Which three piece of codes are equivalent to line 2?

**public interface** Foo {  
 **int *k*** = 4; */\* Line 2 \*/* 1. **final int *k*** = 4;  
 2. **public int *k*** = 4;  
 3. **static int *k*** = 4;  
 4. **abstract int *k*** = 4; //not allowed here  
 5. **volatile int *k*** = 4; //not allowed here  
 6. **protected int *k*** = 4; //not allowed here  
  
}

**// 1, 2, 3.**

abstract , volatile , protected, private не можна використовувати в public інтерфейсі.

Which are valid declarations of a char?

1. **char** c1 = 064770;

7. char b = 10;

8. char a = 0xFFFF;

3. **char** c3 = 0xbeef;

6. **char** c6 = **'\uface'**;

2. **char** c2 = **'face'**; //no  
4. **char** c4 = \u0022; //no  
5. **char** c5 = **'**\i**face'**; //no

**public** **class** Test {       
    {   
        System.out.println("1");   
    }   
   
    Test () {   
        System.out.println("2");   
    }   
       
    **static** {   
        System.out.println("3");   
    }   
       
    {   
        System.out.println("4");   
    }   
   
    **public** **static** **void** main(String[] args) {   
        **new** Test();   
    }   
}

// 3 1 4 2

Последовательность приблизительно такая, сначала Класслоадер загружает класс вместе с блоком статической инициализации и на консоль выводится 3, потом при создании объекта выводятся блоки 1 и 4, согласно их расположению в классе, потом выполняется собственно сам конструктор. Собственно компилятор вам и так все покажет :)

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        **char** ch = '0';   
        **while** (ch <= 9)   
            System.out.print(ch++);   
    }   
}

*Пояснение: Начальным значением переменной ch является число 48 - код символа '0' в кодировочной таблице.  
Это число больше 9, поэтому тело цикла не будет выполнено ни разу.*

**class** A {}   
**class** B **extends** A {}   
   
**public** **class** Test {    
    **static** **public**  **void** main(String args[]) {   
        B b = **new** B();   
        A a = b;   
        **if** (a.hashCode() == b.hashCode())   
            System.out.print("Passed");  //Passed  
    }   
}

*Пояснение: Ссылки a и b указывают на один и тот же объект, поэтому в консоль выведется строка "Passed". Экземпляру класса A можно присвоить экземпляр класса B в силу полиморфизма.*

**public** **class** Test {   
    **static** { i = 5; }   
    **static** **int** i = 6;   
   
    **public** **static** **void** main(String[] args) {   
        System.out.println(i); // 6  
    }   
}

*Пояснение: Блоки статической инициализации и инициализация статических переменных выполняются в том же порядке, в котором они идут в тексте класса. Поэтому первой будет выполнена команда i=5, а затем i=6.*

**public** **class** Starter **extends** Thread {   
    **private** **int** x = 2;   
   
    **public** **static** **void** main(String[] args) **throws** Exception {   
        **new** Starter().makeItSo();   
    }   
   
    **public** Starter() {   
         x = 5;   
         start();   
    }   
   
    **public** **void** makeItSo() **throws** Exception {   
        join();   
        x = x - 1;  // 9  
        System.out.println(x);   
    }   
   
    **public** **void** run() { x \*= 2; }   
 }

*Пояснение: Сначала происходит создание нового объекта - new Starter() - и сразу же вызывается его метод - makeItSo()*

**public** **class** Test {   
    **private** **static** Boolean b1, b2;   
       
    **public** **static** **void** main (String[] args) {   
        **if** (b1 || !b2 || !b1 || b2) {   
            System.out.println(**true**);   
        }   
        **else** {   
            System.out.println(**false**);   
        }   
    }   
}

*Пояснение: При использовании объектов-обёрток в операциях, предназначенных для примитивов, вызываются методы, возвращающие примитивное значение. В данном случае booleanValue(). Это вызовет выбрасывание NullPointerException потому что класу обертке Boolean по умолчанию присваевается значение null.*

Контракт методов equals и hashCode такой: "equal objects must have equal hash codes".

**class** A {   
02.     **public** **void** process() { System.out.print("A,"); }   
03. }   
04. **public** **class** B **extends** A {   
05.     **public** **void** process() **throws** IOException {  //compilation error  
06.         **super**.process();   
07.         System.out.print("B,");   
08.         **throw** **new** IOException();   
09.     }   
10.    
11.     **public** **static** **void** main(String[] args) {   
12.         **try** { **new** B().process(); }   
13.         **catch** (IOException e) { System.out.println("Exception"); }   
14.     }   
15. }

*Пояснение: При переопределении метода нельзя добавлять проверяемое (checked) исключение.*

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        **byte** a = 3;   
        **short** b = 4;   
        compute(a, b);   
    }   
   
    **public** **static** **void** compute(**short** x, **short** y) {   
        System.out.println("Short: " + (x + y) + (x + y)); //77  
    }   
}

*Пояснение: "Short: " + (x + y) + (x + y) состоит из двух опреаторов +, которые левоассоциативны, то есть выражение эквивалентно (("Short: " + (x + y)) + (x + y)). Когда одним из аргументов + является строка, JVM выполняет конкатенацию строк. См. JLS 15.18.1*

**public** **class** Main {   
    **public** **static** **void** main(String args[]) {   
        **byte** b = 0;   
        **while** (++b > 0);   
        System.out.println(b); //-128  
    }   
}

*Пояснение: Переменные типа byte хранят значение в диапазоне -128 .. 127. Когда значение переменной "b" в цикле дойдет до максимального значение этого типа (127), тогда следующий инкремент (++) установит значение на минимальное (-128). Так сказать начнёт новый отсчёт.*

**class** Base {   
    **public** String name = "Base";   
    **public** String getName() {   
        **return** name;   
    }   
}   
   
**class** Sub **extends** Base {   
    **public** String name = "Sub";   
    **public** String getName() {   
        **return** name;   
    }   
}   
   
**public** **class** Program {   
    **public** **static** **void** main(String[] args) {   
        Sub s = **new** Sub();   
        Base b = s;   
        System.out.println(b.name);  //Base  
    }   
}

*Пояснение: При обращении к полям используется тип ссылки, а не тип реального объекта. Если бы вместо обращения к полю был вызов переопределенного метода получения значения поля, то благодаря позднему связыванию мы получили бы значение из реального объекта.*

String data[];   
data = {"one", "two", "three"};  //compile error  
System.out.println(data[1]);

*Пояснение: Список инициализации разрешается использовать или при объявлении массива:  
String data[] = {"one", "two", "three"};  
или при создании массива с помощью оператора new:  
data = new String[] {"one", "two", "three"};*

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        ElectricInverter inverter = **new** ElectricInverter();   
        **int** AC = 220;   
        System.out.println(inverter.invert(AC)); // 0 (GROUND)  
    }   
}   
   
**class** ElectricInverter {       
    **public** **static** **final** **int** AC = ~220;   
    **public** **static** **final** **int** DC = -110;   
    **public** **static** **final** **int** GROUND = 0;   
       
    **int** invert(**int** type) {   
        **if** (type == AC) {   
            **return** DC;   
        }   
        **else** **if** (type == DC) {   
            **return** AC;   
        }   
        **return** GROUND;   
    }   
}

*Пояснение: Оператор ~ выполняет побитовую инверсию целых чисел - меняет все биты в двоичном представлении числа на противоположные, поэтому 220 != ~220.  
Так как все сравнения в методе invert() будут ложными, он вернёт константу GROUND = 0.*

**public** **class** Main **implements** Runnable /\*throws IterruptedException\*/ {   
    **public** **void** run() {   
        System.out.println("Hello");   
        Thread.currentThread().sleep(100);           
    }           
    **public** **static** **void** main(String... args) **throws** InterruptedException {   
        **new** Thread(**new** Main()).start();   
    }   
}

**public** **class** D {   
    **public** **static** **void** main(String[] args) {   
        E e = **new** E();   
        e.someVariable = 100;   
        e.doIt(e);   
        System.out.println(e.someVariable); // 101  
    }   
}   
   
**class** E {   
    **public** **int** someVariable = 10;   
   
    **public** **void** doIt(E aE) {   
        aE.someVariable++;   
    }   
   
    E() { // пустий конструктор дозволено.  
    }   
}

**class** Test {   
    **public** **static** **void** main( String[] args ) {   
        List list = **new** ArrayList( 3 );    
        list.add( **new** Integer( 100 ) );   
        list.add( **new** Integer( 200 ) );   
        list.add( **new** Integer( 300 ) );   
        list.add( **new** Integer( 400 ) );    
   
        System.out.println( list.size() ); //4  
    }   
}

*Пояснение: Мы заведомо определяем размерность списка, которая равна 3. Но в случае выхода за границу, размерность будет динамически расширена. Компиляция пройдет успешно и будет напечатано 4.*

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **int** a = 10;   
        **int** b = 100;   
        **double** c = (**double**) (a/b);   
        String str = String.format("%1.4f", c);   
        System.out.println(str); //0,0000  
    }   
}

*Пояснение: В данном случае результат деления 2-ух целых чисел будет также округляться до целого, а затем приводиться к double (т. о. получаем 0). Чтобы на консоль вывелось 0,1000 необходимо изменить строку double c = (double) (a/b) на double c = (double) a/b.*

**public** **class** Test {   
    **static** **void** m(**int** ... a) {   
        System.out.println("1");   
    }   
   
    **static** **void** m(Integer ... a) {   
        System.out.println("2");   
    }   
   
    **public** **static** **void** main(String[] args) {   
        m(1, 2); //compile error  
    }   
}

*Пояснение: Для вызова m(1,2) могут быть применены оба метода - и m(int ... a), и m(Integer ... a). Ни один из этих методов не является наиболее специфическим, поэтому возникает ошибка компиляции (ambiguous methods).  
P.S. Ранние версии JDK8 содержали ошибку, из-за которой этот код успешно компилировался вопреки спецификации.*

**public** **class** Test {   
    **static** Boolean bo1 = **new** Boolean("true");   
    **static** Boolean bo2 = **new** Boolean(**false**);   
   
    **public** **static** **void** main(String[] args) {   
        Boolean bo3 = **new** Boolean(bo1);   
        Boolean bo4 = **new** Boolean("bo2");   
        System.out.println(bo1.equals(bo3)); //true  
        System.out.println(bo2.equals(bo4)); //true  
    }   
}

Какие из следующих строк можно вставить вместо комментария для успешной компиляции?

**interface** Animal {   
    **void** eat();   
}   
    
// Сюда вставить код   
   
**public** **class** HouseCat **implements** Feline {   
    **public** **void** eat() { }   
}

interface Feline extends Animal { }

interface Feline extends Animal { void eat(); }

*Пояснение: В классе, который реализовывает интерфейс, определяется функция void eat(), которую*

*декларируется в интерфейсе*

**import** java.text.NumberFormat;   
   
**class** Main {   
    **public** **static** **void** main(String[] args) {   
        **float** f = 123.45678f;   
        NumberFormat numberFormat = NumberFormat.getInstance();   
        numberFormat.setMaximumFractionDigits(2);   
        System.out.println(numberFormat.format(f)); //123.46 - round  
    }   
}

**public** **class** Test {   
   
    **static** **int** b = Test.a;   
    **static** **final** **int** a = 3;   
       
    **public** **static** **void** main(String... args) {   
        System.out.println("a = " + a + ", b = " + b); // 3, 3  
    }   
       
}

*Пояснение:*

На первый взгляд может показаться, что правильным будет ответ "a = 3, b = 0", поскольку статические поля инициализируются в порядке объявления и полю **b** присваивается ещё не проинициализированное значение **a**.

