Оглавление

[База 4](#_Toc204299331)

[**OVERLOADING - העמסת שיטות** 4](#_Toc204299332)

[DEEP COPY: 5](#_Toc204299333)

[STATIC 5](#_Toc204299334)

[STATIC FINAL 6](#_Toc204299335)

[FINAL CLASS 6](#_Toc204299336)

[**ARRAY - מערכים** 7](#_Toc204299337)

[**STRING - מערכים** 7](#_Toc204299338)

[**ИНКАПСУЛЯЦИЯ עקרון ההכמסה** 8](#_Toc204299339)

[**RETURN NEW object in GETTER** 9](#_Toc204299340)

[**ПРИВЕДЕНИЕ ТИПА -** המרה מפורשת 9](#_Toc204299341)

[**INHERITANCE - НАСЛЕДОВАНИЕ -"ירושה"** 10](#_Toc204299342)

[**פולימורפיזם - ПОЛИМОРФИЗМ** - 13](#_Toc204299343)

[ПРИВЕДЕНИЕ ТИПОВ 13](#_Toc204299344)

[ABSTRACT 16](#_Toc204299345)

[**INTERFACE -"ממשק"** 18](#_Toc204299346)

[**INTERFACE -- COMPARABLE -- SORTER** 21](#_Toc204299347)

[**ENUM** 22](#_Toc204299348)

[**Исключения -- Exceptions – חריגות** 25](#_Toc204299349)

[\_0\_ начало: 31](#_Toc204299350)

[Clock 31](#_Toc204299351)

[TRIANGLE 32](#_Toc204299352)

[\_1\_ overloading 33](#_Toc204299353)

[Bona maatika / (deep copy). DATE 33](#_Toc204299354)

[Bona maatika / (DEEP COPY). POINT 34](#_Toc204299355)

[\_2\_ MURKAVIM 35](#_Toc204299356)

[Сложные отношения (Композиция) 35](#_Toc204299357)

[יחס מורכב - композиция 35](#_Toc204299358)

[MESSAGE 37](#_Toc204299359)

[PRIANGLE 38](#_Toc204299360)

[Student 39](#_Toc204299361)

[TABLE 40](#_Toc204299362)

[KITA 40](#_Toc204299363)

[SCHOOL 41](#_Toc204299364)

[\_3\_ STATIC 42](#_Toc204299365)

[МУЗЫКАЛЬНЫЕ ДИСКИ 42](#_Toc204299366)

[\_3\_ STATIC COUNTER 43](#_Toc204299367)

[\_3\_ STATIC KIDS 44](#_Toc204299368)

[\_3\_ STATIC **LINE** 45](#_Toc204299369)

[\_3\_ STATIC **MyClass** 46](#_Toc204299370)

[\_3\_ STATIC TestRunner 46](#_Toc204299371)

[\_3\_ STATIC 47](#_Toc204299372)

[\_4\_MAARAH 48](#_Toc204299373)

[ZOO 48](#_Toc204299374)

[PIXEL 50](#_Toc204299375)

[SUBJECT 52](#_Toc204299376)

[\_4\_ ERUSHA 54](#_Toc204299377)

[CIRCLE - CONUS 54](#_Toc204299378)

[\_4\_ ERUSHA 54](#_Toc204299379)

[**\_5\_ERUSHA – STATIC--CAR** 57](#_Toc204299380)

[\_6\_ POLIMORPHISM 59](#_Toc204299381)

[-- TOY STORE 59](#_Toc204299382)

[СЮДА СМОТРИ ТУТ ИНСТАНС ОФ 62](#_Toc204299383)

[Работа с классами Point и Vector 65](#_Toc204299384)

[\_6\_ POLIMORPHISM 67](#_Toc204299385)

[-- АЕРОПОРТ И САМОЛЕТЫ 67](#_Toc204299386)

[\_6\_ POLIMORPHISM -- UPCASTING DOWNCASTING 69](#_Toc204299387)

[\_7\_ ABSTRACT 70](#_Toc204299388)

[Shape 70](#_Toc204299389)

[\_7\_ ABSTRACT TEST -- ONE - TWO 70](#_Toc204299390)

[--NODE-- TREK 71](#_Toc204299391)

[-- TOY WAREHOUSE-- 73](#_Toc204299392)

[\_8\_ INTERFACE 76](#_Toc204299393)

[-- Animals 76](#_Toc204299394)

[INTERFACE INSTANCEOF 76](#_Toc204299395)

[\_8\_ INTERFACE -- Pnimiya (static) 77](#_Toc204299396)

[\_8\_ INTERFACE -- IOne Itwo --Подьебка 78](#_Toc204299397)

[\_8\_ INTERFACE -- FIRST SECOND --Подьебка 79](#_Toc204299398)

[\_8\_ INTERFACE -- BigB –Подьебка 79](#_Toc204299399)

[\_8\_ INTERFACE -- COMPARE 80](#_Toc204299400)

[– Transport 80](#_Toc204299401)

[\_9\_ ENUM 82](#_Toc204299402)

[Программа голосования Евровидения 82](#_Toc204299403)

[**\_10\_ Exception** 84](#_Toc204299404)

[Ex1: Mammal 84](#_Toc204299405)

[Ex2: ABCDE с экзамена. Понять какого типа переданный оъект 85](#_Toc204299406)

# База

Уровни доступа: **בקרת גישה**

Принцип инкапсуляции: (עקרון ההכמסה)

קיימות 4 אפשרויות לבקרת גישה לעצם

❑ **private** – הגישה מותרת רק לשיטות החברות במחלקה.

❑ protected – הגישה מותרת גם לשיטות ממחלקות יורשות.

❑ package –הגישה מותרת גם לשיטות שבמחלקות השייכות לאותה חבילה מחדל.

❑ public – הגישה מותרת ל "כל העולם" , כלומר לשיטה כלשהיא.

С помощью конструктора создаются экземпляры класса

בעזרת הפעולה הבונה יוצרים מופעים של המחלקה

Конструктор, как следует из названия – это метод, принадлежащий классу

Имя конструктора совпадает с именем класса, и он не возвращает никакого значения❑

Сигнатура конструктора уникальна:Состоит из модификатора доступа, имени класса и параметров, которые он принимает

Конструктор принимает параметры, связанные со свойствами класса

Свойства класса: הצהרה על תכונות

סיכום בנאיCONSTRUCTOR-

הוא שיטה מיוחדת המוגדרת ע"י המתכנת ונקראת אוטומטית ע"י המערכת בעת יצירת עצם:

❑ שם שיטת ה- constructor כשם המחלקה.

❑ ה- constructor אינה מחזירה ערך כלשהו, אך היא יכולה לקבל פרמטרים, ולכן ניתן

להעמיס אותה.

❑ Constructor שאינו מקבל פרמטרים, מכניס ערכי ברירת מחדל לתכונות של העצם הנוצר.

❑ constructor ברירת מחדל (constructor default (הוא constructor שמופעל כאשר אין

הגדרה מפורשת של ה- constructor במחלקה.

# **OVERLOADING - העמסת שיטות**

НЕ является перегрузкой (отличается только тип возвращаемого значения):

public void method(int x) { }

public int method(int x) { return x; } *//* Ошибка компиляции!

НЕ является перегрузкой (отличается только модификатор доступа):

**private** void method(int x) { }

public void method(int x) { } *//* Ошибка компиляции!

Правильная перегрузка (отличаются параметры):

public void method(int x) { }

public void method(String s) { }

public void method(int x, String s) { }

Напечатаются одинаково  
System.out.println(m);

System.out.println(m.toString());

Всякие подьебки:

|  |  |
| --- | --- |
| public class Book {  **private** String title;  public Book(String title) {  this.title = title;}  Public static String toString() {  return this.title;  }} | Тут косяк, что Public – всегда должен идти с маленькой буквы. И toString статиком делать это тоже зашквар |

# DEEP COPY:

- שיטה בונה מעתיקה- CONSTRUCTOR COPY - ГЛУБОКОЕ КОПИРОВАНИЕ

public Date(Date d) { *//* העתקה בנאי

this.day = d.day;

this.month = d.month;

this.year= d.year;

}

public Message(String from, String to, String content,Date messageDate) {

this.from = from;

this.to = to;

this.content = content;

this.messageDate = new Date(messageDate); *//*פעולה בונה מעתיקה

}

public Date(Date d) *//* העתקה בנאי{

this.day = d.day;

this.month = d.month;

this.year= d.year;}

# STATIC

|  |  |
| --- | --- |
| public staticint **getCount**() **-** Статический метод  Вызывается через имя класса: **Line.getCount**()  Может обращаться только к статическим полям класса  Не требует создания объекта для вызова | publicint **getCount**() **-** Обычный метод экземпляра  Вызывается через объект: **line1.getCount**()  Может обращаться ко всем полям класса  Требует создания объекта для вызова |
| В данном случае правильнее использовать public staticint **getCount**(), потому что:  Поле **count** статическое (общее для всего класса)  Нет необходимости создавать объект для получения общего счетчика  Логичнее получать общее количество линий через класс, а не через конкретный объект  Поэтому второй метод (нестатический) следует удалить. | |

תכונות המוגדרות ע"י המציין static מייצגות מאפיינים כלל מחלקתיים נקראות

תכונות מחלקה )בניגוד לתכונות רגילות הנקראות תכונות עצם(.

מתי נרצה להגדיר תכונה כסטטית?

❑ שיטות המוגדרות ע"י המציין static יכולות להיקרא בהקשר לשם המחלקה, ללא יצירת עצם.

משיטה סטטית ניתן לגשת רק לתכונות מחלקה.

static в Java означает, что это поле принадлежит самому классу, а не конкретному объекту

**Без** static**:**

class Car {

**private** String color = "WHITE"; *//* каждая машина имеет свой личный цвет, не обязательно белый.

}

Car car1 = new Car(); *//* car1.color = "WHITE"

Car car2 = new Car(); *//* car2.color = "WHITE"

car1.changeColor("RED"); *//* теперь car1.color = "RED", а car2.color всё еще "WHITE"

**Со** static**:**

class Car {

**private** static String color = "WHITE";} *//* один цвет для **ВСЕХ** машин

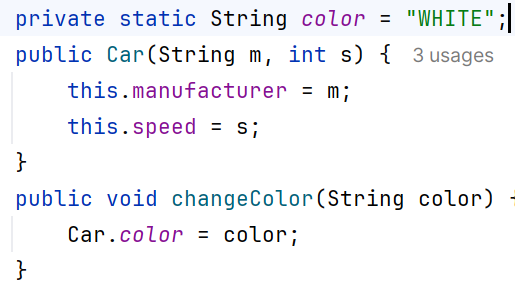
Car car1 = new Car(); *//* Car.color = "WHITE"

Car car2 = new Car(); *//* тот же Car.color = "WHITE"

car1.changeColor("RED"); *//* меняет Car.color на "RED" для ВСЕХ машин!

*//* теперь и car1 и car2 будут красными

Если статик - то this. прописывать в методе не надо

 Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

# STATIC FINAL

**private** static final int SPEED\_UP=5 - это правильное использование static, потому что:

Это константа, которая одинакова для всех машин.Она не меняется (final)

Нет смысла хранить одно и то же значение в каждом объекте

# FINAL CLASS

* Это класс, от которого **нельзя наследоваться**.
* Класс, определенный как final, сам **может наследовать** от другого класса, но **от него наследоваться уже нельзя**.

public final class DeadMachine {..}

Пример из жизни: Класс String является final. Почему?Ответ:Создатели языка хотели предотвратить возможность создания другого класса, который бы наследовался от String и изменял его фундаментальное поведение. Это гарантирует, что строки в Java всегда ведут себя предсказуемо и безопасно, что критически важно, так как они используются повсеместно (например, в сетевых соединениях, доступе к файлам и т.д.).

public class ChessAlgorithm {  
 **enum** ChessPlayer {*WHITE*, *BLACK*}  
 public final ChessPlayer getFirstPlayer() {  
 return ChessPlayer.*WHITE*;  
 }  
}

«белые всегда ходят первыми»

# **ARRAY - מערכים**

int[] arr = new int[5]; *//* массив длины 5

arr[0] = 1; *//* set - установка значения

int x = arr[0]; *//* get - получение значения

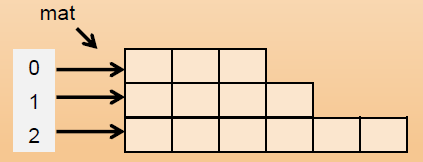
*//* arr[5] = 1; *//* ArrayIndexOutOfBoundsException - индекс 5 вне диапазона [0-4]

|  |  |  |
| --- | --- | --- |
| Пример | Использование | Описание |
| int[] digits ={ 0,1,2,3,4,5,6,7,8,9} | в списке инициализации | длина массива и значения известны заранее |
| int[] digits = new int [7] | массив будет выделен во время выполнения и программа во время производства | длина известна, но не значения |
| int[] digits;  digits = new int[num]; | используем переменную, выделение позже во время выполнения | длина неизвестна и значения тоже |

Типичное исключение- חריגה אופיינית NullPointerException -

הריגה מסוג הצבעה על מקום ריק исключение типа указатель на пустое место-

**מערכים דו-ממדיים לא מלבנייםДВУМЕРНЫЕ НЕРАВНОМЕРНЫЕ МАССИВЫ -**



int[][] mat = new int[3][];

mat[0] = new int[3];

mat[1] = new int[4];

mat[2] = new int[6];

int [][] mat = { { 0,1,2}, {1,1,1,1 }, {1,2,3,4,5,6 } };

# **STRING - מערכים**

**יצירת מחרוזות** overloading

|  |  |  |  |
| --- | --- | --- | --- |
| Конечное значение | Пример | подпись | Описание |
| str2 = ""; | String str2 = new String(); | String() | конструктор создающий пустую строку |
| str1 = "hello world"; | String str1 =  new String( "hello world"); | String (String str) | конструктор создающий новую строку - копию строки,  которая передана |
| str3 = "no need new" | String str 3 = "no need new"; | String (String str) | дополнительный способ (удобный) |

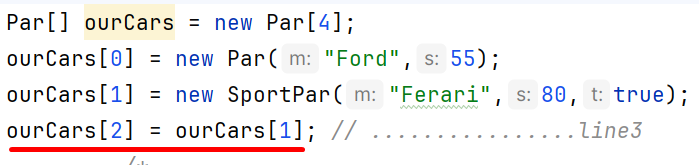
|  |  |
| --- | --- |
| תאור | השיטה |
| מחזירה את מספר התווים במחרוזת (כולל רווחים( | int **length**() |
| מחזירה את התו הנמצא במחרוזת באינדקס נתון | char **charAt**(int index) |

# **ИНКАПСУЛЯЦИЯ עקרון ההכמסה**

Инкапсуляция – כימוס

Его основная идея заключается в объединении данных (атрибутов, полей) и методов (функций), которые работают с этими данными, в единый компонент — класс. При этом детали реализации скрываются от внешнего мира, а доступ к данным предоставляется через четко определенный публичный интерфейс (public методы, такие как геттеры и сеттеры).

|  |  |
| --- | --- |
| Изображение выглядит как текст, снимок экрана, Шрифт, число  Контент, сгенерированный ИИ, может содержать ошибки. | Для line1 - прописан геттер, с помощью которого можно вынуть Manufacturer.  Для line2 - так как все поля у нас PRIVATE - оно их не подтянет.  ЭТО ИНКАПСУЛЯЦИЯ. Легко чинится этим: |
| Изображение выглядит как текст, Шрифт, линия, График  Контент, сгенерированный ИИ, может содержать ошибки. |



Строка **ourCars**[**2**] **= ourCars**[**1**] создает не новый объект, а копирует ссылку. То есть:

**ourCars**[**1**] содержит SportPar("Ferari",80,true)

После присваивания **ourCars**[**2**] и **ourCars**[**1**] указывают на ОДИН И ТОТ ЖЕ объект

Если нужно действительно создать независимую копию объекта, надо:

реализовать метод глубокого копирования

|  |  |
| --- | --- |
| public class Message { *//* внутри одного класса можно и без this. Но с this кошернее  **private** String from;  **private** String to;  **private** String content;  **private** Date messageDate;  public Message(String from, String to, String content, Date messageDate) {  this.from = from;  this.to = to;  this.content = content;  this.messageDate = new Date(messageDate); } | |
| public String toString(){  String s = "";  s += "--"+messageDate.toString()+"--"+ "\n";  s += "From:"+from+"\n";  s += "To:"+to+"\n";  s += content;  return s;} | public String toString(){  String s = "";  s += "--"+this.messageDate.toString()+"--"+ "\n";  s += "From:"+this.from+"\n";  s += "To:"+this.to+"\n";  s += this.content;  return s;} |

|  |  |
| --- | --- |
| **RETURN NEW object in GETTER** | |
| public Point getPointDown(){  return this.pointDown;  }  Возвращает прямую ссылку на внутренний объект **pointDown**. Это означает, что:   * Внешний код получит доступ к оригинальному объекту * Изменения этого объекта извне повлияют на состояние класса **Priangle** * Нарушается инкапсуляция | public Point getPointDown(){  return new Point(this.pointDown);  }  Создает и возвращает новую копию объекта **pointDown**. Это означает, что:   * Внешний код получит копию объекта * Изменения возвращенного объекта НЕ повлияют на состояние класса **Priangle** * Инкапсуляция сохраняется |

# **ПРИВЕДЕНИЕ ТИПА -** המרה מפורשת

*//* Не сработает - нужно явное приведение типа

*//* לא יעבוד - צריך המרה מפורשת

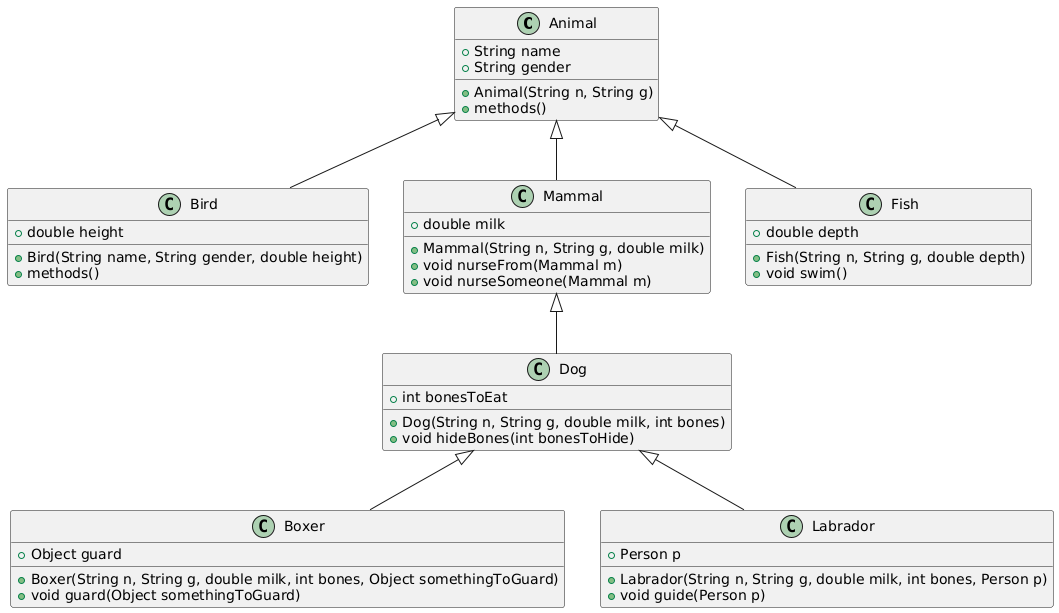
SportPar myCar = ourCars[1]; *//* ❌

*//* Правильно - явно указываем приведение к SportPar

SportPar myCar = (SportPar)ourCars[1]; *//* ✅

# **INHERITANCE - НАСЛЕДОВАНИЕ -"ירושה"**

Термины:

1. תת-מחלקה (subclass) - подкласс, дочерний класс
2. מחלקת-על (super class) - суперкласс, родительский класс
3. צאצא (descendant) - потомок

כל מחלקה יורשת - каждый класс-наследник

את כל האיברים - все элементы/члены

המופיעים במחלקת העל שלה - которые появляются в его суперклассе

אף על פי שאינם כתובים בה - даже если они не написаны в нем

ומוסיפה מאפיינים שייחודיים לה - и добавляет характеристики, уникальные для него

public class Mammal extends Animal {..} - объявление класса наследника.

public class Animal{ *//* у свойств родителя надо вместо прайват писать **PROTECTED**

protected String gender;

protected String name;

}

|  |  |
| --- | --- |
| Конструкторы НЕ наследуются в Java! | |
| *//* Конструктор родительского класса  public Animal(String name, String gender) {  this.name = name;  this.gender = gender;  } | *//* Конструктор класса-наследника  public Mammal(String name, String gender, double milk) {  super(name, gender); *//* конструктор родителя  this.milk = milk; *//* Инициализация собственных полей |

|  |  |
| --- | --- |
| Пустой супер | |
| public Dog (String name, String gender, double milk, int bones){  **super**()**;**  this.bones=bones;  } | Такую дичь можно изобразить только когда у родителей параметры задаются по умолчанию.  Преимущество - можно override -  **"ההעמסה"** перегрузкой писать свои конструкторы |

|  |  |
| --- | --- |
| **A B** | |
| public class A {  public A() {  System.out.println("A");  }  } | public class B extends A {  public B() {  System.out.println("B");  }  } |
| *//* Создание объекта класса B  B b1 = new B();  **Итоговый вывод:**  A *//* Super();  B *//* יופעל כברירת מחדל - (будет вызвано по умолчанию) | |

1. Когда вы создаете объект **B**, сначала вызывается конструктор суперкласса **A**.
2. Вызов конструктора **A** приводит к выполнению

**System.**out**.println**(**"A"**)**;** что выводит "A".

1. Затем управление возвращается в коструктор **B**, выполняется  **System.**out**.println**(**"B"**)**;** что выводит "B".

