

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

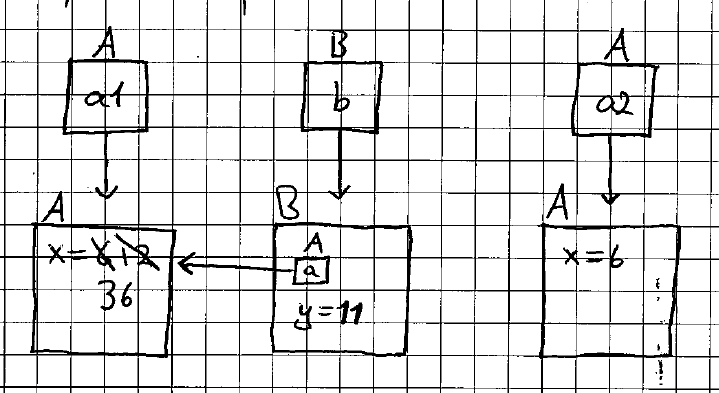
|  |  |
| --- | --- |
| class A{ | |
| private **int** x;  // Обычный конструктор  public A(**int** x){  this.x = x;  }  // **Конструктор глубокого копирования**  // בנאי העתקה עמוקה  public A (A a){  this.x = a.getX();  }  // или можно a.x, так как мы находимся в том же классе  // או אפשר a.x, כי אנחנו נמצאים באותה מחלקה | public **int** getX(){  return this.x;  // или можно return x, тут нет "затенения"  // או אפשר return x, אין כאן "הצללה"  }  public void setX(**int** x){  this.x = x;  }  public String toString() {  return "x: <"+this.x+">";  }  } |

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

A) - возникнет NullPointerException, так как this.a еще не инициализирован.

B) Правильный конструктор - создает поверхностную копию (shallow copy), что в данном случае требуется.



36

6

36 11

C) Тоже правильный - создает новую глубокую копию а, нарушает требуемую связь между объектами.

Изображение выглядит как зарисовка, диаграмма, черно-белый

Контент, сгенерированный ИИ, может содержать ошибки.

36

6

12 11

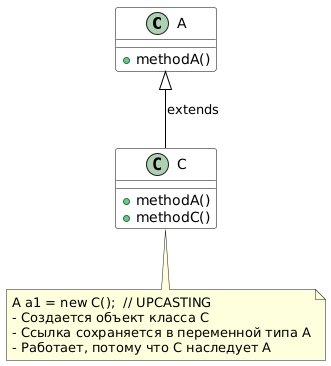
***g***)Продолжи без связи с предыдущим заданием. Дано 3 действия.

(1) A a1 = new C();

(2) B b1 = (B)(new A));

(3) **B b2 = new D**()**;**

***Предположим, что все строки кода правильные и нарисуйте диаграмму наследования. Ниже приведены несколько диаграмм наследования между классами A, B, C и D. Для каждой диаграммы нарисуйте UML диаграмму и выберите правильный ответ/решение***



(1) **A a1 = new C**()**;**

        // Здесь создается объект класса C

        // Переменной типа A присваивается ссылка на объект класса C

        // Это возможно, если класс C является наследником класса A

        // Приведение типов происходит автоматически

(восходящее преобразование **UPCASTING**)

(2) **B b1 =** (**B**)(**new A**())**;**

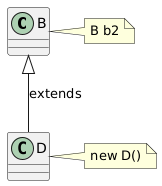


        // Создается объект класса A

        // Затем явное **ПРИВЕДЕНИЯ ТИПА** к классу B с помощью (B)

крч тут сложная хуйня, посмотри описание дальше

(3) **B b2 = new D**()**;**



        // Создается объект класса D

        // Переменной типа B присваивается ссылка на объект класса D

        // Это возможно, если класс D является наследником класса B

        // Приведение типов происходит автоматически

(восходящее преобразование, **UPCASTING**)

