Part 9

Class inheritance

Learning Objectives

- You know that in the Java programming language every class inherits the Object class, and you understand why every object has methods toString, equals, and hashCode.
- You are familiar with the concepts of inheritance, superclass, and subclass.
- You can create classes that inherit some of their properties from another class.
- You can call a constructor or method that is defined in a superclass.
- You know how an object's executed method is determined, and you are familiar with the concept of polymorphism.
- You can assess when to use inheritance, and you can come up with an example that is ill-suited for inheritance.

Classes are used to clarify the concepts of the problem domain in object-oriented programming. Every class we create adds functionality to the programming language. This functionality is needed to solve the problems that we encounter. An essential idea behind object-oriented programming is that **solutions rise from the interactions between objects which are created from classes**. An object in object-oriented programming is an independent unit that has a state, which can be modified by using the methods that the object provides. Objects are used in cooperation; each has its own area of responsibility. For instance, our user interface classes have so far made use of `Scanner` objects.

Every Java class extends the class Object, which means that every class we create has at its disposal all the methods defined in the Object class. If we want to change how these methods are defined in Object function, they must be overriden by defining a new implementation for them in the newly created class. The objects we create receive the methods `equals` and `hashCode`, among others, from the Object class.

Every class derives from `Object`, but it's also possible to derive from other classes. When we examine the API (Application Programming Interface) of Java's ArrayList, we notice that `ArrayList` has the superclass `AbstractList`. `AbstractList`, in turn, has the class `Object` as its superclass.

```
java.lang.Object
└
  java.util.AbstractCollection<E>
  └
    java.util.AbstractList<E>
    └
      java.util.ArrayList<E>
```

Each class can directly extend only one class. However, a class indirectly inherits all the properties of the classes it extends. So the `ArrayList` class derives from the class `AbstractList`, and indirectly derives from the classes `AbstractCollection` and `Object`. So `ArrayList` has at its disposal all the variables and methods of the classes `AbstractList`, `AbstractCollection`, and `Object`.

You use the keyword `extends` to inherit the properties of a class. The class that receives the properties is called the subclass, and the class whose properties are inherited is called the superclass.

Let's take a look at a car manufacturing system that manages car parts. A basic component of part management is the class `Part`, which defines the identifier, the manufacturer, and the description.

```
public class Part {

    private String identifier;
    private String manufacturer;
    private String description;

    public Part(String identifier, String manufacturer, String description) {
        this.identifier = identifier;
        this.manufacturer = manufacturer;
        this.description = description;
    }

    public String getIdentifier() {
        return identifier;
    }

    public String getDescription() {
        return description;
```

```
    }

    public String getManufacturer() {
        return manufacturer;
    }
}
```

One part of the car is the engine. As is the case with all parts, the engine, too, has a manufacturer, an identifier, and a description. In addition, each engine has a type: for instance, an internal combustion engine, an electric motor, or a hybrid engine.

The traditional way to implement the class `Engine`, without using inheritance, would be this.

```
public class Engine {

    private String engineType;
    private String identifier;
    private String manufacturer;
    private String description;

    public Engine(String engineType, String identifier, String manufacturer, St
        this.engineType = engineType;
        this.identifier = identifier;
        this.manufacturer = manufacturer;
        this.description = description;
    }

    public String getEngineType() {
        return engineType;
    }

    public String getIdentifier() {
        return identifier;
    }

    public String getDescription() {
        return description;
    }

    public String getManufacturer() {
        return manufacturer;
```

```
    }
}
```

We notice a significant amount of overlap between the contents of `Engine` and `Part`. It can confidently be said the `Engine` is a special case of `Part`. **The Engine is a Part**, but it also has properties that a Part does not have, which in this case means the engine type.

Let's recreate the class `Engine` and, this time, use inheritance in our implementation. We'll create the class `Engine` which inherits the class `Part`: an engine is a special case of a part.

```
public class Engine extends Part {

    private String engineType;

    public Engine(String engineType, String identifier, String manufacturer, St
        super(identifier, manufacturer, description);
        this.engineType = engineType;
    }

    public String getEngineType() {
        return engineType;
    }
}
```

The class definition `public class Engine extends Part` indicates that the class `Engine` inherits the functionality of the class `Part`. We also define an object variable `engineType` in the class `Engine`.

The constructor of the Engine class is worth some consideration. On its first line we use the keyword `super` to call the constructor of the superclass. The call `super(identifier, manufacturer, description)` calls the constructor `public Part(String identifier, String manufacturer, String description)` which is defined in the class Part. Through this process the object variables defined in the superclass are initiated with their initial values. After calling the superclass constructor, we also set the proper value for the object variable `engineType`.

