# 1 вопрос (30 баллов)

В этом вопросе 4 пункта. Между нет никакой связи. Ответьте на вопросы.

Изображение выглядит как символ, линия, зарисовка, оригами

Содержимое, созданное искусственным интеллектом, может быть неверным.

1.А) Дано классы X и Y. Y наследует от X. Знаем, что следующая строка компилируется верно:

**X x1 = new Y**()**;**

Напротив каждого утверждения оцените правильно или ошибка. Обьясните ваши ответы. Если ошибка – почините его, если возможно.

|  |  |
| --- | --- |
| i) Object o = x1;  + | (i -לא ניתן לכתוב את ההוראה, מכיוון שדרושה המרה מפורשת.  Невозможно выполнить инструкцию, так как требуется явное преобразование (типов) |
| Нет ошибки, потому что X наследует от Object. upCast (i - | |
| ii) X x2 = x1;  + | (ii - לא ניתן לכתוב את ההוראה, מכיוון שדרושה המרה מפורשת. |
| Нет ошибки, потому что параметр х1 это типа Х | |
| iii) Y y1 = x1; - | (iii + לא ניתן לכתוב את ההוראה, מכיוון שדרושה המרה מפורשת. |
| Y y1 = (Y) x1; + | אפשר לעשות רק עם המרה מפורשת בחזרה ל-Y |

1.B) Есть 4 строки, которые компилируются. У нас есть 5 вариантов с деревом наследования. Нужно нарисовать UML, если нельзя скомпилировать – обяьсните почему. Напишите вид ошибки.

|  |
| --- |
| (1) G g = new E(); |
| (2) F f1 = (F)(new E()); |
| (3) F f2 = new D(); |
| (4) E e = new D(); |

|  |  |  |
| --- | --- | --- |
|  | I | |
| (1) G g = new E(); | שגיאת חידור (compilation). G יורשת מ .E |
| (2) F f1 = (F)(new E()); | שגיאת ריצה. F יורשת מ .E ומנסים המרה למטה. |
| (3) F f2 = new D(); | שגיאת חידור (compilation). F אין קשר ל .D |
|  | II | |
| Все работает | |
|  | III | |
| (1) G g = new E(); | שגיאת חידור (compilation). G יורשת מ .E |
| (2) F f1 = (F)(new E()); | שגיאת ריצה. F יורשת מ .E ומנסים המרה למטה. |
|  | III | |
| Невозможное наследование | |
| Изображение выглядит как диаграмма, дизайн  Содержимое, созданное искусственным интеллектом, может быть неверным. | IV | |
| (1) G g = new E(); | שגיאת חידור (compilation). G יורשת מ .E |
| (2) F f1 = (F)(new E()); | שגיאת ריצה. F יורשת מ .E ומנסים המרה למטה. |
|  | V | |
| (2) F f1 = (F)(new E()); | שגיאת ריצה. F יורשת מ .E ומנסים המרה למטה. |
| (3) F f2 = new D(); | שגיאת חידור (compilation). F אין קשר ל .D |

1.C) Выберите один из следующих вариантов и укажите, какой из них является наиболее подходящим в соответствии с принципами объектно-ориентированного программирования (ООП). Обоснуйте свой ответ.

|  |  |
| --- | --- |
| public class City{  \_\_\_(\*)\_\_\_ **int** employees = 20;  }  public class Company extends City{  public void poll(){  syso(”There are ” + employees”);  }} | 1) private, public или protected. 2) public или private. 3) public или protected. **4**)protected**.** 5) public. 6) Написать только private. |

מגדירים את התכונה רק כ- protected. כך שרק מחלקות שיורשים מ- City יהיה להן גישה ישירה לתכונה.

public тоже скомпилирует, но это нарушит не «принцип», а скорее пожелание инкапсуляции – где доступ должен быть минимально необходимым.