**Пример исполнения:**

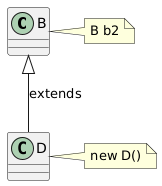
// (1) **A a1 = new C**()**; UPCASTING**  
Animal animal = new Dog(); // Dog является Animal

animal.makeSound(); // Можно вызвать общие методы Animal

// animal.bark(); // Нельзя вызвать специфические методы Dog

// (2) **B b1 =** (**B**)(**new A**())**;** **DOWNCASTING**

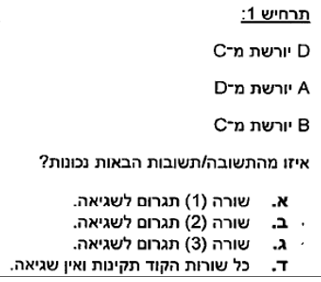
// (3) **B b2 = new D**()**; UPCASTING** // то же что и (1)



Mammal mammal = new Dog(); // Dog наследуется от Mammal

mammal.feedBaby(); // Можно вызвать методы Mammal

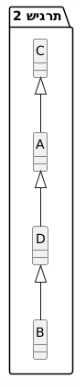
// mammal.bark(); // Нельзя вызвать методы Dog



Изображение выглядит как зарисовка, диаграмма, домкрат, шаблон

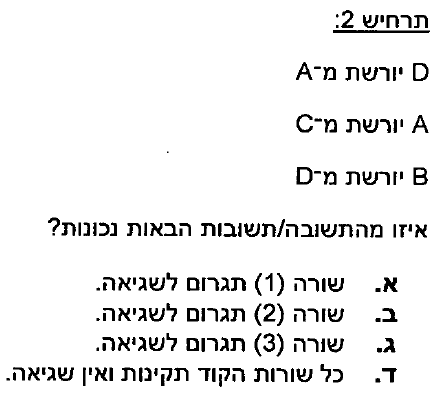
Контент, сгенерированный ИИ, может содержать ошибки.

(1) A a1 = new C(); // -



(2) B b1 = (B)(new A)); // -

(3) **B b2 = new D**()**;** // **-**

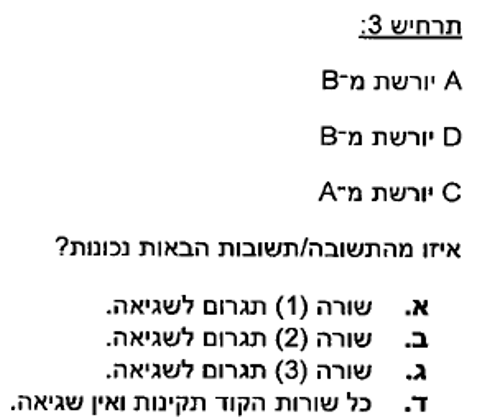
Все строки приведут к ошибке - ответ א ב ג

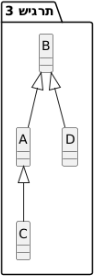
(1) A a1 = new C(); // -

(2) B b1 = (B)(new A)); // -

(3) **B b2 = new D**()**;** // **-**

Все строки приведут к ошибке - ответ א ב ג

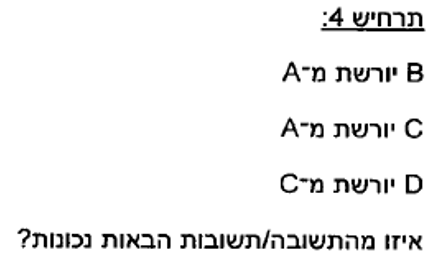




(1) A a1 = new C(); //+

(2) B b1 = (B)(new A)); // +

(3) **B b2 = new D**()**;** // **+**



Изображение выглядит как зарисовка, диаграмма, домкрат, шаблон

Контент, сгенерированный ИИ, может содержать ошибки.

!!!!!

(1) A a1 = new C(); // +

(2) B b1 = (B)(new A)); // -

(3) **B b2 = new D**()**;** // **-**

**БЕЗОПАСНОЕ ПРИВЕДЕНИЕ ТИПА -** (**2**) **B b1 =** (**B**)(**new A**))**;** // **- ???**

Приведение типов B b1 = (B)(new A()); сработает только в одном случае - когда мы на самом деле создаем объект класса B, но через переменную типа A. Вот пример:

A a = new B(); // Сначала создаем объект B, но храним его в переменной типа A

B b1 = (B)(a); // Теперь это сработает, потому что в 'a' реально хранится объект B

class Animal { }

class Dog extends Animal { }

// Это сработает:

Animal animal = new Dog(); // Создали собаку, положили в переменную Animal

Dog dog = (Dog)(animal); // Можно привести обратно к Dog, потому что там реально собака

// Это вызовет ошибку:

Animal animal2 = new Animal(); // Создали просто животное

Dog dog2 = (Dog)(animal2); // Нельзя превратить животное в собаку!