На самом деле, поскольку поле **a** является *константой*, то компилятор сразу подставляет его значение. Т.е. фактически первая строка компилируется как  
  
static int b = 3;  
  
и правильным будет ответ "a = 3, b = 3"

**class** A {   
    **private** **static** **void** test1() {    
        System.out.println("A.test1");   
    }    
       
    **static** **void** test2() {   
        test1();   
    }    
}    
   
**public** **class** B **extends** A {    
    **void** test1() {   
        System.out.println("B.test1");   
    }    
    **static** **void** test2() {   
        **super**.test2();   
    }    
    **public** **static** **void** main(String[] args) {    
        A a = **new** B(); a.test2();    
    }   
}

// compile error

*Пояснение: Нельзя использовать ключевое слово super в static методах.*

**public** **class** TestClass {   
    **public** **static** **void** main(String[] args) {   
        Base sub = **new** Sub();   
        sub.test();   
    }   
}   
   
**class** Base {   
    **public** **static** **void** test() {   
        System.out.println("Base.test()");   
    }   
}   
   
**class** Sub **extends** Base {   
    **public** **static** **void** test() {   
        System.out.println("Sub.test()");   
    }   
}

// Base.test()

*Пояснение: Для вызова статических методов используется механизм раннего связывания, при котором вызываемый метод определяется исходя из типа переменной-ссылки, а не из фактического типа присвоенного объекта.*

**public** **class** Test {   
    **public** **static** **void** main(String[] s) {   
        A a = **new** B();   
        a.b();   
    }   
}   
   
**class** A {   
    **void** a() {   
        System.out.println("A-a");   
    }   
       
    **void** b() {   
        System.out.println("A-b");   
        a();   
    }   
}   
   
**class** B **extends** A {   
    **void** a() {   
        System.out.println("B-a");   
    }   
       
    **void** b() {   
        System.out.println("B-b");   
        **super**.b();   
    }   
}

// B-b, A-b, B-a

*Пояснение: В примере создаётся объект класса B, в котором перегружены методы a() и b(). Поэтому при обращении к этим методам будут вызываться их перегруженные варианты. Условно последовательность действий можно описать так:*

01. A a = **new** B(); // Создаём экземпляр класса B.   
02. a.b();         // Вызываем метод b(). В текущем объекте метод перегружен.   
                   // Происходит вызов перегруженного метода ("B-b")   
03. **super**.b();     // Вызываем метод b() из суперкласса ("A-b")   
04. a();           // Вызываем метод a(). В текущем объекте метод перегружен.   
                   // Происходит вызов перегруженного метода ("B-a")

**public** **class** Sentence {   
    **public** **static** **void** main(String[] args) {   
        String str1 = "Zero";   
        String str2 = "Zero";   
   
        **boolean** b1;   
   
        **if** (str1 == str2)   
            b1 = **true**;   
        **else**   
            b1 = **false**;   
        System.out.println(b1);   
   
        String str3 = "Zero";   
        String str4 = "Zero1";   
   
        **boolean** b2;   
   
        **if** (str3 == str4)   
            b2 = **true**;   
        **else**   
            b2 = **false**;   
        System.out.println(b2);   
    }   
}

//true false

*Пояснение: Объекты классов должны создаваться явно. В представленном вопросе, обьекты создаются неявно: String str1 = "Zero"; String str2 = "Zero"; приводит к присвоению второй ссылочной переменной, адреса, сохраненного в первой ссылочной переменной, из-за оптимизации кода компилятором. При вышеприведенном способе инициализации переменной, объект создается только один раз. Если есть другая переменная, с тем же способом присвоения и таким же набором символов(т.е. - "Zero"), строка будет использоваться совместно. А это значит, что str1==str2 = true(ссылка на одну и ту же область памяти). Однако, равенство обьектов подобным образом обычно не проверется. Содержимое объектов обычно проверяется с помощью метода equals()(например, str1.equals(str2)), который с этой целью переопределяется.*

**public** **class** Test {   
    **final** String s;   
       
    **public** Test() {   
        s = "default";   
    }   
       
    **public** Test(String s) {   
    }   
       
    **public** **static** **void** main(String[] args) {   
        System.out.println(**new** Test().s);   
    }   
}

//compile error

Поля с модификатором final необходимо инициализировать или при объявлении, или в \_каждом\_ конструкторе.   
Чтобы избежать ошибки компиляции нужно изменить неверный конструктор, например, так:  
  public Test(String s) {  
this.s = s;  
}

**public** **class** TestWrap {   
    **public** **static** **void** main(String[] args) {   
        Integer ii = 1000;   
        Integer jj = 2000;   
   
        System.out.print((ii \* 2 == jj) ? "yes " : "no ");   
        System.out.print((jj / 2 == ii) ? "yes " : "no ");   
        System.out.print((ii \* 2 == jj) ^ (jj / 2 == ii) ? "yes " : "no ");   
    }   
}

// yes, yes, no

*Пояснение: При выполнении любых арифметических операций (умножение, деление, ...) объекты классов-оболочек (Integer) автоматически разворачиваются в значения примитивного типа (int). Результатом любой арифметической операции также будет значение примитивного типа.  
Если с помощью == сравниваются значение примитвного типа и объект-оболочка, то объект автоматически разворачивается, и происходит сравнение двух примитивов.  
Результатом операции ^ (исключающее ИЛИ) будет false, так как оба операнда true.*

**public** **class** Test {   
  **public** **static** **void** main(String[] args) {   
    String s1 = **new** String("string");   
    String s2 = **new** String("STRINg");   
           
    StringBuilder sb1 = **new** StringBuilder("test");   
    StringBuilder sb2 = **new** StringBuilder("test");   
           
    System.out.println(s1.equalsIgnoreCase(s2) && sb1.equals(sb2)   
        && s2.charAt(s2.length()) == 'g');   
  }   
}

//false

*Пояснение: Метод equalsIgnoreCase() вернёт true, так как при таком сравнении регистр не учитывается.   
Метод equals() в StringBuilder и StringBuffer не переопределён и не позволяет осуществлять сравнение, так что возвращает всегда false.  
Благодаря && дальнейшая проверка не выполняется, что позволяет избежать StringIndexOutOfBoundsException, так как последним доступным элементом в строке может быть [s2.length()-1].*

**public** **class** Foo {   
    **public** **static** **void** main(String[] args) {   
        **for** (**int** k = 1; k <= 10; k++) {   
            **if** (k % 5 == 0) **break** label;   
            System.out.print(k + " ");   
        }   
   
    label:   
        {   
            System.out.print("stop!");   
        }   
    }   
}

//error

*Пояснение: Компиляция выполнится с ошибкой.  
Метки (label) используются в связке с break (или continue) для выхода (или пропуска итерации) в циклах for, while, do-while, к которым этот break (continue) относится.*

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        System.out.println("4/2=" + 4/2 == "4/2=2");   
    }   
}

//*Пояснение: Деление: 4/2 → 2  
Приведение типов: 2 → "2"  
Конкатенация: "4/2=" + "2" → "4/2=2"  
Сравнение: "4/2=2" == "4/2=2" → true  
Всё это происходит на этапе компиляции - в скомпилированном байт-коде строк нет.*

**byte** b1 = 127;   
b1 +=1;

//-128

*Пояснение:*

127 - это максимальное значение для переменной типа **byte**. Число 128 уже не входит в диапазон значений, но, благодаря использованию дополнительного кода для представления отрицательных чисел, оно рассматривается как -128.

**Такой же результат получается при преобразовании (byte)128.**

**public** **class** Switch {   
    **public** **static** **void** main(String[] args) {   
        **int** i = 3;   
        **switch**(i) {   
            **default**:   
                System.out.println("default");   
            **case** 1:   
                System.out.println("1");   
            **case** 2:   
                System.out.println("2");   
        }   
    }   
}

//default  
1  
2

*Пояснение: Ключевы слова case и default могут идти в любом порядке. Если не использовать ключевое слово break, то программа будет идти на следующий по порядку case.*

String a = **new** String("hello");   
String b = **new** String(a);   
String c = a;   
**char**[] d = { 'h', 'e', 'l', 'l', 'o' };

* (a == c)
* a.equals(b)

**public** **abstract** **class** One {   
    **public** **abstract** **int** doJob();   
}   
**class** Two **extends** One { }

//error

*Пояснение: Неабстрактный класс должен реализовывать абстрактные методы базовых классов/интерфейсов*

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **int** a = 10;   
        **int** b = 100;   
        **double** c = (**double**) (a/b);   
        String str = String.format("%1.4f", c);   
        System.out.println(str);   
    }   
} //0.0000

*Пояснение: В данном случае результат деления 2-ух целых чисел будет также округляться до целого, а затем приводиться к double (т. о. получаем 0). Чтобы на консоль вывелось 0,1000 необходимо изменить строку double c = (double) (a/b) на double c = (double) a/b.*

Выберите все возможные варианты создания класса исключений.

наследовать класс Throwable

наследовать класс Exception

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **int** i = 0;   
        **int** j = 1;   
        System.out.println(i += (j < i) ? (2) : (3)); //1   
    }   
} //3

*Пояснение: В данном примере используется условное выражение (условие) ? (значение1) : (значение2) . В этом выражении перед знаком вопроса ставится условие, а после него через : идут 2 возвращаемых значения. Первое будет возвращено в случае если условие вернёт значение true, в противном случае будет возвращено 2-ое значение.*

Каково будет значение переменной count после выполнения данного кода?

**int** count = 1, i = 0;   
**do** {   
    count \*= ++i;   
    **if** (count > 5) **break**;   
} **while** (i <= 4);

//6

Каким будет результат выполнения программы?

**public** **class** Tenor **extends** Singer {   
    **public** **static** String sing() { **return** "fa"; }   
    **public** **static** **void** main(String[] args) {   
        Tenor t = **new** Tenor();   
        Singer s = **new** Tenor();   
        System.out.println(t.sing() + " " + s.sing());   
    }   
}   
**class** Singer { **public** **static** String sing() { **return** "la"; } }

//fa la

*Пояснение:* ***Полиморфизм не применяется к статическим методам!!!***

Что будет напечатано следующим кодом?

