor you need to call the constructor of the Person class. That is, the call to super(name) will represent a call to the constructor of the Person class.

When calling a constructor, after the word super, the arguments passed are listed in parentheses. In this case, the call to the base class constructor must occur at the very beginning in the derived class constructor. This way, setting the employee name is delegated to the base class constructor.

Moreover, even if the derived class does not perform any other work in the constructor, as in the example above, it is still necessary to call the constructor of the base class.

Using classes:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
        tom.display();
        Employee sam = new Employee("Sam");
        sam.display();
    }
}
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    public  Employee(String name){
        super(name);     // если базовый класс определяет конструктор
                         //  то производный класс должен его вызвать
    }
}
```

A derived class has access to all methods and fields of the base class (even if the base class is in a different package) except those defined with the **private** modifier. In this case, the derived class can also add its own fields and methods:

```java
public class Program{

    public static void main(String[] args) {

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
        sam.work();     // Sam works in Microsoft
    }
}
class Person {

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

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    public void work(){
        System.out.printf("%s works in %s \n", getName(), company);
    }
}
```

In this case, the Employee class adds a company field, which stores the employee's place of work, as well as a work method.

Access modifiers and encapsulation

All class members in Java - fields and methods - have access modifiers. In previous topics, we have already encountered the public modifier. Access modifiers allow you to set the permissible scope for class members, that is, the context in which a given variable or method can be used.

Java uses the following access modifiers:

- public: public, public class or class member. Fields and methods declared with the public modifier are visible to other classes from the current package and from external packages.

- private: A private class or class member, the opposite of the public modifier. A private class or class member is accessible only from code in the same class.

- protected: such a class or class member is accessible from anywhere in the current class or package or in derived classes, even if they are in other packages

- Default modifier. The absence of a modifier for a class field or method implies that the default modifier is applied to it. Such fields or methods are visible to all classes in the current package.

Let's look at access modifiers using the following program as an example:

```
public class Program{

    public static void main(String[] args) {

        Person kate = new Person("Kate", 32, "Baker Street", "+12334567");
        kate.displayName();     // норм, метод public
        kate.displayAge();      // норм, метод имеет модификатор по умолчанию
        kate.displayPhone();    // норм, метод protected
        //kate.displayAddress();  // ! Ошибка, метод private

        System.out.println(kate.name);      // норм, модификатор по умолчанию
        System.out.println(kate.address);   // норм, модификатор public
        System.out.println(kate.age);       // норм, модификатор protected
        //System.out.println(kate.phone);   // ! Ошибка, модификатор private
    }
}
class Person{

    String name;
    protected int age;
    public String address;
    private String phone;

    public Person(String name, int age, String address, String phone){
        this.name = name;
        this.age = age;
        this.address = address;
        this.phone = phone;
    }
    public void displayName(){
        System.out.printf("Name: %s \n", name);
    }
    void displayAge(){
        System.out.printf("Age: %d \n", age);
    }
    private void displayAddress(){
        System.out.printf("Address: %s \n", address);
    }
    protected void displayPhone(){
        System.out.printf("Phone: %s \n", phone);
    }}
```

In this case, both classes are located in one package - the default package, so in the Program class we can use all the methods and variables of the Person class, which have the default modifier, public and protected. And fields and methods with the private modifier in the Program class will not be available.

If the Program class were located in another package, then only fields and methods with the public modifier would be available to it.

The access modifier must precede the rest of the variable or method definition.

Encapsulation

It would seem, why not declare all variables and methods with the public modifier so that they are available anywhere in the program, regardless of the package or class? Take for

example the age field, which represents age. If another class has direct access to this field, then there is a possibility that during program operation it will be passed an incorrect value, for example, a negative number. Such data modification is not advisable. Or we want some data to be directly accessible so that we can display it on the console or simply find out its value. In this regard, it is recommended to restrict access to data as much as possible in order to protect it from unwanted access from the outside (both to get the value and to change it). The use of various modifiers ensures that the data is not distorted or changed inappropriately. This type of hiding of data within a scope is called encapsulation.

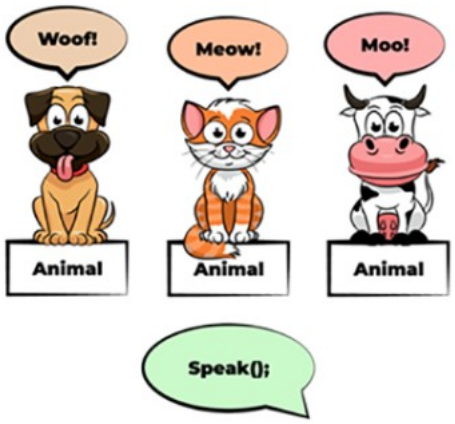
So, as a rule, instead of directly applying fields, access methods are used. For example:

