# **STATIC**

|  |  |
| --- | --- |
| public static **int getCount**() **-** Статический метод  Вызывается через имя класса: **Line.getCount**()  Может обращаться только к статическим полям класса  Не требует создания объекта для вызова | public **int 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 static **int 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

}

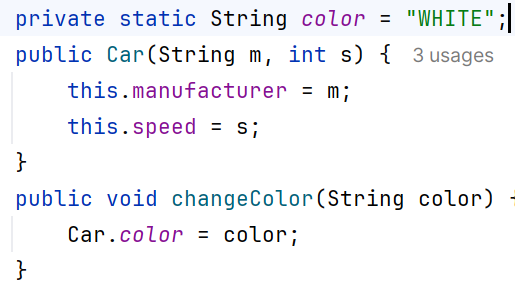
Car car1 = new Car(); // Car.color = "WHITE"

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.

# **FINAL**

# privatestaticfinal **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

# **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());

# **ARRAY - מערכים**

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

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

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

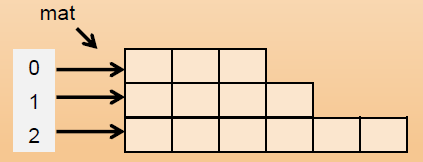
// arr[5] = 1; // ArrayIndexOutOfBoundsException - индекс 5 вне диапазона [0-4]

|  |  |  |
| --- | --- | --- |
| Пример | Использование | Описание |
| **int**[] digits ={ 0,1,2,3,4,5,6,7,8,9} | в списке инициализации (эффективнее - время вычисления) | 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) |

**:УРОВНИ ДОСТУПАבקרת גישה**

קיימות 4 אפשרויות לבקרת גישה לעצם

❑ private – הגישה מותרת רק לשיטות החברות במחלקה.

❑ protected – הגישה מותרת גם לשיטות ממחלקות יורשות.

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

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

# סיכום בנאי - CONSTRUCTOR

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.

# - 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{

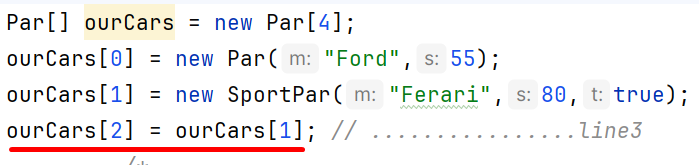
this.day = d.day;

this.month = d.month;

this.year= d.year;}

# **PRIVATE. ИНКАПСУЛЯЦИЯ עקרון ההכמסה**

|  |  |
| --- | --- |
| 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

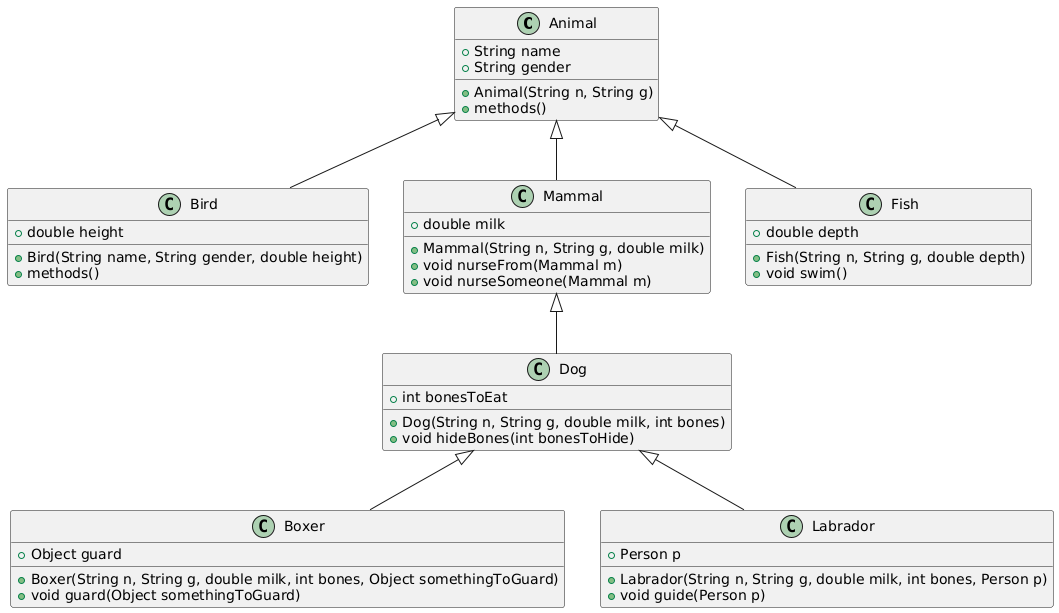
SportPar myCar = ourCars[1]; // ❌

// 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

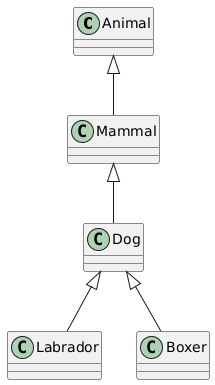
Animal myCat = new Cat(); // Создаем объект подкласса Cat

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[4];

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, ...); }}

|  |  |
| --- | --- |
| // **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;

    }

\_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());  
 }  
  
 }

\_0\_начало: 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;  
 }  
}

# \_1\_ overloading 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

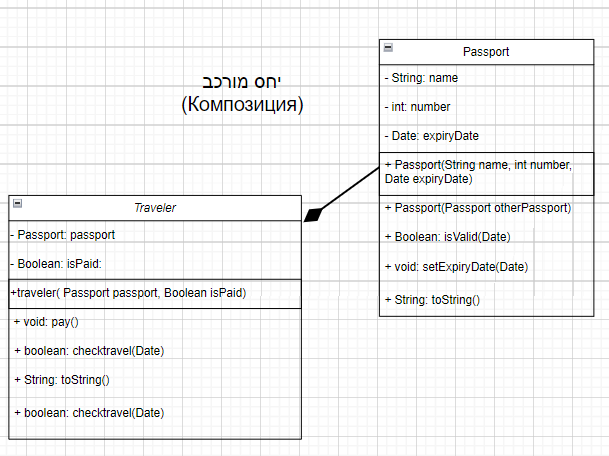
Для реализации этих классов используйте класс Date, который вы написали ранее. Вам нужно добавить к нему копирующий конструктор.

