Part 9

Interfaces

Learning Objectives

- You're familiar with the concept of an interface, can define your own interfaces, and implement an interface in a class.
- You know how to use interfaces as variable types, method parameters and method return values.
- You're aware of some of the interfaces that come with Java.

We can use interfaces to define behavior that's required from a class, i.e., its methods. They're defined the same way that regular Java classes are, but "`public interface ...`" is used instead of "`public class ...` " at the beginning of the class. Interfaces define behavior through method names and their return values. However, they don't always include the actual implementations of the methods. A visibility attribute on interfaces is not marked explicitly as they're always `public`. Let's examine a *Readable* interface that describes readability.

```
public interface Readable {
    String read();
}
```

The `Readable` interface declares a `read()` method, which returns a String-type object. Readable defines certain behavior: for example, a text message or an email may be readable.

The classes that implement the interface decide *how* the methods defined in the interface are implemented. A class implements the interface by adding the keyword *implements* after the class name followed by the

name of the interface being implemented. Let's create a class called `TextMessage` that implements the `Readable` interface.

```
public class TextMessage implements Readable {
    private String sender;
    private String content;

    public TextMessage(String sender, String content) {
        this.sender = sender;
        this.content = content;
    }

    public String getSender() {
        return this.sender;
    }

    public String read() {
        return this.content;
    }
}
```

Since the `TextMessage` class implements the `Readable` interface (`public class TextMessage implements Readable`), the `TextMessage` class *must* contain an implementation of the `public String read()` method. Implementations of methods defined in the interface must always have public as their visibility attribute.

An Interface Is a Contract of Behaviour

When a class implements an interface, it signs an agreement. The agreement dictates that the class will implement the methods defined by the interface. If those methods are not implemented in the class, the program will not function.

The interface defines only the names, parameters, and return values of the required methods. The interface, however, does not have a say on the internal implementation of its methods. It is the responsibility of the programmer to define the internal functionality for the methods.

In addition to the `TextMessage` class, let's add another class that implements the `Readable` interface. The `Ebook` class is an electronic implementation of a book that containing the title and pages of a book. The ebook is read page by page, and calling the `public String read()` method always returns the next page as a string.

```
public class Ebook implements Readable {
    private String name;
    private ArrayList<String> pages;
    private int pageNumber;

    public Ebook(String name, ArrayList<String> pages) {
        this.name = name;
        this.pages = pages;
        this.pageNumber = 0;
    }

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

    public int pages() {
        return this.pages.size();
    }

    public String read() {
        String page = this.pages.get(this.pageNumber);
        nextPage();
        return page;
    }

    private void nextPage() {
        this.pageNumber = this.pageNumber + 1;
        if(this.pageNumber % this.pages.size() == 0) {
            this.pageNumber = 0;
        }
    }
}
```

Objects can be instantiated from interface-implementing classes just like with normal classes. They're also used in the same way, for instance, as an ArrayList's type.

```
TextMessage message = new TextMessage("ope", "It's going great!");
System.out.println(message.read());
```

```
ArrayList<TextMessage> textMessage = new ArrayList<>();
textMessage.add(new TextMessage("private number", "I hid the body.");
```

Sample output

```
It's going great!
```

```
ArrayList<String> pages = new ArrayList<>();
pages.add("Split your method into short, readable entities.");
pages.add("Separate the user-interface logic from the application logic.");
pages.add("Always program a small part initially that solves a part of the prob
pages.add("Practice makes the master. Try different out things for yourself and

Ebook book = new Ebook("Tips for programming.", pages);

int page = 0;
while (page < book.pages()) {
    System.out.println(book.read());
    page = page + 1;
}
```

Sample output

```
Split your method into short, readable entities.
Separate the user-interface logic from the application logic.
Always program a small part initially that solves a part of the problem.
Practice makes the master. Try different out things for yourself and
work on your own projects.
```

Programming exercise:

TacoBoxes (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.