```
public class Program{

    public static void main(String[] args) {

        Person kate = new Person("Kate", 30);
        System.out.println(kate.getAge());      // 30
        kate.setAge(33);
        System.out.println(kate.getAge());      // 33
        kate.setAge(123450);
        System.out.println(kate.getAge());      // 33
    }
}
class Person{

    private String name;
    private int age = 1;

    public Person(String name, int age){

        setName(name);
        setAge(age);
    }
    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name;
    }
    public int getAge(){
        return this.age;
    }
    public void setAge(int age){
        if(age > 0 && age < 110)
            this.age = age;
    }
}
```

And then, instead of directly working with the name and age fields in the Person class, we'll work with methods that set and return the values of these fields. Methods setName, setAge and the like are also called mutaters, since they change the values of a field. And the methods getName, getAge and the like are called accessors, since with their help we get the value of the field.

Moreover, we can put additional logic into these methods. For example, in this case, when the age changes, a check is made to see how well the new value fits within the acceptable range.

Polymorphism and method overriding



Polymorphism is an object-oriented programming (OOP) concept that allows objects of different classes to be treated as objects of a common class. In Java, polymorphism means that a method or operator can have multiple implementations depending on the type of object on which the operation is performed.

There are two types of polymorphism in Java:

Method overloading: This occurs when there are multiple methods in a class with the same name but different parameters. The correct method to call is determined at compile time based on the arguments passed to the method.

Method Overriding: This occurs when a subclass provides a new implementation of a method that is already defined in its superclass. The correct method to call is determined at run time depending on the type of object being referenced.

In Java, polymorphism is achieved through inheritance and the use of interface types. Polymorphism allows objects to be treated as objects of their base class, which allows common code to be written, making it more flexible and reusable. This allows you to create more reusable and maintainable code, and also improves code readability.

Imagine that you have a class “Animal” and two subclasses “Cat” and “Dog”. With polymorphism, you can write one piece of code that works with any object of type "Animal", even if that object is actually a "Cat" or "Dog". This means that the code is not specific to any one type of object and can be used with any object of the same class or subclass.

```
public class Animal {
    public void makeSound() {
        System.out.println("Animal sound");
    }
}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();
        Animal cat = new Cat();

        animal.makeSound();  // Animal sound
        dog.makeSound();     // Bark
        cat.makeSound();     // Meow
    }
}
```

In this example, the Animal class has a makeSound method that displays a text display of the sound that one of the animals plays. The Dog and Cat classes extend Animal and override the makeSound method, providing their own implementation. When an instance of Dog or Cat is created and the makeSound method is called on it, the method implementation appropriate for the object type is executed. This is polymorphism in action.

The ability to treat objects of different classes as objects of a common class provides greater flexibility when writing code and makes it easier to write generic algorithms that can work with objects of different types.

Task 1: Creating an Object and Using Getters and Setters

Create a class Person that contains fields name and age. Implement getters and setters for these fields. Then, create an object of the Person class and use getters and setters to set and retrieve field values.

Task 2: Encapsulation

Extend the Person class from the previous task by making the name and age fields private. Add a method printDetails that prints information about the person (name and age). Create an object and use the printDetails method to display information.

Task 3: Inheritance

Create a class Student that inherits from the Person class. Add an additional field studentId and a method getStudentId. Override the printDetails method to also display information about the student ID.

Task 4: Polymorphism

Create an interface Printable containing the printDetails method. Implement this interface in the Person and Student classes. Then, create an array of Printable objects and use a loop to call the printDetails method for each object in the array, regardless of their actual type.

Task 5: Abstract Class

Create an abstract class Shape containing an abstract method calculateArea. Implement two subclasses of this abstract class: Circle and Rectangle. Each subclass should implement the calculateArea method to calculate the area of a circle and rectangle, respectively.

Task 6: Interfaces

Create an interface Resizable containing a resize method. Implement this interface in the Circle and Rectangle classes from the previous task. Let the resize method change the size of the respective shapes.