**public** **class** Main {   
    **public** **static** **void** var(Integer x, **int** y) {   
        System.out.println("Integer int");   
    }   
   
    **public** **static** **void** var(Object... x) {   
        System.out.println("Object");   
    }   
   
    **public** **static** **void** var(**int**... x) {   
        System.out.println("int... x");   
    }   
   
    **public** **static** **void** var(Integer... x) {   
        System.out.println("Integer...");   
    }   
   
    **public** **static** **void** main(String... args) {    
        **int** i = 0;   
        Integer i2 = 127;   
        var(i, i2);   
    }   
} // Integer int

*Пояснение: При вызове перегруженных (overloaded) методов компилятор:  
1) сначала ищет метод, типы параметров которого в точности совпадают с аргументами;  
если такого метода не нашлось, то  
2) затем ищет метод с параметрами, которые могут быть получены с помощью допустимых при вызове преобразований (widening, boxing+widening, unboxing+widening);  
если и такого метода не нашлось, то   
3) затем анализируются методы с переменным количеством аргументов.  
  
В данном примере компилятор останавливается на втором шаге (срабатывают преобразования int→Integer, Integer→int).*

Какой результат компиляции и запуска следующего кода с использованием параметров a = 0 , b = 3?

**public** **void** divide(**int** a, **int** b) {   
    **try** {   
        **int** c = a / b;   
    } **catch** (Exception e) {   
        System.out.print("Exception ");   
    } **finally** {   
        System.out.println("Finally");   
    }   
} //Finally

*Пояснение: Блок finally выполняется всегда за исключением некоторых вариантов 0/3 = 0 не выкидывает исключительной ситуации*

Можно ли наследоваться от класса **java.lang.String**?

*Пояснение: Нет, так как класс String объявлен как final.*

**public** **class** Bar {   
    **public** **static** **void** main(String args[]) {   
    label:   
        **for** (**int** i = 0; i < 5; ++i) {   
            **for** (**int** j = 0; j < 5; ++j) {   
                **if** (i > 2) **break** label;   
                System.out.print(j);   
            }   
            System.out.print(" ");   
        }   
    }   
}// 01234 01234 01234

*Пояснение: Оператор break с меткой позволяет обойти идентифицируемый блок кода. Программа выдаст строку 01234 01234 01234, а затем выйдет из всех циклов и продолжит выполнение.*

Какие модификаторы доступа разрешены для класса, если он является единственным в файле.

public, package-private(по умолчанию)

Какой результат выполнения данного кода?

**public** **class** Starter **extends** Thread {   
    **private** **int** x = 2;   
   
    **public** **static** **void** main(String[] args) **throws** Exception {   
        **new** Starter().makeItSo();   
    }   
   
    **public** Starter() {   
         x = 5;   
         start();   
    }   
   
    **public** **void** makeItSo() **throws** Exception {   
        join();   
        x = x - 1;   
        System.out.println(x);   
    }   
   
    **public** **void** run() { x \*= 2; }   
 } //9

*Пояснение: Сначала происходит создание нового объекта - new Starter() - и сразу же вызывается его метод - makeItSo()*

Что напечатает следующий код?

**public** **class** Test {   
   
    **private** String name;   
   
    Test(String name) {   
        **this**.name =  name;   
    }   
   
    **public** **void** test(**final** Test test) {           
        test = **new** Test("three");   
    }   
   
    **public** String toString() {   
        **return** name;   
    }   
   
    **public** **static** **void** main(String[] args) {   
        Test t1 = **new** Test("one");   
        Test t2 = **new** Test("two");   
        t1.test(t2);   
        System.out.println(t2);   
    }   
       
}  //error

*Пояснение: final параметр в методе test, говорит о том что ссылка на объект test не может быть изменена, поэтому строка test = new Test("three") приведёт к ошибке на этапе компиляции.*

Какой результат выполнения программы, если 49 - код единицы в ASCII ?

**public** **class** A {   
    **public** **static** **void** main(String[] args) {   
           
        **byte**[] a={ 49,50,51,52,53,54,55 };   
           
        String b=**new** String(a);      // 1    
        System.out.println(b);   
           
        String c=**new** String(a,2,4);  // 2   
        System.out.println(c);   
    }   
   
} // 1234567

// 3456

*Пояснение: В классе String имеется конструктор, который инициализирует строку 8-разрядным byte-массивом:  
public String(byte bytes[], int offset, int length)*

Какой результат будет получен при компиляции и выполнении данного кода:

**public** **class** Main{   
    **final** **int** x = 1;   
   
    **public** **static** **void** main(String [] str) {   
        **int** x = 2;                            //1   
        **for** (x = 3 ; x < 5 ; x++);            //2   
        System.out.println("x=" + x);   
   
    }       
}  //5

x - константа экземпляров класса, а в методе Main - локальная переменная, объявленая как int x.

Цикл for, в данном случае, не делает ничего кроме инкремента x, после завершения итераций x будет равен 5.

Что будет выведено на экран?

**class** A {   
    **int** x = 1;   
   
    **public** **void** printX() {   
        System.out.println(getX());   
    }   
   
    **public** **int** getX() {   
        **return** x;   
    }   
}   
   
**class** B **extends** A {   
    **int** x = 2;   
   
    **public** **int** getX() {   
        **return** x + 1;   
    }   
}   
   
**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        A classA = **new** B();   
        classA.printX();   
    }   
} //3

*Пояснение: Метод printX() выводит на экран значение, которое возвращает метод getX(). При этом вызывается перекрытый метод из класса B, который возвращает x+1, причём значение поля x также берётся из класса B.*

Что будет выведено на экран?

**public** **class** Tasks {   
    **public** **static** Tasks instance = **new** Tasks();   
    **private** **static** **final** **int** DELTA = 5;   
    **private** **static** **int** BASE = 7;   
    **private** **int** x;   
   
    **public** Tasks() {   
        x = BASE + DELTA;   
    }   
    **public** **static** **int** getBASE() {   
        **return** BASE;   
    }   
    **public** **static** **void** main(String[] args) {   
        System.out.println(Tasks.instance.x);   
    }   
} //5

*Пояснение: Инициализация статических полей осуществляется в том порядке, в котором они записаны.  
В этом примере первым будет проинициализировано поле instance, а уже затем - BASE. Поэтому в момент вызова конструктора Tasks() поле BASE ещё содержит 0.  
Поле DELTA является переменной-константой, поэтому компилятор сразу подставляет в выражение вместо DELTA его значение.  
В итоге получаем: x = 0 + 5;*

**class** Main {   
    **public** **static** **void** main(String[] args) {   
        Integer i = **new** Integer("10");   
        **if** (i.toString() == i.toString()) {   
            System.out.println("Равный");   
        } **else** {   
            System.out.println("Неравный");   
        }   
    }   
} // Неравный

*Пояснение: Метод toString() возвращает эквивалент String этого объекта String. Каждый раз, когда он вызывается, создаётся новый объект. Оператор == сравнивает две ссылки на объект, а не реальное содержимое String. Таким образом, сравнение возвращает false, выполняется оператор else, и выводится "Неравный".*

Какого типа вложенных классов не существует (укажите все подходящие варианты)?

соседний класс

глобальный вложенный класс

Что будет выведено на экран следующим кодом:?

**class** Outer {   
   **int** outer\_x = 100;   
   **void** test() {   
       **for**(**int** i = 0; i < 5; i ++) {   
           **class** Inner{   
               **void** display() {   
                   System.out.print("outer\_x = " + outer\_x + "; ");   
               }   
           }   
           Inner inner = **new** Inner();   
           inner.display();   
       }   
   }   
}   
**public** **class** InnerClassDemo {   
     **public** **static** **void** main(String args[]) {   
         Outer outer = **new** Outer();   
         outer.test();   
     }   
} // outer\_x = 100; outer\_x = 100; outer\_x = 100; outer\_x = 100; outer\_x = 100;

Какой результат выполнения следующего кода:

**public** **class** MainClass {   
    **public** **static** **void** main(String args[]) {   
        **int** k = 2;   
        System.out.println( k << 3 >> k == k >> 1 << k);   
    }   
} //true

1. k << 3 = 16 (0000 0010b << 3 = 0001 0000b)

2. 16 >> 2 = 4 (0001 0000b >> 2 = 0000 0100b)

3. k >> 1 = 1 (0000 0010b >> 1 = 0000 0001b)

4. 1 << k = 4 (0000 0001b << 2 = 0000 0100b)

Поскольку 4 = 4 правильный ответ true.

Какой результат выполнения данной программы ?

**public** **class** Test {   
    **private** **void** method1() {   
        MyClass obj = **new** MyClass();   
        obj.x = 1;   
        method2(obj);   
        System.out.println("obj.x="+obj.x);   
    }   
   
    **private** **void** method2(MyClass param) {   
        param.x = 2;   
        param = **new** MyClass();   
        param.x = 3;   
    }   
   
    **class** MyClass {   
        **int** x;   
    }   
   
    **public** **static** **void** main(String[] args) {   
        **new** Test().method1();   
    }   
}// obj.x=2

В разных строчках метода method2() переменная param ссылается на РАЗНЫЕ объекты. По шагам:  
param.x = 2; // присвоить 2 полю x объекта param (он же obj)  
param = new MyClass(); // создать новый объект, ссылку занести в param  
param.x = 3; // присвоить 3 полю x объекта param (уже новый объект), объект obj при этом не меняется

**package** tutorial.base;   
   
**public** **class** TypesTutorial {   
   
    **public** **static** **void** main(String... args) {   
        A alpha = **new** B();   
    }   
}   
   
**class** A {   
    A(){   
        System.out.print("A");   
        a();   
    }   
       
    **void** a() {   
        System.out.print("a");   
    }   
}   
   
**class** B {   
    B() {   
        System.out.print("B");   
        a();   
    }   
       
    **void** a() {   
        System.out.print("b");   
    }   
} //compile error

*Ошибка компиляции в методе main, так как классы не связаны отношением наследования.*

Какой код нужно вставить на место строки 1, чтобы корректно получить строковое представление переменной l?

**public** **class** Test {    
    **static** **public**  **void** main(String args[]) {   
        **long** l = 0;   
        String s = **new** String();   
        // 1   
    }   
}

s = "" + l;

s = Long.toString(l);

Каким будет результат работы программы?

**public** **class** MyTest {   
    **public** **static** **void** main(String[] args) {           
        label1: **for** (**int** i = 0; i < 3; i++) {   
            **if** (i == 1)   
                **continue** label1;    
            label2:   
                System.out.print("TEST ");   
            label3:;   
        }   
    }   
} // TEST TEST

*Пояснение: Конструкция continue label1: передаст управление на метку и цикл будет продолжен.  
Если вместо continue поставить break - TEST будет напечатано один раз.*

Какие из следующих методов выбрасывают исключение InterruptedException?

wait()

sleep()

**public** **class** Exam  {   
    **public** **static** **void** main(String[] arg) {   
        Object obj = **null**;   
        String str = **new** String("str");   
        str = (String) obj;   
        obj = **new** String("obj");   
        System.out.print(obj + ", " + str);     
    }   
} obj, null

Какой результат работы следующей программы:

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **final** Integer i = 1;   
        **switch** (1) {   
            **case** i:   
                System.out.println("1");   
                **break**;   
            **default**:   
                System.out.println("default");   
        }   
    }   
} // compile error

*Пояснение: В case-секциях оператора switch можно использовать переменные, если они объявлены как final. Это условие не выполняется в случае переменных, для которых нужен autounboxing.*

Какой результат выполнения данного кода:

System.out.println( 0.3 == 0.1d + 0.1d + 0.1d );

//false

*Пояснение: Правильный ответ на этот вопрос связан со способом представления дробных чисел в памяти компьютера. Напомним, что они хранятся в двоичном коде.  
  