The `super` call bears some resemblance to the `this` call in a constructor; `this` is used to call a constructor of this class, while `super` is used to call a constructor of the superclass. If a constructor uses the constructor of the superclass by calling `super` in it, the `super` call must be on the first line of the constructor. This is similar to the case with calling `this` (must also be the first line of the constructor).

Since the class `Engine` extends the class `Part`, it has at its disposal all the methods that the class `Part` offers. You can create instances of the class `Engine` the same way you can of any other class.

```
Engine engine = new Engine("combustion", "hz", "volkswagen", "VW GOLF 1L 86-91"
System.out.println(engine.getEngineType());
System.out.println(engine.getManufacturer());
```

Sample output

```
combustion
volkswagen
```

As you can see, the class `Engine` has all the methods that are defined in the class `Part`.

Programming exercise:
ABC (2 parts)

Points
2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Let's practice creating and inheriting classes.

Part 1: Creating classes

Create the following three classes:

- Class `A`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void a()`, which prints a string "A".
- Class `B`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void b()`, which prints a string "B".
- Class `C`. Class should have no object variables nor should you specify a constructor for it. It only has the method `public void c()`, which prints a string "C".

```
A a = new A();
B b = new B();
C c = new C();

a.a();
b.b();
c.c();
```

Sample output

```
A
B
C
```

Part 2: Class inheritance

Modify the classes so that class B inherits class A, and class C inherits class B. In other words, class A will be a superclass for class B, and class B will be a superclass for class C.

```
C c = new C();

c.a();
c.b();
c.c();
```

Sample output

```
A
B
C
```

Exercise submission instructions

How to see the solution

Access modifiers private, protected, and public

If a method or variable has the access modifier `private`, it is visible only to the internal methods of that class. Subclasses will not see it, and a subclass has no direct means to access it. So, from the Engine class there is no way to directly access the variables identifier, manufacturer, and description, which are defined in the superclass Part. The programmer cannot access the variables of the superclass that have been defined with the access modifier private.

A subclass sees everything that is defined with the `public` modifier in the superclass. If we want to define some variables or methods that are visible to the subclasses but invisible to everything else, we can use the access modifier `protected` to achieve this.

Calling the constructor of the superclass

You use the keyword `super` to call the constructor of the superclass. The call receives as parameters the types of values that the superclass constructor requires. If there are multiple constructors in the superclass, the parameters of the super call dictate which of them is used.

When the constructor (of the subclass) is called, the variables defined in the superclass are initialized. The events that occur during the constructor call are practically identical to what happens with a normal constructor call. If the superclass doesn't provide a non-parameterized constructor, there must always be an explicit call to the constructor of the superclass in the constructors of the subclass.

We demonstrate in the example below how to call `this` and `super`. The class `Superclass` includes an object variable and two constructors. One

of them calls the other constructor with the `this` keyword. The class `Subclass` includes a parameterized constructor, but it has no object variables. The constructor of `Subclass` calls the parameterized constructor of the `Superclass`.

```
public class Superclass {

    private String objectVariable;

    public Superclass() {
        this("Example");
    }

    public Superclass(String value) {
        this.objectVariable = value;
    }

    public String toString() {
        return this.objectVariable;
    }
}
```

```
public class Subclass extends Superclass {

    public Subclass() {
        super("Subclass");
    }
}
```

```
Superclass sup = new Superclass();
Subclass sub = new Subclass();

System.out.println(sup);
System.out.println(sub);
```

Sample output

```
Example
Subclass
```

Calling a superclass method

You can call the methods defined in the superclass by prefixing the call with `super`, just as you can call the methods defined in this class by prefixing the call with `this`. For example, when overriding the `toString` method, you can call the superclass's definition of that method in the following manner:

```
@Override
public String toString() {
    return super.toString() + "\n  And let's add my own message to it!";
}
```

Programming exercise:

Person and subclasses (5 parts)

Points 5/5

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Part 1: Person

Create a class `Person`. The class must work as follows:

```
Person ada = new Person("Ada Lovelace", "24 Maddox St. London W1S 2QN");
Person esko = new Person("Esko Ukkonen", "Mannerheimintie 15 00100 Helsinki
System.out.println(ada);
System.out.println(esko);
```

Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
Esko Ukkonen
  Mannerheimintie 15 00100 Helsinki
```

Part 2: Student

Create a class `Student`, which inherits the class `Person`.

At creation, a student has 0 study credits. Every time a student studies, the amount of study credits goes up. The class must act as follows:

```
Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 9002
System.out.println(ollie);
System.out.println("Study credits " + ollie.credits());
ollie.study();
System.out.println("Study credits "+ ollie.credits());
```

Sample output

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
Study credits 0
Study credits 1
```

Part 3: Student's toString

In the previous task, `Student` inherits the toString method from the class `Person`. However, you can also overwrite an inherited method, replacing it with your own version. Write a version of toString method specifically for the `Student` class. The method must act as follows:

```
Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 9002
System.out.println(ollie);
ollie.study();
System.out.println(ollie);
```

Sample output

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 0
```

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 1
```

Part 4: Teacher

Create a class `Teacher`, which inherits the class `Person`.