In the exercise template you'll find Interface `TacoBox` ready for your use. It has the following methods:

- the method `int tacosRemaining()` return the number of tacos remaining in the box.
- the method `void eat()` reduces the number of tacos remaining by one. The number of tacos remaining can't become negative.

```
public interface TacoBox {
    int tacosRemaining();
    void eat();
}
```

Part 1: Triple taco box

Implement the class `TripleTacoBox`, that implements the `TacoBox` interface. `TripleTacobox` has a constructor with no parameters. `TripleTacobox` has an object variable `tacos` which is initialized at 3 when the constructor is called.

Part 2: Custom taco box

Implement the class `CustomTacoBox`, that implements the `TacoBox` interface. `CustomTacoBox` has a constructor with one parameter defining the initial number of tacos in the box(`int tacos`).

Exercise submission instructions

How to see the solution

Interface as Variable Type

The type of a variable is always stated as its introduced. There are two kinds of type, the primitive-type variables (int, double, ...) and reference-

type variables (all objects). We've so far used an object's class as the type of a reference-type variable.

```
String string = "string-object";
TextMessage message = new TextMessage("ope", "many types for the same object");
```

An object's type can be other than its class. For example, the type of the `Ebook` class that implements the `Readable` interface is both `Ebook` and `Readable`. Similarly, the text message also has multiple types. Because the `TextMessage` class implements the `Readable` interface, it has a `Readable` type in addition to the `TextMessage` type.

```
TextMessage message = new TextMessage("ope", "Something cool's about to happen)
Readable readable = new TextMessage("ope", "The text message is Readable!");
```

```
ArrayList<String> pages = new ArrayList<>();
pages.add("A method can call itself.");

Readable book = new Ebook("Introduction to Recursion", pages);

int page = 0;
while (page < book.pages()) {
    System.out.println(book.read());
    page = page + 1;
}
```

Because an interface can be used as a type, it's possible to create a list that contains objects of the interface's type.

```
ArrayList<Readable> readingList = new ArrayList<>();

readingList.add(new TextMessage("ope", "never been programming before..."));
readingList.add(new TextMessage("ope", "gonna love it i think!"));
readingList.add(new TextMessage("ope", "give me something more challenging! :)"
readingList.add(new TextMessage("ope", "you think i can do it?"));
readingList.add(new TextMessage("ope", "up here we send several messages each d
```

```
ArrayList<String> pages = new ArrayList<>();
pages.add("A method can call itself.");

readingList.add(new Ebook("Introduction to Recursion.", pages));

for (Readable readable: readingList) {
    System.out.println(readable.read());
```

Note that although the `Ebook` class that inherits the `Readable` interface class is always of the interface's type, not all classes that implement the `Readable` interface are of type `Ebook`. You can assign an object created from the `Ebook` class to a `Readable`-type variable, but it does not work the other way without a separate type conversion.

```
Readable readable = new TextMessage("ope", "TextMessage is Readable!"); // work
TextMessage message = readable; // doesn't work

TextMessage castMessage = (TextMessage) readable; // works if, and only if, rea
```

Type conversion succeeds if, and only if, the variable is of the type that it's being converted to. Type conversion is not considered good practice, and one of the few situation where it's use is appropriate is in the implementation of the `equals` method.

Interfaces as Method Parameters

The true benefits of interfaces are reaped when they are used as the type of parameter provided to a method. Since an interface can be used as a variable's type, it can also be used as a parameter type in method calls. For example, the `print` method in the `Printer` class of the class below gets a variable of type `read`.

```
public class Printer {
    public void print(Readable readable) {
        System.out.println(readable.read());
    }
}
```

The value of the `print` method of the `printer` class lies in the fact that it can be given *any* class that implements the `Readable` interface as a parameter. Were we to call the method with any object instantiated from a class that inherits the Readable class, the method would function as desired.

```
TextMessage message = new TextMessage("ope", "Oh wow, this printer knows how to

ArrayList<String> pages = new ArrayList<>();
pages.add("Values common to both {1, 3, 5} and {2, 3, 4, 5} are {3, 5}.");
Ebook book = new Ebook("Introduction to University Mathematics.", pages);

Printer printer = new Printer();
printer.print(message);
printer.print(book);
```

Sample output

```
Oh wow, this printer knows how to print these as well!
Values common to both {1, 3, 5} and {2, 3, 4, 5} are {3, 5}.
```