То есть приведение типов работает, только когда мы точно знаем, что объект реально является экземпляром того класса, к которому мы его приводим. Можно проверить это с помощью instanceof:

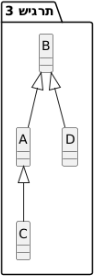
**if** (animal instanceof Dog) {

Dog dog = (Dog)(animal); // Безопасное приведение

}

Ключевой момент: всегда смотрите:

1. Что реально создается (после **new**)
2. К какому типу мы пытаемся привести
3. Есть ли прямая связь наследования между этими классами в правильном направлении

 B b1 = (B)(new A()); - что создается объект A который приводят к Б.

Поэтому в 3 тархише он сработает. Типа создали собаку и привели к ее типу животных. Нет противоречий.

Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.

В чем прикол с protectedchar **t1**? Давайте разберем:

Модификатор protected в Java дает доступ к полю:

1. Внутри того же пакета
2. В классах-наследниках (даже если они в другом пакете)

В нашем случае это важно, потому что:

1. **TwoTav** наследует **Tav** и использует **t1** в методе print()
2. Если бы поле было private, то **TwoTav** не смог бы получить доступ к **t1**
3. Если бы поле было public, то любой класс мог бы изменить значение **t1**, что нарушило бы инкапсуляцию

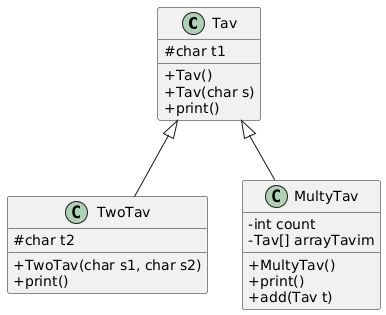
Так что protected здесь - это "золотая середина" между private и public, которая позволяет наследникам работать с полем, но защищает его от доступа извне.

В чем прикол **пустого конструктора** Tav?

|  |  |  |
| --- | --- | --- |
| Состояние массива tavim: | | |
| Индекс | Объект | Вывод print()| |
| [0] | TwoTav(A,B) | A B |
| [1] | Tav(C) | C |
| [2] | TwoTav(C,D) | C D |
| [3] | TwoTav(E,F) | E F |
| [4] | MultyTav | G H, G H, F G |

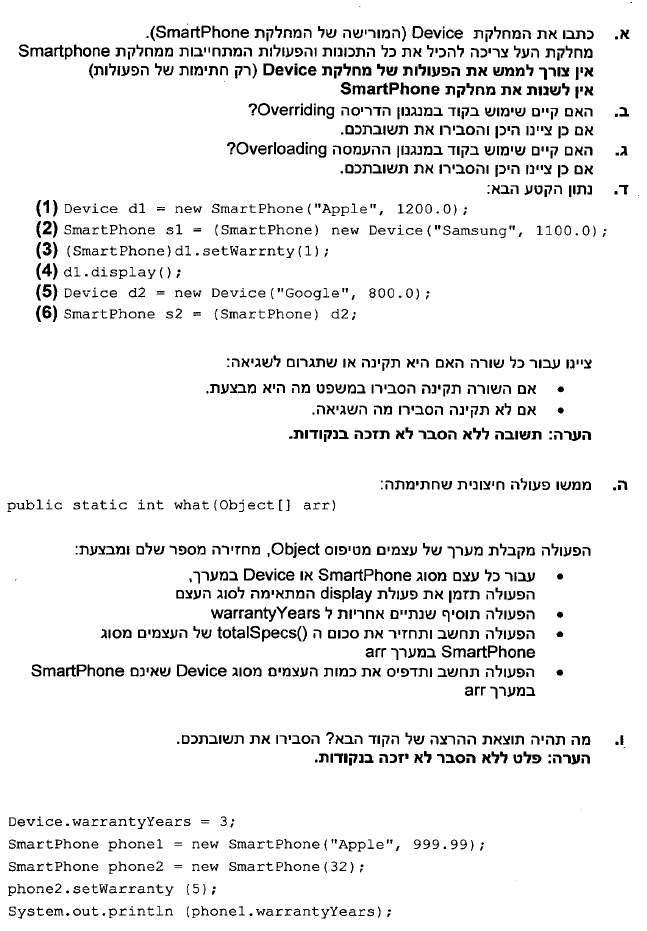
|  |  |  |
| --- | --- | --- |
| Состояние массива subtavim: | | |
| Индекс | Объект | Вывод print()| |
| [0] | TwoTav(G,H) | G,H |
| [1] | TwoTav(G,H) | G,H |
| [2] | TwoTav(F,G) | F,G |
| [3] | null |  |
| [4] | null |  |