**Замечание о super**()**:**

* В данном случае **super**() вызывается автоматически, если вы не укажете его явно.
* Если в классе **A** не будет конструктора по умолчанию, то вам нужно будет явно вызывать **super**(**...**) с нужными параметрами в конструкторе **B**.

|  |  |
| --- | --- |
| **A B** | |
| class A {  public A() {  System.out.println("A");  }  public A(int i){  System.out.println("A + Param "+i);  }} | class B extends A {  public B() {  System.out.println("B");  }  public B(int i){  System.out.println("B + Param" + i);  }} |
| *//* Создание объекта класса B  B b1 = new B(); *//* A  *//* B  B b2 = new B(5); *//* A  *//* B +Param 5 | |

**Создание B b2 = new B**(**5**)**;**:

* + Вызывается конструктор **B**(int **i**), который по умолчанию также вызывает конструктор **A** (выводится "A").
  + Затем выводится "B + Param5".

|  |  |
| --- | --- |
| **A B** | |
| class A {  public A() {  System.out.println("A");  }  public A(int i){  System.out.println("A + Param "+i);  }} | class B extends A {  public B() {  System.out.println("B");  }  public B(int i){  super(i);  System.out.println("B + Param" + i); }} |
| *//* Создание объекта класса B  B b3 = new B(5); *//* A + Param 5  *//* B +Param 5 | |

|  |  |
| --- | --- |
| @Override | |
| public String toString() {  return super.toString() + "milk: " + this.milk + "\n";  } | Чтобы не дублировать родительский стринг |
| public class Animal {  public String toString() {  return "name: " + this.name + "\ngender: " + this.gender + "\n";  }}  public class Mammal extends Animal {  public String toString() {  return super.toString() + "milk: " + this.milk + "\n";  }} | Второй метод переопределит родителя и будет печатать то, что мы захотим и так, как захотим |

**Методы по умолчанию:**

public boolean **equals**(Object **obj**) {

*//* Сравнивает текущий объект с переданным

}

public String toString() {

*//* Возвращает строковое представление объекта

}

**Печать по красоте:**

System.out.print("zoo:zoo: " + animal1.toString() + mammal1.toString());

можно заменить на:

System.out.print("zoo:zoo: " + animal1 + mammal1);

1. **תת מחלקה אינה יכולה לגשת לאיברים במחלקת-על שמוגדרים private** - Подкласс не может обращаться к элементам суперкласса, которые определены как **private**.
2. **אם לתת-מחלקה עברו בירושה איברים שאינם מתאימים לה, היא יכולה לשנותם** - Если подклассу унаследованы элементы, которые ему не подходят, он может их изменить.
3. **שיטה-בונה של תת-מחלקה קוראת תחילה לשיטה-בונה של מחלקת העל שלה** - Конструктор подкласса сначала вызывает конструктор своего суперкласса.

# **פולימורפיזם - ПОЛИМОРФИЗМ** -

Возможность обращаться через определенный тип к объектам этого типа, а также к объектам его подклассов. Пример полиморфизма:

Animal myAnimal = new Animal(); *//* Создаем объект суперкласса

Animal myDog = new Dog(); *//* Создаем объект подкласса Dog

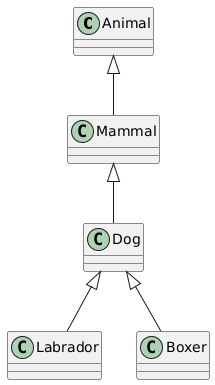
Animal myCat = new Cat(); *//* Создаем объект подкласса Cat

myAnimal.sound(); *//* Вывод: Животное издает звук

myDog.sound(); *//* Вывод: Собака лает

myCat.sound(); *//* Вывод: Кошка мяукает

**Пример с массивом:**  
Мы можем создать массив типа **Animal** и поместить в него объекты типов **Mammal** или **Dog**, так как они являются наследниками **Animal**.



public class Zoo {

public static void main(String[] args) {

Animal[] zoo = new Animal[6];

zoo[0] = new Animal("Joe", "male");

zoo[1] = new Mammal("Snow", "female", 25);

zoo[2] = new Mammal("Mitzi", "female", 30);

zoo[3] = new Animal("Bob", "male");

*//* Также можно добавить объекты подклассов

zoo[4] = new Dog("Laki", "male", 20, 4);

zoo[5] = new Boxer("Bono", "female", 25, 4, ...); }}

Object[] zoo = new Object[4]; - норм. Т.к. все классы наследуются от Object

# ПРИВЕДЕНИЕ ТИПОВ

|  |  |
| --- | --- |
| *//* **Upcasting** (преобразование вверх)  המרות כלפי מעלה  Animal myAnimal = new Dog();  Mammal mam = new Mammal("kuku", "male", 50);  Animal myAnimal = mam; *//* Upcasting | *//* **Downcasting** (преобразование вниз)  המרות כלפי מטה  Dog myDog = (Dog) myAnimal; |

|  |  |
| --- | --- |
| *//* **Upcasting** (преобразование вверх)  Mammal mam = new Mammal("kuku", "male", 50);  Animal myAnimal = mam; *//* Upcasting | **Объект myAnimal**:  Тип: **Animal**  Доступны только методы и поля, определенные в классе **Animal**. Например, метод **getMilk**() недоступен через **myAnimal**. |

Пример:

class Animal { *//* Суперкласс

void sound() {

System.out.println("Животное издает звук");

}}

class Dog extends Animal { *//* Подкласс

@Override

void sound() {

System.out.println("Собака лает");

}

void bark() {

System.out.println("Гав-гав!");

}}

public static void main(String[] args) {

*//* Upcasting (преобразование вверх)

Animal myAnimal = new Dog(); *//* Объект Dog приводится к типу Animal

myAnimal.sound(); *//* Вывод: Собака лает

*//* myAnimal.bark(); *//* Ошибка! Метод bark() недоступен через тип Animal

*//* Downcasting (преобразование вниз)

**if** (myAnimal instanceof Dog) {

Dog myDog = (Dog) myAnimal; *//* Приводим обратно к типу Dog

myDog.bark(); *//* Вывод: Гав-гав! } }}

**Безопасное приведение типов**

**Upcasting** (преобразование вверх): Безопасно, объект подкласса всегда является объектом суперкласса.  
Dog pushok = new Dog("Пушок", "male", 30);

Animal animalPushok = pushok; *//* Безопасно

**Downcasting** (преобразование вниз): Небезопасно, так как объект суперкласса не всегда является объектом подкласса.  
Animal rob = new Animal("rob", "male");

Dog dogRob = (Dog) rob; *//* Ошибка: ClassCastException

Коротко:

Upcasting — безопасно, выполняется автоматически.

Downcasting — небезопасно, требует проверки instanceof.

**Преобразование вниз** (**Downcasting**)**:**  
Пример допустимого преобразования:

Father f = new Son();

Son s = (Son) f; *//* Безопасно, если f действительно является объектом Son

Пример недопустимого преобразования:

Father f = new Father();

Son s = (Son) f; *//* Ошибка выполнения: f не является объектом Son

**Все классы в джава по умолчанию наследуют от** Object

Object[] collection = new Object[4];

collection[0] = new Integer(42); *//* Upcasting

collection[1] = new Double(4.5); *//* Upcasting

collection[2] = new Character('B'); *//* Upcasting

collection[3] = new Object(); *//* Объект

ב. ציין איזה מנגנון של תכנות מונחה *עצמים בא לידי ביטוי בהגדרת הפעולה* ()*total במחלקה Toy*

*והפעולה* ()*total במחלקה ThinkToy.*

Ответ -: это механизм полиморфизма (פולימורפיזם), потому что: У нас есть один и тот же метод total() в базовом классе и в классе-наследнике В классе-наследнике мы переопределяем (override) этот метод.

Метод имеет одинаковую сигнатуру, но разную реализацию в разных классах

התשובה: זהו מנגנון של פולימורפיזם, מהסיבות הבאות:

1. יש לנו את אותה פעולה total() במחלקת הבסיס ובמחלקה היורשת
2. במחלקה היורשת אנחנו דורסים (override) את הפעולה הזו
3. לפעולה יש אותה חתימה, אבל מימוש שונה במחלקות השונות

זהו מנגנון של פולימורפיזם - Это механизм полиморфизма

בגלל ש - потому что

יש לנו - у нас есть

את אותה פעולה - тот же самый метод

במחלקת הבסיס - в базовом классе

ובמחלקה היורשת - и в классе-наследнике

אנחנו דורסים - мы переопределяем

את הפעולה הזו - этот метод

לפעולה יש - метод имеет

אותה חתימה - такую же сигнатуру

אבל מימוש שונה - но разную реализацию

במחלקות השונות - в разных классах

*ג. האם אפשר לממש את הפעולה* ()*total במחלקה ThinkToy בעזרת הפעולה* ()*total במחלקה Toy*

*? אם כן – ממש אותה, אם לא- נמק מדוע*

Перевод ответа: "Да, можно! Есть два способа реализации:

1. С помощью super :

@Override

public double total() {

return super.total() + 10;

}

1. Или с прямым использованием полей:

@Override

public double total() {

return this.price \* this.q + 10;

}

Оба способа правильные, но использование super более рекомендуется потому что:

* Код более понятный. Если мы изменим реализацию в родительском классе, изменение отразится и в дочернем классе. Это предотвращает дублирование кода"

 @Override

    public double total(){

        return super.total()+10;

    }

# ABSTRACT

Абстрактный класс - это класс, от которого нельзя создать экземпляр напрямую. Он служит базовым классом для других классов и может содержать:

1. Абстрактные методы (без реализации) - которые должны быть реализованы в дочерних классах
2. Обычные методы с реализацией
3. Поля данных
4. Конструкторы

Если класс содержит хотя бы один абстрактный метод, класс должен быть объявлен как абстрактный.

Абстрактный класс не обязан содержать только абстрактные методы, он может также включать обычные методы.

Абстрактный класс нельзя инстанцировать напрямую (нельзя создать его экземпляр).

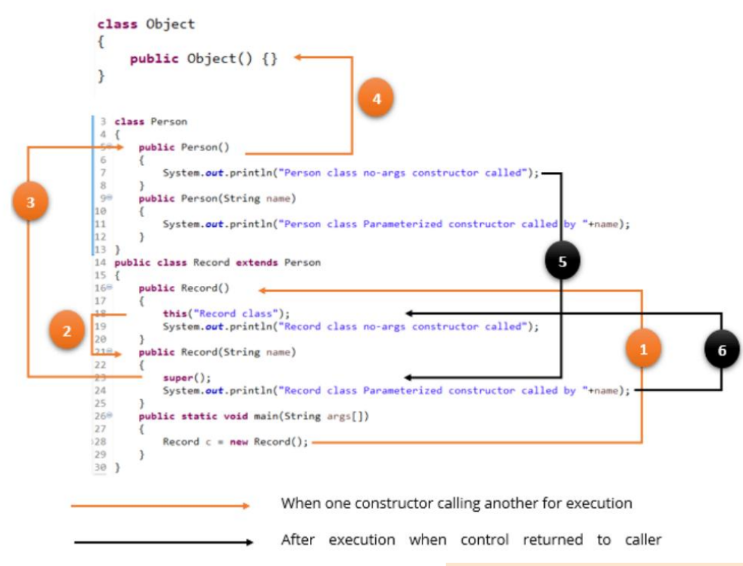
Определение класса как абстрактного гарантирует, что все подклассы будут реализовывать абстрактные методы.

Важные аспекты абстрактных классов:

1. Абстрактный метод объявляется без реализации и должен быть реализован в подклассах.
2. Необходимо правильно определять наследование от абстрактного класса:
   * Следует точно определить все абстрактные методы из родительского класса
   * Если подкласс не реализует все абстрактные методы, он тоже должен быть объявлен абстрактным
   * От этого подкласса также нельзя будет создать экземпляры
3. По сути, объявляя класс абстрактным, мы гарантируем, что в любом неабстрактном подклассе будут реализованы все необходимые абстрактные методы.

**Итоги:**

1. Абстрактный класс - это класс, представляющий абстрактную концепцию.
2. Чтобы создать абстрактный класс, необходимо добавить ключевое слово **abstract** в его объявление.
3. Нельзя создавать экземпляры абстрактных классов.
4. Абстрактный класс может содержать абстрактные методы.
5. Объявление абстрактного метода выполняется с помощью ключевого слова **abstract** в его сигнатуре.
6. Абстрактные функции - это основной инструмент абстрактных классов. Они создают общий контракт, которому должны следовать все наследники, которые не являются абстрактными.



# **INTERFACE -"ממשק"**

**Интерфейс** (**Interface**) — это **виртуальный** класс, который может содержать только абстрактные (**abstract**) методы с доступом public, и поля, которые являются static и final с доступом public. Методы, определенные в интерфейсе, не могут быть: **private**, final, static или protected.

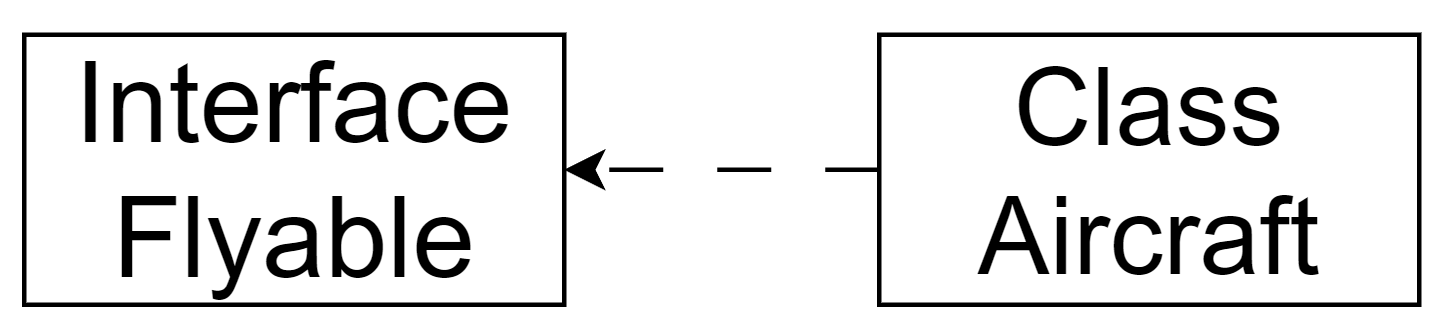
**Преимущество интерфейсов** - можно создать **множественное наследование** с помощью использования интерфейсов.

Interface היא מחלקה מדומה שיכולה להכיל אך ורק פעולות מופשטות )אבסטרקטיות **abstract** )

בעלות הרשאת הגישה public , ותכונות שהן static ו- final בעלות הרשאת גישה public.

יתרון לממשקים - ניתן ליצור הורשה מרובה באמצעות השימוש ב-Interfaces.

* Все определенные методы являются **abstract** и имеют уровень доступа public!
* Все поля являются static, final и имеют уровень доступа public



Можно создавать экземпляры интерфейсов, но кроме методов из интерфейса ничего другого вызвать не получится:

|  |  |
| --- | --- |
| public interface Flyable {  public void startFlying(); } |  |
| public **abstract** class Bird implements Flyable {  public void eat(){  System.out.println("Bird is eating");  }  public **abstract** void makeSound();  *//* **fly**() **- остается наследнику.** } |
| public class Eagle extends Bird{   @Override  publicvoid **fly**(){ System.out.println("Eagle soars high");   }  @Override  public void makeSound() {  System.out.println("Eagle screeches!");   }} |

Также класс может реализовывать несколько интерфейсов

class Bird extends Animal implements Flyable , Movable;

**Права класса, реализующего Interface**

* Если в интерфейсе определены константы, реализующий класс получает их автоматически.
* Реализация интерфейса позволяет обращаться к объектам реализующего класса, используя тип интерфейса.

|  |
| --- |
| public interface Flyable {  int *MAX\_ALTITUDE* = 10000; public void startFlying();  } |
| System.out.println(Flyable.*MAX\_ALTITUDE*); *//* 10000 |

Абстрактный класс **может** реализовывать методы интерфейса, но **не обязан** этого делать.

* Он может оставить реализацию всех или части методов интерфейса для классов, которые будут от него наследоваться.

|  |  |
| --- | --- |
| public interface Flyable {  public void fly(); } | public class Bird extends Animall implements Flyable {  public Bird(String name, int age, String type) {  super(name, age, type);  }  @Override  public void startFlying() {  System.out.println(getName()+" FLY!!!"); }} |
| public static void main(String[] args) {  Flyable fly1 = new Bird("Soroka", 1, "black");  fly1.startFlying(); *//* Soroka FLY!!! | |

INTERFACE & **INSTANCEOF**

 instanceofдля проверки, реализует ли объект определённый интерфейс (например, Flyable).

|  |  |
| --- | --- |
| Object e = new Eagle();  **if** (e instanceof Flyable){  ((Flyable)e).fly();  } | *//* Eagle soars high |

INTERFACE & **COMPARABLE**

- Интерфейс, определяющий объекты, которые можно сравнивать между собой

public interface Comparable{

*//* Возвращает true, если данный объект равен переданному объекту (в качестве пар-а)

    public boolean isEqual(Object **obj**);

*//* Возвращает true, если данный объект меньше переданного объекта (в качестве пар-а)

public boolean isLessThan(Object **obj**);

}

Пример:

public class Date implements Comparable{

Требуется приведение типа, чтобы обращаться к объекту как к Date.