The class must act as follows:

```
Teacher ada = new Teacher("Ada Lovelace", "24 Maddox St. London W1S 2QN", 1
Teacher esko = new Teacher("Esko Ukkonen", "Mannerheimintie 15 00100 Helsin
System.out.println(ada);
System.out.println(esko);

Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 9002

int i = 0;
while (i < 25) {
  ollie.study();
  i = i + 1;
}
System.out.println(ollie);
```

Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
  salary 1200 euro/month
Esko Ukkonen
  Mannerheimintie 15 00100 Helsinki
  salary 5400 euro/month
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 25
```

Part 5: List all Persons

Write a method `public static void printPersons(ArrayList<Person> persons)` in the Main class. The

method prints all the persons on the list given as the parameter. Method must act as follows when invoked from the `main` method:

```
public static void main(String[] args) {
    ArrayList<Person> persons = new ArrayList<Person>();
    persons.add(new Teacher("Ada Lovelace", "24 Maddox St. London W1S 2QN",
    persons.add(new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 9002

    printPersons(persons);
}
```

Sample output

```
Ada Lovelace
  24 Maddox St. London W1S 2QN
  salary 1200 euro/month
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  Study credits 0
```

Exercise submission instructions

How to see the solution

The actual type of an object dictates which method is executed

An object's type decides what the methods provided by the object are. For instance, we implemented the class `Student` earlier. If a reference to a `Student` type object is stored in a `Person` type variable, only the methods defined in the `Person` class (and its superclass and interfaces) are available:

```
Person ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028");
ollie.credits();        // DOESN'T WORK!
```

```
ollie.study();              // DOESN'T WORK!
```

So an object has at its disposal the methods that relate to its type, and also to its superclasses and interfaces. The Student object above offers the methods defined in the classes Person and Object.

In the last exercise we wrote a new `toString` implementation for Student to override the method that it inherits from Person. The class Person had already overriden the toString method it inherited from the class Object. If we handle an object by some other type than its actual type, which version of the object's method is called?

In the following example, we'll have two students that we refer to by variables of different types. Which version of the toString method will be executed: the one defined in Object, Person, or Student?

```
Student ollie = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 90028");
System.out.println(ollie);
Person olliePerson = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 900
System.out.println(olliePerson);
Object ollieObject = new Student("Ollie", "6381 Hollywood Blvd. Los Angeles 900
System.out.println(ollieObject);

Object alice = new Student("Alice", "177 Stewart Ave. Farmington, ME 04938");
System.out.println(alice);
```

Sample output

```
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  credits 0
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  credits 0
Ollie
  6381 Hollywood Blvd. Los Angeles 90028
  credits 0
Alice
  177 Stewart Ave. Farmington, ME 04938
  credits 0
```

The method to be executed is chosen based on the actual type of the object, which means the class whose constructor is called when the object is created. If the method has no definition in that class, the version of the method is chosen from the class that is closest to the actual type in the inheritance hierarchy.

Polymorphism

Regardless of the type of the variable, the method that is executed is always chosen based on the actual type of the object. Objects are polymorphic, which means that they can be used via many different variable types. The executed method always relates to the actual type of the object. This phenomenon is called polymorphism.

Let's examine polymorphism with another example.

You could represent a point in two-dimensional coordinate system with the following class:

```
public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int manhattanDistanceFromOrigin() {
        return Math.abs(x) + Math.abs(y);
    }

    protected String location(){
        return x + ", " + y;
    }

    @Override
    public String toString() {
        return "(" + this.location() + ") distance " + this.manhattanDistanceFr
```

```
    }
```

The `location` method is not meant for external use, which is why it is defined as protected. Subclasses will still be able to access the method. Manhattan distance means the distance between two points if you can only travel in the direction of the coordinate axes. It is used in many navigation algorithms, for example.

A colored point is otherwise identical to a point, but it contains also a color that is expressed as a string. Due to the similarity, we can create a new class by extending the class Point.