Если попробовать перевести 0.1 в двоичную систему счисления, то получится БЕСКОНЕЧНАЯ двоичная дробь 0,000110011001100110011001100... Поскольку в памяти компьютера число представляется с помощью КОНЕЧНОГО количества двоичных разрядов (например, под значение типа double отводится 64 бита), то бесконечная двоичная дробь просто округляется. То число, которое получается в результате округления, будет чуть-чуть (на одну многомиллионную) отличаться от 0.1. Аналогично, двоичное представление для 0.3 будет чуть-чуть отличаться от 0.3.  
Получается, что в примере сравниваются два числа, каждое из которых примерно равны 0.3, но они не равны между собой.  
Если взять числа с другим двоичным представлением, будет другой результат. Например, System.out.println( 0.375 == 0.125d + 0.125d + 0.125d ) выводит true.*

Какой результат выполнения программы:

**public** **class** Foo {   
    **public** **static** **void** main(String[] args) {   
        **for** (**int** k = 1; k <= 10; k++) {   
            **if** (k % 5 == 0) **break** label;   
            System.out.print(k + " ");   
        }   
   
    label:   
        {   
            System.out.print("stop!");   
        }   
    }   
} //compile error

*Пояснение: Компиляция выполнится с ошибкой.  
Метки (label) используются в связке с break (или continue) для выхода (или пропуска итерации) в циклах for, while, do-while, к которым этот break (continue) относится.*

**public** **class** Main {   
   
    **static** **class** X {   
        **static** String x = Y.y;   
    }   
   
    **static** **class** Y {   
        **static** String y = X.x;   
    }   
   
    **public** **static** **void** main(String[] args) **throws** Throwable {   
        Y.y = "0";   
        System.out.println(X.x);   
    }   
} //null

*Пояснение: 1)сначала Y.y инициализирован null   
2)X.x инициализируется Y.y то есть null  
3)Y.y инициализируется X.x то есть null  
2)после Y.y = "0" , X.x же остаётся null  
результат : null*

Что произойдет во время компиляции и выполнения данного кода?

**class** Base {   
    **public** String name = "Base";   
    **public** String getName() {   
        **return** name;   
    }   
}   
   
**class** Sub **extends** Base {   
    **public** String name = "Sub";   
    **public** String getName() {   
        **return** name;   
    }   
}   
   
**public** **class** Program {   
    **public** **static** **void** main(String[] args) {   
        Sub s = **new** Sub();   
        Base b = s;   
        System.out.println(b.name);   
    }   
} // Base

*Пояснение: При обращении к полям используется тип ссылки, а не тип реального объекта. Если бы вместо обращения к полю был вызов переопределенного метода получения значения поля, то благодаря позднему связыванию мы получили бы значение из реального объекта.*

**import** java.util.Arrays;   
   
**class** Generic<T **extends** Number> {    
    **private** T arr[] = {1, 2, 3, 4, 5};   
    
    **public** String toString(){    
        **return** Arrays.toString(arr);   
    }    
}    
   
**public** **class** Test {    
    **public** **static** **void** main(String [] args) {    
        Generic<Double> obj = **new** Generic<Double>();   
        System.out.println(obj);    
    }    
} //compile error

*Пояснение: Ошибка компиляции при объявлении массива.  
Списки инициализации не могут быть использованы для инициализации параметризованных массивов.*

**import** java.io.ByteArrayOutputStream;   
**import** java.io.ObjectOutputStream;   
**import** java.io.Serializable;   
   
**public** **class** NewClass {   
    **public** **static** **void** main(String[] args) **throws** Exception {   
        ByteArrayOutputStream bOut = **new** ByteArrayOutputStream();   
        ObjectOutputStream oOut = **new** ObjectOutputStream(bOut);   
        Whole object = **new** Whole();   
        oOut.writeObject(object);  //private void writeObject()  
    }   
}   
   
**class** Whole **implements** Serializable {   
    **public** **void** writeObject(ObjectOutputStream out) {   
        System.out.println("Whole.writeObject()");   
    }   
}

Программа выполнится без ошибок и ничего не появится

По умолчанию класс ObjectOutputStream использует стандартный алгоритм для сохранения значений полей сериализуемого объекта.  
Если в классе имеется специальный метод, сигнатура которого в точности соответствует  
**private** void writeObject(java.io.ObjectOutputStream stream);  
то он и будет вызван для сохранения значений вместо стандартного алгоритма.   
А если использовать другой модификатор, как это сделано в примере, то ничего подобного не происходит. Звернення до приватного методу йде за допомогою рефлексії.

**public** **class** Test {   
   
    **private** String name;   
   
    Test(String name) {   
        **this**.name =  name;   
    }   
   
    **public** **void** test(Test test) {           
        test = **new** Test("three");   
    }   
   
    **public** String toString() {   
        **return** name;   
    }   
   
    **public** **static** **void** main(String[] args) {   
        Test t1 = **new** Test("one");   
        Test t2 = **new** Test("two");   
        t1.test(t2);   
        System.out.println(t2);   
    }   
       
} //**two**

*Пояснение: Ссылки в метод передаются по значению и изменять их переназначая на новые объекты нельзя.*

Что выведет на экран следующий код?

**public** **class** Test{   
    **public** **static** **void** main(String[] args){       
        **char** c1 = '1';   
        **char** c2 = '\u0031';   
        **char** c3 = 49;   
        System.out.println(c1 + c2 + c3);   
    }   
} //147

*Пояснение: Важно знать, что значение для переменной типа char можно задать несколькими способами: непосредственно символом, unicode-кодом этого символа, а также целым числом. Так же важно знать, что* ***значения типа char не "склеиваются", а складываются.***

Какой результат выполнения программы:

**public** **class** StartClass {   
    **public** **static** **void** main(String[] args) {   
        Double d1 = 1d;   
        Double d2 = 1d;   
        System.out.println(d1 == d2);   
    }   
} //false

**Пул** есть только для **Boolean, Byte, Char, Short, Integer**

Какой результат выполнения данного кода:

**int** i1 = 012;   
**int** i2 = 20;   
System.out.println(i1 \* i2); //200

если начинать число с "0", то оно считается введённым в восьмеричной системе.  
12 в восьмеричной == 10 в десятичной.  
Потому и 200.

поскольку в восьмеричной системе нет цифр 8 и 9, то нельзя написать 018, например

Какой результат попытки компиляции и запуска программы:

**public** **class** MainClass {   
    **public** **static** **void** test(**int** ...a) {   
        System.out.println("int...");   
    }   
   
    **public** **static** **void** test(Integer ...a) {   
        System.out.println("Integer...");   
    }   
   
    **public** **static** **void** test(Number ...a) {   
        System.out.println("Number...");   
    }   
   
    **public** **static** **void** test(Object ...a) {   
        System.out.println("Object...");   
    }   
   
    **public** **static** **void** main(String args[]){   
        Number n = **new** Integer(1);   
        test(n.intValue());   
    }   
} // compile error

*Пояснение: Очевидно, что каждый из четырёх методов является потеницально применимым для вызова test(int).  
Задача компилятора в этом случае - выбрать наиболее специализированный метод.  
В случае методов с переменным количеством аргументов компилятор рассматривает, как соотносятся базовые типы и выбирает из них наиболее узкий, при этом тип фактического параметра не учитывается  
Так, из типов Integer, Number, Object наиболее узким будет Integer. Но примитивные типы не являются подтипами ссылочных типов, поэтому из типов int и Integer компилятор не может выбрать наиболее узкий и генерирует ошибку компиляции.*

Какой будет результат компиляции и выполнения следующего кода?

**import** **static** java.lang.Byte.\*;   
**import** **static** java.lang.Integer.\*;   
   
**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        System.out.println(MAX\_VALUE);   
    }   
} //compile error

В коде происходит импорт всех статических членов классов Byte и Integer.   
В обоих классах есть статическое поле MAX\_VALUE, отсюда и двусмысленность - компилятор не понимает какое именно и из какого класса использовать MAX\_VALUE.

Какой результат выполнения данного кода:

Integer i = 5000;   
System.out.println(i.hashCode());  //5000

*Пояснение: Классы-оболочки целочисленных типов (****Byte****,* ***Character****,* ***Short****,* ***Integer****) в качестве хэш-кода возвращают хранящееся в них значение.*

Что будет напечатано в результате выполнения данного кода?

**import** java.util.Arrays;   
**class** ArraysComparing {   
    **public** **static** **void** main(String...args) {   
        **int**[] i1[] = {{1,2,3}, {0,0,0}};   
        **int**[][] i2 = {{1,2,3}, {0,0,0,}};   
        **int**[][] i3 = **new** **int**[2][3];   
        System.arraycopy(i1, 0, i3, 0, i3.length);   
        System.out.println(Arrays.equals(i1, i2));   
        System.out.println(Arrays.equals(i1, i3));   
        System.out.println(Arrays.deepEquals(i1, i2));   
    }   
} //false true true

*Пояснение: В массивах i1 и i2 ссылки на вложенные массивы разные, поэтому первый equals вернет false.   
Во втором случае ссылки будут одинаковыми, соответственно true.   
В третьем случае производится сравнение элементов, все одинаковы, поэтому true*

Каков результат выполнения кода?

**import** java.util.\*;   
   
**public** **class** TestIterator{   
    **public** **static** **void** main(String[] args){   
        **final** List synchList = Collections.synchronizedList(**new** ArrayList());   
        fillList(synchList, 10);   
        Thread anotherThread = **new** Thread(){   
            **public** **void** run(){   
                **try**{   
                    Thread.sleep(300);   
                }**catch** (InterruptedException e){   
                    System.out.println("anotherThread interrupted");   
                    **return**;   
                }   
                    synchList.remove(1);   
                    synchList.remove(7);   
            }   
        };   
        anotherThread.start();   
        Iterator iterator = synchList.iterator();   
        **while**(iterator.hasNext()){   
            System.out.print(iterator.next());   
            **try**{   
                Thread.sleep(100);   
            } **catch** (InterruptedException e) {   
                System.out.println("Main thread interrupted");   
                **return**;   
            }   
        }   
    }   
    **static** **void** fillList(List list, **int** n){   
        **for**(**int** i = 1; i <= n; i++)   
            list.add(i);   
    }   
}  // runtime error

*Пояснение: Т.к. несмотря на то, метод класса Collections возвращает синхронизированную копию коллекции, итератор остается не синхронизированным. В результате будет выброшен ConcurrentMofificationException.*

После выполнения какой строки только один объект будет доступен для сборки мусора (Garbage Collection)?