**private** int day, month;

*//* .. конструктор и тд

public boolean isEqual(Object **obj**){

Date other = (Date) **obj**;

return (this.day == other.day) && (this.month == other.month);

}

public boolean isLessThan(Object **obj**){

Date other = (Date) **obj**;

**if** (this.month < other.month) return true;

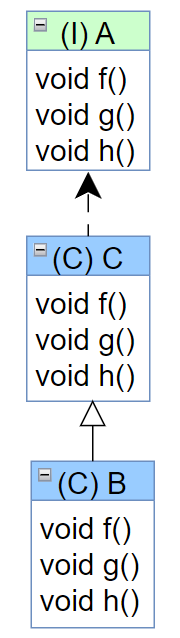
**if** (this.month > other.month) return false;

return (this.day < other.day);

}

}

Подьебки:



public interface A {

public void f();

public void g();

public void h();

}

public class C implements A {

public C(){};

public void f() {}

public void g() {}

public void h() {}

}

public class B extends C{

**private** int x;

**private** String y;

public B(int x, String y){}

public B(){}

}

public static void main(String[] args) {  
 int x; String y;  
 1. C b1 = new B(); *//* `B` является подклассом `C`, поэтому переменная типа `C` может ссылаться на объект типа `B` (полиморфизм).  
 2. A[] arrA = new A[3]; *//* массив, который может содержать ссылки на объекты, реализующие интерфейс `A`.  
 3. *//* arrA[0] = new B(x,y); ERROR: COMPILATION - x обьявлен, но не инициализирован  
 4. arrA[1] = new B(); *//* `B` реализует интерфейс `A` (через наследование от `C`).  
 5. arrA[2] = new C(); *//* `C` напрямую реализует интерфейс `A`  
 6. *//* arrA[0].f(); ERROR: NullPointerException - 1. `arrA[0]` не был проинициализирован (из-за ошибки в строке выше), поэтому по умолчанию содержит `nul`  
 arrA[2].g();  
 arrA[1].h();  
}

# **INTERFACE -- COMPARABLE -- SORTER**

**Что нужно для универсальной сортировки?**

* **Интерфейс Comparable** — реализуют классы, объекты которых можно сравнивать (например, String, Integer, Date уже реализуют Comparable).
* **Класс-сортировщик** (например, Sorter) — реализует универсальный метод сортировки для массивов объектов, реализующих Comparable.

public class Sorter {  
 static void **sort**(int[] arr){}  
 static void **sort**(double[] arr){}  
 static void **sort**(String[] arr){}  
 static void **sort**(Date[] arr){}  
}

Это позволяет разным классам вести себя одинаково с точки зрения внешнего кода, даже если они не связаны наследованием.

public interface Comparable

{

*// возвращает* true*, если объекты равны*

    public boolean isEqual(Object **obj**);

*// возвращает* true*, если текущий объект меньше переданного*

    public boolean isLessThan(Object **obj**);

}

* Такой интерфейс задаёт контракт: любой класс, который его реализует, обязан уметь сравнивать себя с другим объектом (isEqual, isLessThan).
* Это позволяет сортировать, искать, сравнивать объекты разных классов, если они реализуют этот интерфейс.

# **ENUM**

* Сокращение от enumeration (перечисление) - специальный тип данных
* Позволяет определить группу именованных констант под одним типом
* Используется для представления фиксированных значений (например: дни недели, цвета, времена года...)

**Основные свойства Enum:**

* Enum - это специальный тип данных с фиксированными значениями
* Каждый элемент **enum** является константой - public static final
* Конструкторы всегда **private** явно или неявно
* Можно добавлять конструкторы, методы и поля
* Можно использовать методы .values() для получения всех значений
* Можно сравнивать элементы через == или **equals**()
* Нельзя создавать нвоый экземпляр через **new** - שגיאת הידור

Простой пример:

|  |
| --- |
| public **enum** DaysWeek {  *SUNDAY*, *MONDAY*, *TUESDAY*, *WEDNESDAY*,  *THURSDAY*, *FRIDAY*, *SATURDAY*; |
| public static void main(String[] args){  DaysWeek today = DaysWeek.*SUNDAY*;  **if** (today == DaysWeek.*SUNDAY*) System.out.println("Good week!"); |
| switch (today) {  case *SUNDAY*:  System.out.println("Good week!");  **break**;  case *FRIDAY*:  System.out.println("Good weekend!");  **break**;  default:  System.out.println("Good day!");  }}} |

**enum** наследует Enum – поэтому есть возмоность юзать .values():

|  |
| --- |
| DaysWeek[] days = DaysWeek.*values*(); **for**(int i = 0; i< days.**length**; i++) {  System.out.println(days[i]); } *//* SUNDAY MONDAY TUESDAY WEDNESDAY… |

Пример Светофор:

|  |  |
| --- | --- |
| public **enum** TrafficLight {  *RED*("Stop"),  *GREEN*("Drive"),  *YELLOW*("Ready");  **private** final String meaning;  *//* конструктор  TrafficLight(String meaning){  this.meaning = meaning;  }  public String getMeaning(){  return meaning;  } | public static void main(String[] args) {  System.out.println(  TrafficLight.*RED*.getMeaning()  ); *//* Stop  System.out.println(  TrafficLight.*RED*  ); *//* RED  } } |

Упражнение подъебка 1:

|  |  |
| --- | --- |
| public **enum** Enums{  ONE, TWO, THREE, ONE, FOUR;  } | В енуме нельзя делать дубли. |

Упражнение подъебка 2:

|  |  |
| --- | --- |
| public class A{}  public **enum** Enums extends A{  ABC, BCD, CDE, DEF;  } | В енуме нельзя наследовать другие классы. Он наследует только Enum |

Упражнение подъебка 3:

|  |  |
| --- | --- |
| public **enum** Levels {  **private** TOP,  public MEDIUM,  protected BOTTOM;  } | В енуме нельзя ставить уровни доступа  לא ניתן להגדיר לקבועים ב **enum** הרשאת גישה! |

Упражнение подъебка 4:

|  |  |
| --- | --- |
| public **enum** Enums {  *A*, *B*, *C*;  **private** Enums() {  System.out.println(1);  }}  class Test {  public static void main(String[] args) {  Enum en = Enums.*B*;  }} | 1. **При первом обращении к enum'у** (Enums.B) все константы **enum**'а инициализируются **одновременно** 2. **Конструктор вызывается для каждой константы:**  * Создается объект A → конструктор выводит 1 * Создается объект B → конструктор выводит 1 * Создается объект C → конструктор выводит 1  1. **Результат:** 111 (три единицы подряд) |
| **בעת גישה ל Enums נטענים כל הקבועים לזכרון ומופעלת הפעולה הבונה לכולם**  **Ключевая особенность enum:**   * Все константы **enum**'а создаются **при первом обращении к любой из них** * Каждая константа - это отдельный объект, для которого вызывается конструктор * Порядок создания: слева направо (A, потом B, потом C) * Enum константы - это **синглтоны** (создаются только один раз) * Если бы потом снова обратились к Enums.A или Enums.C, ничего бы не вывелось, так как объекты уже созданы. | |

Упражнение подъебка 5:

|  |  |
| --- | --- |
| public **enum** Directions {  *NORTH*, *SOUTH*, *WEST*, *EAST*;  **private** Directions(){  System.out.println(1);  }} public class Test{  public static void main(String[] args){  Directions d1 = new Directions(); }} | שגיאת הידור  לא ניתן ליצור מופע מ  **enum** |

Упражнение подъебка 6:

|  |  |
| --- | --- |
| public **enum** MyEnums {  *FIRST*, *SECOND*, *THIRD*, *FOURTH*; } class Test3{  public static void main(String[] args){  MyEnums[] myEnums = new MyEnums[4];  **for** (int i=0; i< myEnums.**length**; i++){  System.out.println(myEnums[i]);  }}} | null  null  null  null |

Упражнение подъебка 7:

|  |  |
| --- | --- |
| public **enum** Levels {  *TOP*, *MEDIUM*(10), *BOTTOM*(20,30);  int i, j;  **private** Levels(){}  **private** Levels(int i){  this.i = i;  this.j = j;  }  **private** Levels(int i, int j){  this.i = i;  this.j = j;  } | public static void main(String[] args){  System.out.println(Levels.*TOP*.i); *//* 0  System.out.println(Levels.*TOP*.j); *//* 0  System.out.println(Levels.*MEDIUM*.i);*//* 10  System.out.println(Levels.*MEDIUM*.j);*//* 0  System.out.println(Levels.*BOTTOM*.i);*//* 20  System.out.println(Levels.*BOTTOM*.j);*//* 30  } } |

Упражнение подъебка 8:

|  |  |
| --- | --- |
| public **enum** ABC {  *ABC*;  public int i = 1;  {  i++;  }  {  ++i;   }  **private** ABC() {  i = i++ + ++i;  } | *//* i=1  *//* i было 1, становится 2  *//* i было 2, становится 3  *//* СЛОЖНОЕ ВЫРАЖЕНИЕ!  Перед выполнением: i = 3  i++ - возвращает текущее значение (3), потом увеличивает i до 4  ++i - сначала увеличивает i с 4 до 5, потом возвращает 5  Сложение: 3 + 5 = 8  Присваивание: i = 8 |
| class Test4{  public static void main(String[] args){  System.out.println(ABC.*ABC*.i); *//* 8  } | |

**Ключевые моменты:**

* **Поля инициализируются первыми**
* **Блоки инициализации выполняются по порядку**
* **Конструктор выполняется последним**
* **i++ vs ++i** - разница в том, когда возвращается значение (до или после инкремента)

Пример с сортировкой:

# **Исключения -- Exceptions – חריגות**

**Compilation Error**, ArithmeticException:

|  |  |
| --- | --- |
| public static void main(String[] args) {  int j;  **for** (int i=0; i<6; i++){  j = j/i;  } | 1. переменная j – не инициализированна  "The local variable j may not have been initialized"  - שגיא חידור **Compilation Error**  2. даже если поправят - на ноль делить нельзя  ArithmeticException - **Runtime Exception**  **שגיאת זמן ריצה** |

NullPointerException

|  |  |
| --- | --- |
| public class t2\_group { static class Group{  **private** String[] groupMembers;  **private** int numOfMembers;  public Group(int numOfMembers, String[] groupMembers){…}  public int getNumOfMembers(){ return numOfMembers; }  public void addMember(String newMember){}  public void removeMember(int numOfTheMemberToRemove){  numOfMembers--;  groupMembers[numOfTheMemberToRemove] = null;  } } | |
| public static void main(String[] args) {  String[] grpMembers = {"Dan", "Din", "Dina"};  Group dansGroup = new Group(3, grpMembers);  dansGroup.**remove**(3);  dansGroup.removeMember(3);  } } | .**remove**(3); - метода **remove** не существует. **Compilation Error**  В этом массиве нет такого итема. есть 0, 1, 2 .NullPointerException  **שגיאת זמן ריצה** |

Чинится дополнительной проверкой:

|  |
| --- |
| public void removeMember(int numOfTheMemberToRemove){  **if** (numOfTheMemberToRemove < groupMembers.**length**){  numOfMembers--;  groupMembers[numOfTheMemberToRemove] = null;  } **else** { System.out.println("Нету индекса!"); }} |

В идеале так:

|  |
| --- |
| public void removeMember(int numOfTheMemberToRemove) throws Exception {  **if** (numOfTheMemberToRemove >= groupMembers.**length**)  throw new Exception  (“Never try to **remove** a member ” +  “that his number is >= the number” +  “ **of** members in the group” +  “(“ + groupMembers.**length** + “)”);  numOfMembers--;  groupMembers[numOfTheMemberToRemove] = null;  } |

|  |
| --- |
| *// Хотим удалить первых четырех участников*  try{  **for** (int i = 0; i < 4; i++)          dansGroup.removeMember(i);  }  catch(Exception *e*){      System.out.println("removeMember() failed to **remove**" +                        "the " + i + " member");  } |

try, catch и также finally

**try**  -  Фрагмент кода, который мы пытаемся выполнить (и в котором может быть выброшено исключение)

**catch** -  Фрагмент кода, обрабатывающий исключение, которое было поймано

**finally** -  Фрагмент кода, который выполняется в любом случае, независимо от того, было ли выброшено исключение или нет

try {

*// код, который может выбросить исключение*

} catch (ExceptionType *e*) {

*// обработка исключения*

} finally {

*// код, выполняющийся всегда*

}

|  |
| --- |
| public static int div(int a, int b){  try {  a = a/b;  }  catch (Exception e){  System.out.println("there is a problem!");  }  finally {  System.out.println("print anyway");  return a;  } } |
| public static void main(String[] args) {  *div*(5,2); *//*print anyway  System.out.println(*div*(8,2)); *//* print anyway 4  System.out.println(*div*(2,0)); }} *//* there is a problem! print anyway 2 |

זריקת חריגות – throw -

**Кто выбрасывает исключения?** Сама Java-среда, Программист

**Как программист выбрасывает исключения сам?**

throw new Exception() или throw new Exception("some message")

Exception e = new Exception();

... throw e;

|  |
| --- |
| public class t4\_div2 {  *//* Метод, который делит на 2. Выбрасывает исключение, если число нечётное  public int divideBy2(int n) throws Exception{  **if** ((n%2) != 0)  throw new Exception("The number is odd");  return n/2;  } |
| public static void main(String[] args) {  try {  int halfN;  t4\_div2 myObj = new t4\_div2();  **for** (int i=2; i<10; i++){  halfN = myObj.divideBy2(i);  System.out.println("halfN = " + halfN);  } }  catch (Exception e){  System.out.println("Ошибка: " + e.getMessage()); } }} |
| Выведет:  halfN = 1  Ошибка: The number is odd |

Когда внутри появляется ошибка – выполнение цикла останавливается

|  |
| --- |
| **for** (int i=2; i<10; i++){      try {          int halfN = myObj.divideBy2(i);          System.out.println("halfN = " + halfN);      }      catch (Exception *e*){          System.out.println("Ошибка для числа " + i + ": " + e.getMessage());      }  } |
| Выведет:  halfN = 1  Ошибка: The number is odd  halfN = 1  Ошибка для числа 3: The number is odd  halfN = 2 … |

Пример Калькулятор – обработка отрицательных значений

|  |  |
| --- | --- |
| public class Calculator {  *//* Проверка валидности числа  **private** void validateNumber(int number) throws Exception {  **if** (number < 0) {  throw new Exception("Ошибка: Отрицательные числа запрещены!");  }  **if** (number > 100) {  throw new Exception("Ошибка: Числа больше 100 запрещены!"); } } | |
| *//* Вычитание  public int subtract(int a, int b) throws Exception {  validateNumber(a);  validateNumber(b);  int result = a - b;  validateNumber(result);  return result;  } | *//* Деление  public int divide(int a, int b) throws Exception {  validateNumber(a);  validateNumber(b);  **if** (b == 0) {  throw new Exception("Ошибка: Деление на ноль запрещено!");  }  int result = a / b;  validateNumber(result);  return result;  }} |
| public static void main(String[] args) {  Calculator calc = new Calculator();  *//* Тест вычитания с отрицательным результатом  try {  int result = calc.subtract(10, 20);  System.out.println("10 - 20 = " + result);  } catch (Exception e) {  System.out.println("Вычитание: " + e.getMessage());  }  System.out.println("qweqwe"); } | |
| Выведет:  Вычитание: Ошибка: Отрицательные числа запрещены!  qweqwe | |

**Бросание исключения** (**חריגה היא עצם**)

**throw new Exception**()

* Декларация о бросании исключения в определённой ситуации
* Различные типы исключений:
* IOException, FileNotFoundException
* IndexOutOfBoundsException, NullPointerException (RuntimeException)
* Все исключения наследуются от класса Exception
* Можно писать собственные классы исключений

Упражнение подъебка

|  |  |
| --- | --- |
| public class Example {  public void printOnlyIfItIs5(int number){  **if** (number != 5)  throw new Exception(); *//* ← ПРОБЛЕМА ЗДЕСЬ  System.out.println("It is 5!");  } } | |
| public void printOnly5(int number){  **if** (number!=5)  throw new RuntimeException();   System.out.println("It is 5!"); } | public void print5(int number) throws Exception{  **if** (number!=5)  throw new Exception();  System.out.println("It is 5!"); }} |
| public static void main(String[] args) {  *printOnly5*(5); *//* It is 5!  *printOnly5*(50); *//* .RuntimeException (выведет после 5ки)  } | |

**Основные методы в классе Exception**

1. **Конструктор без параметров** Exception()
2. **Конструктор, принимающий строку как параметр** Exception(String s)
3. **getMessage**() - возвращает сообщение об ошибке
4. **printStackTrace**() - выводит трассировку стека (где произошла ошибка)

**Эти методы наследуются всеми классами, представляющими исключения**

Разные исключение можно ловить по разному. Также их может быть несколько.

public void f() throws AException, BException{...}

try{...}

catch (Exception e)

catch(IOException ioe){...}

catch(NullPointerException npe){...}

catch(FileNotFoundException fnfe){...}

|  |  |
| --- | --- |
| Дерево наследования исключений:  java.lang.Object  |  +-java.lang.Throwable  |  +-java.lang.Exception  |  +-java.lang.RuntimeException  |  +-java.lang.IndexOutOfBoundsException | Пример:  try{  *//*code to try with possibly throw GrandFatherException, FatherException, or SonException  }  catch (SonException se) { ...}  catch (FatherException fe) { ...}  catch (GrandFatherException gfe) { ...} |

**Различные типы исключений в наследованииПравило:** Если метод выбрасывает различные типы исключений, между которыми есть отношения наследования, объявление родительского класса означает объявление также всех его потомков.**Пример:**Если метод выбрасывает одно из исключений:

* GrandFatherException (дедушка)
* FatherException (отец)
* SonException (сын)

То **достаточно объявить только** GrandFatherException:

public void f() throws GrandFatherException {

*//* Можно выбрасывать любое из наследованных исключений

throw new FatherException();

*//* или

throw new SonException();

*//* или

throw new GrandFatherException();

}

**Объяснение:**

* FatherException extends GrandFatherException
* SonException extends FatherException
* Поэтому объявив throws GrandFatherException, мы автоматически покрываем все дочерние исключения

**Преимущество:** Упрощает сигнатуру метода и делает код более читаемым, когда есть иерархия исключений.

Если когда ловим исключение типа Exception, ловятся все исключения (так как все наследуются от него), почему бы не использовать всегда catch (Exception e)? **Ответ:** Просто, потому что часто мы хотим разную обработку для разных типов исключений.

|  |
| --- |
| try {  *// код*  } catch (FileNotFoundException *e*) {      System.out.println("Файл не найден!");  } catch (IOException *e*) {      System.out.println("Ошибка ввода-вывода!");  } catch (Exception *e*) {      System.out.println("Неизвестная ошибка!");  } |

**Создание пользовательского исключения MyException**

|  |
| --- |
| public class MyException extends Exception{  public MyException(){  super("This is my Exception");  }  public MyException(String msg){  super(msg);}} |