Let's make another class called `ReadingList` to which we can add interesting things to read. The class has an `ArrayList` instance as an instance variable, where the things to be read are added. Adding to the reading list is done using the `add` method, which receives a `Readable`-type object as its parameter.

```
public class ReadingList {
    private ArrayList<Readable> readables;

    public ReadingList() {
        this.readables = new ArrayList<>();
    }

    public void add(Readable readable) {
        this.readables.add(readable);
    }

    public int toRead() {
        return this.readables.size();
```

```
    }
}
```

Reading lists are usually readable, so let's have the `ReadingList` class implement the `Readable` interface. The `read` method of the reading list reads all the objects in the `readables` list, and adds them to the string returned by the `read()` method one-by-one.

```
public class ReadingList implements Readable {
    private ArrayList<Readable> readables;

    public ReadingList() {
        this.readables = new ArrayList<>();
    }

    public void add(Readable readable) {
        this.readables.add(readable);
    }

    public int toRead() {
        return this.readables.size();
    }

    public String read() {
        String read = "";

        for (Readable readable: this.readables) {
            read = read + readable.read() + "\n";
        }

        // once the reading list has been read, we empty it
        this.readables.clear();
        return read;
    }
}
```

```
ReadingList jonisList = new ReadingList();
jonisList.add(new TextMessage("arto", "have you written the tests yet?"));
jonisList.add(new TextMessage("arto", "have you checked the submissions yet?"))

System.out.println("Joni's to-read: " + jonisList.toRead());
```

Sample output

```
Joni's to-read: 2
```

Because the `ReadingList` is of type `Readable`, we're able to add `ReadingList` objects to the reading list. In the example below, Joni has a lot to read. Fortunately for him, Verna comes to the rescue and reads the messages on Joni's behalf.

```
ReadingList jonisList = new ReadingList();
int i = 0;
while (i < 1000) {
    jonisList.add(new TextMessage("arto", "have you written the tests yet?"));
    i = i + 1;
}

System.out.println("Joni's to-read: " + jonisList.toRead());
System.out.println("Delegating the reading to Verna");

ReadingList vernasList = new ReadingList();
vernasList.add(jonisList);
vernasList.read();

System.out.println();
System.out.println("Joni's to-read: " + jonisList.toRead());
```

Sample output

```
Joni's to-read: 1000
Delegating the reading to Verna

Joni's to-read:0
```

The `read` method called on Verna's list goes through all the `Readable` objects and calls the `read` method on them. When the `read` method is called on Verna's list it also goes through Joni's reading list that's included in Verna's reading list. Joni's reading list is run through by calling its `read` method. At the end of each `read` method call, the read list is cleared. In this way, Joni's reading list empties when Verna reads it.

As you notice, the program already contains a lot of references. It's a good idea to draw out the state of the program step-by-step on paper and outline how the `read` method call of the `vernasList` object proceeds!

Programming exercise:
Interface In A Box (4 parts)

Points
4/4

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: Packables

Moving houses requires packing all your belongings into boxes. Let's imitate that with a program. The program will have boxes, and items to pack into those boxes. All items must implement the following Interface:

```
public interface Packable {
    double weight();
}
```

Add the Interface to your program. Adding a new Interface is quite similar to adding a new class. Instead of selecting *new Java class* just select *new Java interface*.

Create classes `Book` and `CD`, which implement the Interface. Book has a constructor which is given the author (String), name of the book (String), and the weight of the book (double) as parameters. CD has a constructor which is given the artist (String), name of the CD (String), and the publication year (int). The weight of all CDs is 0.1 kg.