```
public class ColorPoint extends Point {

    private String color;

    public ColorPoint(int x, int y, String color) {
        super(x, y);
        this.color = color;
    }

    @Override
    public String toString() {
        return super.toString() + " color: " + color;
    }
}
```

The class defines an object variable in which we store the color. The coordinates are already defined in the superclass. We want the string representation to be the same as the Point class, but to also include information about the color. The overriden `toString` method calls the `toString` method of the superclass and adds to it the color of the point.

Next, we'll add a few points to a list. Some of them are "normal" while others are color points. At the end of the example, we'll print the points on the list. For each point, the toString to be executed is determined by the actual type of the point, even though the list knows all the points by the `Point` type.

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Point> points = new ArrayList<>();
        points.add(new Point(4, 8));
        points.add(new ColorPoint(1, 1, "green"));
```

```
        points.add(new ColorPoint(2, 5, "blue"));
        points.add(new Point(0, 0));

        for (Point p: points) {
            System.out.println(p);
        }
    }
}
```

Sample output

```
(4, 8) distance 12
(1, 1) distance 2 color: green
(2, 5) distance 7 color: blue
(0, 0) distance 0
```

We also want to include a three-dimensional point in our program. Since it has no color information, let's derive it from the class `Point`.

```
public class Point3D extends Point {

    private int z;

    public Point3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
    protected String location() {
        return super.location() + ", " + z;    // the resulting string has the
    }

    @Override
    public int manhattanDistanceFromOrigin() {
        // first ask the superclass for the distance based on x and y
        // and add the effect of the z coordinate to that result
        return super.manhattanDistanceFromOrigin() + Math.abs(z);
    }

    @Override
    public String toString() {
        return "(" + this.location() + ") distance " + this.manhattanDistanceFr
```

```
    }
}
```

So a three-dimensional point defines an object variable that represents the third dimension, and overrides the methods `location`, `manhattanDistanceFromOrigin`, and `toString` so that they also account for the third dimension. Let's now expand the previous example and add also three-dimensional points to the list.

```
public class Main {

    public static void main(String[] args) {
        ArrayList<Point> points = new ArrayList<>();
        points.add(new Point(4, 8));
        points.add(new ColorPoint(1, 1, "green"));
        points.add(new ColorPoint(2, 5, "blue"));
        points.add(new Point3D(5, 2, 8));
        points.add(new Point(0, 0));


        for (Point p: points) {
            System.out.println(p);
        }
    }
}
```

Sample output

```
(4, 8) distance 12
(1, 1) distance 2 color: green
(2, 5) distance 7 color: blue
(5, 2, 8) distance 15
(0, 0) distance 0
```

We notice that the `toString` method in Point3D is exactly the same as the toString of Point. Could we save some effort and not override toString? The answer happens to be yes! The Point3D class is refined into this:

```
public class Point3D extends Point {

    private int z;

    public Point3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
    protected String location() {
        return super.location() + ", " + z;
    }

    @Override
    public int manhattanDistanceFromOrigin() {
        return super.manhattanDistanceFromOrigin() + Math.abs(z);
    }
}
```

What happens in detail when we call the toString method of a three-dimensional point? The execution advances in the following manner.

1. Look for a definition of toString in the class Point3D. It does not exist, so the superclass is next to be examined.
2. Look for a definition of toString in the superclass point. It can be found, so the code inside the implementation of the method is executed
    - so the exact code to be executed is `return "("+this.location()+") distance "+this.manhattanDistanceFromOrigin();`
    - the method `location` is executed first
    - look for a definition of `location` in the class Point3D. It can be found, so its code is executed.
    - This `location` calls the `location` of the superclass to calculate the result
    - next we look for a definition of `manhattanDistanceFromOrigin` in the Point3D class. It's found and its code is then executed
    - Again, the method calls the similarly named method of the superclass during its execution

As we can see, the sequence of events caused by the method call has multiple steps. The principle, however, is clear: The definition for the method is first searched for in the class definition of the actual type of the object. If it is not found, we next examine the superclass. If the definition

cannot be found there, either, we move on to the superclass of this superclass, etc...

Quiz:
Inheritance

Points:
1/1

What does the program print?

```
public class Counter {

    public int addToNumber(int number) {
        return number + 1;
    }

    public int subtractFromNumber(int number) {
        return number - 1;
    }
}

----------------------

public class SuperCounter extends Counter {

    @Override
    public int addToNumber(int number) {
        return number + 5;
    }

}

----------------------

public static void main(String[] args) {
    Counter counter = new Counter();
    Counter superCounter = new SuperCounter();
    int number = 3;
    number = superCounter.subtractFromNumber(number);
    number = superCounter.subtractFromNumber(number);
    number = counter.addToNumber(number);
    System.out.println(number);
}
```

Select the correct answer

2

3

4

7

✓ 8

9

The answer is correct

Submitted The number of tries is not limited

Quiz:
Polymorphism

Points:
1/1

What does the program print?