# \_0\_ начало:

## Clock

public class Clock {  
 **private** int hour;  
 **private** int minutes;  
 public Clock(int hour, int minutes){  
 this.hour = hour;  
 this.minutes = minutes;  
 }  
  
 public int getHour() { return hour; }  
 public void setHour(int hour) { this.hour = hour; }  
  
 public int getMinutes() { return minutes; }  
  
 public void setMinutes(int minutes) { this.minutes = minutes; }  
  
 public boolean assertHour(Clock c1, Clock c2){  
 return c1.getHour() == c2.getHour();  
 }  
  
 public int pereshMinutes(Clock c1, Clock c2){  
 return Math.abs(c1.getHour()\*60 + c1.getMinutes() - c2.getHour()\*60 - c2.getMinutes());  
 }  
  
 public String toString(){  
 return "<" + hour + ":" + minutes + ">";  
 }  
  
 public static void main(String[] args) {  
 Clock c1 = new Clock(12, 30);  
 Clock c2 = new Clock(12, 45);  
 Clock c3 = new Clock(14, 20);  
  
 System.out.println(c1.assertHour(c1, c2));  
 System.out.println(c1.assertHour(c1, c3));  
  
 System.out.println(c1.pereshMinutes(c1, c2));  
 System.out.println(c1.toString());  
 System.out.println(c2.toString());  
 }  
  
 }

## TRIANGLE

public class Triangle {  
 **private** double a, b, c;  
  
 public Triangle(double a, double b, double c) {  
 this.a = a;  
 this.b = b;  
 this.c = c;  
 }  
  
 public double getA() {  
 return a;  
 }  
 public void setA(double a) {  
 this.a = a;  
 }  
 public double getB() { return b; }  
 public void setB(double b) { this.b = b; }  
 public double getC() { return c; }  
 public void setC(double c) { this.c = c; }  
  
 public boolean isEquilateral() {  
 return a == b && c == a && b == c;  
 }  
  
 public double getPerimeter() {  
 return a+b+c;  
 }  
  
 public String toString(){  
 return "A=" + a + " B=" + b + " C=" + c +  
 (isEquilateral() ? "\nравносторонний" : "\nне равносторонний");  
 }  
  
 public static void main(String[] args) {  
 Triangle t1 = new Triangle(12, 30, 10);  
 System.out.println(t1.toString());  
 }  
 }

\_1\_ overloading

Bona maatika / (deep copy). DATE  
public class Date {  
 **private** int day;  
 **private** int month;  
 **private** int year;  
  
 public Date(int day, int month, int year){  
 this.day = day;  
 this.month = month;  
 this.year = year;  
 }  
  
 *//* BonaMaatika  
 public Date(Date other){  
 this.year = other.year;  
 this.month = other.month;  
 this.day = other.day;  
 }  
  
 *//* Геттеры  
 public int getDay() { return day; }  
 public int getMonth() { return month; }  
 public int getYear() { return year; }  
  
 *//* Сеттеры  
 public void setDay(int day) { this.day = day; }  
 public void setMonth(int month) { this.month = month; }  
 public void setYear(int year) { this.year = year; }  
  
  
 public boolean isTheSame(Date other) {  
 return this.day == other.day &&   
 this.month == other.month &&   
 this.year == other.year;  
 }   
   
 @Override  
 public String toString(){  
 return day + "/" + month + "/" + year;  
 }  
}

## Bona maatika / (DEEP COPY). POINT

public class Point {  
 **private** int x;  
 **private** int y;  
 public Point(int x, int y){  
 this.x = x;  
 this.y = y;  
 }  
  
 public int getX() { return x; }  
 public int getY() { return y; }  
 public void setX(int x) { this.x = x; }  
 public void setY(int y) { this.y = y; }

*//* Shita Bona Maatika  
 public Point(Point other){  
 this.x = other.x;  
 this.y = other.y;  
 }  
  
 public double distance(Point other) {  
 double dx = this.getX() - other.getX();  
 double dy = this.getY() - other.getY();  
 return Math.sqrt(dx \* dx + dy \* dy);  
 }  
  
 public String toString(){  
 return "<"+this.x+":"+this.y+">";  
 }  
  
  
  
 public static void main(String[] args) {  
 Point p1 = new Point(4,9);  
 Point p2 = p1;  
 Point p3 = new Point(p1);  
 p2.setX(5);  
 p2.setY(8);  
 System.out.println(p1); *//* <5:8>  
 System.out.println(p2.toString()); *//* <5:8>  
 System.out.println(p3.toString()); *//* <4:9>  
 *//* p1 ו-p2 מצביעים על אותו אובייקט (השמת הפניה)  
 }  
}

# \_2\_ MURKAVIM

## Сложные отношения (Композиция)

Вам нужно реализовать следующие классы:

* Passport - представляет паспорт путешественника
* Traveler - представляет обычного путешественника

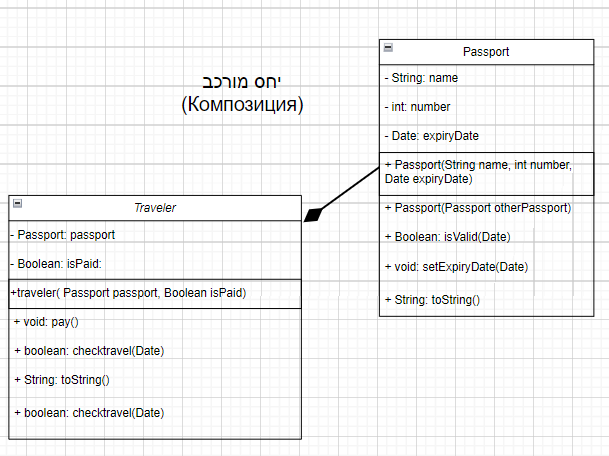
Для реализации этих классов используйте класс Date, который вы написали ранее. Вам нужно добавить к нему копирующий конструктор.

Отношения и связи между классами показаны на следующем рисунке:

Напоминание: знак a → b представляет отношение композиции (a состоит из b, то есть: как минимум одно из свойств a имеет тип b)

*Примечание: композиция - это более сильная форма агрегации, где компонент* (*часть*) *не может существовать без контейнера* (*целого*)*. В данном случае паспорт является частью путешественника и не может существовать отдельно от него в контексте этой системы.*

## יחס מורכב - композиция



class Passport{  
 **private** String name;  
 **private** int number;  
 **private** Date expiryDate;  
 public Passport(String name, int number, Date expiryDate){  
 this.name = name;  
 this.number = number;  
 this.expiryDate = new Date(expiryDate);  
 }  
 public boolean isValid(Date dateChecked){  
 return dateChecked.getYear() < expiryDate.getYear() ||   
 (dateChecked.getYear() == expiryDate.getYear() && (dateChecked.getMonth() < expiryDate.getMonth() ||   
 (dateChecked.getMonth() == expiryDate.getMonth() && dateChecked.getDay() <= expiryDate.getDay()))); }  
 public void setExpiryDate(Date newExpiryDate){  
 *//* установка даты (глубокое копирование)  
 this.expiryDate = new Date(newExpiryDate);  
 }  
 Passport(Passport passport){  
 *//* метод глубокого копирования  
 this.name = passport.name;  
 this.number = passport.number;  
 this.expiryDate=new Date(passport.expiryDate);  
 }  
  
 @Override  
 public String toString(){  
 return ("Name:" +name + "\nPass. Num: " + number + "\nExp Date: " + expiryDate);  
 }}  
  
class Traveler{  
 **private** Passport passport;  
 **private** Boolean isPaid;  
 public Traveler(Passport passport, Boolean isPaid){  
 this.passport = passport;  
 this.isPaid = isPaid;  
 }  
 void pay(){ this.isPaid = true; }  
 boolean isPaid(){ return isPaid; }  
 boolean checkTravel(Date travelDate){  
 *//* возвращает ТРУ если дата валидная и билеты проплатили  
 return passport.isValid(travelDate) && isPaid(); }  
 @Override  
 public String toString(){  
 return passport.toString() + " \nPaid **for** ticket - " + (isPaid ? "Yes":"No"); }}  
public class T1\_SummarySemestrA {  
 public static void main(String[] args) {  
 *//* Создаем дату  
 Date expiryDate1 = new Date(1, 1, 2024); *//* срок действия паспорта  
 Date travelDate = new Date(1, 1, 2023); *//* дата путешествия  
  
 *//* Создаем паспорт  
 Passport passport1 = new Passport("Leonid", 336540331, expiryDate1);  
  
 *//* Создаем путешественника (без оплаты)  
 Traveler trav1 = new Traveler(passport1, false);

*//* Проверяем возможность путешествия до оплаты  
 System.out.println("Информация о путешественнике:");  
 System.out.println(trav1.toString());  
 System.out.println("\nМожно путешествовать 1.1.2023 (до оплаты): " +   
 trav1.checkTravel(travelDate));  
 trav1.pay(); *//* Оплачиваем и проверяем снова

System.out.println("\nМожно путешествовать 1.1.2023 (после оплаты): " +   
 trav1.checkTravel(travelDate)); } }

## MESSAGE

public class Message {  
 **private** String from;  
 **private** String to;  
 **private** String content;  
 **private** Date messageDate;  
 public Message(String from, String to, String content, Date messageDate) {  
 this.from = from;  
 this.to = to;  
 this.content = content;  
 this.messageDate = new Date(messageDate);  
 }  
 public Date getMessageDate(){  
 return new Date(this.messageDate);  
 }  
 public void setMessageDate(Date messageDate){  
 this.messageDate.setDay(messageDate.getDay());  
 this.messageDate.setMonth(messageDate.getMonth());  
 this.messageDate.setYear(messageDate.getYear());  
 }  
 public String toString(){  
 String s = "";  
 s += "--"+this.messageDate.toString()+"--"+ "\n";  
 s += "From:"+this.from+"\n";  
 s += "To:"+this.to+"\n";  
 s += this.content;  
 return s;  
 }  
 public void reply(String rep, Date d){  
 String temp = this.from;  
 this.from = this.to;  
 this.to = temp;  
 this.setMessageDate(d);  
 this.content = rep + "\n->\n"+this.content;  
 }  
 public void forward(String to, Date d){  
 this.from = this.to;  
 this.to = to;  
 this.setMessageDate(d);  
 this.content = "forward>>"+this.content;  
 }  
 public static void main(String[] args) {  
 Date d = new Date(25,3,2021);  
 Message m = new Message("Gidi", "Meir", "Hi! How are you?", d);  
 System.out.println(m);  
 d.setDay(30);  
*//* m.reply("Все хорошо!", d);  
*//* System.out.println(m);  
 m.forward("Yoni", d);  
 System.out.println(m);  
 }}

--30/3/2021--

From:Meir

To:Yoni

forward>>Hi! How are you?

--30/3/2021--

From:Meir

To:Gidi

Все хорошо! ->

->

--25/3/2021--

From:Gidi

To:Meir

Hi! How are you?

## PRIANGLE

public class Priangle {  
  
 **private** Point pointUp;  
 **private** Point pointDown;  
 public Priangle(Point pointUp, Point pointDown){  
 this.pointUp=new Point(pointUp);  
 this.pointDown=new Point(pointDown);  
 }  
 public Point getPointUp(){  
 return new Point(this.pointUp);  
 }  
  
 public Point getPointDown(){  
 return new Point(this.pointDown);  
 }  
  
 public Point getThirdPoint(){  
 int x = this.pointUp.getX();  
 int y = this.pointDown.getY();  
 return new Point(x,y);  
 }  
  
 public double areaOfpriangle(){  
 double base1=0, base2=0;  
 Point p = this.getThirdPoint();  
 base1 = this.pointUp.distance(p);  
 base2 = this.pointDown.distance(p);  
 return (base1\*base2)/2;  
 }  
  
  
 public static void main(String[] args) {  
  
 }  
}

## Student

public class Student {  
 **private** String name; *//* имя студента  
 **private** String id; *//* ID студента  
 **private** char gender; *//* пол (M/F)  
 **private** Date birthDay; *//* дата рождения  
  
 public Student(String name, String id, char gender, Date birthDay) {  
 this.name = name;  
 this.id = id;  
 this.gender = gender;  
 this.birthDay = new Date(birthDay); *//* создаем копию даты  
 }  
  
 public String getId() { return id; }  
 public char getGender() { return gender; }  
  
 /\* Метод для получения даты рождения \* @return дата рождения \*/  
 public Date getBirthDay() {  
 return new Date(birthDay); *//* возвращаем копию даты  
 }  
 /\* Метод для проверки одинаковой даты рождения  
 \* @param otherStudent - другой студент для сравнения  
 \* @return true если даты рождения совпадают, false если нет  
 \*/  
 public boolean twinBirth(Student otherStudent) {  
 return this.birthDay.isTheSame(otherStudent.birthDay);  
 }  
  
 public String toString() {  
 return "Студент: " + name +  
 ", ID: " + id +  
 ", Пол: " + gender +  
 ", Дата рождения: " + birthDay;  
 }  
  
 public static void main(String[] args) {  
 *//* 1. Создаем трех студентов согласно таблице  
 Student student1 = new Student("Tal", "111", 'F', new Date(15, 9, 2004));  
 Student student2 = new Student("Ron", "222", 'M', new Date(3, 3, 2005));  
 Student student3 = new Student("Ben", "333", 'M', new Date(15, 9, 2004));  
  
 *//* 2. Выводим информацию о студентах  
 System.out.println(student1);  
 System.out.println(student2);  
 System.out.println(student3);  
  
 *//* 3. Проверяем, родились ли student1 и student3 в один день  
 **if** (student1.twinBirth(student3)) {  
 System.out.println(student1.getId() + " and " + student3.getId() + " are twin birth!");  
 } }}

## TABLE

public class Table {  
 **private** int line;  
 **private** char tur;  
 public Table(int line, char tur){  
 **if** (line < 1 || line > 7) {  
 throw new IllegalArgumentException("Строка должна быть от 1 до 7");  
 }  
 **if** (tur < 'A' || tur > 'E') {  
 throw new IllegalArgumentException("Столбец должен быть от A до E");  
 }  
 this.line = line;  
 this.tur = tur;  
 }  
 public int getLine() { return line; }  
 public char getTur() { return tur; }

## KITA

class Kita{  
 **private** int count;  
 **private** Table[] t;  
 public Kita(int count, Table[] t){  
 **if** (count > 35) {  
 throw new IllegalArgumentException("В классе не может быть больше 35 столов");  
 }  
 this.count = count;  
 this.t = t;  
 }  
 public int howMany(){  
 return count;  
 }  
 public boolean isOk(int n){  
 return n<=35;  
 }  
 public void addTable(Table t){  
 *//* Добавляет новый стол в массив t  
 *//* 1. Создаем новый массив на 1 больше текущего  
 Table[] newT = new Table[count + 1];  
 *//* 2. Копируем все старые столы  
 **for**(int i = 0; i < count; i++) {  
 newT[i] = this.t[i];  
 }  
 *//* 3. Добавляем новый стол  
 newT[count] = t;  
 *//* 4. Обновляем ссылку и счетчик  
 this.t = newT;  
 count++;  
 }  
}

## SCHOOL

class School {  
 **private** int count; *//* количество классов  
 **private** Kita[] kitot; *//* массив классов  
  
 *//* б. конструктор  
 public School(int maxKitot) {  
 this.kitot = new Kita[maxKitot]; *//* a. инициализация массива  
 this.count = 0; *//* b. инициализация счетчика  
 }  
  
 *//* в. получить класс по индексу  
 public Kita getKita(int i) {  
 return kitot[i];  
 }  
  
 *//* г. добавить класс в школу  
 public void addClass(Kita k) {  
 **if** (count < kitot.**length**) {  
 kitot[count] = k;  
 count++;  
 } **else** {  
 throw new IllegalStateException("Школа заполнена");  
 }  
 }  
  
 *//* д. подсчет всех столов в школе  
 public int getTotalTables() {  
 int total = 0;  
 **for** (int i = 0; i < count; i++) {  
 total += kitot[i].howMany();  
 }  
 return total;  
 }  
  
 *//* е. найти класс для 42 учеников (нужно минимум 21 стол)  
 public int findClassFor42() {  
 **for** (int i = 0; i < count; i++) {  
 **if** (kitot[i].howMany() >= 21) { *//* 42/2 = 21 стол минимум  
 return i;  
 }  
 }  
 return -1; *//* если не найден подходящий класс  
 }  
}

\_3\_ STATIC

МУЗЫКАЛЬНЫЕ ДИСКИ  
 **private** String name;  
 **private** int myAmount;  
 **private** static int classDiscBox = 0;  
 public static final int INITIAL\_AMOUNTS\_OF\_DISCS = 3;  
 public ClassMember(String name)  
 {  
 this.name = name;  
 this.myAmount = INITIAL\_AMOUNTS\_OF\_DISCS;  
 classDiscBox += INITIAL\_AMOUNTS\_OF\_DISCS;  
 }  
 public String getName()  
 {  
 return this.name;  
 }  
 public int getStudentAmount()  
 {  
 return this.myAmount;  
 }  
  
 public static int getClassDiscBox()  
 {  
 return ClassMember.classDiscBox;  
 }  
 public void enterDiscs (int amount)  
 {  
 ClassMember.classDiscBox += amount;  
 this.myAmount += amount;  
 }  
  
 public static void main(String[] args) {  
 ClassMember[] members = new ClassMember[3];  
 members[0] = new ClassMember("Moshe"); *//* 3  
 members[1] = new ClassMember("Dvir"); *//* 6  
 members[2] = new ClassMember("Michal"); *//* 9  
 **for** (int i=0; i<members.**length**; i++)  
 {  
 members[i].enterDiscs(i);  
 }  
 System.out.println(members[2].getStudentAmount()); *//* 3+2=5  
 System.out.println(members[2].getClassDiscBox()); *//* 9+1+2=12  
 }

\_3\_ STATIC COUNTERpublic class Counter {  
 public static int count=0;  
  
 /\* Конструктор класса Counter  
 \* Принцип работы:  
 \* 1. При создании каждого нового объекта  
 \* 2. Увеличивает счетчик на 1  
 \*/  
 public Counter() {  
 count++;  
 }  
  
 /\* Метод для получения количества созданных объектов  
 \* @return количество созданных объектов  
 \* Метод статический, так как работает со статической переменной  
 \*/  
 public static int getCount() {  
 return count;  
 }  
  
  
 public static void main(String[] args) {  
 *//* Генерируем случайное число от 1 до 50  
 int randomCount = (int)(Math.random() \* 50) + 1;  
   
 *//* Создаем случайное количество объектов Counter  
 **for**(int i = 0; i < randomCount; i++) {  
 new Counter();  
 }  
   
 *//* Выводим количество созданных объектов  
 *//* прикол в том, чтобы обращатся напрямую к классу. т.к. это статик  
 System.out.println("Создано объектов: " + Counter.getCount());  
 }  
}

\_3\_ STATIC KIDSpublic class Kids {  
 **private** String name; *//* имя ребенка  
 **private** int age; *//* возраст ребенка  
 **private** int place; *//* место ребенка в семье  
 **private** static int *numChildren* = 0; *//* количество детей  
  