01. **public** **class** Test {   
02.      Test ags = **null**;   
03.      **public** **static** **void** main(String argv[]) {   
04.           Test a1 = **new** Test();   
05.           Test a2 = **new** Test();   
06.           Test a3 = **new** Test();   
07.           a1.ags = **new** Test();   
08.           a2.ags = a1.ags;   
09.           a3.ags = a2.ags;   
10.           a1 = **null**;   
11.           a2 = **null**;   
12.           a3 = **null**;   
13.      }   
14. }  //10

*Пояснение: После 9-ой строки: a1.ags, a2.ags, a3.ags указывают на объект созданный в строке 7. В 10-ой строке ссылке a1 присваивается значение null. Теперь на объект, созданный в строке 4, нет ссылок. То есть он становится доступным для сборщика. Его поле ags в расчет не берется, так как объект, на который оно указывает, будет доступен посредством ссылок a2.ags, a3.ags*

Что выведет программа в консоль?

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        **boolean** b = **true**;   
        **int** x = 0;   
   
        **do** {   
            **if** (x++ > 5)   
                b = **false**;   
            System.out.print(x);   
        } **while** (b);   
    }   
} //1234567

*Пояснение: Очевидно, что числа на экран будут выводиться последовательно. Остаётся только разобраться, какое число будет выведено первым, а какое – последним.  
Поскольку при каждой проверке условия переменная x будет увеличиваться, то первым выведется число 1.  
На последней итерации условие x++ > 5 должно стать истинным. Поскольку значением выражения x++ является исходное значение переменной, то это будет при x = 6. Но на экран уже будет выведено новое, увеличенное значение, т.е. 7.  
Получаем ответ: 1234567.*

Что будет результатом запуска такого кода?

**package** tutorial.base;   
   
**public** **class** TypesTutorial {   
    **public** **static** **void** main(String... args) {   
        A alpha = **new** B(0);   
    }   
}   
   
**class** A {  // або добавляти A(){ }  
    A(**int** x){            // - 1 -   
        a(x);            // - 2 -   
    }   
       
    **void** a(**int** x) {   
        System.out.println("A-a: " + x);   
    }   
}   
   
**class** B **extends** A {   
    B(**int** x) {           // - 3 -   
        a(x);            // - 4 - // або міняти на super(x);  
    }   
       
    **void** a(**int** x) {   
        System.out.println("B-a: " + x);   
    }   
} // compile error 4

*Пояснение: В классе A отсутствует конструктор по-умолчанию, поэтому конструктор класса B (потомок) первым делом должен явно вызвать один из обьявленных конструкторов класса-родителя.*

Что выведет следующий код?

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        recur(99);   
    }   
    **public** **static** **void** recur(**int** a) {   
        **if** (a <= 100) {   
            System.out.println("a=" + a);   
            recur(++a);   
            System.out.println("a=" + a);   
        }   
    }   
} 99, 100, 101, 100

*Пояснение: Сначала выдаст значение а = 99,а после префиксного инкремента выполнится снова метод recur с параметром 100, который выведет а=100, следующий recur будет вызван с параметром 101, но условие выполняться не будет, так как передаваемый параметр больше 100, и будет выведено на экран значение параметра а=101, закончится выполнение этого метода, и программа возвратится к предыдущему выполнению этого метода, и выдаст значение а=100, таким, каким бы выдало его, если бы метод recur и вовсе не вызывался сам в себе, а просто переменная а увеличилась бы на 1.*

Каким будет результат выполнения следующей программы?

**import** java.util.\*;   
**public** **class** TestFormatter {           
    **public** **static** **void** main(String... args){   
        Integer I1 = 0;   
        Integer I2 = -1;   
        Integer I3 = 1;   
        Formatter f = **new** Formatter();   
        f.format("%1$b ", I1.toString())   
         .format("%1$b ", I2.toString())   
         .format("%1$b ", I3.toString());   
        System.out.println(f.toString());           
    }   
} // true, true, true

*Пояснение: Последовательность %b используется для вывода логического значения. Соответствующий аргумент обрабатывается следующим образом:  
- если передаётся булево значение, то результатом будет его строковое представление ("true" или "false");  
- если передаётся null, то результатом будет "false";  
- во всех остальных случаях (как и в данном примере) результатом будет "true".*

Что выведет на экран следующий код?

**public** **class** Test {   
   
    **private** String name;   
   
    Test(String name) {   
        **this**.name =  name;   
    }   
   
    **public** **void** test(Test test) {           
        test = **new** Test("three");   
    }   
   
    **public** String toString() {   
        **return** name;   
    }   
   
    **public** **static** **void** main(String[] args) {   
        Test t1 = **new** Test("one");   
        Test t2 = **new** Test("two");   
        t1.test(t2);   
        System.out.println(t2); //**two**  
    }   
       
}

*Пояснение: Ссылки в метод передаются по значению и изменять их переназначая на новые объекты нельзя.*

**class** Test2 {   
    **static** **int** N(Integer x) {   
        **return** x;   
    }   
}   
   
**public** **class** Test {   
    **static** **int** M(Integer v){   
        **return** v;   
    }   
    **public** **static** **void** main(String[] args){   
        Integer x = M(100);     // 1   
        Integer z = N(100);     // 2 //compile error  
        System.out.println(x + " " + z);   
    }   
}

Метод N відноситься до іншого класу.

Что произойдет, если скомпилировать и запустить следующий код:

**byte** b1 = 127;   
b1 +=1; // -128

Что будет выведено на экран следующим кодом?

**import** java.util.EnumMap;   
**import** java.util.Map;   
   
**enum** Types { A, B, C }   
   
**public** **class** Test {   
    **static** Integer value;   
    **public** **static** **void** main(String args[]) {   
        Map<Types, Integer> m = **new** EnumMap<Types, Integer>(Types.**class**);   
        m.put(Types.A, value);   
        System.out.println(m); // {A=null}  
    }   
} *Пояснение: В коллекции EnumMap имеется параметризованный конструктор, который задаёт используемый тип элементов перечисления.  
Этот тип сохраняется в экземпляре коллекции, и используется для контроля значения ключа во время работы программы:*

Какой результат выдаст этот код:

**int** []a = {5,5};   
**int** b = 1;   
a[b] = b = 0;   
System.out.println(Arrays.toString(a)); [5, 0]

Что выведет программа в консоль?

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        **boolean** b = **true**;   
        **int** x = 0;   
   
        **do** {   
            **if** (x++ > 5)   
                b = **false**;   
            System.out.print(x); //1234567  
        } **while** (b);   
    }   
}

*Пояснение: Очевидно, что числа на экран будут выводиться последовательно. Остаётся только разобраться, какое число будет выведено первым, а какое – последним.  
Поскольку при каждой проверке условия переменная x будет увеличиваться, то первым выведется число 1.  
На последней итерации условие x++ > 5 должно стать истинным. Поскольку значением выражения x++ является исходное значение переменной, то это будет при x = 6. Но на экран уже будет выведено новое, увеличенное значение, т.е. 7.  
Получаем ответ: 1234567.*

Каким будет результат компиляции и выполнения данного кода?

**public** **class** A {   
    **public** **static** **void** main(String[] args) {   
        **for** (**int** i = 0; ++i > 0;) {   
            System.out.println(i);   
        }   
    }   
}

Напечатает все числа от 1 до 231 - 1

*Пояснение:*

Значение типа int занимает 32 бита (4 байта). Всего возможно 232 различных значений, из них половина положительных и половина отрицательных, причём отрицательными считаются числа, у которых старший бит равен единице:

00000000 00000000 00000000 000000002 = 010  
00000000 00000000 00000000 000000012 = 110  
. . .  
01111111 11111111 11111111 111111112 = 214748364710 = 231–1  
10000000 00000000 00000000 000000002 = –214748364810 = –231  
. . .  
11111111 11111111 11111111 111111112 = –110

Поэтому во время выполнения цикла после положительного числа 2147483647 получается отрицательное число –2147483648, которое не удовлетворяет условию продолжения цикла и он заканчивает свою работу.

Что произойдет при попытке запустить/откомпилировать следующий программный код?

1.**import** **static** java.lang.System.\*;   
2.**public** **class** A {   
3.    **public** **static** **void** main(String[] args){   
4.        B b1 = **new** B("one","two");   
5.        B b2 = **new** B("one", "two");   
6.        B b3 = b1;   
7.        out.println(b1 == b2);   
8.        out.println(b1 == b3);   
9.        out.println(b2 == b3);   
10.       out.println(b1.equals(b2));   
11.       out.println(b1.equals(b3));   
12.       out.println(b3.equals(b2));   
13.  }   
14.}   
   
1.**class** B {   
2.    **public** B(String prop1, String prop2){   
3.        **this**.prop1 = prop1;   
4.        **this**.prop2 = prop2;   
5.    }   
6.    **private** String prop1 = **null**;   
7.    **private** String prop2 = **null**;   
8.}

false  
true  
false  
false  
true  
false

*Пояснение: Оператор == проверяет, находятся ли объекты в одной и той же области памяти. Тогда, логично, что строки 7 и 9 возвращают false. Так как b3 и b1 указывают на один и тот же объект, то 8-я строка возвращает true.****Для строк 10, 11 и 12 выполняются аналогичные операции, так как метод equals() не переопределен в классе B.***

Что будет выведено на экран?

**public** **class** Tasks {   
    **public** **static** Tasks instance = **new** Tasks();   
    **private** **static** **final** **int** DELTA = 5;   
    **private** **static** **int** BASE = 7;   
    **private** **int** x;   
   
    **public** Tasks() {   
        x = BASE + DELTA;   
    }   
    **public** **static** **int** getBASE() {   
        **return** BASE;   
    }   
    **public** **static** **void** main(String[] args) {   
        System.out.println(Tasks.instance.x); //5  
    }   
}

*Пояснение: Инициализация статических полей осуществляется в том порядке, в котором они записаны.  
В этом примере первым будет проинициализировано поле instance, а уже затем - BASE. Поэтому в момент вызова конструктора Tasks() полеBASE ещё содержит 0.  
Поле DELTA является переменной-константой, поэтому компилятор сразу подставляет в выражение вместо DELTA его значение.  
В итоге получаем: x = 0 + 5;*

Каким будет результат выполнения кода:

**package** tests;   
   
**import** java.util.Hashtable;   
   
**public** **class** Test {   
  **public** **static** **void** main(String[] args) {   
    Hashtable ht = **new** Hashtable();   
    ht.put("1", "2");   
    ht.put("2", "3");   
    ht.put("3", "4");   
    ht.put("4", "2");   
       
    **if**(ht.contains("1")){   
      System.out.print("1");   
    }   
    **if**(ht.contains("2")){   
      System.out.print("2");   
    }   
    **if**(ht.contains("3")){   
      System.out.print("3");   
    }   
    **if**(ht.contains("4")){   
      System.out.print("4");   
    }   
  }   
} //234

*Пояснение: На экран выведется 234, так как метод contains в классе Hashtable проверяет наличие значений в списке values, а не keys.*

Какой результат выполнения данного кода:

**class** One {   
    **public** **static** **int** j = 90;   
    One()   
    { j = 12; }   
}   
**public** **class** Test **extends** One {   
    **public** **static** **void** main(String args[]) {   
        **new** Test();   
        System.out.println(One.j);   
    }   
} //12

Выберите из приведенных все модификаторы, которые применимы для top-level классов (не вложенных).

public

final

*Пояснение: Модификаторы private и protected применимы только к вложенным классам-членам (member class), модификатор transient – только к полям*

Какой результат выполнения данного кода:

System.out.println(0.0/0.0 + 10);  //NaN

Что произойдет в результате попытки компиляции и выполнения программы?