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
```

```
    public String toString() {
        return name + " (" + age + " years) ";
    }
}

------------------

public class Student extends Person {

    private int credits;

    public Student(String name, int age, int credits) {
        super(name, age);
        this.credits = credits;
    }

    @Override
    public String toString() {
        return super.toString() + credits + " credits";
    }

}

-------------------

public static void main(String[] args) {
    Student student = new Student("Kenny", 23, 140);

    Person person = student;
    Object object = student;

    System.out.print(student + ", ");
    System.out.print(person + ", ");
    System.out.println(object);
}
```

Select the correct answer

✓ Kenny (23 years) 140 op, Kenny (23 years) 140 op, Kenny (23 years) 140 op

Student@728edb84, Person@728edb84, Object@728edb84

Student@728edb84, Student@728edb84, Student@728edb84

Kenny (23) , Kenny (23) , Kenny (23)

Kenny (23) 140 credits, Kenny (23) , Kenny (23)

Kenny (23) 140 credits, Kenny (23) , Object@728edb84

The answer is correct

Submitted The number of tries is not limited

Quiz:
Polymorphism in your own words

Points:
1/1

Explain the concept of polymorphism. Use some Java examples, and also explain how the examples would work if Java did not have polymorphism.

Your answer should be at least 50 words

Your answer

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class, even if they are actually instances of derived classes. It enables one interface to be used for different underlying forms (data types).

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent

Words: 161

Submitted The number of tries is not limited

When is inheritance worth using?

Inheritance is a tool for building and specializing hierarchies of concepts; a subclass is always a special case of the superclass. If the class to be created is a special case of an existing class, this new class could be created by extending the existing class. For example, in the previously discussed car part scenario an engine **is** a part, but an engine has extra functionality that not all parts have.

When inheriting, the subclass receives the functionality of the superclass. If the subclass doesn't need or use some of the inherited functionality, inheritance is not justifiable. Classes that inherit will inherit all the methods and interfaces from the superclass, so the subclass can be used in place of the superclass wherever the superclass is used. It's a good idea to keep the inheritance hierarchy shallow, since maintaining and further developing the hierarchy becomes more difficult as it grows larger. Generally speaking, if your inheritance hierarchy is more than 2 or 3 levels deep, the structure of the program could probably be improved.

Inheritance is not useful in every scenario. For instance, extending the class `Car` with the class `Part` (or `Engine`) would be incorrect. A car **includes** an engine and parts, but an engine or a part is not a car. More

generally, **if an object owns or is composed of other objects, inheritance should not be used**.

When using inheritance, you should take care to ensure that the Single Responsibility Principle holds true. There should only be one reason for each class to change. If you notice that inheriting adds more responsibilities to a class, you should form multiple classes of the class.

Example of misusing inheritance

Let's consider a postal service and some related classes. `Customer` includes the information related to a customer, and the class `Order` that inherits from the `Customer` class and includes the information about the ordered item. The class `Order` also has a method called `postalAddress` which represents the postal address that the order is shipped to.

```
public class Customer {

    private String name;
    private String address;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

```
public class Order extends Customer {

    private String product;
    private String count;

    public Order(String product, String count, String name, String address) {
        super(name, address);
```

```
        this.product = product;
        this.count = count;
    }

    public String getProduct() {
        return product;
    }

    public String getCount() {
        return count;
    }

    public String postalAddress() {
        return this.getName() + "\n" + this.getAddress();
    }
}
```

Above inheritance is not used correctly. When inheriting, the subclass must be a special case of the superclass; an order is definitely not a special case of a customer. The misuse shows itself in how the code breaks the single responsibility principle: the `Order` class is responsible both for maintaining the customer information and the order information.

The problem becomes very clear when we think of what a change in a customer's address would cause.

In the case that an address changes, we would have to change *every* order object that relates to that customer. This is hardly ideal. A better solution would be to encapsulate the customer as an object variable of the `Order` class. Thinking more closely on the semantics of an order, this seems intuitive. *An order has a customer*.

Let's modify the `Order` class so that it includes a reference to a `Customer` object.

```
public class Order {

    private Customer customer;
    private String product;
    private String count;

    public Order(Customer customer, String product, String count) {
        this.customer = customer;
```

```
        this.product = product;
        this.count = count;
    }

    public String getProduct() {
        return product;
    }

    public String getCount() {
        return count;
    }

    public String postalAddress() {
        return this.customer.getName() + "\n" + this.customer.getAddress();
    }
}
```

This version of the `Order` class is better. The method `postalAddress` uses the *customer* reference to obtain the postal address instead of inheriting the class `Customer`. This helps both the maintenance of the program and its concrete functionality.