1.D) Исходя из определения класса, реализуйте в мейне команды, чтобы получить такой вывод:

// 1) val 3, count 5

// 2) val 2, count 5

// 3) val -999, count 5

// 4) val 5, count 5

|  |  |
| --- | --- |
| public class Stam {  private **int** val;  private static **int** *count* = 0;  public Stam() {  *count*++;  this.val=*count*;  }  public Stam(boolean over){  **if** (!over){  *count*++;  this.val = *count*;  }**else**{  this.val = -999;  } }  public void display(){  System.out.println("val=" + val+", count ="+*count*);  }{ | Stam s1 = new Stam(); Stam s2 = new Stam(); Stam s3 = new Stam(); Stam s4 = new Stam(); Stam s5= new Stam();  // 1) val 3, count 5 s3.display();  // 2) val 2, count 5 s2.display();   Stam d5 = new Stam(true);  // 3) val -999, count 5 d5.display(); // 4) val 5, count 5 s5.display(); |

הבנאי הראשון תמיד סופר ומוסיף כמה אוביאקטים אותחלו מסוג Stam. ושומר את המספר הזו, בערך של val.

בבנאי השני שמקבל ערך בוליאני, אם הארך TRUE, אפשר לצור את האובריאקט בלי שינוי לסופר, אבל ערךval 999 -

# 1 вопрос (30 баллов): ABCDE с экзамена.

|  |  |
| --- | --- |
| public class A {  protected int x;  public A(){  this.x = 9;  System.*out*.println("A.x = "+this.x);  }  public A(int x){  this.x = x;  System.*out*.println("A.x = "+this.x);  }  public int getX(){ return this.x;}  public int foo(){return this.x;} } | public class B extends A {  public B(){}  public B(int x){  super(x);  }  public int foo(){return this.x+1;} } |
| public class C extends B {  public C(){}  public C(int x){  super(x);  }  public int foo(){return this.x+2;}  public int bar(){ return this.x;} } | public class D extends C {  public D(){  this.x++;  System.*out*.println("D.x = " + this.x);  }  public D(int x){  super(x);  System.*out*.println("D.x = "+this.x);  }  public D(int x, int y){  super();  this.x = this.x + x + y;  System.*out*.println("D.x = "+this.x);  }  public int foo(){return this.x-1;}  } |
| public class E extends C {  public E(){}  public int bar(){ return this.x;} }  Изображение выглядит как диаграмма, дизайн  Содержимое, созданное искусственным интеллектом, может быть неверным. |

A) Понять какого типа переданный оъект.  
 *\* Метод возвращает: Если m типа A то 1, B - 2, C - 3, D - 4, E - 5  
 \* Чтобы проверить тип объекта, следует использовать методы getX, bar, foo.  
 \* Нельзя использовать оператор instanceof и методы класса Object.  
 \* Нельзя изменять классы A, B, C, D, E.  
 \* Предполагается, что m принадлежит одному из классов A, B, C, D, E и не равен null.  
\*/*public static int getType(Object m){  
 *//* Сначала пытаемся проверить на самые "продвинутые" типы: C, D, E.  
 *//* Их общая черта - наличие метода bar(). У A и B его нет.  
 *//* Поэтому мы используем try-catch: если приведение к типу C не удастся,  
 *//* значит, это точно A или B.  
 try {  
 *//* Приводим объект к типу C, чтобы получить доступ к методам.  
 *//* Если 'm' - это A или B, здесь произойдет ClassCastException.  
 C obj = (C) m;  
 int fooValue = obj.foo();  
 int barValue = obj.bar();  
  
 *//* Теперь у нас есть значения foo() и bar(), и мы можем различить C, D, E.  
  