public Tav(){  
 // благодаря пустому конструктору можно наследовать без  
 // обязательного обозначения (super)  
}  
 public static class MultyTav extends Tav {  
 private **int** count = 0;  
 private Tav[] arrayTavim;  
  
 public MultyTav() {  
 this.arrayTavim = new Tav[5];  
 }  
------------------------  
 public static void main(String[] args) {  
 MultyTav tavim = new MultyTav();  
 Tav t = new TwoTav('A', 'B');  
 tavim.add(t); // A B  
  
 t = new Tav('C');  
 tavim.add(t); // A B, C  
  
 t = new TwoTav('C', 'D');  
 tavim.add(t); // A B, C, C D  
  
 t = new TwoTav('E', 'F');  
 tavim.add(t); // A B, C ,C D, E F  
  
 MultyTav subtavim = new MultyTav();  
 TwoTav t2 = new TwoTav('G', 'H');  
 subtavim.add(t2); // G H  
 t=t2;  
  
 subtavim.add(t2); // G H, G H, F G  
 t=new TwoTav('F', 'G');  
 subtavim.add(t);  
  
 tavim.add(subtavim);  
 tavim.print(); // [A B], [C] ,[C D], [E F], ([G H], [G H], [F G])  
 // в итоге будет 5 итемов в массиве. но итемы не "соло буквы" - а объекты  
 // если бы было переполнение - была бы ошибка.  
 }  
}



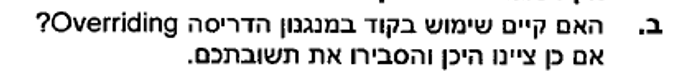
Изображение выглядит как текст, Шрифт, снимок экрана, документ

Контент, сгенерированный ИИ, может содержать ошибки.



public class Device{  
 protected String brand; -- protected т.к. получили его в наследнике через this.brand  
 private Double price; -- private просто по умолчанию.  
 private String category; -- private т.к. в коде был указан ГЕТТЕР.   
 static **int** *warrantyYears* = 10; -- static т.к. гарантия одинакова для ВСЕХ устройств.

Можно получить через Device.warrantyYears  
// Constructors  
 public Device(String brand, Double price)  
 public Device(String brand, Double price, String category)   
// Methods  
 public String getCategory();  
 public **int** calculatePerformance();  
 public double applyDiscount(double percentage);

!!!!!!метод public **int** warrantyYears() - не нужен, значение указано статиком.  
 public void display();  
   
 כן. למשל דרסנו בתת מחלקו Smarptphone. פעולות האלה:

Smartphone.display()

Smartphone.applyDiscount(double percentage)



Да, в коде есть Overloading (перегрузка методов). Давайте посмотрим все случаи:

1. В классе **Device**:

public Device(String brand, Double price)

public Device(String brand, Double price, String category)

1. В классе **Smartphone**:

public Device(String brand, Double price)

public Device(**int** cameraMP)

Изображение выглядит как текст, Шрифт, белый, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

(1) Device d1 = new Smartphone("Apple", 1200.0); // Smartphone brand: Apple

(2) Smartphone s1 = (Smartphone) new Device("Samsung", 1100.0);

(3) (Smartphone)d1.setWarranty(1);

((Smartphone)d1).setWarranty(1); - должны быть еще одни скобки. Вывода бы не было.