Now, when a customer changes, all you need to do is change the customer information; there is no need to change the orders.

Programming exercise:

Warehousing (7 parts)

Points 7/7

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

The exercise template contains a class `Warehouse`, which has the following constructors and methods:

- **public Warehouse(double capacity)** - Creates an empty warehouse, which has the capacity provided as a parameter; an invalid capacity (<=0) creates a useless warehouse, with the the capacity 0.
- **public double getBalance()** - Returns the balance of the warehouse, i.e. the capacity which is taken up by the items in the warehouse.

- **public double getCapacity()** - Returns the total capacity of the warehouse (i.e. the one that was provided in the constructor).
- **public double howMuchSpaceLeft()** - Returns a value telling how much space is left in the warehouse.
- **public void addToWarehouse(double amount)** - Adds the desired amount to the warehouse; if the amount is negative, nothing changes, and if everything doesn't fit, then the warehouse is filled up and the rest is "thrown away" / "overflows".
- **public double takeFromWarehouse(double amount)** - Take the desired amount from the warehouse. The method returns much we actually **get**. If the desired amount is negative, nothing changes and we return 0. If the desired amount is greater than the amount the warehouse contains, we get all there is to take and the warehouse is emptied.
- **public String toString()** - Returns the state of the object represented as a string like this `balance = 64.5, space left 123.5`

In this exercise we build variations of a warehouse based on the `Warehouse` class.

Part 1: Product warehouse, step 1

The class `Warehouse` handles the functions related to the amount of a product. Now we want a product name for the product and a way to handle the name. **Let's write ProductWarehouse as a subclass of Warehouse!** First, we'll just create a private object variable for the product name, a constructor, and a getter for the name field:

- **public ProductWarehouse(String productName, double capacity)** - Creates an empty product warehouse. The name of the product and the capacity of the warehouse are provided as parameters.
- **public String getName()** - Returns the name of the product.

Remind yourself of how a constructor can run the constructor of the superclass as its first action!

Example usage:

```
ProductWarehouse juice = new ProductWarehouse("Juice", 1000.0);
juice.addToWarehouse(1000.0);
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
System.out.println(juice);           // balance = 988.7, space left 11.3
```

Sample output

Juice
balance = 988.7, space left 11.3

Part 2: Product warehouse, step 2

As we can see from the previous example, the `toString()` inherited by the ProductWarehouse object doesn't (obviously!) know anything about the product name. *Something must be done!* Let's also add a setter for the product name while we're at it:

- **public void setName(String newName)** - sets a new name for the product.
- **public String toString()** - Returns the state of the object represented as a string like this `Juice: balance = 64.5, space left 123.5`

The new `toString()` method could be written using the getters inherited from the superclass, which would give access to values of inherited, but still hidden fields. However, the superclass already has the desired functionality to provide a string representation of the warehouse state, so why bother recreating that functionality? Just take advantage of the inherited `toString()`.

Remind yourself of how to call an overridden method in a subclass!

Usage example:

```
ProductWarehouse juice = new ProductWarehouse("Juice", 1000.0);
juice.addToWarehouse(1000.0);
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left
```

Sample output

Juice
Juice: balance = 989.7, space left 10.299999999999955

Part 3: Change History, step 1

Sometimes it might be useful to know how the inventory of a product changes over time: Is the inventory often low? Are we usually at the limit? Are the changes in inventory big or small? Etc. Thus we should give the `ProductWarehouse` class the ability to remember the changes in the amount of a product.

Let's begin by creating a tool that aids in the desired functionality.

The storing of the change history could of course have been done using an `ArrayList<Double>` object in the class *ProductWarehouse*, however, we want our own *specialized tool* for this purpose. The tool should be implemented by encapsulating the `ArrayList<Double>` object.

Public constructors and methods of the `ChangeHistory` class:

- **public ChangeHistory()** creates an empty `ChangeHistory` object.
- **public void add(double status)** adds provided status as the latest amount to remember in the change history.
- **public void clear()** empties the history.
- **public String toString()** returns the string representation of the change history. *The string representation provided by the ArrayList class is sufficient.*

Part 4: Change History, step 2

Build on the `ChangeHistory` class by adding analysis methods:

- **public double maxValue()** returns the largest value in the change history. If the history is empty, the method should return zero.
- **public double minValue()** returns the smallest value in the change history. If the history is empty, the method should return zero.
- **public double average()** returns the average of the values in the change history. If the history is empty, the method should return zero.

The methods should not modify the order of the encapsulated list.

Part 5: Product warehouse with history, step 1

Implement `ProductWarehouseWithHistory` as a subclass of `ProductWarehouse`. In addition to all the previous features this new

warehouse also provides services related to the change history of the warehouse inventory. The history is managed using the `ChangeHistory` object.