Remember to implement the Interface `Packable` in both of the classes. The classes must work as follows:

```
public static void main(String[] args) {
    Book book1 = new Book("Fyodor Dostoevsky", "Crime and Punishment", 2);
    Book book2 = new Book("Robert Martin", "Clean Code", 1);
    Book book3 = new Book("Kent Beck", "Test Driven Development", 0.5);
```

```
    CD cd1 = new CD("Pink Floyd", "Dark Side of the Moon", 1973);
    CD cd2 = new CD("Wigwam", "Nuclear Nightclub", 1975);
    CD cd3 = new CD("Rendezvous Park", "Closer to Being Here", 2012);

    System.out.println(book1);
    System.out.println(book2);
    System.out.println(book3);
    System.out.println(cd1);
    System.out.println(cd2);
    System.out.println(cd3);
```

Prints:

Sample output

```
Fyodor Dostoevsky: Crime and Punishment
Robert Martin: Clean Code
Kent Beck: Test Driven Development
Pink Floyd: Dark Side of the Moon (1973)
Wigwam: Nuclear Nightclub (1975)
Rendezvous Park: Closer to Being Here (2012)
```

NB: The weight is not printed

Part 2: Box

Make a class called `Box`. Items implementing the `Packable` interface can be packed into a box. The `Box` constructor takes the maximum capacity of the box in kilograms as a parameter. The combined weight of all items in a box cannot be more than the maximum capacity of the box.

Below is an example of using a box:

```
public static void main(String[] args) {
    Box box = new Box(10);

    box.add(new Book("Fyodor Dostoevsky", "Crime and Punishment", 2)) ;
    box.add(new Book("Robert Martin", "Clean Code", 1));
    box.add(new Book("Kent Beck", "Test Driven Development", 0.7));
```

```
        box.add(new CD("Pink Floyd", "Dark Side of the Moon", 1973));
        box.add(new CD("Wigwam", "Nuclear Nightclub", 1975));
        box.add(new CD("Rendezvous Park", "Closer to Being Here", 2012));

        System.out.println(box);
    }
```

Prints

Sample output

```
Box: 6 items, total weight 4.0 kg
```

NB: As the weights are saved as a double, the calculations might have some small rounding errors. You don't need to worry about them.

Part 3: Box weight

If you made an class variable `double weight` in the Box class, replace it with a method which calculates the weight of the box:

```
public class Box {
    //...

    public double weight() {
        double weight = 0;
        // calculate the total weight of the items in the box
        return weight;
    }
}
```

When you need the weight of the box, for example when adding a new item to the box, you can just call the weight method.

The method could also return the value of an object variable. However here we are practicing a situation, where we do not have to maintain an object variable explicitly, but can calculate its value as needed. After the next exercise storing the weight as an object variable would not necessary work anyway. After completing the exercise have a moment to think why that is.

Part 4: A Box is packable too!

Implementing the `Packable` Interface requires a class to have the method `double weight()`. We just added this method to the Box class. This means we can make the Box packable as well!

Boxes are objects, which can contain objects implementing the `packable` Interface. Boxes implement this Interface as well. So **a box can contain other boxes!**

Try this out. Make some boxes containing some items, and add some smaller boxes to a bigger box. Try what happens, when you put a box in itself. Why does this happen?

Exercise submission instructions

How to see the solution

Interface as a return type of a method

Interfaces can be used as return types in methods — just like regular variable types. In the next example is a class `Factory` that can be asked to construct different objects that implement the `Packable` interface.

```
import java.util.Random;

public class Factory {

    public Factory() {
        // Note that there is no need to write an empy constructor without
        // parameters if the class doesn't have other constructors.
        // In these cases Java automatically creates a default constructor for
        // the class which is an empty constructor without parameters.
    }

    public Packable produceNew() {
        // The Random-object used here can be used to draw random numbers.
        Random ticket = new Random();
        // Draws a number from the range [0, 4). The number will be 0, 1, 2, or
```

```
        int number = ticket.nextInt(4);

        if (number == 0) {
            return new CD("Pink Floyd", "Dark Side of the Moon", 1973);
        } else if (number == 1) {
            return new CD("Wigwam", "Nuclear Nightclub", 1975);
        } else if (number == 2) {
            return new Book("Robert Martin", "Clean Code", 1);
        } else {
            return new Book("Kent Beck", "Test Driven Development", 0.7);
        }
    }
```

The Factory can be used without exactly knowing what different kind of Packable classes exist. In the next example there is a class Packer that gives a box of things. A packer defines a factory which is used to create the things:

```
public class Packer {
    private Factory factory;

    public Packer() {
        this.factory = new Factory();
    }

    public Box giveABoxOfThings() {
         Box box = new Box(100);

         int i = 0;
         while (i < 10) {
             Packable newThing = factory.produceNew();
             box.add(newThing);

             i = i + 1;
         }

         return box;
    }
}
```

Because the packer does not know the classes that implement the interface `Packable`, one can add new classes that implement the interface without changing the packer. The next example creates a new

class that implements the Packable interface `ChocolateBar`. The factory has been changed so that it creates chocolate bars in addition to books and CDs. The class `Packer` works without changes with the updated version of the factory.

```
public class ChocolateBar implements Packable {
    // Because Java's automatically generated default constructor is enough,
    // we don't need a constructor

    public double weight() {
        return 0.2;
    }
}
```

```
import java.util.Random;

public class Factory {
    // Because Java's automatically generated default constructor is enough,
    // we don't need a constructor

    public Packable produceNew() {

        Random ticket = new Random();
        int number = ticket.nextInt(5);

        if (number == 0) {
            return new CDDisk("Pink Floyd", "Dark Side of the Moon", 1973);
        } else if (number == 1) {
            return new CDDisk("Wigwam", "Nuclear Nightclub", 1975);
        } else if (number == 2) {
            return new Book("Robert Martin", "Clean Code", 1 );
        } else if (number == 3) {
            return new Book("Kent Beck", "Test Driven Development", 0.7);
        } else {
            return new ChocolateBar();
        }
    }
}
```

Quiz:
Interfaces and polymorphism

Points:
1/1

In the previous section we talked about polymorphism. Think about how polymorphism works together with interfaces. Explain with examples how polymorphism occurs in programs using interfaces.

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 or interface. When combined with interfaces, polymorphism enables the use of multiple classes through a common interface, providing flexibility and scalability in your code.

How Polymorphism Works with Interfaces
Interface Definition: An interface defines a contract that classes must follow. Each implementing class can provide its own version of the methods defined in the interface.

Words: 383

Submitted The number of tries is not limited

Reducing the dependencies between classes

Using interfaces in programming enables reducing dependencies between classes. In the previous example the Packer does not depend on the classes that implement the Packable interface. Instead, it just depends on the interface. This makes possible to add new classes that implement the interface without changing the Packer class. What is more, adding new Packable classes doesn't affect the classes that use the Packer class.

Built-in Interfaces

Java offers a considerable amount of built-in interfaces. Here we'll get familiar with four commonly used interfaces: List, Map, Set, and Collection.

The List Interface

The List interface defines the basic functionality related to lists. Because the ArrayList class implements the `List` interface, one can also use it through the `List` interface.

```
List<String> strings = new ArrayList<>();
strings.add("string objects inside an arraylist object!");
```

As we can see fom the Java API of List, there are many classes that implement the `List` interface. One list that is familiar to computer scientists is a linked list. A linked list can be used through the List interface exactly the same way as an object created from ArrayList.

```
List<String> strings = new LinkedList<>();
strings.add("string objects inside a linkedlist object!");
```

From the perspective of the user, both implementations of the `List` interface work the same way. The interface *abstracts* their inner functionality. The internal structures of `ArrayList` and `LinkedList` differ quite a bit. ArrayList saves objects to an array where fetching an object with a specific index is very fast. On the other hand LinkedList constructs a list where each element contains a reference to the next element in the list. When one searches for an object by index in a linked list, one has to go though the list from the beginning until the index.

One can see noticeable performance differences between list implementations if the lists are big enough. The strength of a linked list is that adding to it is always fast. ArrayList, on the other hand, is backed by an array, which needs to be resized each time it gets full. Resizing the array requires creating a new array and copying the values from the old

array to the new one. On the other hand, searching objects by index is much faster in an array list compared to a linked list.

For the problems that you encounter during this course you should almost always choose ArrayList. However, "interface programming" is beneficial: implement your programs so that you'll use the data structures through the interfaces.

Programming exercise:

List as a method parameter

Points 1/1

In the mainProgram class, implement a class method `returnSize`, which is given a List-object as a parameter, and returns the size of the list as an integer.

The method should work as follows:

```
List<String> names = new ArrayList<>();
names.add("First");
names.add("Second");
names.add("Third");

System.out.println(returnSize(names));
```

Sample output

```
3
```

Exercise submission instructions

How to see the solution

The Map Interface

The Map interface defines the basic behavior associated with hash tables. Because the HashMap class implements the `Map` interface, it can also be accessed through the `Map` interface.

```
Map<String, String> maps = new HashMap<>();
maps.put("ganbatte", "good luck");
maps.put("hai", "yes");
```

The keys to the hash table are obtained using the `keySet` method.

```
Map<String, String> maps = new HashMap<>();
maps.put("ganbatte", "good luck");
maps.put("hai", "yes");

for (String key : maps.keySet()) {
    System.out.println(key + ": " + maps.get(key));
}
```

Sample output

```
ganbatte: good luck
hai: yes
```

The `keySet` method returns a set of elements that implement the `Set` interface. You can use a for-each statement to go through a set that implements the `Set` interface. The hash values can be obtained from the hash table using the `values` method. The `values` method returns a set of elements that implement the `Collection` interface. Let's take a quick look at the `Set` and `Collection` interfaces.

Programming exercise:

Map as a method parameter

Points 1/1

In the class MainProgram implement a class method `returnSize` which gets a Map-object as a parameter, and returns its size as an integer.

The method should work as follows:

```
Map<String, String> names = new HashMap<>();
names.put("1", "first");
names.put("2", "second");

System.out.println(returnSize(names));
```

Sample output

```
2
```

Exercise submission instructions

How to see the solution

The Set Interface

The Set interface describes functionality related to sets. In Java, sets always contain either 0 or 1 amounts of any given object. As an example, the set interface is implemented by HashSet. Here's how to go through the elements of a set.

```
Set<String> set = new HashSet<>();
set.add("one");
set.add("one");
set.add("two");

for (String element: set) {
    System.out.println(element);
}
```

Sample output

```
one
two
```

Note that HashSet in no way assumes the order of a set of elements. If objects created from custom classes are added to the HashSet object, they must have both the `equals` and `hashCode` methods defined.

Programming exercise:

Set as method parameter

Points 1/1

In the Main-class, implement the static method `returnSize`, which receives a Set object as a parameter and returns its size.

The method should work e.g. like this:

```
Set<String> names = new HashSet<>();
names.add("first");
names.add("first");
names.add("second");
names.add("second");
names.add("second");

System.out.println(returnSize(names));
```

Prints:

Sample output

```
2
```

Exercise submission instructions

How to see the solution

The Collection Interface

The Collection interface describes functionality related to collections. Among other things, lists and sets are categorized as collections in Java — both the List and Set interfaces implement the Collection interface. The Collection interface provides, for instance, methods for checking the existence of an item (the method `contains`) and determining the size of a collection (the method `size`).

The Collection interface also determines how the collection is iterated over. Any class that implements the Collection interface, either directly or

indirectly, inherits the functionality required for a `for-each` loop.

Let's create a hash table and iterate over its keys and values.

```
Map<String, String> translations = new HashMap<>();
translations.put("ganbatte", "good luck");
translations.put("hai", "yes");

Set<String> keys = translations.keySet();
Collection<String> keyCollection = keys;

System.out.println("Keys:");
for (String key: keyCollection) {
    System.out.println(key);
}

System.out.println();
System.out.println("Values:");
Collection<String> values = translations.values();

for (String value: values) {
    System.out.println(value);
}
```

Sample output

```
Keys:
ganbatte
hai

Values:
yes
good luck
```

In the next exercise, we build functionality related to e-commerce and practice using classes through their interfaces.

Programming exercise:

Online shop (8 parts)

Points 9/9

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 this exercise we'll create program components, that can be used to run an online store.

Part 1: Warehouse

Create the class `Warehouse` with the following methods:

- `public void addProduct(String product, int price, int stock)`, which adds a product to the warehouse with the price and stock balance given as parameters.
- `public int price(String product)`, which returns the price of the product it received as a parameter. If the product hasn't been added to the warehouse, the method must return -99.

The products in the warehouse (and in the next part their stock) must be stored in a variable of the type `Map<String, Integer>`! The object created can be a HashMap, but its type must be the Map-interface, rather than any implementation of that interface.

```
Warehouse warehouse = new Warehouse();
warehouse.addProduct("milk", 3, 10);
warehouse.addProduct("coffee", 5, 7);

System.out.println("prices:");
System.out.println("milk: " + warehouse.price("milk"));
System.out.println("coffee: " + warehouse.price("coffee"));
System.out.println("sugar: " + warehouse.price("sugar"));
```

Prints:

Sample output

```
prices:
milk: 3
coffee: 5
sugar: -99
```

Part 2: Products stock balance

Save the stock balance of products in a variable with the `Map<String, Integer>` type, in the same way the prices were stored. Supplement the warehouse with the following methods:

- `public int stock(String product)` returns the current remaining stock of the product in the warehouse. If the product hasn't been added to the warehouse, the method must return 0.
- `public boolean take(String product)` reduces the stock remaining for the product it received as a parameter by one, and returns true if there was stock remaining. If the product was not available in the warehouse the method returns false. A products stock can't go below zero.

An example of the warehouse in use:

```
Warehouse warehouse = new Warehouse();
warehouse.addProduct("coffee", 5, 1);

System.out.println("stock:");
System.out.println("coffee:  " + warehouse.stock("coffee"));
System.out.println("sugar: " + warehouse.stock("sugar"));

System.out.println("taking coffee " + warehouse.take("coffee"));
System.out.println("taking coffee " + warehouse.take("coffee"));
System.out.println("taking sugar " + warehouse.take("sugar"));

System.out.println("stock:");
System.out.println("coffee:  " + warehouse.stock("coffee"));
System.out.println("sugar: " + warehouse.stock("sugar"));
```

Prints:

Sample output

```
stock:
coffee:  1
sugar: 0
taking coffee true
taking coffee false
taking sugar false
stock:
coffee:  0
sugar: 0
```

Part 3: Listing the products

Let's add one more method to the warehouse:

- `public Set<String> products()` returns the names of the products in the warehouse as a *Set*

This method is easy to implement with HashMap. You can get the products in the warehouse from either the Map storing the prices or the one storing current stock, by using the method `keySet()`

An example use case:

```
Warehouse warehouse = new Warehouse();
warehouse.addProduct("milk", 3, 10);
warehouse.addProduct("coffee", 5, 6);
warehouse.addProduct("buttermilk", 2, 20);
warehouse.addProduct("yogurt", 2, 20);

System.out.println("products:");

for (String product: warehouse.products()) {
    System.out.println(product);
}
```

Sample output

```
products:
buttermilk
yogurt
coffee
milk
```

Part 4: Item

Items can be added to the shopping cart (which we'll add soon). An item is a product with a quantity. You for example add an item representing one bread to the cart, or add an item representing 24 coffees.

Create the class `Item` with the following methods:

- `public Item(String product, int qty, int unitPrice)`; a constructor that creates an item corresponding to the product given as a parameter. *qty* tells us how many of the product are in the item, while *unitPrice* is the price of a single product.
- `public int price()` return the price of the item. You get the items price by multiplying its unit price by its quantity(qty).
- `public void increaseQuantity` increases the quantity by one.
- `public String toString()` returns the string representation of the item. which must match the format shown in the example below.

An example of the Item class being used:

```
Item item = new Item("milk", 4, 2);
System.out.println("an item that contains 4 milks has the total price of "
System.out.println(item);
item.increaseQuantity();
System.out.println(item);
```

Sample output

```
an item that contains 4 milks has the total price of 8
milk: 4
milk: 5
```

NB: The *toString* is formatted like this: *product: qty* — price is not included in the string representation.

Part 5: Shopping cart

We finally get to implement the shopping cart class!

Internally, `ShoppingCart` stores products added there as *Item-objects*. ShoppingCart must have an instance variable with either the `Map<String, Item>` type, or the `List<Item>` type. Don't add any other instance variable to the ShoppingCart class, besides the List or Map used to store the items.

NB: If you save the items in a Map type variable, you'll finds its `values()` method to be quite useful for going though all the items objects stored in it for both this part of the exercise and the next.

First let's give `ShoppingCart` a constructor with no parameters and these methods:

- `public void add(String product, int price)` adds an item to the cart that matches the product given as a parameter, with the price given as a parameter.
- `public int price()` returns the total price of the shopping cart.

```
ShoppingCart cart = new ShoppingCart();
cart.add("milk", 3);
cart.add("buttermilk", 2);
cart.add("cheese", 5);
System.out.println("cart price: " + cart.price());
cart.add("computer", 899);
System.out.println("cart price: " + cart.price());
```

Sample output

```
cart price: 10
cart price: 909
```

Part 6: Printing the cart

Implement the method `public void print()` for the shopping cart. The method prints the *Item-objects* in the cart. The order they are printed in is irrelevant. E.g the print of the cart in the previous example would be:

Sample output

```
buttermilk: 1
cheese: 1
computer: 1
milk: 1
```

NB: the number printed is the quantity in the cart, not the price!

Part 7: One item per product

Let's change our cart so that if a product is being added thats already in the cart, we don't add a new item, but instead update item already in the cart by calling its *increaseQuantity()* method.

E.g:

```
ShoppingCart cart = new ShoppingCart();
cart.add("milk", 3);
cart.print();
System.out.println("cart price: " + cart.price() + "\n");

cart.add("buttermilk", 2);
cart.print();
System.out.println("cart price: " + cart.price() + "\n");

cart.add("milk", 3);
cart.print();
System.out.println("cart price: " + cart.price() + "\n");

cart.add("milk", 3);
cart.print();
System.out.println("cart price: " + cart.price() + "\n");
```

Sample output

```
milk: 1
cart price: 3

buttermilk: 1
milk: 1
cart price: 5

buttermilk: 1
milk: 2
cart price: 8

buttermilk: 1
milk: 3
cart price: 11
```

So in the example above, we first added milk and buttermilk and they get their own Item-objects. When more milk is added to to cart, instead of adding new items we increase the quantity in the item representing milk.

Part 8: Store

We now have all the parts we need for our "online store", except the store itself. Let's make that next. Our store has a warehouse that includes all our products. For each 'visit' we have a shopping cart. Every time the customer chooses a product its added to their cart if its available in the warehouse. At the same time, the stock in the warehouse is reduced by one.

Below you'll find a template for a text-based user interface for our store. Create a `Store` class for your project and copy-paste the code below there.

```
import java.util.Scanner;

public class Store {

    private Warehouse warehouse;
    private Scanner scanner;

    public Store(Warehouse warehouse, Scanner scanner) {
        this.warehouse = warehouse;
        this.scanner = scanner;
    }

    // the method that handles the customers visit to the store.
    public void shop(String customer) {
        ShoppingCart cart = new ShoppingCart();
        System.out.println("Welcome to the store " + customer);
        System.out.println("our selection:");

        for (String product : this.warehouse.products()) {
            System.out.println(product);
        }

        while (true) {
            System.out.print("What to put in the cart (press enter to go to
            String product = scanner.nextLine();
            if (product.isEmpty()) {
                break;
            }

            // Add code here that adds the product to the cart,
            // If there is any in the warehouse, and reduces the stock in t
            // Don't touch any of the other code!
        }

        System.out.println("your shoppingcart contents:");
```

```
        cart.print();
        System.out.println("total: " + cart.price());
    }
```

The following is a main method that stocks the stores warehouse and sends John to shop in the store.

```
Warehouse warehouse = new Warehouse();
    warehouse.addProduct("coffee", 5, 10);
    warehouse.addProduct("milk", 3, 20);
    warehouse.addProduct("cream", 2, 55);
    warehouse.addProduct("bread", 7, 8);

    Scanner scanner = new Scanner(System.in);

    Store store = new Store(warehouse, scanner);
    store.shop("John");
```

The store is almost done. The method `public void shop(String customer)` has a part you need to complete, marked with comments. In the marked part, add code that checks if the product requested by the customer is available and has stock in the warehouse. If so, reduce the products stock in the warehouse and add the product to the shopping cart.

In reality an online store would be implemented a little differently. Web-apps have an HTML-page as a user interface, and clicks there are sent to a server application. There are several courses related to web development available at the University Of Helsinki.

Exercise submission instructions

How to see the solution

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

→ 3. Object polymorphism

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