**class** TreeTest {   
    **private** **static** Set<String> set = **new** TreeSet<String>();   
    **public** **static** **void** main(String[] args) {   
        set.add("one");   
        set.add("two");   
        set.add("three");   
        set.add("\u000a");   
        set.add("\u000d");   
        set.add("\u000c");   
        set.add("1");   
        set.add("2");   
        set.add("3");           
        **for** (String string : set) {   
            System.out.print(string + " ");   
        }   
    }    
} //compile error

*Пояснение: \u000a соответствует символу перевода строки.  
\u000d соответствует символу возврата каретки.  
Именно так эти символы воспринимает компилятор, поэтому использовать их в строке не стоит.*

Является ли пустой файл разрешенным для компиляции файлом исходного кода? - yes.

Что напечатает следующий код:

**public** **class** Funcs **extends** java.lang.Math {   
    **public** **int** add(**int** x, **int** y) {   
        **return** x + y;   
    }   
    **public** **int** sub(**int** x, **int** y) {   
        **return** x - y;   
    }   
    **public** **static** **void** main(String[] a) {   
        Funcs f = **new** Funcs();   
        System.out.println("" + f.add(1, 2));   
    }   
} //compile error

*Пояснение: Класс java.lang.Math объявлен как final, то есть наследоваться от него нельзя.*

System.out.println(Math.sqrt(-2.0)); //NaN  
System.out.println(1.0/0.0);  //Infinity

В каких строчках будет ошибка компиляции?

**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        **char** c = (**char**) 1; // ok  
        **boolean** boo = (**boolean**) 1; // error  
        **byte** b = (**byte**) **true**;    // error  
    }   
}

Что произойдет при выполнении такого кода:

**import** java.util.\*;   
**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        List a = **new** ArrayList<Double>();// 4   
        a.add("1.5");// 5   
        List<Double> b = **new** ArrayList();// 6   
        b.add("1.5");// 7  //error  
        System.out.println(a + " " + b);   
    }   
}

*Пояснение: Ошибка компиляции возникает в 7-ой строке, потому что в коллекцию "b" можно добавлять только элементы типа Double и производного от него, а в коллекцию "a" можно добавлять элементы любого типа, так параметризованный тип Double в ней повышается до Object.*

Что будет выведено на экран в результате выполнения программы ?

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        Formatter formatter = **new** Formatter(Locale.ROOT);   
        formatter.format("%.2E\n", 100.0/3.0);      //1   
        formatter.format("%.2f", 100.0/3.0);        //2   
        System.out.println(formatter);   
    }   
} // 3.33E+01  
 // 33.33

Какой результат выполнение данного кода?

**import** java.util.Arrays;   
   
**class** Generic<T **extends** Number> {    
    **private** T arr[] = {1, 2, 3, 4, 5}; //error  
    
    **public** String toString(){    
        **return** Arrays.toString(arr);   
    }    
}    
   
**public** **class** Test {    
    **public** **static** **void** main(String [] args) {    
        Generic<Double> obj = **new** Generic<Double>();   
        System.out.println(obj);    
    }    
} //compile error

*Пояснение: Ошибка компиляции при объявлении массива.  
Списки инициализации не могут быть использованы для инициализации параметризованных массивов.*

Что произойдет при попытке откомпилировать/запустить следующий код:

**abstract** **class** Animal {   
    **static** {   
        System.out.println("Inside Animal");   
    }   
}   
   
**class** Cat **extends** Animal {   
    **static** {   
        System.out.println("Inside Cat");   
    }   
}   
   
**class** Dog **extends** Animal {   
    **static** {   
        System.out.println("Inside Dog");   
    }   
}   
   
**public** **class** Main {   
     **public** **static** **void** main(String[] args){   
          Animal cat = **new** Cat();   
          Animal dog = **new** Dog();   
     }   
}

В консоли будет выведено: Inside Animal Inside Cat Inside Dog

*Пояснение: Блок статической инициализации вызывается всего один раз, когда загружается класс.*

Что выведет на экран следующий код ?

**class** TT {   
    **private** **static** **int** count = 0;   
    **private** **final** **int** id = ++count;   
   
    **private** **void** methodA () {   
        System.out.println("TT.methodA " + id);   
    }   
   
    **public** **void** methodTT(TT t) {   
        t.methodA();   
    }   
}   
   
   
**public** **class** Test {   
    **public** **static** **void** main(String[] args) {   
        TT t1 = **new** TT();   
        TT t2 = **new** TT();   
        t2.methodTT(t1);   
    }       
} // TT.methodA 1

*Пояснение: При выполнении кода создадутся два объекта класса TT с полями у первого класса id = 1 у второго id = 2. При вызове метода methodTT() объекта t2 в метод передастся ссылка на объект t1. Далее у объекта t1 вызовется метод methodA() который напечатает TT.methodA 1.*

Являются ли массивы в Java однородными?

Да. За исключением случаев, разрешённых полиморфизмом.

*Пояснение: Массивы в Java однородные, то есть все элементы в массиве должны быть одного типа, за исключением случаев, разрешённых полиморфизмом. Иными словами, когда вы создаёте массив, вы должны определить тип элементов, которые могут в нём содержаться. Если тип не примитивный, а определённый через класс, то реальный массив, помимо объектов заданного класса, может содержать также объекты подклассов того класса, который был указан в объявлении массива.*

Что напечатается в результате компиляции и выполнения следующего кода?

**public** **class** Test {   
    **private** **static** Boolean b1, b2;   
       
    **public** **static** **void** main (String[] args) {   
        **if** (b1 || !b2 || !b1 || b2) {   
            System.out.println(**true**);   
        }   
        **else** {   
            System.out.println(**false**);   
        }   
    }   
} //runtime error. NPE.

*Пояснение: При использовании объектов-обёрток в операциях, предназначенных для примитивов, вызываются методы, возвращающие примитивное значение. В данном случае booleanValue(). Это вызовет выбрасывание NullPointerException.*

Какой результат работы следующей программы:

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **final** Integer i = 1;   
        **switch** (1) {   
            **case** i:   
                System.out.println("1");   
                **break**;   
            **default**:   
                System.out.println("default");   
        }   
    }   
} //compile error

*Пояснение: В case-секциях оператора switch можно использовать переменные, если они объявлены как final. Это условие не выполняется в случае переменных, для которых нужен autounboxing.*

Рассмотрим код:

**public** **class** MyCalculator {   
    **public** **static** **void** main(String[] args) {   
     // Insert here   
    }   
}

Что из нижеуказанного надо вставить в код, чтобы напечаталось 3.1416?

System.out.printf("%6.4f", Math.PI);

Что будет выведено следующей программой?

**public** **class** Main {   
  {   
    System.out.print("1 ");   
  }   
   
  Main(){   
    System.out.print("2 ");   
  }   
   
  **public** **static** **void** main(String[] args) {   
    System.out.print("3 ");   
    Main m = **new** Main();   
    System.out.print("4 ");   
  }   
   
  **static** {   
    System.out.print("5 ");   
  }   
} //5 3 1 2 4

*Пояснение: Первым инициализируется статический блок, далее управление передается main, вызывается конструктор и доинициализирует класс, вызывается функция внутри конструктора, окончание работы.*

Что произойдет при попытке скомпилировать и запустить данный код? Считается, что файл outstream уже создан.

**import** java.io.\*;   
   
**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **try** {   
            FileInputStream fis = **new** FileInputStream("outstream");   
            InputStreamReader isr = **new** InputStreamReader(fis);   
            BufferedReader br = **new** BufferedReader(isr);   
            **try** {   
                **do** {   
                    String str = br.readLine();   
                    System.out.println(str);   
                } **while** (str != **null**);                
            }   
            **finally** {   
                fis.close();   
                isr.close();   
                br.close();   
            }   
        } **catch**(IOException e){   
            e.printStackTrace();   
        }   
    }   
} // compile error

*Пояснение: Ошибка компиляции произойдёт из-за того, что переменная str объявлена внутри цикла do.. while().  
Объявлять переменные в цикле, конечно же, разрешается. Только и использоваться они должны внутри тела цикла. А в этом примере переменная str используется за пределами блока, в котором она была объявлена - в условии продолжения цикла.*

Что выведется на экран при выполнении следующего кода?

**public** **class** GoodbyeWorld {       
    **public** **static** **void** main(String[] args) {   
        System.out.print(A.i);   
        System.out.print(B.i);   
    }   
}   
   
**class** A {   
    **static** {   
        i = 2;   
    }   
    **static** **int** i = 1;   
};   
   
**class** B {   
    **static** **int** i = 1;   
    **static** {   
        i = 2;   
    }   
}; //12

*Пояснение: Инициализация всех статических полей и блоки статической инициализации выполняются друг за другом, в том порядке, в котором они записаны.  
Поэтому в классе A поле i будет иметь значение 1, а в классе B — 2.*

Зачем нужна аннотация @Override когда вы переопределяете (или реализовываете) методы?

Чтобы своевременно обнаружить изменение в сигнатуре метода у класса-предка (или интерфейса).

**public** **class** Cycle {   
    **public** **static** **void** main(String...args) {   
        **int** i = 1;   
        **do** **while**(i < 1)   
            System.out.println("i = " + ++i);   
        **while** (i > 1);   
    }   
}//Программа выполнится, но ничего не выведет на экран

*Пояснение: Пониманию исходного кода мешает отстутствие фигурных скобок. Расставим их:*

**do** {   
     **while** (i < 1) {  // false при i = 1   
         System.out.println("i = " + ++i);   
     }   
} **while** (i > 1);  // false при i = 1

Теперь становится очевидно, что программа выполнится успешно, но ничего не выведет на экран.

Какой результат выполнения данного кода:

Boolean b = **new** Boolean("Hello");   
System.out.println(b);  //false

true тільки тоді, коли передається "true" без врахування регістру.

Что будет выведено на экран в результате выполнения кода?

**import** java.util.TreeSet;   
   
**public** **class** TestTreeSet {   
   
    **public** **static** **void** main(String[] args) {   
        TreeSet<Item> set = **new** TreeSet<Item>();   
        set.add(**new** Item(2));   
        set.add(**new** Item(5));   
        set.add(**new** Item(2));   
        System.out.println(set);   
    }   
   
    **static** **class** Item {   
        **int** n;   
        Item(**int** n) {   
            **this**.n = n;   
        }   
        **public** String toString() {   
            **return** "Item " + n;   
        }   
    }   
}// runtime error

*Пояснение: Будет выброшен ClassCastException, т.к. в TreeSet для упорядочивания элементов используется метод compareTo(или compare), соответственно, в нем могут храниться объекты, реализующие интерфейс Comparable, либо необходимо передавать свою реализацию Comparator-а в конструкторе.*

Что выведет в консоль следующий код?