Public constructors and methods:

- **public ProductWarehouseWithHistory(String productName, double capacity, double initialBalance)** creates a product warehouse. The product name, capacity, and initial balance are provided as parameters.

Set the initial balance as the initial balance of the warehouse, as well as the first value of the change history.

- **public String history()** returns the product history like this `[0.0, 119.2, 21.2]`. Use the string representation of the ChangeHistory object as is.

NB in this initial version the history is not yet working properly; currently it only remembers the initial balance.

Usage example:

```
// the usual:
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice"
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left

// etc

// however, history() still doesn't work properly:
System.out.println(juice.history()); // [1000.0]
// so we only get the initial state of the history set by the constructor..
```

Sample output

```
Juice
Juice: balance = 989.7, space left 10.299999999999955
[1000.0]
```

Part 6: Product warehouse with history, step 2

It's time to make history! The first version didn't know anything but the initial state of the history. Expand the class with the following methods

- **public void addToWarehouse(double amount)** works just like the method in the `Warehouse` class, but we also record the changed state to the history. **NB:** the value recorded in the history should be the warehouse's balance after adding, not the amount added!
- **public double takeFromWarehouse(double amount)** works just like the method in the `Warehouse` class, but we also record the changed state to the history. **NB:** the value recorded in the history should be the warehouse's balance after removing, not the amount removed!

Usage example:

```
// the usual:
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice"
juice.takeFromWarehouse(11.3);
System.out.println(juice.getName()); // Juice
juice.addToWarehouse(1.0);
System.out.println(juice);           // Juice: balance = 989.7, space left

// etc

// and now we have the history:
System.out.println(juice.history()); // [1000.0, 988.7, 989.7]
```

Sample output

```
Juice
Juice: balance = 989.7, space left 10.299999999999955
[1000.0, 988.7, 989.7]
```

Remember how an overriding method can take advantage of the overridden method!

Part 7: Product warehouse with history, step 3

Expand the class with the method

- **public void printAnalysis()**, which prints history related information for the product in the way presented in the example.

Usage example:

```
ProductWarehouseWithHistory juice = new ProductWarehouseWithHistory("Juice"
juice.takeFromWarehouse(11.3);
juice.addToWarehouse(1.0);
//System.out.println(juice.history()); // [1000.0, 988.7, 989.7]

juice.printAnalysis();
```

Sample output

```
Product: Juice
History: [1000.0, 988.7, 989.7]
Largest amount of product: 1000.0
Smallest amount of product: 988.7
Average: 992.8
```

Exercise submission instructions

How to see the solution

Abstract classes

Sometimes, when planning a hierarchy of inheritance, there are cases when there exists a clear concept, but that concept is not a good candidate for an object in itself. The concept would be beneficial from the point of view of inheritance, since it includes variables and functionality that are shared by all the classes that would inherit it. On the other hand, you should not be able to create instances of the concept itself.

An abstract class combines interfaces and inheritance. You cannot create instances of them — you can only create instances of subclasses of an abstract class. They can include normal methods which have a method

body, but it's also possible to define abstract methods that only contain the method definition. Implementing the abstract methods is the responsibility of subclasses. Generally, abstract classes are used in situations where the concept that the class represents is not a clear independent concept. In such a case you shouldn't be able to create instances of it.

To define an abstract class or an abstract method the keyword `abstract` is used. An abstract class is defined with the phrase `public abstract class *NameOfClass*`; an abstract method is defined by `public abstract returnType nameOfMethod`. Let's take a look at the following abstract class called `Operation`, which offers a structure for operations and executing them.

```
public abstract class Operation {

    private String name;

    public Operation(String name) {
        this.name = name;
    }

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

    public abstract void execute(Scanner scanner);
}
```

The abstract class `Operation` works as a basis for implementing different actions. For instance, you can implement the plus operation by extending the `Operation` class in the following manner.

```
public class PlusOperation extends Operation {

    public PlusOperation() {
        super("PlusOperation");
    }

    @Override
    public void execute(Scanner scanner) {
        System.out.print("First number: ");
        int first = Integer.valueOf(scanner.nextLine());
```

```
        System.out.print("Second number: ");
        int second = Integer.valueOf(scanner.nextLine());

        System.out.println("The sum of the numbers is " + (first + second));
    }
}
```