 /\* Конструктор класса Kids  
 \* Принцип работы:  
 \* 1. Инициализирует поля объекта  
 \* 2. Увеличивает счетчик детей  
 \*/  
 public Kids(String name, int age, int place) {  
 this.name = name;  
 this.age = age;  
 this.place = place;  
 *numChildren*++;  
 }  
 /\* Метод для получения количества детей  
 \* @return количество детей  
 \*/  
 public static int getNumChildren() {  
 return *numChildren*;  
 }  
  
 /\* Метод для строкового представления ребенка  
 \* @return строка с информацией о ребенке  
 \*/  
 public String toString() {  
 return name + " - возраст: " + age + ", место: " + place;  
 }  
  
 public static void main(String[] args) {  
 *//* Создаем трех детей семьи Коэн  
 Kids tal = new Kids("Таль", 16, 1);  
 Kids gil = new Kids("Гиль", 13, 2);  
 Kids dina = new Kids("Дина", 11, 3);  
  
 *//* Выводим информацию о каждом ребенке  
 System.out.println(tal); System.out.println(gil); System.out.println(dina);  
   
 *//* Выводим общее количество детей  
 System.out.println("Всего детей: " + Kids.*getNumChildren*());  
 }  
}

\_3\_ STATIC **LINE**

public class Line {  
 **private** Point p1;  
 **private** Point p2;  
 public static int count = 0;  
 *//* constructor  
 public Line(){};  
 public Line(int x1, int y1, int x2, int y2){  
 this.p1 = new Point(x1, y1);  
 this.p2 = new Point(x2, y2);  
 count++;  
 }  
 прикол в том, что оно и без гетера вернет все по вызову.  
*//* public static int getCount() {   
*//* return count;  
*//* }  
  
 public static void main(String[] args) {  
 Line line1 = new Line(100,100,50,150);  
 Line line2 = new Line(100,100,150,150);  
 System.out.println(count); *//* 2  
 System.out.println("No. **of** lines = "+ Line.count); *//* 2  
 Line ln1 = new Line();  
 System.out.println("No. **of** lines = "+ ln1.count); *//* 2  
 }  
}

\_3\_ STATIC **MyClass**

public class MyClass {  
 **private** int num;  
 **private** String name;  
 public static int *found*;  
 public MyClass(int n, String name) {  
 this.num = n;  
 this.name = name;  
 }  
 public void setNum(int n) { this.num = n; }  
 public int getNum() { return num; }

public static void print() {

System.out.println("Printing");

}  
  
 public void print(String msg) {  
 System.out.println(msg);  
 }  
  
 public static void main(String[] args) {  
 MyClass a = new MyClass(1, "first"); *//* Конструктор с int и String  
 a.setNum(20); *//* Метод установки числа  
 a.print("Printing"); *//* Метод print -- Printing  
 MyClass.print(); *//* Статический метод print -- Printing  
 System.out.println(a.name + " " + a.getNum()); *//* Геттер для num -- first 20  
 System.out.println(MyClass.*found*); *//* Статическое поле found  
 }  
}

\_3\_ STATIC TestRunner

public class TestRunner {  
  
 static public int succeeded;  
 static public int failed;  
 public TestRunner() {  
 }  
 public TestRunner(int s, int f) {  
 succeeded=s; failed=f;  
 }  
 public String toString() {  
 return succeeded + " " + failed;  
 }  
 public static void main(String[] args) {  
 TestRunner tr = new TestRunner(1, 99);  
 TestRunner.succeeded = 99;  
 System.out.print(new TestRunner()); *//* 99 99  
 }  
}

\_3\_ STATIC

Вопрос - что оно скомпилирует из этих 4 вариантов?

A) 1000 1000 999 999 B) 999 998 999 999 C) 999 998 0 0 **D**) **Программа не скомпилируется** E) 999 999 999 998

Тут подъебка в том, что в конструкторе this на static ставить нельзя. А в методе можно (в Car меняли цвет).

Иначе было бы 999 998 999 999

|  |  |
| --- | --- |
| class AccountA{  **private** int accountNumber = 0;  **private** static int *currentAccountNumber*=1000;  public AccountA(){  this**.*currentAccountNumber*--;** XXXX  this.accountNumber = *currentAccountNumber*;  }  public int getNum(){  return this.accountNumber;  } } | class AccountB{  **private** int accountNumber = 0;  **private** int currentAccountNumber=1000;  public AccountB(){  this.currentAccountNumber--;  this.accountNumber = currentAccountNumber;  }   public int getNum(){  return this.accountNumber;  } } |

public class T5\_SummarySemestrA {  
 public static void main(String[] args) {  
 AccountA a1 = new AccountA();  
 AccountA a2 = new AccountA();  
 AccountB b1 = new AccountB();  
 AccountB b2 = new AccountB();  
 System.out.println(a1.getNum()+" "+  
 a2.getNum()+" "+  
 b1.getNum()+" "+  
 b2.getNum()  
 );  
 }  
}

התשובה הנכונה היא D - התוכנית לא תעבור את שלב החידור

הסיבה:

* אי אפשר להשתמש ב-this עבור שדה static
* שדה static שייך למחלקה ולא לאובייקט ספציפי
* במקום this.currentAccountNumber צריך לכתוב פשוט currentAccountNumber או AccountA.currentAccountNumber

בדוגמה של Car זה עבד כי זה היה בתוך מתודה רגילה ולא בתוך constructor.

אם הקוד היה נכתב נכון (בלי this לשדה static), התוצאה הייתה 999 998 999 999 (אפשרות B).

Ключевые термины:

* שדה static - статическое поле
* constructor - בנאי
* מחלקה - класс
* אובייקט - объект
* שלב החידור - этап компиляции

\_4\_MAARAH

ZOOpublic class Animal {  
 **private** String name; *//* Name **of** the animal  
 **private** int age; *//* Age **of** the animal  
 **private** String type; *//* Type **of** the animal  
 public Animal(String name, int age, String type) {  
 this.name = name;  
 this.age = age;  
 this.type = type;  
 }  
 public String getName() { return name;}  
 @Override  
 public String toString() {  
 return "Animal{name='" + name + "', age=" + age + ", type='" + type + "'}";  
 }  
}  
class Zoo {  
 **private** static final int MAX\_ANIMALS = 100; *//* Maximum number **of** animals allowed  
 **private** Animal[] animals; *//* Array to store animals  
 **private** int count; *//* Number **of** animals in the zoo  
 *//* Constructor  
 public Zoo() {  
 animals = new Animal[MAX\_ANIMALS]; *//* Initialize the array  
 count = 0; *//* Initial animal count  
 }  
  
 *//* Add an animal to the zoo  
 public void addAnimal(Animal animal) {  
 **if** (count < MAX\_ANIMALS) {  
 animals[count] = animal; *//* Add the animal to the array  
 count++; *//* Increase the count  
 System.out.println(animal.getName() + " was added to the zoo.");  
 } **else** {  
 System.out.println("The zoo is full. Cannot add more animals.");  
 }  
 }  
  
 *//* Remove an animal by index  
 public void removeAnimal(int index) {  
 **if** (index >= 0 && index < count) {  
 System.out.println(animals[index].getName() + " was removed from the zoo.");  
 *//* Сдвиг массива влево, чтобы заполнить пустоту от удаленного итема  
 **for** (int i = index; i < count - 1; i++) {  
 animals[i] = animals[i + 1];  
 }  
 animals[count - 1] = null; *//* обнуляем последний элемент массива  
 count--; *//* Decrease the count  
 } **else** {  
 System.out.println("Invalid index.");  
 }  
 }  
 *//* Return the number **of** animals in the zoo  
 public int getAnimalCount() {  
 return count;  
 }  
  
 *//* Display all animals using a basic **for** loop  
 public String displayAnimals() {  
 String result = "Animals in the zoo:\n";  
 **for** (int i = 0; i < count; i++) {  
 result += animals[i].toString() + "\n";  
 }  
 return result;  
 }  
  
 public static void main(String[] args) {  
 *//* Create a Zoo object  
 Zoo zoo = new Zoo();  
  
 *//* Create Animal objects  
 Animal enot = new Animal("Enot", 2, "Rakoon");  
 Animal lion = new Animal("Lion", 5, "Simba");  
 Animal snake= new Animal("Snake", 3, "Anakonda");  
  
 *//* Add animals to the zoo  
 zoo.addAnimal(enot); zoo.addAnimal(lion); zoo.addAnimal(snake);  
  
 *//* Display current list **of** animals  
 System.out.println(zoo.displayAnimals());  
  
 *//* Remove an animal by index  
 zoo.removeAnimal(1);  
  
 *//* Display updated list **of** animals  
 System.out.println(zoo.displayAnimals());  
  
 *//* Display the number **of** animals  
 System.out.println("Number **of** animals in the zoo: " + zoo.getAnimalCount());  
 }  
}

## PIXEL

public class Pixel {  
 **private** int red; *//* Red component  
 **private** int green; *//* Green component  
 **private** int blue; *//* Blue component  
   
 public Pixel(int red, int green, int blue) {  
 this.red = red;  
 this.green = green;  
 this.blue = blue;  
 }  
  
 public int getRed() { return red; }  
 public int getGreen() { return green; }  
 public int getBlue() { return blue; }  
  
 /\* Проверяет, является ли пиксель чисто красным  
 \* @return true если пиксель красный, false иначе  
 \*/  
 public boolean isRed() {  
 return red > 0 && green == 0 && blue == 0;  
 }  
 public boolean isGreen() { return green > 0 && red == 0 && blue == 0; }  
 public boolean isBlue() { return blue > 0 && red == 0 && green == 0; }  
  
 public boolean isOther() {  
 return !isRed() && !isGreen() && !isBlue();  
 }  
  
 /\* Тестирование класса Pixel  
 \* Принцип работы:  
 \* 1. Создаем тестовые пиксели разных цветов  
 \* 2. Проверяем методы определения цвета  
 \* 3. Создаем и тестируем картины  
 \*/  
 public static void main(String[] args) {  
 *//* Create test pixels  
 Pixel redPixel = new Pixel(255, 0, 0);  
 Pixel greenPixel = new Pixel(0, 255, 0);  
 Pixel bluePixel = new Pixel(0, 0, 255);  
 Pixel mixedPixel = new Pixel(123, 45, 67);  
   
 *//* Test the methods  
 System.out.println("Red Pixel isRed: " + redPixel.isRed()); *//* true  
 System.out.println("Green Pixel isGreen: " + greenPixel.isGreen()); *//* true  
 System.out.println("Blue Pixel isBlue: " + bluePixel.isBlue()); *//* true  
 System.out.println("Mixed Pixel isOther: " + mixedPixel.isOther()); *//* true  
   
 *//* Create a Picture object and test isBalanced method  
 Picture picture = new Picture("Test Picture", "Test Artist", 3, 3);  
   
 *//* Test 1 - Balanced Picture (3 red, 3 green, 3 blue)  
 Pixel[][] firstPicture = {  
 { redPixel, greenPixel, bluePixel },  
 { redPixel, greenPixel, bluePixel },  
 { redPixel, greenPixel, bluePixel }  
 };  
 picture.createPic(firstPicture);  
 System.out.println("First Picture Test: " + picture.isBalanced()); *//* true  
 *//* Test 2 - Unbalanced Picture  
 Pixel[][] secondPicture = {  
 { redPixel, redPixel, redPixel },  
 { redPixel, greenPixel, bluePixel },  
 { redPixel, greenPixel, bluePixel }  
 };  
 picture.createPic(secondPicture);  
 System.out.println("Second Picture Test: " + picture.isBalanced()); *//* false  
 }}  
  
class Picture {  
 **private** String picName; *//* название картины  
 **private** String painterName; *//* имя художника  
 **private** Pixel[][] pixels; *//* массив пикселей  
 \* @param width - ширина картины  
 \* @param height - высота картины  
 public Picture(String picName, String painterName, int width, int height) {  
 this.picName = picName;  
 this.painterName = painterName;  
 this.pixels = new Pixel[height][width];  
 }  
 /\* Создает картину из готового массива пикселей  
 \* @param pixels - двумерный массив пикселей  
 \*/  
 public void createPic(Pixel[][] pixels) {  
 this.pixels = pixels;  
 }  
  
 public String getPicName() { return picName; }  
 public String getPainterName() { return painterName; }  
  
 /\* Проверяет, сбалансирована ли картина  
 \* @return true если количество красных, зеленых и синих пикселей равно  
 public boolean isBalanced() {  
 int redCount = 0, greenCount = 0, blueCount = 0;  
   
 **for**(int i = 0; i < pixels.**length**; i++) {  
 **for**(int j = 0; j < pixels[i].**length**; j++) {  
 **if**(pixels[i][j].isRed()) redCount++;  
 **if**(pixels[i][j].isGreen()) greenCount++;  
 **if**(pixels[i][j].isBlue()) blueCount++;  
 } }  
 return redCount == greenCount && greenCount == blueCount; }  
}

## SUBJECT

public class Subject {  
 **private** String subName; *//* proffession name  
 **private** int grade;  
  
 public Subject(String subName, int grade) {  
 this.subName = subName;  
 this.grade = grade;  
 }  
 public int getGrade() {return grade;}  
}  
  
class ReportCard {  
 **private** String stuname;  
 **private** Subject[] subArray;  
  
 public ReportCard(String stuname, Subject[] subArray) {  
 this.stuname = stuname;  
 this.subArray = subArray;  
 }  
  
 public ReportCard(String stuname, int num) {  
 this.stuname = stuname;  
 this.subArray = new Subject[num];  
 }  
  
 public double average() {  
 double sum = 0;  
 int count = 0;  
 **for** (int i = 0; i < subArray.**length**; i++) {  
 **if** (subArray[i] != null) {  
 sum += subArray[i].getGrade();  
 count++;  
 }  
 }  
 **if** (count == 0) return 0;  
 return sum / count;  
 }  
  
 public Boolean isExcellent() {  
 return average() >= 90;  
 }  
  
 public static void PrintExcellent(ReportCard[] reportCards) {  
 **for** (int i = 0; i < reportCards.**length**; i++) {  
 **if** (reportCards[i].isExcellent()) {  
 System.out.println(reportCards[i].getStuname());  
 }  
 }  
 }  
 public String getStuname() { return stuname; }  
}  
class TestReportCard {  
 public static void main(String[] args) {  
 *//* Create subjects **for** student 1  
 Subject[] subjects1 = {  
 new Subject("Math", 95),  
 new Subject("Physics", 90),  
 new Subject("Chemistry", 88)  
 };  
  
 *//* Create subjects **for** student 2  
 Subject[] subjects2 = {  
 new Subject("Math", 85),  
 new Subject("Physics", 75),  
 new Subject("Chemistry", 80)  
 };  
  
 *//* Create subjects **for** student 3  
 Subject[] subjects3 = {  
 new Subject("Math", 98),  
 new Subject("Physics", 92),  
 new Subject("Chemistry", 96)  
 };  
  
 *//* Create ReportCard objects **for** each student  
 ReportCard student1 = new ReportCard("Alice", subjects1);  
 ReportCard student2 = new ReportCard("Bob", subjects2);  
 ReportCard student3 = new ReportCard("Charlie", subjects3);  
  
 *//* Create an array **of** ReportCard objects  
 ReportCard[] reportCards = { student1, student2, student3 };  
  
 *//* Print the names **of** students with excellent grades  
 System.out.println("Students with excellent grades:");  
 ReportCard.PrintExcellent(reportCards);  
 }  
}

\_4\_ ERUSHA

CIRCLE - CONUS  
class Circle{  
 protected Point center;  
 protected int radius;  
 public Circle(Point center, int radius){  
 this.center = center;  
 this.radius = radius;  
 }  
 public double getArea(){  
 return Math.PI \* radius \* radius; *//* Площадь круга  
 }  
 public double getPerimeter(){  
 return 2 \* Math.PI \* radius; *//* Периметр круга  
 }  
  
 @Override  
 public String toString() {  
 return "The Circle:\n" +  
 "The center **of** the circle: (" + center.getX() + "," + center.getY() + ")\n" +  
 "The radius: " + radius;  
 }  
}  
  
class Cylinder extends Circle {  
 protected int height;  
 public Cylinder(Point center, int radius, int height) {  
 super(center, radius); *//* Вызов конструктора суперкласса Circle  
 this.height = height;  
 }  
   
 @Override  
 public double getArea() {  
 return (2 \* super.getArea()) + (getPerimeter() \* height); *//* Площадь поверхности  
 }  
  
 public double getVolume() {  
 return super.getArea() \* height; *//* Объем  
 }  
 *//* public double getVolume() { -так тоже скомпилирует  
 *//* return getArea() \* height; *//* Объем  
 *//* }  
  
 @Override  
 public String toString() {  
 return "The Cylinder:\n" +  
 "The center **of** the base: (" + center.getX() + ", " + center.getY() + ")\n" +  
 "The radius **of** the base: " + radius + "\n" +  
 "Height: " + height; }}

\_4\_ ERUSHA

class Cone extends Cylinder {  
 public Cone(Point center, int radius, int height) {  
 super(center, radius, height); *//* Вызов конструктора суперкласса Cylinder  
 }  
 @Override  
 public double getArea() {  
 double slantHeight = Math.sqrt(radius \* radius + height \* height); *//* Наклонная высота  
 return (Math.PI \* radius \* slantHeight) + super.getArea(); *//* Площадь поверхности  
 }  
 @Override  
 public double getVolume() {  
 return (1.0 / 3) \* super.getArea() \* height; *//* Объем  
 }  
 @Override  
 public String toString() {  
 return "The Cone:\n" +  
 "The center **of** the base: (" + center.getX() + ", " + center.getY() + ")\n" +  
 "The radius **of** the base: " + radius + "\n" +  
 "Height: " + height;  
 }}  
public class Circles {  
 public static void main(String[] args) {  
 *//* Создание цилиндра  
 Point cylinderCenter = new Point(100, 100);  
 Cylinder cylinder = new Cylinder(cylinderCenter, 1, 10);  
   
 System.out.println(cylinder.toString());  
 System.out.println("Surface Area **of** Cylinder: " + cylinder.getArea());  
 System.out.println("Volume **of** Cylinder: " + cylinder.getVolume());  
  
 *//* Создание конуса  
 Point coneCenter = new Point(50, 50);  
 Cone cone = new Cone(coneCenter, 1, 5);  
   
 System.out.println(cone.toString());  
 System.out.println("Surface Area **of** Cone: " + cone.getArea());  
 System.out.println("Volume **of** Cone: " + cone.getVolume());

ПОЛИМОРФИЗМ

Object[] arr1 = new Object[2];

        Circle[] arr2 = new Circle[2];

        Circle circle = new Circle(new Point(1,1),5);

        Cylinder cylinder = new Cylinder (new Point(1,1),5,3);

        arr1[0] = circle;

        arr1[1] = cylinder;

        arr2[0] = circle;

        arr2[1] = cylinder;

        System.out.println(arr1[0]);

        System.out.println(arr1[0].getArea());

*//* Ошибка возникает, потому что массив arr1 имеет тип Object[],

*//* и компилятор не знает, что объекты в этом массиве имеют метод getArea()

*//* Чтобы исправить это, нужно привести объект к типу Circle.