 *//* Проверяем на E: у него foo() от C (x+2), а bar() свой (x+1).  
 *//* Поэтому foo == bar + 1  
 if (fooValue == barValue + 1) {  
 return 5; *//* Это E  
 }  
  
 *//* Проверяем на D: у него foo() свой (x-1), а bar() от C (x).  
 *//* Поэтому foo == bar - 1  
 if (fooValue == barValue - 1) {  
 return 4; *//* Это D  
 }  
  
 *//* Если это не D и не E, а приведение к C удалось, значит, это C.  
 return 3; *//* Это C  
  
 } catch (ClassCastException e) {  
 *//* Мы попали сюда, значит, 'm' - это не C и не его потомки (D, E).  
 *//* Остались только варианты A или B.  
 *//* Приводим к типу A, так как B является его потомком.  
 A obj = (A) m;  
 int fooValue = obj.foo();  
 int xValue = obj.getX();  
  
 *//* Проверяем на B: у него foo() свой (x+1), а getX() от A (x).  
 *//* Поэтому foo == getX + 1  
 if (fooValue == xValue + 1) {  
 return 2; *//* Это B  
 }  
  
 *//* Если это не B, то по условию задачи остается только A.  
 return 1; *//* Это A  
 }  
}  
public static void main(String[] args) {  
 A a = new A(10); B b = new B(10); C c = new C(10); D d = new D(10);  
 System.*out*.println("Тип объекта a: " + *getType*(a)); *//* Ожидаем 1  
 System.*out*.println("Тип объекта b: " + *getType*(b)); *//* Ожидаем 2  
 System.*out*.println("Тип объекта c: " + *getType*(c)); *//* Ожидаем 3  
 System.*out*.println("Тип объекта d: " + *getType*(d)); *//* Ожидаем 4

## С) – что выведет компилятор?

A a1 = new B(); // A.x = 9  
A a2 = new E(); // A.x = 9  
A a3 = new D(); // A.x = 9 D.x = 10  
A a4 = new D(5); // A.x = 5 D.x = 5  
A a5 = new D(3,7); // A.x = 9 D.x = 19

## S10\_Coach – big MAAKAV

|  |  |  |
| --- | --- | --- |
| public class Coach {  protected int num;  protected int sessions;  protected String specialty;  public Coach(int num, int sessions, String specialty) {  this.num = num;  this.sessions = sessions;  this.specialty = specialty;  }  public void setNum(int n) {  this.num = n; } | public Coach(Coach other) {  this.num = other.num;  this.sessions = other.sessions;  this.specialty = other.specialty; }  @Override public String toString() {  return "Coach: " + this.num + ", sessions: " + this.sessions + ", specialty: " + this.specialty; } | |
| public void reduceSessions(int s) {  if (s > 0) {  this.sessions = Math.max(0, this.sessions - s);  }} |
| public class Athlete extends Coach {  **private** Coach personalCoach;   public Athlete(int n, int sessions, Coach personalCoach) {  super(n, sessions, "Athlete");  this.personalCoach = personalCoach;  this.sessions = Math.max(0, personalCoach.sessions);  }  public String toString() {  return "Athlete: " + num + ", sessions: " + sessions +  ", Coach: " + personalCoach.num +  ", sessions: " + personalCoach.sessions +  ", specialty: " + personalCoach.specialty; }} | |  |

А B)Задание: исходя из этого кода распиши что будет до изменения, и что – после. Сделай таблицу маакав.

|  |  |
| --- | --- |
| Coach[] c = new Coach[4]; c[0] = new Coach(123, 12, "Jym"); c[1] = new Coach(234, 8, "CrossFit"); c[2] = new Athlete(345, 6, new Coach(c[0])); c[3] = new Athlete(456, 10, c[1]);  System.*out*.println("\nOld Team:"); **for** (int i = 0; i < c.length; i++) {  System.*out*.println(c[i]); }  c[0] = new Coach(567, 8, "Studio"); c[1].setNum(678); c[2].reduceSessions(3); c[3].reduceSessions(11); System.*out*.println("\nNew Team:"); **for** (int i = 0; i < c.length; i++) {  System.*out*.println(c[i]); } |  |

Old Team:

Coach: 123, sessions: 12, specialty: Jym

Coach: 234, sessions: 8, specialty: CrossFit

Athlete: 345, sessions: 12, Coach: 123, sessions: 12, specialty: Jym

Athlete: 456, sessions: 8, Coach: 234, sessions: 8, specialty: CrossFit

New Team:

Coach: 567, sessions: 8, specialty: Studio

Coach: 678, sessions: 8, specialty: CrossFit

Athlete: 345, sessions: 9, Coach: 123, sessions: 12, specialty: Jym

Athlete: 456, sessions: 0, Coach: 678, sessions: 8, specialty: CrossFit

C) Какие механизмы объектно-ориентированного программирования были использованы в этом вопросе? Укажите >= 3, объясните, где именно в коде они используются.  
   
  
**1. Наследование (Inheritance)**

* **Объяснение:** Наследование — это механизм, который позволяет одному классу (дочернему) перенимать свойства и методы другого класса (родительского). Это способствует повторному использованию кода и построению иерархии классов.
* **Где в коде:**  public class Athlete extends Coach

**2. Полиморфизм (Polymorphism)**

* **Объяснение:** Полиморфизм (в данном контексте) позволяет объектам дочерних классов использоваться там, где ожидаются объекты родительского класса, и при этом вызывать свою, специфическую реализацию методов.
* **Где в коде:**

1. **Массив родительского типа:** В классе TestEx2 создается массив типа Coach, который хранит как объекты Coach, так и объекты Athlete.

**Строка 4:** Coach[] c = new Coach[4];

**Строки 8-9:** c[2] = new Athlete(...) и c[3] = new Athlete(...). Здесь объект Athlete неявно приводится к типу Coach.

1. **Переопределение методов (Method Overriding):** Оба класса имеют метод toString(). При вызове System.out.println(c[i]) для каждого элемента массива вызывается та версия метода toString(), которая соответствует фактическому типу объекта (либо Coach, либо Athlete).

**3. Композиция (Composition)**

**Объяснение:** Композиция — это создание сложных объектов путем включения в них объектов других классов. Это отношение типа "has-a" (у объекта "есть" другой объект).

**Где в коде:** Класс Athlete содержит в себе объект класса Coach в качестве одного из своих полей. **Строка 4:** private Coach personalCoach;

# 2 вопрос (35 баллов)



**А.** Для каждого из пяти представленных ниже классов необходимо определить:

**a.** Заголовок (сигнатуру) класса

**b.** Свойства (атрибуты/поля) класса

**c.** Заголовки (сигнатуры) всех методов, которые должны быть в классе.(**15 баллов**)

|  |  |
| --- | --- |
| public interface Colorable {  void colortofill(String color); } | public interface Rotatable {  void toleft(double degree);  void toright(double degree); } |
| public class Figure2D {  protected String color;  protected Point coord1; } | public class Circle extends Figure2D implements Colorable{  private **int** radius;  public void colortofill(String color) {} } |
| public class Square extends Figure2D implements Colorable, Rotatable{  protected **int** radius;  public void colortofill(String color) {};  public void toleft(double degree) {}  public void toright(double degree) {} } | public class Line extends Figure2D implements Rotatable{  private Point coord2;  public void toleft(double degree) {}  public void toright(double degree) {} } |
| public class Point {  private **int** x;  private **int** y; } |  |

**Б.**  Для каждой из строк (1–15) необходимо указать:

* Является ли строка корректной или нет.
* Если строка некорректна — необходимо обосновать причину и, если возможно, предложить исправление.
* Указать тип ошибки: ошибка компиляции / ошибка времени выполнения



|  |  |
| --- | --- |
| 1. Figure2D c = new Circle("red", new Point(50,50), 8); | + |
| 2. Line line = new Line("blue", new Point(100,100), new Point(200,200)); | + |
| 3. Square d2 = new Square("orange", new Point(45,45), 4); | + |
| 4. Colorable col1 = new Colorable();   * Нельзя создать объект из интерфейса. | compilation error |
| 5. Rotatable rot1; | + |
| 6. Colorable col2 = d2; - пример המרה מרומזת | + |
| 6.1. col2.toleft(5); - не смотря на то, что d2 – квадрат, компилятор  Видит в нем только объект интерфейса Colorable. | compilation error |
| 7. rot1 = line; | + |
| 8. col2 = (Colorable)c; | + |
| 9. rot1.setColor("green");   * Нет такого метода для него. Можно rot1.toleft() | - compilation error |
|  |
| 10. col2.colortofill("red"); | + |
| 11. col2.toleft(Math.PI/2);  сol2 после 8й строки – это ТОЛЬКО объект интерфейса Colorable | - compilation error |
|  |
| 12. ((Square)d2).toleft(Math.PI/4); | + |
| 13. Figure2D d = c; | + |
| 14. ((Line)d).setColor("red");  המרה מפורשת למטה לא לצורך שיחזור.  ((circle)d).setColor(“red”) – а так можно. | - runtime error |
| 15. d.colortofill("Yellow");  Вы пытаетесь вызвать метод colortofill() у переменной d.  Компилятор смотрит на **тип переменной d**, а он — Figure2D.  Компилятор проверяет: "Есть ли в классе Figure2D метод colortofill()?   * **Ответ: Нет.** Метод colortofill() определен в интерфейсе Colorable, а не в родительском классе Figure2D.   ((Colorable)d).colortofill("Yellow") – а так можно. | - compilation error |



**В.** Напишите внешнюю функцию с именем handleFigures, которая получает массив объектов Figure2D и выполняет следующие действия:

* Печатает точку cood1 всех фигур синего цвета ("blue").
* Если фигуру можно залить, закрашивает ее в красный цвет ("red").
* Если фигуру можно вращать, поворачивает ее на 30 градусов влево.
* Если фигура является Circle (Кругом), печатает радиус круга.
* Если это Line (Линия), печатает ее вторую точку.

Функция возвращает количество фигур, которые можно вращать, но нельзя залить.

public static **int** handleFigures(Figure2D[] arr) {  
 Figure2D curr;  
 **int** count = 0;  
 for(**int** i=0; i< arr.**length**; i++) {  
 curr = arr[i];  
 // 1: Напечатать координаты синих фигур  
 **if** ("blue".**equals**(curr.getColor())) System.out.print(curr.getPoint());

// либо  
**if** (curr.getColor().**compareTo**("blue")==0) System.out.print(curr.getPoint());

// 2: Если форму можно заполнить, заполнить цветом "red"  
 **if** (curr instanceof Colorable) {  
 Colorable c = (Colorable) curr;  
 c.colortofill("red");  
 }  
 // 3: Повернуть фигуру на 30 градусов влево, если возможно  
 **if** (curr instanceof Rotatable) {  
 Rotatable r = (Rotatable) curr;  
 r.toleft(30);  
 }  
 // 4: Если это Circle, напечатать радиус  
 **if** (curr instanceof Circle) System.out.println( ((Circle)curr).getRadius() );  
 // 5: Если это Line, напечатать вторую точку  
 **if** (curr instanceof Line) System.out.println(((Line) curr).getCoord2());  
 // 6: Подсчет фигур, которые можно повернуть, но нельзя закрасить  
 **if** (curr instanceof Rotatable && !(curr instanceof Colorable)) {  
 count++;  
 }  
 }  
 return count;  
}

# 3 вопрос (30 баллов)

Есть 2 класса А и B

|  |  |
| --- | --- |
| public class A {  private **int** val;  public A(){  this.val=1;  }  public A(**int** val){  this.val = val;  }  public **int** getVal(){  return this.val;  } | public class B extends A {  private String st; public B() {  this.st = "B"; } public B(String st, **int** val){  super(val);  this.st=st;  } public String getSt(){  return this.st; } |
| public boolean **equals**(Object other){  System.out.println("A-Object");  **if** (other instanceof A)  return this.val == ((A)other).val;  return false;  } } |
| public boolean **equals**(Object other) {  System.out.println("B-object");  **if** (other instanceof B){  return this.st.**equals**( ((B)other).st) && this.getVal() == (((B) other).getVal());  }  return false;  } |
| public boolean **equals**(A other){  System.out.println("B-A");  **if** (other instanceof B)  return this.st.**equals**( ((B)other).st) && this.getVal()==((B)other).getVal();  return false;  }} |
| public boolean **equals**(B other){  System.out.println("B-B");  return this.st.**equals**(other.st) && this.getVal()==other.getVal();  }} |