**import** java.util.List;   
**import** java.util.concurrent.CopyOnWriteArrayList;   
   
**public** **class** Test {   
   
    **private** **static** **void** removeAndPrint(List<String> list) {   
        **for** (String str : list) {   
            **if** (str.equals("two")) {   
                list.remove("three");   
            }   
        }   
        System.out.println(list);   
    }   
   
    **public** **static** **void** main(String[] args) {   
        List<String> list = **new** CopyOnWriteArrayList<String>();   
           
        list.add("one");   
        list.add("two");   
        list.add("three");   
        list.add("four");   
   
        removeAndPrint(list);   
   
    }   
} // one, two, four

*Пояснение: CopyOnWriteArrayList - это разновидность List нормально переносит вызов метода remove() при перемещении по списку, не выдавая исключения ConcurrentModificationException.*

Выведется ли текст, помещенный в блок условного оператора?

**public** **static** **void** main(String[] args){   
       **int** x = 18;   
       **int** y = x++;   
       **if** (x == 18 && y > 10){   
              System.out.println("y = " + y);   
              System.out.println("x = " + x);   
       }   
} //no

*Пояснение: После присвоения значения переменной "y", значение "x" будет увеличено на один(т.е. = 19).*

Что произойдет при попытке скомпилировать и запустить данный код:

**import** java.util.\*;   
**public** **class** Clazz {   
  **public** **static** **void** main(String[] args) {   
    List arrayList = **new** ArrayList();    
    arrayList.add("str1");    
    arrayList.add("str2");    
    arrayList.add("str3");    
   
    **for** (**int** i = 0; i < arrayList.size(); i++)     
        arrayList.remove(i);    
   
    System.out.println(arrayList.size());    
  }   
}//1

*Пояснение: Исходное состояние списка: (0:"str1"; 1:"str2"; 2:"str3"), size() = 3  
i = 0 → удаляем элемент "str1" → (0:"str2"; 1:"str3"), size() = 2  
i = 1 → удаляем элемент "str3" → (0:"str2"), size() = 1  
i = 2 → конец цикла, т.к. i > size()  
Ответ: size() = 1*

Каким будет результат компиляции и запуска следующей программы?

**public** **class** SleepMain {   
    **public** **static** **void** main(String... args) {   
        Thread t = **new** MyThread();   
        **for** (**int** i = 1; i <= 5; i++) {   
            System.out.print(i + " ");   
            **try** {   
                t.sleep(1000);   
            } **catch** (InterruptedException e) {   
                System.out.println("Interrupted in main");   
            }   
        }   
    }   
   
    **static** **class** MyThread **extends** Thread {   
        **public** **void** run() {   
            **for** (**int** i = 1; i <= 5; i++) {   
                System.out.print(i + " ");   
                **try** {   
                    Thread.sleep(1000);   
                } **catch** (InterruptedException e) {   
                    System.out.println("Interrupted in myThread");   
                }   
            }   
        }   
    }   
} //12345

*Пояснение: Для запуска второго потока надо было вызвать метод t.start(). Без этого работает только исходный поток.  
Метод sleep() класса Thread статический, потому по команде t.sleep() будет засыпать текущий поток, т.е. main.*

Что будет выведено в результате компиляции и запуска:

**public** **class** Quest {   
    **public** **static** **void** main() {   
        System.out.print("А");   
   }   
    **public** **static** **void** main(String args) {   
        System.out.print("Б");   
    }   
    **public** **static** **void** main(String[] args) {   
        System.out.print("В");   
    }   
}  //B

*Пояснение: Методы main() являются корректно перегруженными, поэтому никаких ошибок не произойдет, а будет вызван только метод, запускающий приложение.*

Какие из приведенных выражений будут истинными при выполнении кода:

String a = **new** String("hello");   
String b = **new** String(a);   
String c = a;   
**char**[] d = { 'h', 'e', 'l', 'l', 'o' };

// a == c

// a.equals(b)

Выберите из приведенных все модификаторы, которые применимы для top-level классов (не вложенных).

// public

// final

Каким будет результат выполнения данного кода:

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **int** var = 1;   
        System.out.println("Переменная var (постфиксный инкремент): " + var++);   
        System.out.println("Переменная var (префиксный инкремент): " + ++var); //1   
        **int** var1 = 1;   
        **int** var2 = 1;   
        **if** (var1++ == ++var2)  //2   
            System.out.println("Переменные равны");   
        **else**    
            System.out.println("Переменные не равны");   
    }   
}

Переменная var (постфиксный инкремент): 1  
Переменная var (префиксный инкремент): 3  
Переменные не равны

*Пояснение: Операторы инкремента и декремента имеют 2 формы: постфиксная и префиксная. Разница между формами в том, что постфиксная форма оператора выполняется после вычисления выражения, а префиксная до. Т.о. строка с постфиксным выражением var++ даст нам исходное значение переменной var (1), а префиксное — новое значение (3).  
По этой же причине при сравнении значений выражений var1++ и ++var2 они будут не равны.*

Каким будет результат компиляции и выполнения данного кода?

01. **class** A {   
02.     **public** **void** process() { System.out.print("A "); }   
03. }   
04. **public** **class** B **extends** A {   
05.     **public** **void** process() **throws** RuntimeException {   
06.         **super**.process();   
07.         **if** (**true**) **throw** **new** RuntimeException();   
08.         System.out.print("B ");   
09.     }   
10.    
11.     **public** **static** **void** main(String[] args) {   
12.         **try** { ((A)**new** B()).process(); }   
13.         **catch** (Exception e) { System.out.print("Exception "); }   
14.     }   
15. } // A Exception

*Пояснение: Хотя при переопределении метода нельзя добавлять исключение, на RuntimeException (исключениe времени исполнения) это ограничениe не распространяется.*

Какой результат выполнения данного кода?

**abstract** **class** Test{   
    **private** **static** **int** getHalf(**int** i){   
        **return** i/2;   
    }   
    **public** **static** **void** main(String[] str){   
        **int** half=getHalf(0);   
        System.out.println("Result is:"+half);   
    }   
} // result = 0.

*Пояснение: Абстрактный класс может не содержать абстрактных методов. Если не создавать экземпляров этого класса, то код выполняется и работает.*

Что произойдет при попытке скомпилировать и запустить данный код:

**import** java.util.\*;   
**public** **class** Clazz {   
  **public** **static** **void** main(String[] args) {   
    List arrayList = **new** ArrayList();    
    arrayList.add("str1");    
    arrayList.add("str2");    
    arrayList.add("str3");    
   
    **for** (**int** i = 0; i < arrayList.size(); i++)     
        arrayList.remove(i);    
   
    System.out.println(arrayList.size());    
  }   
} // 1

*Пояснение: Исходное состояние списка: (0:"str1"; 1:"str2"; 2:"str3"), size() = 3  
i = 0 → удаляем элемент "str1" → (0:"str2"; 1:"str3"), size() = 2  
i = 1 → удаляем элемент "str3" → (0:"str2"), size() = 1  
i = 2 → конец цикла, т.к. i > size()  
Ответ: size() = 1*

Что произойдет при вызове System.arraycopy() с одним и тем же массивом в качестве src и dist?

Метод відпрацює коректно.

**public** **class** Test {   
    **static** String s;   
       
    **static** **void** go() {   
        System.out.println(s);   
        go();   
    }   
   
    **public** **static** **void** main(String[] args) {   
        go();   
    }   
} //SOE. Безкінечної рекурсії в Джава нема.

Что произойдет при компиляции и запуске данного кода?

**abstract** **public** **class** Parent {   
    String s = "hello!";   
    **public** **void** test() {   
        System.out.println(s);   
    }   
}   
   
**class** P {   
    **public** **static** **void** main(String[] args) {   
        Parent p = **new** Parent();   
        p.test();   
    }   
} // compile error

*Пояснение: Нельзя создавать экземпляры абстрактного класса используя оператор new, так как он не определен полностью*

Что произойдет в результате компиляции и выполнения следующего кода?

**import** java.util.\*;   
**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        **try** {   
            ArrayList<Integer> ar = **new** ArrayList<Integer>();   
            List temp = ar; //1   
            temp.add(**new** java.util.Date()); //2   
            temp.add(**new** Float(1.66));   
            Iterator it = ar.iterator();   
            **while** (it.hasNext())   
                System.out.println(it.next());   
            System.out.println(ar.get(0));   
        } **catch** (Exception e) {   
            System.out.println(e.getClass());   
        }   
    }   
} // successfully compile

*Пояснение:*

Коллекция temp не является типизированной, поэтому метод temp.add() успешно выполняется даже для объектов "неправильного" типа.

Исключение возникнет, если при обращении к элементам коллекции ar явно или неявно будет выполняться преобразование к типу Integer. В данном примере такого нет, поэтому обход коллекции также выполнится успешно

Чтобы предотвратить добавление в коллекцию "неправильных" элементов, можно использовать представление с проверкой, например, такое:

List temp = Collections.checkedList(ar, Integer.**class**);

Какой будет результат компиляции и выполнения следующего кода?

**public** **class** Test {   
    **public** **static** **void** main(String... args) {   
        test("A", "B");   
    }   
   
    **public** **static** **void** test(String... str) {   
        System.out.print("A");   
    }   
   
    **public** **static** **void** test(String str1, String str2) {   
        System.out.print("B");   
    }   
   
    **public** **static** **void** test(String str1, String... str2) {   
        System.out.print("C");   
    }   
} //B

*Пояснение: Каждый из трёх методов test() является потенциально применимым для вызова test("A", "B").  
Поскольку среди них есть только один метод с фиксированным количеством аргументов, то именно он и будет вызван.*

Какой результат выполнения данного кода:

**try** {   
    **int** i = **new** Integer(10);   
    System.out.println(i);   
} **catch**(Exception e) {   
    System.out.println("opsss... error");   
} //10

Какой результат выдаст следующий код?

**public** **class** A {   
    **public** **int** i = 0;   
   
    **public** A() {   
        i = 10;   
    }   
   
    **public** **static** **void** main(String[] args) {   
        **int** i = 9;   
        A a = **new** A();   
        **while**(a.i < 10) a.doIt(); //9   
    }   
   
    **public** **static** **void** doIt() {   
        i++;                             //12   
        System.out.println("Hello");   
    }   
} // compile error in 12

Следует помнить, что любое нестатическое поле должно принадлежать какому-то объекту.  
В 9-ой строке происходит обращение к полю i объекта a, при этом не важно, внутри какого (статического или нестатического) метода делается это обращение - ведь целевой объект у нас указан.  
В 12-ой строке тоже происходит обращение к полю i, но объект не указан. Поэтому компилятор пытается подставить ссылку на текущий объект, т.е. this. А поскольку использовать this внутри статического метода нельзя, то это приводит к ошибке.

Каким будет результат выполнения следующей программы?

**public** **class** SwitchTest {   
    **public** **static** **void** main(String[] args) {   
        **byte** k = 0;   
           
        **switch**(k) {   
            **default**:   
                **int** h = 3;   
                System.out.println(h);   
                **break**;   
            **case** 0:   
                **int** h = 7;   
                System.out.println(h);   
            **case** 1:   
                **int** h = 11;   
                System.out.println(h);   
        }   
    }     
} //compile error

*Пояснение: Возникнет ошибка компиляции "h is already defined".  
Локальные переменные, объявленные в блоке switch, имеют общую область видимости.  
Чтобы избежать ошибки компиляции, можно использовать отдельный блок { } для каждого case / default.*

Что произойдет в результате компиляции и выполнения программы:

**public** **class** Main {       
    **public** **static** **void** main(String[] args) {   
        String strA = "text";   
        String strB = "text";   
        strA += "1";   
        strB += "1";   
        System.out.println(strA != strB);   
        strA = "text1";   
        strB = "text1";   
        System.out.println(strA != strB);   
    }   
}

//true

//false

*Пояснение: Во-первых, нужно помнить, что строковые литералы в Java - это, фактически, объекты (String). При этом, если в программе несколько раз встречается один и тот же строковый литерал, то ему будет соответствовать один и тот же объект. Поэтому во втором случае strA == strB (и, соответственно, выводится false).  
  
Во-вторых, нужно помнить, что строки (String) в Java - это неизменяемые объекты. Поэтому, когда к объекту strA "приклеивается" другая строка (strA += "1"), на самом деле создаётся новый объект, содержащий "text1". И когда приклеиваем к объекту strB - создаётся ещё один (другой!) объект, также содержащий "text1". Следовательно, strA != strB, и в первом случае выводится true.  
  
Фактически, команда strA += "1" разворачивается в strA = new StringBuilder().append(strA).append("1").toString(); И если это выполнится два раза, то получится два разных объекта, пусть и с одинаковым содержимым.*

Что будет выведено на экран в результате выполнения программы ?

**public** **class** Main {   
    **public** **static** **void** main(String[] args) {   
        Formatter formatter = **new** Formatter(Locale.ROOT);   
        formatter.format("%.2E\n", 100.0/3.0);      //1   
        formatter.format("%.2f", 100.0/3.0);        //2   
        System.out.println(formatter);   
    }   
}

3.33E+01  
33.33

*Пояснение: Formatter - форматирующий класс в Java. Результатом выполнения строки 1 будет 3.33E+01, т.к. ".2" округляет выражение до 2-го знака после запятой, а Е преобразует выражение в инженерный вид.  
Результатом выполнения строки 2 будет 33.33, здесь также выполняется округление до 2-го знака после запятой.  
Без явного указания Locale мы бы получили результат в соответствии с тем, какой разделитель дробной и целой части установлен в системе. Так, например, в русской версии Windows мы бы получили 33,33.*

Из какой структуры данных "сборщик мусора" удалит все элементы у которых исчезла последняя ссылка на их ключ в этой структуре?

WeakHashMap

*Пояснение:*

**WeakHashMap**, фактически, хранит не пары "ключ - значение", а пары "слабая ссылка на ключ - значение". Особенность слабых ссылок (WeakReference) состоит в том, что они игнорируются сборщиком мусора, т.е. если на объект-ключ нет других ссылок, он уничтожается.

Перед любым обращением к WeakHashMap (get(), put(), size() и т.д.) анализируются невалидные ссылки и соответствующая пара удаляется.

Код для демонстрации:

Map map = **new** WeakHashMap();   
Object obj = **new** Object(); // создаём объект   
map.put(obj, "object"); // кладём его в мапу   
System.out.println(map.size()); // в мапе один элемент   
obj = **null**; // чистим ссылку   
System.gc(); // играемся со сборщиком мусора   
System.runFinalization();   
System.out.println(map.size()); // мапа должна стать пустой

Что будет выведено на экран?

**class** A {   
    **int** i = 0;   
    **public** **int** increment() {   
        **return** ++i;   
    }   
}   
    
**public** **class** B **extends** A {   
    **int** i = 10;   
    **public** **int** increment() {   
        **return** ++i;   
    }   
    
    **public** **static** **void** main(String[] args) {   
        A obj = (A) **new** B();    
        System.out.println(obj.increment());   
    }   
} // 11

*Пояснение: Метод A.increment() перекрывается в классе B.  
Поэтому, вне зависимости от типа переменной obj, будет вызываться метод, соответствующий реальному типу объекта - т.е. B.increment().  
Этот метод будет использовать переменную i, объявленную в том же классе B и имеющую начальное значение 10.  
Результирующее значение - 11.*

Что произойдет в результате компиляции и запуска следующего кода:

**public** **class** Bar **extends** Foo {               // 1   
    **public** **static** **void** main(String[] args) { // 2   
        System.out.print(sum(1, 2));         // 3   
    }   
}   
**class** Foo {   
    **int** sum(**int** x, **int** y) {   
        **return** x + y;   
    }   
} // compile error in 3

*Пояснение: Ошибка компиляции в строке 3, так как нельзя вызывать нестатический метод из статического контекста*

Если переопределить в классе метод finalize(), то для объекта этого класса этот метод будет вызван:

Перед тем, как система сборки мусора уничтожит объект

У каких операторов всегда вычисляются все операнды (выберите все подходящие варианты)?

%

*Пояснение: Оператор || не вычисляет значение второго операнда, если первый операнд имеет значение true.  
Оператор && не вычисляет значение второго операнда, если первый операнд имеет значение false.  
Оператор ?: в зависимости от истинности условия вычисляет значение только одного из двух выражений.*

Какой результат выполнения данного кода:

**int** a = 7;   
**int** b = 4;   
   
System.out.println(-a % b);   
System.out.println(a % -b);   
System.out.println((-a % b) == (a % -b));

//-3

// 3

// false

Что произойдет во время компиляции и выполнения данного фрагмента кода?

**int** i = 0, j = 5;   
tp: **for** (;;) {   
    i++;   
    **for** (;;) {   
        **if** (i > --j) {   
            **break** tp;   
        }   
    }   
    System.out.println("i =" + i + ", j = " + j);   
} //compile error

*Пояснение: Ошибка компиляции из-за недостижимости кода "System.out.println("i =" + i + ", j = " + j);"*

То есть должен быть обязательно какой-то выход из бесконечного цикла. А в первом цикле такого явного выхода нет.

Что произойдет при компиляции и исполнении кода:

**public** **class** GoTest {   
    **public** **static** **void** main(String[] args) {   
        Sente a = **new** Sente(); a.go();   
        Goban b = **new** Goban(); b.go();   
        Stone c = **new** Stone(); c.go();   
    }   
}    
   
**class** Sente **implements** Go {   
    **public** **void** go() { System.out.println("go in Sente"); }   
}    
   
**class** Goban **extends** Sente {   
    **public** **void** go() { System.out.println("go in Goban"); }   
}    
   
**class** Stone **extends** Goban **implements** Go { }   
   
**interface** Go { **public** **void** go(); }

go in Sente  
go in Goban  
go in Goban

Какой будет результат компиляции и выполнения данного кода:

**class** A {   
    **private** **static** **void** test1() {    
        System.out.println("A.test1");   
    }    
       
    **static** **void** test2() {   
        test1();   
    }    
}    
   
**public** **class** B **extends** A {    
    **void** test1() {   
        System.out.println("B.test1");   
    }    
    **static** **void** test2() {   
        **super**.test2();   
    }    
    **public** **static** **void** main(String[] args) {    
        A a = **new** B(); a.test2();    
    }   
} //compile error

*Пояснение: Нельзя использовать ключевое слово* ***super*** *в* ***static*** *методах.*

Какие реализации java.util.Set сортируют элементы в их естественном порядке (или на основании Comparator'а)?

java.util.TreeSet

*Пояснение: TreeSet сортирует элементы (она реализует SortedSet). LinkedHashSet позволяет управлять порядком добавления элементов, но не сортирует их.*

Что произойдет в результате компиляции кода?

1. **package** mail;   
2.   
3. **interface** Box {   
4.     **protected** **void** open();   
5.     **void** close();   
6.     **public** **void** empty();   
7. } //cimpile error in line 4

*Пояснение: Для методов, объявляемых в интерфейсах, разрешается использовать только модификаторы public и/или abstract.*

Что напечатается в результате компиляции и выполнения следующего кода?

**public** **class** Test {   
    **private** **static** Boolean b1, b2;   
       
    **public** **static** **void** main (String[] args) {   
        **if** (b1 || !b2 || !b1 || b2) {   
            System.out.println(**true**);   
        }   
        **else** {   
            System.out.println(**false**);   
        }   
    }   
} //runtime error. NOT compile!!!

Что будет напечатано следующим кодом?

**public** **class** Main {   
   
    **private** **static** **class** A1 {   
        **private** String test() {   
            **return** "A1";   
        }   
    }   
   
    **public** **static** **class** A2 **extends** A1 {   
        **public** String test() {   
            **return** "A2";   
        }   
    }   
   
    **public** **static** **class** A3 **extends** A2 {   
        **public** String test() {   
            **return** "A3";   
        }   
    }   
   
    **public** **static** **void** main(String ... arg) {   
        A1 a1 = **new** A1();   
        System.out.println(a1.test());   
        a1 = **new** A2();   
        System.out.println(a1.test());   
        a1 = **new** A3();   
        System.out.println(a1.test());   
    }   
   
} // A1 A1 A1

*Пояснение:* ***Private****-методы* ***не******наследуются****, их нельзя перекрыть (override) в классах-потомках, для них не используется механизм позднего связывания. Поэтому всегда будет вызываться метод test() из класса A1 — исходя из типа переменной-ссылки, а не фактического типа объекта.*

Что напечатает следующий код:

**int** i = 0;   
System.out.print(++i);   
System.out.print(i++);   
System.out.print(i);

//112

Какие из перечисленных объявлений переменных не допустимы в Java?

int goto;

int if;

int else;

Можна використовувати - int then;

Каким образом можно запретить наследование класса (речь идет о top-level классах) ?

Добавить модификатор final

Добавить модификатор private

Каким будет результат выполнения кода:

**package** tests;   
   
**import** java.util.Hashtable;   
   
**public** **class** Test {   
  **public** **static** **void** main(String[] args) {   
    Hashtable ht = **new** Hashtable();   
    ht.put("1", "2");   
    ht.put("2", "3");   
    ht.put("3", "4");   
    ht.put("4", "2");   
       
    **if**(ht.contains("1")){   
      System.out.print("1");   
    }   
    **if**(ht.contains("2")){   
      System.out.print("2");   
    }   
    **if**(ht.contains("3")){   
      System.out.print("3");   
    }   
    **if**(ht.contains("4")){   
      System.out.print("4");   
    }   
  }   
} // 234

Дан двумерный массив(матрица)

**int**[][] arr = {{1,2,3}, {0,0,0}, {3,2,1}};

Какие из следующих вариантов ответа позволят пройтись по всем элементам массива и вывести их на экран при помощи for-each?

**for** (**int** i[] : arr) {   
    **for** (**int** j : i)   
        System.out.print(j + " ");   
    System.out.println();   
}