Since all the classes that inherit from `Operation` have also the type `Operation`, we can create a user interface by using `Operation` type variables. Next we'll show the class `UserInterface` that contains a list of operations and a scanner. It's possible to add operations to the UI dynamically.

```
public class UserInterface {

    private Scanner scanner;
    private ArrayList<Operation> operations;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
        this.operations = new ArrayList<>();
    }

    public void addOperation(Operation operation) {
        this.operations.add(operation);
    }

    public void start() {
        while (true) {
            printOperations();
            System.out.println("Choice: ");

            String choice = this.scanner.nextLine();
            if (choice.equals("0")) {
                break;
            }

            executeOperation(choice);
            System.out.println();
        }
    }

    private void printOperations() {
        System.out.println("\t0: Stop");
        int i = 0;
        while (i < this.operations.size()) {
```

```
            String operationName = this.operations.get(i).getName();
            System.out.println("\t" + (i + 1) + ": " + operationName);
            i = i + 1;
        }
    }

    private void executeOperation(String choice) {
        int operation = Integer.valueOf(choice);

        Operation chosen = this.operations.get(operation - 1);
        chosen.execute(scanner);
    }
}
```

The user interface works like this:

```
UserInterface userInterface = new UserInterface(new Scanner(System.in));
userInterface.addOperation(new PlusOperation());

userInterface.start();
```

Sample output

```
Operations:
0: Stop
1: PlusOperation
Choice: 1
First number: 8
Second number: 12
The sum of the numbers is 20

Operations:
0: Stop
1: PlusOperation
Choice: 0
```

The greatest difference between interfaces and abstract classes is that abstract classes can contain object variables and constructors in addition to methods. Since you can also define functionality in abstract classes, you can use them to define e.g. default behavior. In the user interface

above storing the name of the operation used the functionality defined in the abstract `Operation` class.

Programming exercise:

DifferentKindsOfBoxes (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In the exercise template you'll find the classes `Item` and `Box`. `Box` is an abstract class, where adding multiple items is implemented by repeatedly calling the `add`-method. The `add`-method, meant for adding a single item, is abstract, so every class that inherits it, must implement it. Your assignment is to edit the `Box`-class and to implement different kinds of boxes based on the `Box` class.

```
import java.util.ArrayList;

public abstract class Box {

    public abstract void add(Item item);

    public void add(ArrayList<Item> items) {
        for (Item item : items) {
            Box.this.add(item);
        }
    }

    public abstract boolean isInBox(Item item);
}
```

Part 1: Editing the Item class

Implement the `equals` and `hashCode` methods for the `Item`-class. They are needed, so that you can use the `contains`-methods of

different lists and collections. Implement the methods in such a way that value of the `weight` instance variable of the `Item`-class isn't considered. *It's probably a good idea to make use of Netbeans's functionality to implement the `equals` and `hashCode` methods*

Part 2: Box with a max weight

Implement the class `BoxWithMaxWeight`, that inherits the `Box` class. BoxWithMaxWeight has a constructor `public BoxWithMaxWeight(int capacity)`, that defines the max weight allowed for that box. You can add an item to a BoxWithMaxWeight when and only when, adding the item won't cause the boxes maximum weight capacity to be exceeded.

```
BoxWithMaxWeight coffeeBox = new BoxWithMaxWeight(10);
coffeeBox.add(new Item("Saludo", 5));
coffeeBox.add(new Item("Pirkka", 5));
coffeeBox.add(new Item("Kopi Luwak", 5));

System.out.println(coffeeBox.isInBox(new Item("Saludo")));
System.out.println(coffeeBox.isInBox(new Item("Pirkka")));
System.out.println(coffeeBox.isInBox(new Item("Kopi Luwak")));
```

Sample output

```
true
true
false
```

Part 3: One item box and the misplacing box

Next, implement the class `OneItemBox`, that inherits the `Box` class. `OneItemBox` has the constructor `public OneItemBox()`, and it has the capacity of exactly one item. If there is already an item in the box, it must not be switched. The weight of the item added to the box is irrelevant.

```
OneItemBox box = new OneItemBox();
box.add(new Item("Saludo", 5));
box.add(new Item("Pirkka", 5));
```

```
System.out.println(box.isInBox(new Item("Saludo")));
System.out.println(box.isInBox(new Item("Pirkka")));
```

Sample output

```
true
false
```

Next implement the class `MisplacingBox`, that inherits the `Box`-class. MisplacingBox has a constructor `public MisplacingBox()`. You can add any items to a misplacing box, but items can never be found when looked for. In other words adding to the box must always succeed, but calling the method `isInBox` must always return false.

```
MisplacingBox box = new MisplacingBox();
box.add(new Item("Saludo", 5));
box.add(new Item("Pirkka", 5));

System.out.println(box.isInBox(new Item("Saludo")));
System.out.println(box.isInBox(new Item("Pirkka")));
```

Sample output

```
false
false
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 2. Interfaces

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

1. Class inheritance

2. Interfaces

3. Object polymorphism

4. Summary

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI