НИУ ИТМО

Факультет программной инженерии и компьютерной техники

**ЛАБОРАТОРНАЯ РАБОТА № 3**

по дисциплине

‘ПРОГРАММИРОВАНИЕ’

Вариант № 121010

*Выполнил:*

Студент группы P3110

Абрабоу Ахмед Елсаид А.И

*Преподаватель:*Сорокин Роман Борисович



Санкт-Петербург, 2021

***Задание:***

***Text

Description automatically generated***

***Диограмма:***Diagram

Description automatically generated

***GitHub//:***

***https://github.com/a0-0/ITMO-Programming-Labs/tree/master/prog%20lab3***

***Код программы:***

***Class Main***import buildings.Factory;  
import characters.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 Police police = new Police();  
 Villagers villagers = new Villagers();  
 Astronaut astronaut = new Astronaut();  
 Znayka znayka = new Znayka();  
 FactoryWorkers factoryWorkers = new FactoryWorkers();  
 Scooperfield scooperfield = new Scooperfield();  
 Factory factory = new Factory();  
 System.*out*.print("Нечего и говорить, что ");  
 police.fearOfRocket();  
 villagers.goToAstronauts(astronaut.getName());  
 villagers.getSeedsFromAstronaut(astronaut.giveSeedsToVillagers());  
 villagers.tellAboutPlantingSites();  
 znayka.giveOrder();  
 villagers.getWeightlessnessDevicesFromAstronaut(astronaut.giveWeightlessnessDevicesToVillagers());  
 villagers.getAntiluniteFromAstronaut(astronaut.giveAntiluniteToVillagers());  
 villagers.getExplainFromAstronaut(astronaut.explainHowToUse());  
 factoryWorkers.arriveAtTheAstronauts();  
 factoryWorkers.tellWhatTheyDecided();  
 factoryWorkers.thinkAboutTheyDecided();  
 }  
}

***Classes of package Characters***

package characters;  
  
import resources.Antilunite;  
import resources.Seeds;  
import resources.WeightlessnessDevices;  
  
import java.util.Objects;  
  
public class Astronaut implements AstronautHelp {  
 private final String name = "космонавтам";  
 private final String instruction = "Пользуйтесь так!";  
 private final Seeds seeds = new Seeds();  
 private final WeightlessnessDevices weightlessnessDevices = new WeightlessnessDevices();  
 private final Antilunite antilunite = new Antilunite();  
  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public WeightlessnessDevices giveWeightlessnessDevicesToVillagers() {  
 return weightlessnessDevices;  
 }  
  
 @Override  
 public Seeds giveSeedsToVillagers() {  
 return seeds;  
 }  
  
 @Override  
 public Antilunite giveAntiluniteToVillagers() {  
 return antilunite;  
 }  
  
 public String explainHowToUse() {  
 return instruction;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Astronaut astronaut = (Astronaut) o;  
 return Objects.*equals*(name, astronaut.name) && Objects.*equals*(seeds, astronaut.seeds) && Objects.*equals*(weightlessnessDevices, astronaut.weightlessnessDevices) && Objects.*equals*(antilunite, astronaut.antilunite);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name, seeds, weightlessnessDevices, antilunite);  
 }  
  
 @Override  
 public String toString() {  
 return "Astronaut{" +  
 "name='" + name + '\'' +  
 ", seeds=" + seeds +  
 ", weightlessnessDevices=" + weightlessnessDevices +  
 ", antilunite=" + antilunite +  
 '}';  
 }  
}

package characters;  
  
import resources.Antilunite;  
import resources.Seeds;  
import resources.WeightlessnessDevices;  
  
public interface AstronautHelp {  
 WeightlessnessDevices giveWeightlessnessDevicesToVillagers();  
  
 Seeds giveSeedsToVillagers();  
  
 Antilunite giveAntiluniteToVillagers();  
}

package characters;  
  
import java.util.Objects;  
  
public class FactoryWorkers extends Villagers {  
 private final String placeOfWork = "скуперфильдовской макаронной фабрики.";  
  
 public void arriveAtTheAstronauts() {  
 System.*out*.println("Вскоре к космонавтам прибыли несколько рабочих со " + placeOfWork);  
 }  
  
 public String sayPlan() {  
 return "прогнать с фабрики Скуперфильда, а макароны будут делать сами без всяких хозяев";  
 }  
  
 public void tellWhatTheyDecided() {  
 System.*out*.println("Они сказали, что решили " + sayPlan() + ".");  
 }  
  
 public void thinkAboutTheyDecided() {  
 System.*out*.println("Чтоб осуществить этот план, им нужно устроить на фабрике невесомость, " +  
 "так как в противном случае полицейские могут помешать им и даже вовсе прогонят их с фабрики.");  
 }  
  
 public boolean arrangeWeightlessness() {  
 return true;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 FactoryWorkers that = (FactoryWorkers) o;  
 return Objects.*equals*(placeOfWork, that.placeOfWork);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(placeOfWork);  
 }  
  
 @Override  
 public String toString() {  
 return "FactoryWorkers{" +  
 "placeOfWork='" + placeOfWork + '\'' +  
 '}';  
 }  
}

package characters;  
  
import resources.Antilunite;  
import resources.Seeds;  
import resources.WeightlessnessDevices;  
  
public interface GetHelpFromAstronaut {  
 void getWeightlessnessDevicesFromAstronaut(WeightlessnessDevices weightlessnessDevices);  
  
 void getAntiluniteFromAstronaut(Antilunite antilunite);  
  
 void getSeedsFromAstronaut(Seeds seeds);  
}

package characters;  
  
import java.util.Objects;  
  
public class Police {  
 private final String policeName = "полицейские";  
  
 public void fearOfRocket() {  
 System.*out*.println(policeName + " боялись теперь и близко " + goToRocket() + ", а не то что " + shootNearRocket() + ".");  
 }  
  
 public String goToRocket() {  
 return "подходить к ракете";  
 }  
  
 public String shootNearRocket() {  
 return "стрелять возле неё";  
 }  
  
 public void preventEvent() {  
 }  
  
 public void banishFromFactory() {  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Police police = (Police) o;  
 return Objects.*equals*(policeName, police.policeName);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(policeName);  
 }  
  
 @Override  
 public String toString() {  
 return "Police{" +  
 "policeName='" + "полиция" + '\'' +  
 '}';  
 }  
}

package characters;  
  
import java.util.Objects;  
  
public class Scooperfield {  
 private String placeOfWork = "Директор макаронной фабрики";  
 private String name = "Скуперфильд";  
  
 public void banishByWorkers() {  
 placeOfWork = "безработный";  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Scooperfield that = (Scooperfield) o;  
 return Objects.*equals*(placeOfWork, that.placeOfWork) && Objects.*equals*(name, that.name);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(placeOfWork, name);  
 }  
  
 @Override  
 public String toString() {  
 return "Scooperfield{" +  
 "placeOfWork='" + placeOfWork + '\'' +  
 ", name='" + name + '\'' +  
 '}';  
 }  
}

package characters;  
  
import buildings.Village;  
import resources.Antilunite;  
import resources.Seeds;  
import resources.WeightlessnessDevices;  
  
import java.util.Objects;  
  
public class Villagers implements GetHelpFromAstronaut {  
 private final String nameOfVillagers = "Деревенские жители";  
 private final String name = "лунтакам";  
 private Seeds seeds;  
 String instruction;  
 WeightlessnessDevices weightlessnessDevices;  
 Antilunite antilunite;  
  
 public void goToAstronauts(String name) {  
 System.*out*.print(nameOfVillagers + " могли беспрепятственно приходить к " + name + " ");  
 }  
  
 public void setWeightlessnessDevices(WeightlessnessDevices weightlessnessDevices) {  
 this.weightlessnessDevices = weightlessnessDevices;  
 }  
  
 public void setAntilunite(Antilunite antilunite) {  
 this.antilunite = antilunite;  
 }  
  
 public void setSeeds(Seeds seeds) {  
 this.seeds = seeds;  
 }  
  
 @Override  
 public void getWeightlessnessDevicesFromAstronaut(WeightlessnessDevices weightlessnessDevices) {  
 setWeightlessnessDevices(weightlessnessDevices);  
 }  
  
 @Override  
 public void getAntiluniteFromAstronaut(Antilunite antilunite) {  
 setAntilunite(antilunite);  
 }  
  
 @Override  
 public void getSeedsFromAstronaut(Seeds seeds) {  
 setSeeds(seeds);  
 System.*out*.println("и получать у них " + seeds.getName() + " гигантских растений.");  
 }  
  
 public void setInstruction(String instruction) {  
 this.instruction = instruction;  
 }  
  
 public void getExplainFromAstronaut(String instruction) {  
 setInstruction(instruction);  
 }  
  
 public void tellAboutPlantingSites() {  
 System.*out*.println("Теперь гигантские семена сажали не только в деревне Нееловке, но и в селе "  
 + Village.*GOLOPYATKINA*.name() + ", "  
 + Village.*BESKHLEBNOV*.name() + ", "  
 + Village.*GOLODAYEVKA*.name() + ", "  
 + Village.*IMPASSABLE*.name() + "и во многих других.");  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Villagers villagers = (Villagers) o;  
 return Objects.*equals*(nameOfVillagers, villagers.nameOfVillagers) && Objects.*equals*(name, villagers.name) && Objects.*equals*(seeds, villagers.seeds) && Objects.*equals*(instruction, villagers.instruction) && Objects.*equals*(weightlessnessDevices, villagers.weightlessnessDevices) && Objects.*equals*(antilunite, villagers.antilunite);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(nameOfVillagers, name, seeds, instruction, weightlessnessDevices, antilunite);  
 }  
  
 @Override  
 public String toString() {  
 return "Villagers{" +  
 "nameOfVillagers='" + nameOfVillagers + '\'' +  
 ", name='" + name + '\'' +  
 ", seeds=" + seeds +  
 ", instruction='" + instruction + '\'' +  
 ", weightlessnessDevices=" + weightlessnessDevices +  
 ", antilunite=" + antilunite +  
 '}';  
 }  
}

package characters;  
  
import java.util.Objects;  
  
public class Znayka {  
 private String name = "Знайка";  
 private String order = "чтоб лунатикам давали не только нужные им семена, но снабжали их приборами невесомости," +  
 " а также антилунитом и объясняли им, как всем этим пользоваться, чтоб защититься от полицейских.";  
  
 public void giveOrder() {  
 System.*out*.println(name + " распорядился, " + order);  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Znayka znayka = (Znayka) o;  
 return Objects.*equals*(name, znayka.name) && Objects.*equals*(order, znayka.order);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name, order);  
 }  
  
 @Override  
 public String toString() {  
 return "Znayka{" +  
 "name='" + name + '\'' +  
 ", order='" + order + '\'' +  
 '}';  
 }  
}

***classes of Package buildings***

package buildings;  
  
import java.util.Objects;  
  
public class Factory {  
 private String nameOfFactory="макаронная фабрика";  
 private boolean weightlessness=false;  
  
 public boolean isWeightlessness() {  
 return weightlessness;  
 }  
  
 public void setWeightlessness(boolean weightlessness) {  
 this.weightlessness = weightlessness;  
 }  
  
 public String getNameOfFactory() {  
 return nameOfFactory;  
 }  
  
 public void setNameOfFactory(String nameOfFactory) {  
 this.nameOfFactory = nameOfFactory;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Factory factory = (Factory) o;  
 return weightlessness == factory.weightlessness && Objects.*equals*(nameOfFactory, factory.nameOfFactory);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(nameOfFactory, weightlessness);  
 }  
  
 @Override  
 public String toString() {  
 return "Factory{" +  
 "nameOfFactory='" + nameOfFactory + '\'' +  
 ", weightlessness=" + weightlessness +  
 '}';  
 }  
}

package buildings;  
  
public class Rocket {  
 public Rocket() {  
 }  
}

package buildings;  
  
public enum Village {  
 *GOLOPYATKINA*,  
 *BESKHLEBNOV*,  
 *GOLODAYEVKA*,  
 *IMPASSABLE*}

***classes of package resources***

package resources;  
  
import java.util.Objects;  
  
public class Antilunite extends Resource {  
 private String name="антилунит";  
 public Antilunite() {  
 super("антилунитом");  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Antilunite that = (Antilunite) o;  
 return Objects.*equals*(name, that.name);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name);  
 }  
  
 @Override  
 public String toString() {  
 return "Antilunite{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

package resources;  
  
public abstract class Resource {  
 private final String name;  
  
 public Resource(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
}

package resources;  
  
import java.util.Objects;  
  
public class Seeds extends Resource {  
 private String name="семена";  
 public Seeds() {  
 super("семена");  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Seeds seeds = (Seeds) o;  
 return Objects.*equals*(name, seeds.name);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name);  
 }  
  
 @Override  
 public String toString() {  
 return "Seeds{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

package resources;  
  
import java.util.Objects;  
  
public class WeightlessnessDevices extends Resource {  
 private String name = "прибор невесомости";  
  
 public WeightlessnessDevices() {  
 super("приборами невесомости");  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 WeightlessnessDevices that = (WeightlessnessDevices) o;  
 return Objects.*equals*(name, that.name);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(name);  
 }  
  
 @Override  
 public String toString() {  
 return "WeightlessnessDevices{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

**Output:**

Нечего и говорить, что полицейские боялись теперь и близко подходить к ракете, а не то что стрелять возле неё.

Деревенские жители могли беспрепятственно приходить к космонавтам и получать у них семена гигантских растений.

Теперь гигантские семена сажали не только в деревне Нееловке, но и в селе GOLOPYATKINA, BESKHLEBNOV, GOLODAYEVKA, IMPASSABLEи во многих других.

Знайка распорядился, чтоб лунатикам давали не только нужные им семена, но снабжали их приборами невесомости, а также антилунитом и объясняли им, как всем этим пользоваться, чтоб защититься от полицейских.

Вскоре к космонавтам прибыли несколько рабочих со скуперфильдовской макаронной фабрики.

Они сказали, что решили прогнать с фабрики Скуперфильда, а макароны будут делать сами без всяких хозяев.

Чтоб осуществить этот план, им нужно устроить на фабрике невесомость, так как в противном случае полицейские могут помешать им и даже вовсе прогонят их с фабрики.

**Вывод:**

Во время выполнения лабораторной работы я укрепила свои знания в ООП, получила опыт в применении enum, интерфейсов и абстрактных классов. Также изучила методы класса Object (toString, equals, hashCode) и научилась их переопределять. Познакомилась с принципами обьектно-ориентированного программирования SOLID.

**Questions**

Covariance and contravariance

Many programming language type systems support subtyping. For instance, if the type Cat is a subtype of Animal, then an expression of type Cat should be substitutable wherever an expression of type Animal is used.

Variance refers to how subtyping between more complex types relates to subtyping between their components. For example, how should a list of Cats relate to a list of Animals? Or how should a function that returns Cat relate to a function that returns Animal?

### Contravariant method parameter type

**class** **CatShelter** **extends** AnimalShelter {

void putAnimal(Object animal) {

*// ...*

}

}

### Covariant method parameter type

**class** **CatShelter** **extends** AnimalShelter {

void putAnimal(**covariant** Cat animal) {

*// ...*

}

}

**Interface**

An interface is more flexible from a client's point of view: any class can implement any interface. But the interface is "stiffer" from the point of view of its developer: it is more difficult to change it (the work of all clients will be broken), you cannot impose restrictions on the client's constructor, and you cannot reuse the code.

**Abstractions**

An abstract class is "stiffer" from the clients' point of view: the client will be forced to abandon the current base class. But an abstract class is "more flexible" from the point of view of its developer: it allows you to reuse the code, restrict the constructor of descendants, allow you to make changes (easily add a virtual method without breaking existing clients), and more clearly define a "contract" with descendants using Template Methods.

**Lambda expressions**

Lambda expressions are anonymous functions (may not be 100% correct in Java, but they add some clarity). In simpler terms, it is a method without declaration, without access modifiers, returning value and name.

In short, they allow you to write a method and use it immediately. It is especially useful in the case of a single method call, since cuts the time to declare and write a method without having to create a class.

**What is a functional interface**

A functional interface is an interface that contains exactly one abstract method, that is, a description of a method without a body. At the same time, static methods and methods are not counted by default, there can be any number of them in the functional interface.

When a method parameter is a functional interface, one of the arguments must be a code block when calling that method.

**SOLID is an abbreviation for:**

**https://ru.stackoverflow.com/questions/900455/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B-solid-%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BD%D1%8B%D0%BC-%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%BC-%D0%BD%D0%B0-%D0%9F%D0%A0%D0%9E%D0%A1%D0%A2%D0%AB%D0%A5-%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%D0%B0%D1%85-%D0%BA%D0%BE%D0%B4%D0%B0-%D0%A1**

**SRP:** Single responsibility principle - "A class should have only one reason to change"

**OCP:** Open / Closed principle - Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

**LSP:** Liskov substitution principle - the ability to substitute any of its subtypes instead of the base type.

**ISP:** Interface segregation principle

**DIP:** Dependency inversion principle - Dependency inversion (You need to depend on abstractions)

**Dependency inversion.** Throughout this principle is explained as "Top-level modules should not depend on lower-level modules. Both should depend on abstractions." As for me, this phrase is rather difficult to understand. What does this mean in practice?

Let's turn to the author of the principle (Bob Martin), and in short he says, "You need to depend on abstractions, not on something specific." Applying this principle, some modules can be easily replaced with others, just changing the dependency module, and then no changes in the low-level module will affect the high-level one.

There should be no variables that store references to specific classes.

There should be no classes derived from concrete classes.

There should be no methods that override a method implemented in one of the base classes.

But at the same time, it is quite normal dependence on the "native" classes of the language. That is, dependence on String is quite normal. If we write classes ourselves, then they can be mutable. We don't want to depend directly on these classes.

**Liskov substitution** - the idea of ПРАВИЛЬНОГО implementing the "RIGHT" polymorphism. That is, you need to implement the inheritance of subtypes so that it would be possible to substitute any of its subtypes instead of the base type.

**Open / Closed** - Software entities must be open for extension, but closed for modification.

What does it mean?

**Closed** for modification means: the only reason why you can change the code of a class \ function \ module is directly changing the function embedded in it or fixing errors in the operation of this function. There can be no other reason.

**Open** for extensibility means: if you need your class \ function \ module to be able to perform the inherent functions in a new environment, they must support it without changing their code.

**Interface segregation**

Interface segregation is about providing each client with a minimal (ideally separate!) Interface.