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)); } }

# \_2\_ MURKAVIM **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?

# \_2\_ MURKAVIM **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) {  
  
 }  
}

# \_2\_ MURKAVIM **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!");  
 } }}

# \_2\_ MURKAVIM 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++;  
 }  
}

# \_2\_ MURKAVIM 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());  
 }  
}

\_4\_ MAARAH 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; }  
}

\_4\_ MAARAH 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

package Exercices.EX1.\_5\_Erusha;  
import Exercices.EX1.\_1\_OverLoading.Point;  
  
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());

\_4\_ ERUSHA 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

}  
}

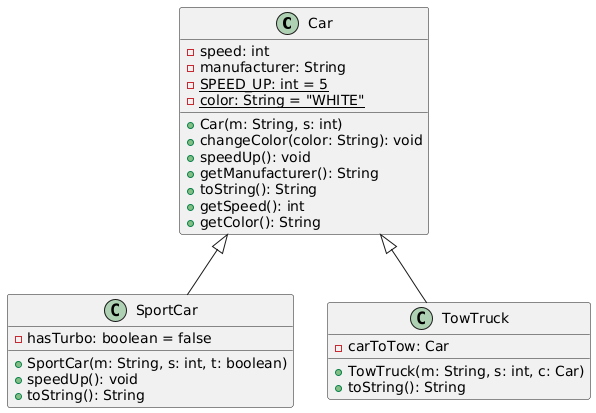
# **ERUSHA -- STATIC**

# **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 \*/  
 }}

# Упражнение 2 TOY STORE

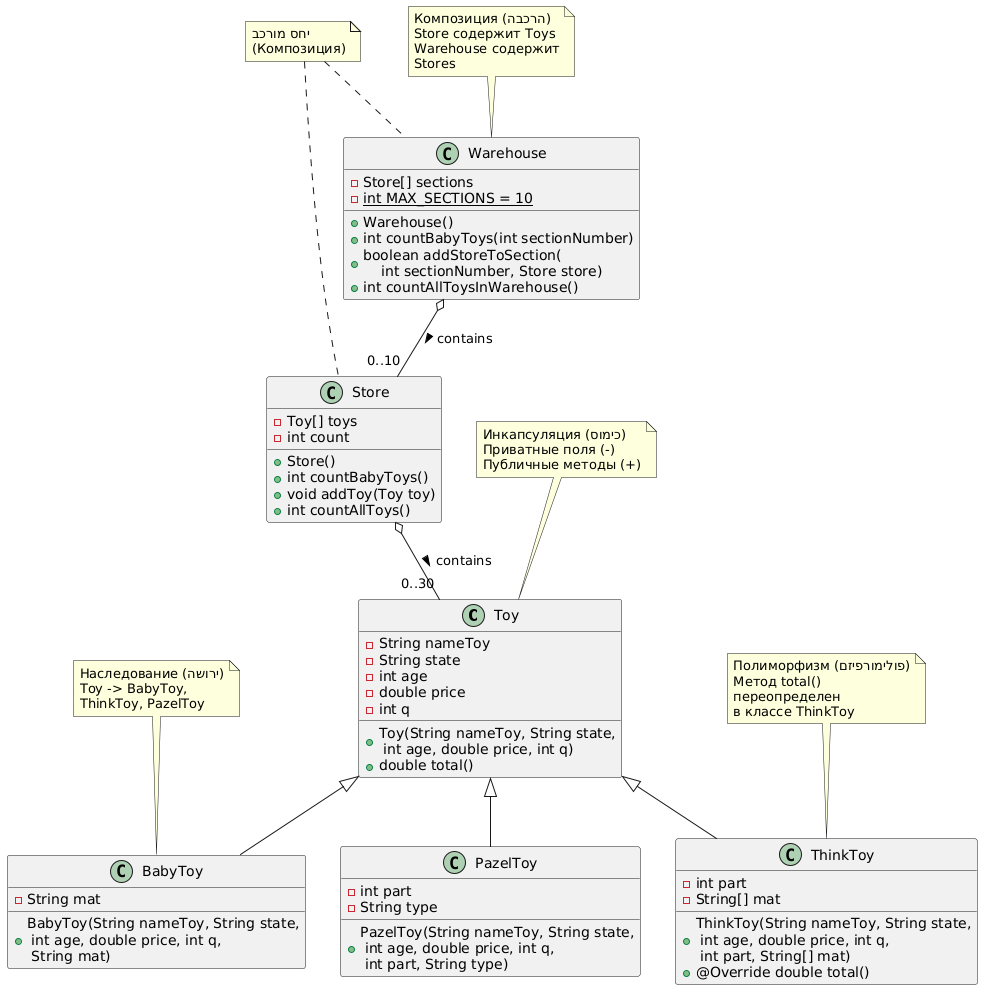
Компания по маркетингу игрушек разрабатывает проект для управления запасами игрушек в компании.  
Ниже частичное определение классов в Java: Игрушка - Toy, игрушка для малышей - BabyToy,  
развивающая игрушка - ThinkToy, игрушка-пазл - PazelToy  
  
A. Реализуйте в Java конструктор в классе BabyToy,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

# Задание 4 - 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)

# 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); }}