A) нарисовать маакав на строки:

|  |  |
| --- | --- |
| A a1 = new A(); |  |
| A a2 = new A(3); |  |
| A ab = new B(); |  |
| B b1 = new B("B", 1); |  |
| B b2 = new B("B", 3); |  |

Б) что напечатает в строках и обьясните:

1) **if** (a1.**equals**(b1)) System.out.println(11); // **"A-**Object**", 11**

A-Object – למחלקה A קיים דריסה, לפעולת 11 **equals** של מחלקה Object, לכן הולכים לשם.

2) **if** (b1.**equals**(a1)) System.out.println(12); // **"B-A".**

יש פעולה **equals** שמקבלת A. אבל היא מחזירה false המקרנ הזה.

Т.к. а1 – не экземпляр B. false на вывод 12.

3) **if** (a1.**equals**(ab)) System.out.println(13); // **"A-**Object**", 13**

יש למחלקה A פעולת **equals** שמרבלת Object אבל הפניה שנשלחו היא בפעול instanceof A לכן הוחזר true

4) **if** (ab.**equals**(a1)) System.out.println(14); // **"B-object"**

**ПОДЪЕБКА!! – ab – это overloading объект типа B. И на него в JAVA по умолчанию вызывается equals**(Object **obj**)**. Внутри сраного B – 3 метода, и когда у нас подьебаный ab – то** Object **obj, а не как в** стандартном2) случае

5) **if** (b1.**equals**(ab)) System.out.println(15); // **"B-A", 15**

**ПОДЪЕБКА!! – ab – здесь это ПЕРЕМЕННАЯ типа А.** A ab = new B();

6) **if** (ab.**equals**(b1)) System.out.println(16); // “B-object”, 16

**ПОДЪЕБКА!! – ab – это overloading объект типа B.**

7) **if** (a1.**equals**(a2)) System.out.println(17); // A-Object

8) **if** (b1.**equals**(b2)) System.out.println(18); // B-B  
  
В) Напишите функцию matchCount которая получает массив типа Object[]. ф-я считает и возвращает сколько из объектов в массиве это экземпляры класса “B” и сколько из них отвечают двум требованиям: val >2 и st!=”B”.

public static **int** matchCount(Object[] arr){  
 **int** count = 0;  
 **if** (arr == null || arr.**length** == 0) return 0;   
 for (**int** i=0; i<arr.**length**; i++){  
 Object **obj** = arr[i];  
 **if** (**obj** instanceof B){  
 B b\_instance = (B) **obj**;  
 **if** (b\_instance.getVal()>2 && ( !"B".**equals**(b\_instance.getSt()))) count++;  
 } } return count;}

**Г)** Используется ли принцип **Перегрузки (Overloading)** **העמסה** - в классах A и B? **Ответ:Да, используется.**

1. **В конструкторах (в обоих классах):**

* В классе A есть два конструктора: A() и A(int val). Это перегрузка конструктора.
* В классе B есть два конструктора: B() и B(String st, int val). Это тоже перегрузка конструктора.

1. **В методе equals (в классе B):**

* В классе B вы написали **три разных метода** с именем equals, но с разными параметрами:
* equals(Object other)
* equals(A other)
* equals(B other)
* Это классический пример **перегрузки метода**.

**Д)** Используется ли принцип **Переопределения (Overriding) דריסה**  в классах A и B? **Ответ:Да, используется.**

1. **Класс A переопределяет метод из Object:**

* В классе A есть метод public boolean equals(Object other).
* Этот метод имеет точно такую же сигнатуру (имя и параметры), как и метод equals в самом главном классе Object.
* Следовательно, класс A **переопределяет** стандартное поведение метода equals.

1. **Класс B переопределяет метод из A:**

* В классе B тоже есть метод public boolean equals(Object other).
* Поскольку B наследуется от A (extends A), этот метод **переопределяет** метод equals(Object other) из класса A. Класс B предоставляет свою, более специфичную логику сравнения.