(4) d1.display(); // brand: Apple price: 1200.0 Smartphone

(5) Device d2 = new Device("Google", 800.0); // создаст новый девайс

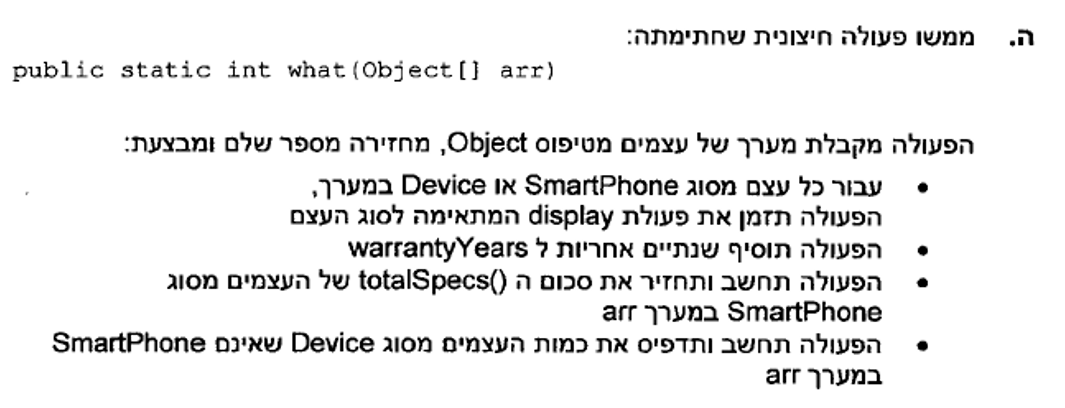
(6) Smartphone s2 = (Smartphone) d2; // нельзя **Device** к **Smartphone**

(2) Smartphone s1 = (Smartphone) new Device("Samsung", 1100.0);

Нельзя привести обычный **Device** к типу **Smartphone**. Это как сказать что любое устройство - это смартфон (часы это устройство, но не смартфон).

А вот обратное преобразование **Device d1 = new Smartphone**(**"Apple", 1200.0**)**;** работает, потому что каждый смартфон является устройством.

הבעיה היא שאנחנו מנסים להמיר Device רגיל לסוג Smartphone, אבל ל-Device אין את השדות והמתודות הספציפיות של סמארטפון (כמו cameraMP). זה כמו לומר שכל מכשיר הוא סמארטפון, מה שלא נכון לוגית (למשל, שעון הוא מכשיר, אבל הוא לא סמארטפון).



 public static **int** what(Object[] arr){

**int** sum = 0;

**int** deviceCount = 0;

            for(Object **obj** : arr) {

**if**(**obj** instanceof Device) {

                    ((Device)**obj**).display();

                    Device.warrantyYears += 2;

                    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**if**(**obj** instanceof Smartphone) {

                        sum += ((Smartphone)**obj**).totalSpecs();

                    } **else** {

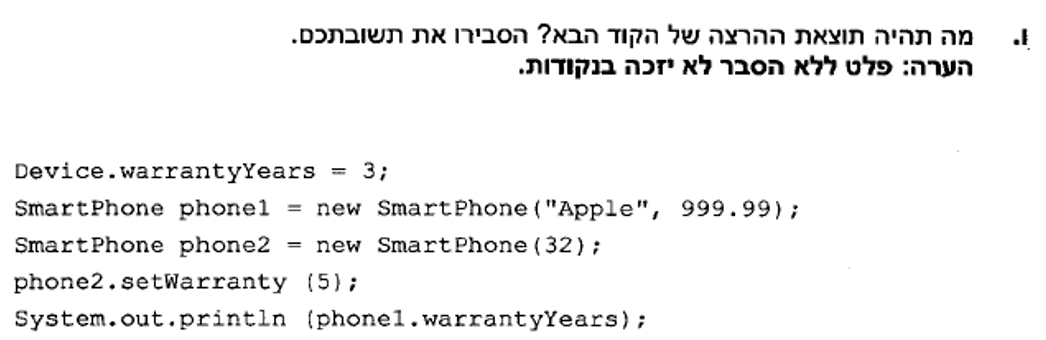
                        deviceCount++;

                    }

                }

            }

            return sum + deviceCount;



 Device.warrantyYears = 3;          // Устанавливаем статическую переменную warrantyYears в 3

            Smartphone phone1 = new Smartphone("Apple", 999.99); // Создаем первый смартфон

            Smartphone phone2 = new Smartphone(32); // Создаем второй смартфон с камерой 32MP

            phone2.setWarranty(5);

            System.out.println(phone1.warrantyYears); // 3