        System.out.println(arr2[0]);

        System.out.println(arr2[0].getArea());

        System.out.println(arr2[1].getArea());

        System.out.println(arr2[1].getVolume()); *//* тоже ошибка. Надо привести его к типу цилиндра. А в массиве обьекты типа круг. А цилиндр - всеголишь его наследник

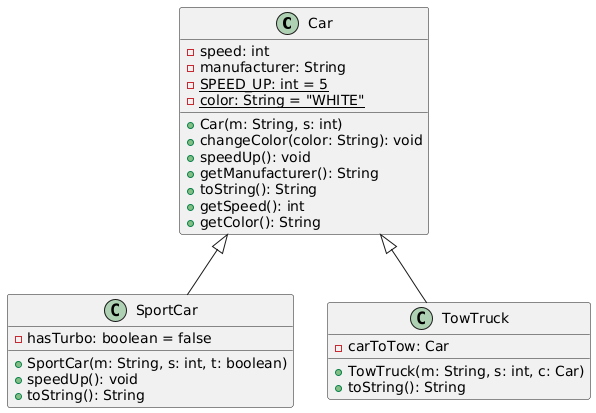
}  
}

# **\_5\_ERUSHA – STATIC--CAR**

class Car {  
 **private** int speed;  
 **private** String manufacturer; *//* היצרן שם  
 **private** static final int SPEED\_UP=5;  
 **private** static String color = "WHITE";

*//* - СТАТИК, значит что если этот цвет изменится- то он изменится не для конкретной тачки, а для ВСЕХ  
 public Car(String m, int s) { *//* конструктор  
 this.manufacturer = m;  
 this.speed = s;  
 }  
 public void changeColor(String color) {  
 this.color = color; } *//* тоже дикая, но вполне себе компилируемая хуйня. Но только в методе. В конструкторе оно не скомпилирует. Лучше так:

public int getSpeed() {  
 return speed;  
 }  
 public String getColor() {  
 return color;  
 }



Car.*color* = color; *//* замена статика

*//* если не пересекается 120, то заплюсит +5км.  
 public void speedUp() {  
 **if** (this.speed + SPEED\_UP <= 120)  
 this.speed += SPEED\_UP;  
 }  
 public String getManufacturer() {  
 return this.manufacturer;  
 }  
 /\* это не сработает, потому что нет геттера на колор. Не смотря на статик.  
 public String toString() {  
 return "Car " + this.manufacturer +  
 " " + this.color + " " + this.speed;  
 } \*/  
 *//* а с этими геттерами сработает  
 public String toString() {  
 return "Car " + this.manufacturer +  
 " " + color + " " + this.speed;  
}

}

class SportCar extends Car {  
 **private** boolean hasTurbo = false;  
 public SportCar(String m, int s, boolean t) {  
 super(m,s);  
 this.hasTurbo = t;  
 }  
 *//*по уму - в таких случаях надо писать @OverRide  
 public void speedUp() {  
 super.speedUp();  
 super.speedUp();  
 super.speedUp(); }  
 *//*по уму - в таких случаях надо писать @OverRide  
 public String toString() {  
 return "Sport " + super.toString() +  
 " Turbo:" + this.hasTurbo;  
 }}  
  
class TowTruck extends Car {  
 **private** Car carToTow;  
 public TowTruck(String m, int s, Car c) {  
 super(m,s);  
 this.carToTow = c;  
 }  
 /\* Здесь line2 не сработает, т.к. надо через геттеры на цвет и скорость  
 это ИНКАПСУЛЯЦИЯ  
 public String toString() {  
 return "TowTruck: " + this.getManufacturer() + *//* ...line1  
 " " + this.color + " " + this.speed; *//* ...line2  
 } \*/  
 public String toString() {  
 return "TowTruck: " + this.getManufacturer() + *//* ...line1  
 " " + this.getColor() + " " + this.getSpeed(); *//* ...line2  
 }}  
class CarShop {  
 public static void main(String[] args) {  
 Car[] ourCars = new Car[4];  
 ourCars[0] = new Car("Ford",55);  
 ourCars[1] = new SportCar("Ferari",80,true);  
 *//* line3 вполне выполнит, но это хуево - будет просто 2 ссылки на 1 и тот же объект  
 *//* по уму нужно сделать глубокое копирование.  
 ourCars[2] = ourCars[1]; *//* ... line3  
 ourCars[3] = new TowTruck("Isuzu",80,null);  
  
 *//* надо сделать явное определение типов, не выполнит.  
 *//* SportCar myCar = ourCars[1]; *//* ...line4  
 SportCar myCar = (SportCar)ourCars[1];  
 **for** (int i = 0; i < ourCars.**length**; i++) {  
 ourCars[i].speedUp();  
 }  
 *//* ебанет на все тачки синий цвет, хотя указана одна.  
 ourCars[2].changeColor("BLUE"); *//* ....line5  
 **for** (int i = 0; i < ourCars.**length**; i++) {  
 System.out.println(ourCars[i]);  
 }  
 /\*  
Car Ford BLUE 60  
*//* была ссылка на 1 объект дважды  
*//* там метод спидап для спортов 3 раза по 5 плюсит  
Sport Car Ferari BLUE 110 Turbo:true -  
Sport Car Ferari BLUE 110 Turbo:true

TowTruck: Isuzu BLUE 85 \*/  
 }}

# \_6\_ POLIMORPHISM

## -- TOY STORE

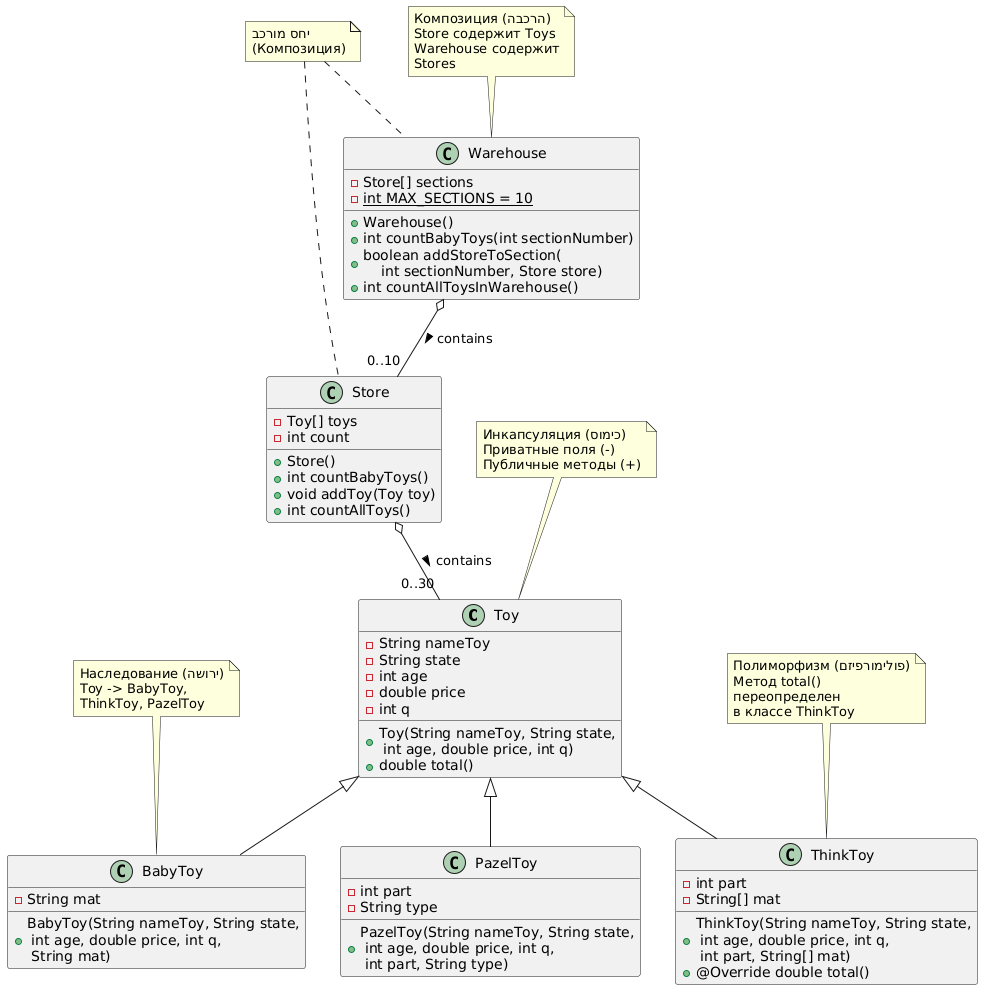
Компания по маркетингу игрушек разрабатывает проект для управления запасами игрушек в компании.  
Ниже частичное определение классов в Java: Игрушка - Toy, игрушка для малышей - BabyToy,  
развивающая игрушка - ThinkToy, игрушка-пазл - PazelToy  
  
A. Реализуйте в Java конструктор в классе BabyToy, который использует конструктор, получающий параметры в классе Toy.

Б. Укажите, какой механизм объектно-ориентированного программирования проявляется в определении метода total() в классе Toy и метода total() в классе ThinkToy.

В. Можно ли реализовать метод total() в классе ThinkToy с помощью метода total() из класса Toy? Если да - реализуйте его, если нет - объясните почему.

Г. Класс Store представляет магазин игрушек, содержащий игрушки всех

типов (BabyToy, ThinkToy, PazelToy).  
 В магазине может быть до 30 видов игрушек. Реализуйте в Java класс Store, который должен включать следующие пункты:  
Г.1 Конструктор класса и его свойства  
Г.2 Метод, возвращающий количество babyToy в магазине  
Г.3 Метод, принимающий новую игрушку и добавляющий её в магазин  
  
D. Класс WareHouse представляет склад из 10 отсеков для игрушек, пронумерованных от 1 до 10.  
D.1 Конструктор класса и его свойства  
D.2 Метод, принимающий номер отсека 1-10 и возвращающий, сколько детских игрушек BabyToy есть в этом отсеке  
D.3 Метод, принимающий номер отсека 1-10 и игрушку, добавляющий игрушку в отсек,  
 при условии, что есть место для игрушки в отсеке.  
D.4 Метод добавления Store в отсек  
D.5 Метод для подсчета всех игрушек во всем складе



class Toy {  
 protected String nameToy;  
 protected String state;  
 protected int age;  
 protected double price;  
 protected int q;  
  
 public Toy() {  
 }  
  
 public Toy(String nameToy, String state, int age, double price, int q) {  
 }  
  
 public double total() {  
 return this.price \* this.q;  
 }  
}  
*//* A. Реализуйте в Java конструктор в классе BabyToy, который использует конструктор,

*//* получающий параметры в классе Toy.  
class BabyToy extends Toy{  
 **private** String mat; *//* игра сделана из 1‑го этого материала  
 *//* в конструкторе мы ОБЯЗАТЕЛЬНО добавляем ВСЕ аргументы супер класса и потом + свои.  
 public BabyToy(String nameToy, String state, int age, double price, int q, String mat) {  
 super(nameToy, state, age, price, q);  
 this.mat = mat;  
 }}  
  
*//* Б. Укажите, какой механизм объектно-ориентированного программирования проявляется в определении метода total() в классе Toy и метода total() в классе ThinkToy.  
*//* В. Можно ли реализовать метод total() в классе ThinkToy с помощью метода total() из класса Toy? Если да - реализуйте его, если нет - объясните почему.

class ThinkToy extends Toy {  
 **private** int part; *//* номер деталей  
 **private** String[] mat; *//* материалы для запчастей игрушек  
 /\* если бы не было +10 (изменения) - мы бы просто унаследовали его.  
 *//* здесь был явный ПОЛИМОРФИЗМ и НАСЛЕДОВАНИЕ  
 public double total() {  
 return this.price \* this.q + 10;  
 } \*/  
 *//* Конструктор  
 public ThinkToy(String nameToy, String state, int age, double price, int q, int part, String[] mat) {  
 super(nameToy, state, age, price, q); *//* вызываем конструктор родителя  
 this.part = part;  
 this.mat = mat;  
 }  
 @Override  
 public double total(){  
 return super.total()+10;  
 }  
}  
class PazelToy extends Toy{  
 **private** int part; *//* количество частей  
 **private** String mat; *//* материал  
 **private** String type; *//* комикс, фотки и тд  
 *//* Конструктор  
 public PazelToy(String nameToy, String state, int age, double price, int q, int part, String mat, String type) {  
 super(nameToy, state, age, price, q); *//* вызываем конструктор родителя  
 this.part = part;  
 this.mat = mat;  
 this.type = type;  
 }}

*//* Г. Класс Store представляет магазин игрушек, содержащий игрушки всех типов (BabyToy, ThinkToy, PazelToy).  
*//* в магазине может быть до 30 видов игрушек.  
*//* Г.1 Конструктор класса и его свойства  
*//* Г.2 Метод, возвращающий количество babyToy в магазине  
*//* Г.3 Метод, принимающий новую игрушку и добавляющий её в магазин  
  
class Store {  
 public Toy[] toys;  
 **private** int count;  
*//* Г.1 Конструктор класса и его свойства  
 public Store () {  
 toys = new Toy[30]; *//* максимум 30 игрушек  
 count = 0; }

СЮДА СМОТРИ ТУТ ИНСТАНС ОФ

*//* Подсчет определенного типа игрушек  
*//* Г.2 Метод, возвращающий количество babyToy в магазине  
 public int countBabyToys(){  
 int babyCount = 0;  
 **for** (int i=0; i<count; i++){  
 **if** (toys[i] instanceof BabyToy) {  
 babyCount++;  
 } }  
 return babyCount; }  
  
*//* Г.3 Метод, принимающий новую игрушку и добавляющий её в магазин  
 public void addToy(Toy toy){  
 **if** (count<30){ *//* проверка или есть место  
 toys[count] = toy;  
 count++; *//* когда добавили - увеличиваем свойство класса count на 1.  
 }**else** {  
 System.out.println("Store full");  
 } }}

/\* D. Класс WareHouse представляет склад из 10 отсеков для игрушек, пронумерованных от 1 до 10.  
D.1 Конструктор класса и его свойства  
D.2 Метод, принимающий номер отсека 1-10 и возвращающий, сколько детских игрушек BabyToy есть в этом отсеке  
D.3 Метод, принимающий номер отсека 1-10 и игрушку, добавляющий игрушку в отсек,  
при условии, что есть место для игрушки в отсеке.  
D.4 Метод добавления Store в отсек  
D.5 Метод для подсчета всех игрушек во всем складе \*/  
class Warehouse{  
 **private** Store[] sections; *//* массив отсеков, в каждом отсеке массив игрушек  
 **private** int MAX\_SECTIONS = 10; *//* всегда 10 отсеков  
  
 *// D.1. Конструктор класса и его свойства*  
 public Warehouse() {  
 sections = new Store[MAX\_SECTIONS];  
 *//* изначально все отсеки пустые (null)  
 }  
  
 *//* D.2 Метод, принимающий номер отсека 1-10 и возвращающий, сколько детских игрушек BabyToy есть в этом отсеке  
 public int countBabyToys(int sectionNumber){  
 **if** (sectionNumber<1 || sectionNumber > MAX\_SECTIONS) return -1 ;  
 int index = sectionNumber-1; *//* преобразуем номер 1-10 в индекс 0-9  
 **if** (sections[index] == null) return -1;  
 return sections[index].countBabyToys();  
 }  
  
 *//* D.3. Метод, принимающий номер отсека 1-10 и игрушку, добавляющий игрушку в отсек,  
 *//* при условии, что есть место для игрушки в отсеке.  
 public void addToMahsanStore(int NUM,Toy toy){  
 **if** (NUM<1 || NUM >10)  
 System.out.println("ERROR - check num");  
 sections[NUM-1].addToy(toy);  
 }  
  
 *//* D.4. Метод добавления Store в отсек  
 public boolean addStoreToSection(int sectionNumber, Store store) {  
 *//* Проверяем корректность номера отсека  
 **if**(sectionNumber < 1 || sectionNumber > MAX\_SECTIONS) {  
 System.out.println("Invalid section number");  
 return false;  
 }  
 int index = sectionNumber - 1;  
 *//* Проверяем, свободен ли отсек  
 **if**(sections[index] == null) {  
 sections[index] = store;  
 return true;  
 } **else** {  
 System.out.println("Section " + sectionNumber + " is already occupied");  
 return false;  
 }  
 }  
  
 *//* D.5 Метод для подсчета всех игрушек во всем складе  
 public int countAllToysInWarehouse() {  
 int totalToys = 0;  
 **for**(int i = 0; i < MAX\_SECTIONS; i++) {  
 **if**(sections[i] != null) {  
 *//* Для каждого Store считаем ненулевые элементы в массиве toys  
 **for**(int j = 0; j < sections[i].toys.**length**; j++) {  
 **if**(sections[i].toys[j] != null) {  
 totalToys++; } } }  
 }  
 return totalToys; }}

public class T2\_SummarySemestrA {  
 public static void main(String[] args) {  
 *//* Создаем склад  
 Warehouse warehouse = new Warehouse();  
   
 *//* Создаем несколько магазинов  
 Store store1 = new Store();  
 Store store2 = new Store();  
   
 *//* Создаем разные игрушки  
 BabyToy babyToy1 = new BabyToy("Кукла", "Новая", 3, 29.99, 5, "Пластик");  
 BabyToy babyToy2 = new BabyToy("Мячик", "Новый", 1, 15.50, 10, "Резина");  
 String[] materials = {"plastic", "metal"};  
 ThinkToy thinkToy = new ThinkToy("Smart Puzzle", "New", 5, 50.0, 1, 10, materials);  
 PazelToy pazelToy = new PazelToy("Космос", "Новый", 7, 45.0, 1, 1000, "Фото", "ololo");  
   
 *//* Добавляем игрушки в первый магазин  
 store1.addToy(babyToy1);  
 store1.addToy(babyToy2);  
   
 *//* Добавляем игрушки во второй магазин  
 store2.addToy(thinkToy);  
 store2.addToy(pazelToy);  
   
 *//* Добавляем магазины в разные секции склада  
 warehouse.addStoreToSection(1, store1);  
 warehouse.addStoreToSection(2, store2);  
   
 *//* Тестируем функционал  
 System.out.println("Количество детских игрушек в секции 1: " + warehouse.countBabyToys(1));  
 System.out.println("Количество детских игрушек в секции 2: " + warehouse.countBabyToys(2));  
 System.out.println("Общее количество игрушек на складе: " + warehouse.countAllToysInWarehouse());  
   
 *//* Пробуем добавить игрушку в существующую секцию  
 BabyToy babyToy3 = new BabyToy("Погремушка", "Новая", 0, 10.99, 3, "Пластик");  
 warehouse.addToMahsanStore(1, babyToy3);  
   
 *//* Проверяем обновленное количество  
 System.out.println("\nПосле добавления новой игрушки:");  
 System.out.println("Количество детских игрушек в секции 1: " + warehouse.countBabyToys(1));  
 System.out.println("Общее количество игрушек на складе: " + warehouse.countAllToysInWarehouse());  
 }  
}

# Работа с классами Point и Vector

Класс Point - определяет точку на плоскости

|  |  |
| --- | --- |
| Метод | Описание |
| Point(int x, int y) | Создает точку на плоскости |
| void setY(int y) | Обновляет значение координаты y на переданное |
| String toString() | Возвращает строковое представление x и y точки |

Класс Vector - определяет тип данных переменного размера для хранения объектов типа Object

Примечание: первый объект хранится в позиции с индексом 0

|  |  |
| --- | --- |
| Метод | Описание |
| Vector() | Создает пустой вектор размером 0 |
| void addElement(Object) | Добавляет объект в конец вектора |
| Object elementAt(int) | Возвращает объект по указанному индексу |
| int size() | Возвращает строковое представление x и y точки |

Задание:

--------

1. Создать программу MyPoints, содержащую только метод main()

2. Программа должна:

- Создать вектор с минимум тремя точками: (1,1), (2,2), (3,3)

- Порядок точек в векторе не важен

- Вывести все точки, обнулив их координату y

- Использовать все методы из интерфейсов

3. Примечание: точная длина вектора неизвестна

4. Формат вывода точек: (x,0)

class Point{  
 int x = 0;  
 int y = 0;  
 public Point(int x, int y){  
 this.x = x;  
 this.y = y;  
 }  
 public void setY(int y){  
 this.y = y;  
 }  
 public String **toString**() {  
 return "X:"+x+" Y:"+y;  
 }}

Пример вывода:------------(1,0) (2,0) (3,0)

class Vector {  
 **private** Object[] elements;  
 **private** int size;  
 *//* Конструктор - создает пустой вектор размером 0  
 public Vector(){  
 elements = new Object[10]; *//* начальная емкость (она >3)  
 size = 0; }  
  
 *//* Метод добавления объекта в конец вектора  
 public void addElement(Object **obj**){  
 **if** (size < elements.**length**) {  
 elements[size++] = **obj**; } }  
  
 public Object elementAt(int index){  
 **if** (index >= 0 && index < size)  
 return elements[index];  
 return null; }  
  
 public int size(){ return size; } }  
  
 class MyPoints{  
 public static void main(String[] args){  
 Vector v = new Vector();  
 *//* добавляем минимум 3 точки  
 v.addElement(new Point(1,1));  
 v.addElement(new Point(2,2));  
 v.addElement(new Point(3,1));  
  
 **for** (int i=0; i< v.size(); i++){  
 Point p = (Point)v.elementAt(i);  
 p.setY(0);  
 System.out.println(p.toString()); } } }

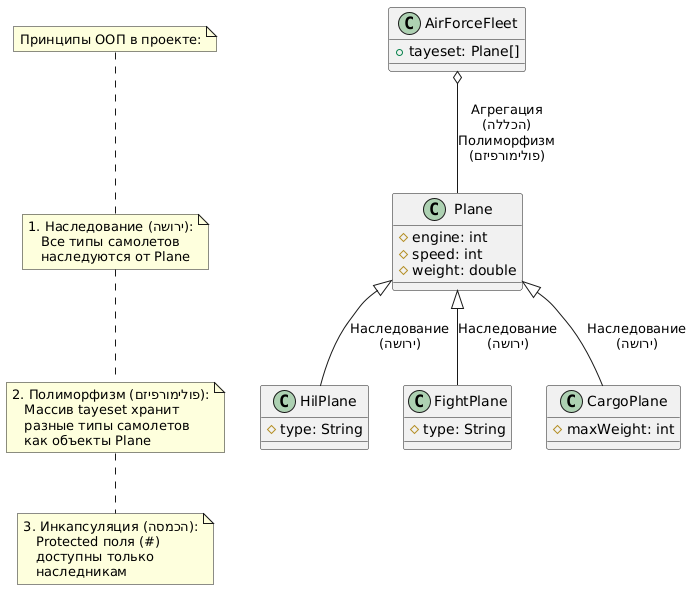
Так сделать нельзя - Метод elementAt возвращает тип Object, а не Point.  
 for (int i=0; i< v.size(); i++){  
 v.elementAt(i).setY(0);  
 System.out.println(v.elementAt(i).**toString**());  
 }  
Java не знает, что возвращаемый объект - это именно Point, и поэтому не может найти метод setY().  
 Нужно обязательно сделать приведение типов (cast)

# \_6\_ POLIMORPHISM

## -- АЕРОПОРТ И САМОЛЕТЫ

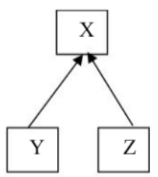
class Plane{  
 protected int engine; *//* номер борта  
 protected int speed; *//* max скорость  
 protected double weight; *//* вес  
 public Plane(int engine, int speed, double weight){  
 this.engine = engine;  
 this.speed = speed;  
 this.weight = weight;  
 }  
 public String toString(){  
 return " engine:" + engine + " speed:" + speed +" weight: "+weight;  
 }  
}  
  
class HilPlane extends Plane{  
 protected String type;  
 public HilPlane(int engine, int speed, double weight, String type){  
 super(engine, speed, weight);  
 this.type=type;  
 }  
 public String toString(){  
 return super.toString() + "type: "+type;  
 }  
}  
  
class FightPlane extends Plane{  
 protected String type;  
 public FightPlane(int engine, int speed, double weight, String type){  
 super(engine, speed, weight);  
 this.type = type;  
 }  
 public String toString(){  
 return super.toString() + "type: "+type;  
 }  
}  
  
class CargoPlane extends Plane{  
 protected int maxWeight;  
 public CargoPlane(int engine, int speed, double weight, int maxWeight){  
 super(engine, speed, weight);  
 this.maxWeight = maxWeight;  
 }}  
  
class AirForceFleet{  
 **private** int numPlane; *//* к-во самолетов в авиапарке  
 public Plane[] tayeset; *//* массив самолетов  
 public AirForceFleet(){  
 this.numPlane = 0;  
 tayeset = new Plane[50];  
 }  
 public void addPlane(Plane p){  
 this.tayeset[numPlane]=p;  
 this.numPlane++;  
 }}  
  
class Test{  
 public static void main(String[] args) {  
 AirForceFleet natBag = new AirForceFleet();  
 natBag.addPlane(new FightPlane(450, 3000, 1, "F16"));  
 natBag.addPlane(new FightPlane(450, 3000, 1, "F14")); natBag.addPlane(new HilPlane(70, 300, 1, "Apache"));  
 natBag.addPlane(new CargoPlane(45, 300, 1, 5000));  
 *//* добавим двумя способами в эскадрилью объекты.  
 Plane p1 = new Plane(45, 300, 1);  
 natBag.addPlane(p1);  
 CargoPlane p2 = new CargoPlane(45, 300, 1, 5000);  
 HilPlane p3 = new HilPlane(45, 300, 1, "aligator");  
 FightPlane p4 = new FightPlane(45, 300, 1, "stinger");  
  
 System.out.println(p1); System.out.println(p2); System.out.println(p3); System.out.println(p4);

natBag.addPlane(p1); natBag.addPlane(p2); natBag.addPlane(p3); natBag.addPlane(p4);  
  
 *//* Подсчет типов самолетов  
 int regularPlanes = 0;  
 int fightPlanes = 0;  
 int hilPlanes = 0;  
 int cargoPlanes = 0;  
  
 **for**(int i = 0; i < 50; i++) {  
 **if**(natBag.tayeset[i] instanceof FightPlane) {  
 fightPlanes++;  
 } **else** **if**(natBag.tayeset[i] instanceof HilPlane) {  
 hilPlanes++;  
 } **else** **if**(natBag.tayeset[i] instanceof CargoPlane) {  
 cargoPlanes++;  
 } **else** **if**(natBag.tayeset[i] instanceof Plane) {  
 regularPlanes++; } }  
  
 *//* Вывод результатов.пришлось заменить public Plane[] tayeset с privat;  
 System.out.println("В авиапарке:");   
 System.out.println("Обычных самолетов: " + regularPlanes);  
 System.out.println("Истребителей: " + fightPlanes);  
 System.out.println("Вертолетов: " + hilPlanes);  
 System.out.println("Грузовых самолетов: " + cargoPlanes); }}



# \_6\_ POLIMORPHISM -- UPCASTING DOWNCASTING

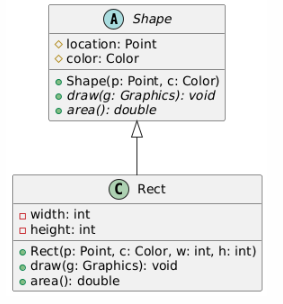
X x = new X();  
 Y y = new Y();  
 Z z = new Z();  
 X xy = new Y(); *//* upcasting+  
 X xz = new Z(); *//* upcasting+  
*//* Y yz = new Z(); *//* ERROR: compilation  
*//* Y y1 = new X(); *//* ERROR: compilation  
*//* Z z1 = new X(); *//* ERROR: compilation  
 X x1 = y; *//* upcasting+  
 X x2 = z; *//* upcasting+  
*//* Y y1 = (Y)x; *//* downcasting- ERROR: ClassCastException  
*//* Z z1 = (Z)x;  
 Y y2 = (Y)x1; *//* downcasting+  
 Z z2 = (Z)x2; *//* downcasting+  
*//* Y y3 = (Y)z; *//* ERROR: compilation  
 Object o = z; *//* upcasting+  
*//* Object o1 = (Y)o; *//* downcasting- ERROR: ClassCastException



# \_7\_ ABSTRACT

## Shape

public **abstract** class Shape {  
 protected Point location;  
 protected Color color;  
 public Shape(Point p, Color c){  
 this.location = p;  
 this.color = c; }  
 public **abstract** void draw(Graphics g);  
 public **abstract** double area();  
}



public class Rect extends Shape{  
 **private** int width;  
 **private** int height;  
 public Rect(Point p, Color c, int w, int h) {  
 super(p, c);  
 this.width = w;  
 this.height = h;  
 }  
 @Override  
 public void draw(Graphics g) {  
 g.setColor(super.color);  
 g.drawRect((int) super.location.getX(), (int) super.location.getY(), this.width, this.height);  
 }  
 @Override  
 public double area() { return 0; }}

# \_7\_ ABSTRACT TEST -- ONE - TWO

public **abstract** class One {  
 public **abstract** int sumOfTwo(int n1, int n2);  
 public **abstract** int sumOfThree(int n1, int n2, int n3);  
 public void disp(){  
 System.out.println("Method **of** class One");  
 }  
}  
  
class Two extends One{  
 @Override  
 public int sumOfTwo(int num1, int num2) { return num1+num2; }  
 @Override  
 public int sumOfThree(int num1, int num2, int num3) {  
 return num1+num2+num3; }}  
  
class Testabstract{  
 public static void main(String args[]){  
 Two **obj** = new Two(); *//* 10  
 System.out.println(**obj**.sumOfTwo(3,7)); *//* 26  
 System.out.println(**obj**.sumOfThree(4,3,19));  
 **obj**.disp(); *//* Method **of** class One }}

## --NODE-- TREK

public **abstract** class Trek {  
 **private** String name;  
 **private** String location;  
 **private** int duration;  
  
 public Trek(String name, String location, int duration) {  
 this.name = name;  
 this.location = location;  
 this.duration = duration;  
 }  
  
 public **abstract** String getDifficulty();  
 public **abstract** Node<String> getRequiredEquipment();  
  
 *//* Prints trek details  
 public void printDetails() {  
 System.out.println("Name: " + name);  
 System.out.println("Location: " + location);  
 System.out.println("Duration: " + duration + " days");  
 System.out.println("Difficulty: " + getDifficulty());  
   
 System.out.print("Required Equipment: ");  
 Node<String> equipment = getRequiredEquipment();  
 while (equipment != null) {  
 System.out.print(equipment.getValue() + ", ");  
 equipment = equipment.getNext();  
 }  
 System.out.println();  
 }  
}

public class DesertTrek extends Trek {  
 **private** double temperature;  
  
 public DesertTrek(String name, String location, int duration, double temperature) {  
 super(name, location, duration);  
 this.temperature = temperature;  
 }  
  
 @Override  
 public String getDifficulty() {  
 return "Medium";  
 }  
  
 @Override  
 public Node<String> getRequiredEquipment() {  
 return new Node<>("Sunscreen", new Node<>("Wide hat"));  
 }  
  
 public double getTemperature() {  
 return temperature;  
 }  
  
 public void setTemperature(double temperature) {  
 this.temperature = temperature;  
 }  
}

public static void main(String[] args) {  
 *//* Create a list **of** treks according to the task example  
 Node<Trek> treks = new Node<Trek>(  
 new MountainTrek("Annapurna", "Nepal", 10, 5000),  
 new Node<Trek>(new DesertTrek("Desert sands", "Morocco", 5, 40)));  
   
 *//* Print information about each trek  
 Node<Trek> pos = treks;  
 while (pos != null) {  
 pos.getValue().printDetails();  
 System.out.println("");  
 System.out.println("->");  
 pos = pos.getNext();  
 }  
   
 *//* Demonstration **of** creating additional treks  
 Trek mountainTrek2 = new MountainTrek("Everest", "Nepal", 15, 8000);  
 Trek desertTrek2 = new DesertTrek("Sahara", "Egypt", 7, 45.5);  
   
 System.out.println("\nAdditional treks:");  
 mountainTrek2.printDetails();

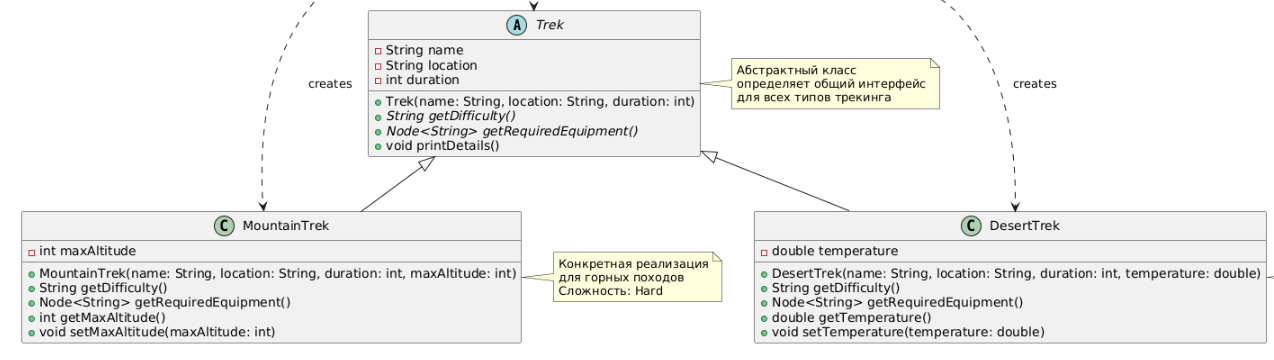
Name: Annapurna

Location: Nepal

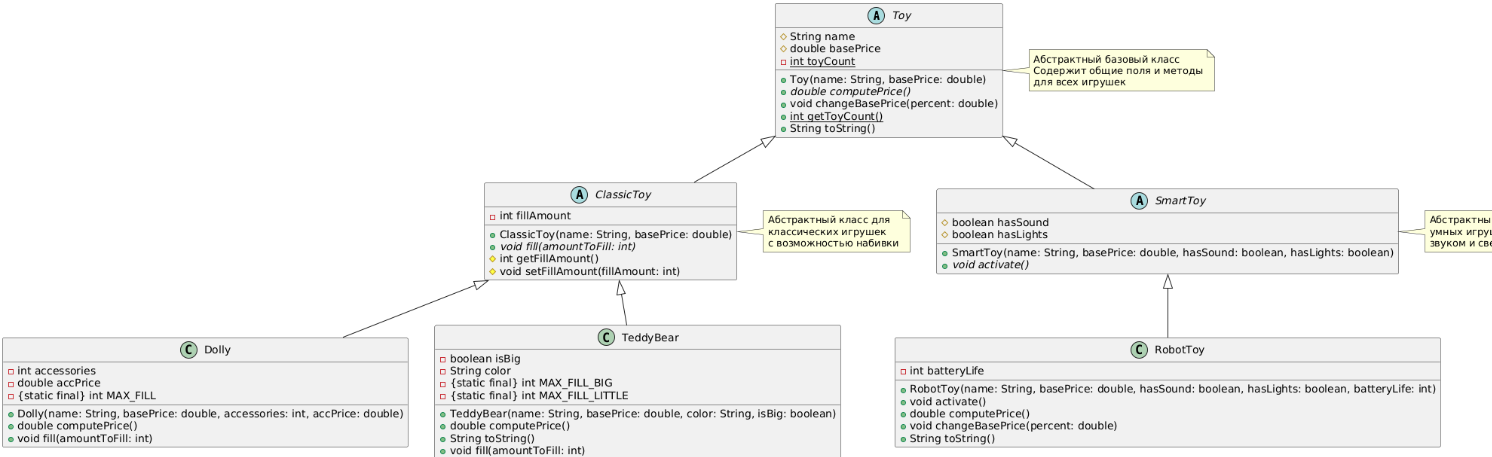
Duration: 10 days

Difficulty: Hard

Required Equipment: Warm coat, Walking sticks, Climbing shoes,



## -- TOY WAREHOUSE--



public **abstract** class Toy {  
 protected String name;  
 protected double basePrice;  
 **private** static int *toyCount*=0;  
  
 public Toy(String name, double basePrice) {  
 this.name = name;  
 this.basePrice = basePrice;  
 *toyCount*++;  
 }  
  
 public **abstract** double computePrice();  
 public void changeBasePrice(double percent){  
 this.basePrice = basePrice \* (1 + percent/100);  
 }  
 public static int getToyCount(){ return *toyCount*; }  
 @Override  
 public String toString() {  
 return this.name+": the price is "+ this.computePrice();  
 }}

public **abstract** class ClassicToy extends Toy{  
 **private** int fillAmount;  
 public ClassicToy(String name, double basePrice){  
 super(name, basePrice);  
 this.fillAmount = 0;  
 }  
 public **abstract** void **fill**(int amountToFill);  
 protected int getFillAmount(){  
 return this.fillAmount;  
 }  
 protected void setFillAmount(int fillAmount){  
 this.fillAmount = fillAmount; }}  
  
public class Dolly extends ClassicToy{  
 **private** int accessories;  
 **private** double accPrice;  
 **private** static final int *MAX\_FILL* = 60;  
  
 public Dolly(String name, double basePrice, int accessories, double accPrice){  
 super(name, basePrice);  
 this.accessories = accessories;  
 this.accPrice = accPrice;  
 }  
  
 @Override  
 public double computePrice(){  
 return basePrice + (accessories \* accPrice);  
 }  
  
 @Override  
 public void **fill**(int amountToFill){  
 int newAmount = getFillAmount() + amountToFill;  
 **if** (newAmount <= *MAX\_FILL*){  
 setFillAmount(newAmount);  
 } **else** {  
 setFillAmount(*MAX\_FILL*); } }}

public static void main(String[] args) {  
 Toy[] toys = new Toy[6];  
 toys[0] = new Dolly("Dolly1", 100, 3, 20);  
 toys[1] = new Dolly("Dolly2", 120, 2, 25);  
 toys[2] = new TeddyBear("Teddy1", 80, "Red", true);  
 toys[3] = new TeddyBear("Teddy2", 90, "Blue", false);  
 toys[4] = new RobotToy("Robot1", 200, true, true, 15);  
 toys[5] = new RobotToy("Robot2", 250, false, true, 8);  
 **for** (int i=0; i<toys.**length**;i++)  
 System.out.println(toys[i].toString());  
  
 toys[1].changeBasePrice(50);  
 System.out.println(toys[1].toString());  
 System.out.println("Toy count: "+Toy.*getToyCount*());  
 ((RobotToy)toys[5]).activate();  
  
 Toy[] toys2 = new Toy[2];  
 toys2[0] = new Dolly("Dolly1", 100, 3, 20);  
 toys2[1] = new Dolly("Dolly2", 120, 2, 25);  
 **for** (int i=0; i<toys2.**length**;i++)  
 System.out.println(toys2[i].toString());  
 System.out.println("Toy count: "+Toy.*getToyCount*());  
  
 *//* Test 1: Testing TeddyBear filling functionality  
 System.out.println("\nTeddyBear Filling Test:");  
 TeddyBear bigBear = new TeddyBear("BigTeddy", 100, "Brown", true);  
 System.out.println("Initial state: " + bigBear);  
 bigBear.**fill**(50);  
 System.out.println("After filling 50: " + bigBear);  
 bigBear.**fill**(40); *//* Trying to exceed maximum  
 System.out.println("After attempting to overfill: " + bigBear);  
  
 *//* Test 2: Testing Robot activation  
 System.out.println("\nRobot Functionality Test:");  
 RobotToy smartRobot = new RobotToy("SmartBot", 300, true, false, 20);  
 System.out.println("Initial state: " + smartRobot);  
 smartRobot.activate();  
 System.out.println("After activation: " + smartRobot);  
  
 *//* Test 3: Testing price changes with different percentages  
 System.out.println("\nPrice Change Test:");  
 Dolly testDolly = new Dolly("TestDolly", 100, 2, 15);  
 System.out.println("Initial price: " + testDolly.computePrice());  
 testDolly.changeBasePrice(10); *//* increase by 10%  
 System.out.println("After 10% increase: " + testDolly.computePrice());  
 testDolly.changeBasePrice(-5); *//* decrease by 5%  
 System.out.println("After 5% decrease: " + testDolly.computePrice());  
  
 *//* Test 4: Testing edge cases  
 System.out.println("\nEdge Cases Test:");  
 RobotToy minPriceRobot = new RobotToy("MinBot", 0, false, true, 1);  
 RobotToy maxPriceRobot = new RobotToy("MaxBot", 1000, true, true, 100);  
 System.out.println("Robot with minimum price: " + minPriceRobot);  
 System.out.println("Robot with maximum price: " + maxPriceRobot);  
  
 *//* Test 5: Testing toy counter  
 System.out.println("\nToy Counter Test:");  
 System.out.println("Current toy count: " + Toy.*getToyCount*());  
  
 *//* Test 6: Testing different battery life impacts on price  
 System.out.println("\nBattery Life Impact Test:");  
 RobotToy lowBatteryRobot = new RobotToy("LowBatBot", 100, true, true, 5);  
 RobotToy highBatteryRobot = new RobotToy("HighBatBot", 100, true, true, 15);  
 System.out.println("Low battery robot price: " + lowBatteryRobot.computePrice());  
 System.out.println("High battery robot price: " + highBatteryRobot.computePrice());  
  
 *//* Final statistics  
 System.out.println("\n=== Final Statistics ===");  
 System.out.println("Final toy count in store: " + Toy.*getToyCount*());

\_8\_ INTERFACE

## -- Animals

|  |  |
| --- | --- |
| public interface ICanFly { public void fly();}  public interface IHasEggs { public void layingEggs();}  **abstract** class Animal {}  public class Mammal extends Animal {}  public class Bat extends Mammal implements ICanFly {  public void fly() {} } |  |
| public class Fish extends Animal implements IHasEggs {  public void layingEggs(){} }  public class Bird extends Animal implements IHasEggs, ICanFly{  public void layingEggs() { }  public void fly() { } } | |

Подъебки:

Animal a = new Bat(); *//* 1. ++ Polymorphism  
*//* Fish f = new IHasEggs(); *//* 2. -- ERROR: Compilation  
 ICanFly cf = new Bird(); *//* 3. ++ Переменная интерфейса может ссылаться на любой объект, реализующий этот интерфейс.  
 cf = new Bat();  
 cf.fly();  
  
 Mammal m = new Bat(); *//* 4. +-  
*//* m.fly(); *//* -- ERROR: Compilation  
 ((Bat)m).fly(); *//* +  
  
 Animal a2 = new Bird(); *//* 5. ++ Polymorphism and type casting  
 ((Bird)a2).fly();  
  
 IHasEggs he = new Bird();*//* 6. +-  
*//* he.fly(); *//* -- ERROR: Compilation  
 he.layingEggs(); *//* +

INTERFACE INSTANCEOF

*//* Метод, который считает летучих животных

public int printICanFly(Animal[] animals){  
 int count = 0;  
 **for** (int i=0; i<animals.**length**; i++){  
 **if** (animals[i] instanceof ICanFly) count++;  
 }  
 return count;  
}

\_8\_ INTERFACE -- Pnimiya (static)

|  |  |
| --- | --- |
| public interface Student {  public void setGroupGrade (int grade);   public int getGroupGrade();  public Teacher getGroupTeacher();  public void setGroupTeacher (Teacher newTeacher);   public boolean hasPermission(); } |  |
| public class Teacher {  **private** String name;  public Teacher(String name){ this.name = name; }  public boolean givePermission(Student student) {  return student.getGroupGrade() > 0; *//* студент должен иметь положительную оценку }}  public class AStudent implements Student {  **private** int id;  **private** String name;  **private** static int *grade*;  **private** static Teacher *groupTeacher*;   public AStudent(int id, String name) {  this.id = id;  this.name = name;  }   public void setGroupGrade(int grade) { AStudent.*grade* = grade; }  public int getGroupGrade() { return AStudent.*grade*; }  public Teacher getGroupTeacher() { return AStudent.*groupTeacher*; }  public void setGroupTeacher(Teacher newTeacher) { AStudent.*groupTeacher* = newTeacher; }  public boolean hasPermission() {  return (AStudent.*grade* > 150) && (AStudent.*groupTeacher* != null && AStudent.*groupTeacher*.givePermission(this));  } } | |

\_8\_ INTERFACE -- IOne Itwo --Подьебка

|  |  |
| --- | --- |
| 1. -- `ITwo a = new ITwo();` -- Это **שגיאת הידור** (ошибка компиляции).  Интерфейсы нельзя инстанцировать напрямую. Нужен класс, реализующий интерфейс.  **אינטרפאסים אי אפשר ליצור אובייקטים מהם** - זה חוק של שפת Java  2. ++ `ITwo b = new C2();` ++  Корректно: C2 реализует ITwo, можно присвоить объект C2 переменной типа ITwo. |  |
| 3. ++ `C3 c = new C4();` ++ (upcasting).  Корректно: C4 наследует C3, поэтому объект C4 можно присвоить переменной типа C3 | |
| 4. -- `C2 d = new C4();` -- Это **שגיאת הידור** (ошибка компиляции).  Некорректно: C4 и C2 не связаны напрямую в иерархии наследования. C4 реализует интерфейс ITwo (как и C2), нельзя присвоить экземпляр C4 переменной типа C2. | |
| 5. --`C4 e = new C3();` -- **שגיאת הידור** (ошибка компиляции).  Ошибка: C3 не является типом C4. Родитель не может автоматически приводиться к дочернему типу. הורה לא יכול להמיר אוטומטית לסוג הבן | |
| 6. --`C4 f = (C4)(new C3());`-- Это **שגיאת ריצה** (ошибка выполнения).  Некорректно: приведение C3 к C4 вызовет ClassCastException во время выполнения, так как объект C3 не является экземпляром класса C4 **המרה כפויה**(**casting**) **יכולה לגרום לשגיאת ריצה** - הקוד יקמפל בהצלחה, אבל בזמן הריצה תתרחש שגיאההקומפיילר לא יכול לדעת בזמן ההידור אם ההמרה תהיה תקינה | |
| 7. `IOne g1 = new C1();` *//* 7. ++  `C4 g2 = new C4();` *//* 7. +  `g1 = g2;` *//* 7. + Код `g1 = g2;` сработал по следующим причинам:  7.1. \*\*Иерархия типов\*\*: Здесь происходит восходящее приведение типов (upcasting).  Переменная `g1` имеет тип `IOne`, а `g2` имеет тип `C4`. 7.2. \*\*C4 реализует IOne (косвенно)\*\*: Как мы обсуждали ранее, класс C4 наследуется от C3, который реализует интерфейс IThree, а IThree расширяет IOne. Таким образом,  объект типа C4 является также объектом типа IOne через цепочку наследования:  C4 → C3 → IThree → IOne 7.3. \*\*Правило полиморфизма\*\*: В Java ссылке на интерфейс (или суперкласс) можно присвоить экземпляр любого класса, который реализует этот интерфейс (или является подклассом). Таким образом, строка `g1 = g2;` означает: "Присвой ссылке на интерфейс IOne объект класса C4, который (косвенно) реализует IOne". Такое присваивание абсолютно корректно с точки зрения системы типов Java. После этой операции через переменную `g1` можно будет вызывать только методы, определенные в интерфейсе IOne  7. 4. (в данном случае только `methodOne()`), несмотря на то, что реальный объект имеет и другие методы.` | |
| 8. -- IOne h1 = new C4(); *//* 8. +-  ITwo h2 = new C2(); *//* 8. +- h2=h1; *//* 8. -- Это **שגיאת ריצה** (ошибка выполнения).  **h1 הוא מסוג IOne ו-h2 הוא מסוג ITwo** - אי אפשר להקצות משתנה מסוג אחד למשתנה מסוג אחר, גם אם שניהם אינטרפאסים | |

\_8\_ INTERFACE -- FIRST SECOND --Подьебка

|  |  |
| --- | --- |
| public class First {  public First(int k) {  System.out.print(k+":First ");}} | public class Second extends First {  public Second(int k) {  System.out.print(k+":Second");}} |

Код в Second не сработает – нет конструктора super(k); **שגיאת הידור** (ошибка компиляции).

|  |  |
| --- | --- |
| public class First {  public First(int k) {  System.out.print(k+":First ");  }} | class Second extends First {  public Second(int k) {  super(5);  System.out.print(k+" :Second "); }} |

Код сработает: 5:First 10 :Second

\_8\_ INTERFACE -- BigB –Подьебка

Здесь еще подьебка в том, что print1 есть в А1 тоже, который не реализует Inter1

|  |  |
| --- | --- |
| 1. ++ Inter1 i1; ++ |  |
| 2. -- Inter2 i2 = new Inter2();  **שגיאת הידור**  Inter i2 = new A3(); |
| 3. -- Inter i2 = new A1();  **שגיאת הידור**  Inter i2 = new A3(); - сработает |
| 4. ++ Inter1 i2 = new A2(); |
| 5. ++ Inter1 i2 = new A2();  ++ i2.print1() |
| 6. ++ A1 a = new A3();   ++ a.print1(); |
| 7. +- A3 b = new A3();   +- A1 c = b; *//* 7. +- upcasting  -- c.print2();  ((Inter2)c).print2(); - сработает, чтобы вызвать метод , вам нужно выполнить даункастинги  ((A3)c).print2(); | |
| 8. -- B1 b = new A1();**שגיאת הידור**  Не каждый объект является объектом B1 (хотя каждый B1 является ). A1A1  A1 a = new B1(); - сработает. Это апкастинг, работает без явного приведения типов | |
| 9. +- BigB bigB = new BigB(); **שגיאת ריצה**   -- bigB.all[0].print1(); -- NullPointerException  bigB.all[0] = new A1(); *//* Или любой другой класс, наследующий A1 bigB.all[0].print1(); *//* Теперь это будет работать | |

\_8\_ INTERFACE -- COMPARE

## – Transport

|  |  |
| --- | --- |
| public interface Comparable {  public int **compareTo**(Object o); }  public interface Movable {  String getSource();  String getDestination();  void move();  String getType();  String details(); } |  |
| public **abstract** class Vehicle implements Movable, Comparable {  **private** int id;  **private** String source;  **private** String destination;  public Vehicle(int id, String source, String destination) {  this.id = id;  this.source = source;  this.destination = destination;  }  public String getSource() {  return source;  }  public String getDestination() {  return destination;  }  public **abstract** String getType();  public **abstract** int **compareTo**(Object o);  public void move() {  String temp = source;  source = destination;  destination = temp;  }  public String details() {  return "Number ["+ this.id + "]\n"+ "Source = [" + source + "]\n"+  "Destination =[" + destination + "]" ;  }} | |
| public class Plane extends Vehicle{  **private** int height;  **private** String origPoint;  **private** String finalPoint;  **private** int id;  **private** String type = "Plane";   public Plane(String source, String destination, int height, int id) {  super(id, source, destination);  this.id = id;  this.height = height;  this.origPoint = source;  this.finalPoint = destination;   }   public String getType() {  return type;  }   public String details(){  return "Movable Type: " + getType()+ "\n"+  super.details() + "\n"+  "Max altitude: [" + height + "]" +"\n";  }   *//* если параметр основного объекта выше чем сравниваемого - вернет 1  @Override  public int **compareTo**(Object o) {  **if** (!(o instanceof Plane)) return 0;  Plane other = (Plane) o;  return Integer.compare(this.height, other.height);  } } | |

# \_9\_ ENUM

## Программа голосования Евровидения

В песенном конкурсе Евровидения страны получают баллы из двух источников:

* Профессиональное жюри
* Голосование зрителей

Некоторые страны получают баллы только из одного источника (например, если получили 0 от жюри).Вам необходимо создать систему, которая представляет голосование стран в финале Евровидения.

**Задачи:**

**1**) **Определите тип enum под названием VoteType с значениями JURY и AUDIENCE**

* Каждое значение получает текстовое описание (например: "Jury points", "Audience points")
* Добавьте свойство типа строка под названием: description
* Добавьте функцию getDescription(), которая возвращает описание каждого значения

**2**) **Создайте класс Country, содержащий:**

* Название страны (name) - строка
* Позиция в рейтинге (rank) - целое число
* Тип голосования (voteType) - VoteType
* Абстрактный метод getPoints, возвращающий целое число
* Метод toString, возвращающий описание данных страны

**3**) **Создайте класс JuryVoted, наследующий от Country и содержащий свойство:**

* juryPoints для баллов от жюри - целое число

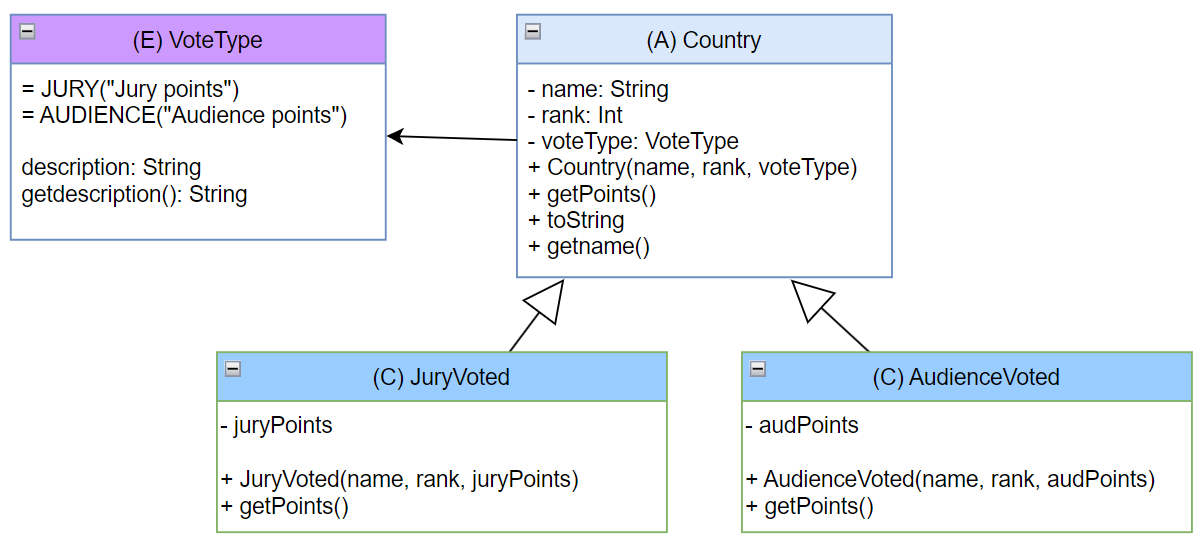
**4**) **Создайте класс AudienceVoted, наследующий от Country и содержащий свойство:**

* audPoints для баллов от зрителей - целое число

**5**) **Напишите главную функцию в классе TestEurovision, которая создает массив стран, участвовавших в финале Евровидения с голосами зрителей и жюри для каждой страны:**

**а.** Отсортируйте массив по названию страны**б.** Функция должна вывести:

* Рейтинг стран
* Тип голосования, который получила каждая страна
* Общее количество баллов каждой страны



|  |  |
| --- | --- |
| public **enum** VoteType {  *AUDIENCE*("Audience points"),  *JURY*("Jury points");  **private** final String description;   VoteType(String description){  this.description = description;  }   public String getDescription(){  return description;  } } | public **abstract** class Country {  **private** String name;  **private** int rank;  **private** VoteType voteType;  public Country(  String name, int rank, VoteType voteType  ){  this.name = name;  this.rank = rank;  this.voteType = voteType;  }  public String getName(){  return this.name;  }  public **abstract** int getPoints();   @Override  public String toString() {  return "Name: "+this.name +", rank: "+this.rank+" voteType: "+this.voteType;  }} |
| public class JuryVoted extends Country {  **private** int juryPoints;  public JuryVoted(String name, int rank, int juryPoints) {  super(name, rank, VoteType.*JURY*);  this.juryPoints = juryPoints;  }  @Override  public int getPoints(){ return juryPoints; }} | |

public class TestEurovision { public static void main(String[] args) {  
 Country[] countries = new Country[]{  
 new JuryVoted("Sweden", 1, 340),  
 new AudienceVoted("Finland", 2, 376),  
 new JuryVoted("France", 3, 280),  
 new AudienceVoted("Norway", 4, 210), };  
  
 *//* Пример сортировки по голосам (low to high)  
 Arrays.**sort**(countries, (c1, c2) ->  
 Integer.*compare*(c1.getPoints(), c2.getPoints()));  
 *//* Sort the countries by votes (high to low)  
 Arrays.**sort**(countries, (c1, c2) ->  
 Integer.*compare*(c2.getPoints(), c1.getPoints()));

*//* Пример сортировки по алфавиту by name (A to Z)  
 Arrays.**sort**(countries, (c1, c2) ->  
 c1.getName().**compareTo**(c2.getName()));

*//* Print the country details one by one  
 **for** (int i = 0; i < countries.**length**; i++) {  
 System.out.println(countries[i].toString()); } }}

# **\_10\_ Exception**

## Ex1: Mammal

**Задача: обработка исключения неправильного пола. Понятно, что если mom является самцом млекопитающего, это исключение** (**нельзя кормиться от самца**)**. Нужно:**

1. Написать код, который обрабатывает случай, когда mom является самцом
2. Создать класс WrongGenderException и использовать его
3. Показать пример использования в классе тестов TestZoo с вызовом метода nurseFrom и ловлей исключения

|  |
| --- |
| public class WrongGenderException extends Exception {  public WrongGenderException() {  super("Неправильный пол: нельзя кормиться от самца!");  }  public WrongGenderException(String message) {  super(message);  }  } |
| public void nurseFrom(Mammal mom) throws WrongGenderException {  **if** (!mom.gender.**equals**("female")) {          throw new WrongGenderException("Мама должна быть самкой!");      }      this.energy += mom.milk \* CAL\_IN\_MILK;      mom.energy -= mom.milk \* CAL\_IN\_MILK;      mom.milk = 0;  } |
| public class TestZoo {      public static void main(String[] *args*) {          Mammal baby = new Mammal("baby", "female");          Mammal dad = new Mammal("dad", "male");            try {              baby.nurseFrom(dad); *// Попытка кормиться от самца*          } catch (WrongGenderException *e*) {              System.out.println("Поймано исключение: " + e.getMessage());              e.printStackTrace();          }            System.out.println("Программа продолжается...");      }} |