Roman Beekeeper

автор тг-канала в t.me/romankh3

16.03.2020 164315 74

Топ-50 Java Core вопросов и ответов на собеседовании. Часть 1

Всем привет, дамы и господа Software Engineers! Давайте поговорим о вопросах на собеседовании. О том, к чему нужно готовиться и что нужно знать. Это отличный повод для того, чтобы повторить или же изучить с нуля эти моменты.

У меня получилась довольно объемная подборка часто задаваемых вопросов об ООП, Java Syntax, исключениях в Java, коллекциях и многопоточности, которую для удобства я разобью на несколько частей.

`Важно:` мы будем говорить только о версии Java до 8. Все нововведения с 9, 10, 11, 12, 13 не будут учитываться здесь. Любые идеи/замечания, как улучшить ответы, приветствуются.

Приятного прочтения, поехали!

Java собеседование: вопросы по ООП

1. Какие особенности есть у Java?

Ответ:

1. ООП концепты:
    1. объектная ориентированность;
    2. наследование;

4. полиморфизм;
5. абстракция.

2. Кроссплатформенность: программа на Java может быть запущена на любой платформе без каких-либо изменений. Единственное, что нужно — установленная JVM (java virtual machine).

3. Высокая производительность: JIT(Just In Time compiler) позволяет высокую производительность. JIT конвертирует байт-код в машинный код и потом JVM стартует выполнение.

4. Мультипоточность: поток выполнения, известный как `Thread`. JVM создает thread, который называется `main thread`. Программист может создать несколько потоков наследованием от класса Thread или реализуя интерфейс `Runnable`.

2. Что такое наследование?

Под наследованием подразумевается, что один класс может наследовать(“extends”) другой класс.

Таким образом можно переиспользовать код с класса, от которого наследуются.

Существующий класс известен как `superclass`, а создаваемый — `subclass`. Также еще говорят `parent` и `child`.

```
public class Animal {
   private int age;
}

public class Dog extends Animal {

}
```

где `Animal` — это `parent`, а `Dog` — `child`.

3. Что такое инкапсуляция?

Такой вопрос часто встречается на собеседовании Java-разработчика. Инкапсуляция — это сокрытие реализации при помощи модификаторов доступа, при помощи геттеров и сеттеров. Это делается для того, чтобы закрыть доступ для внешнего использования в тех местах, где разработчики считают нужным.

Доступный пример из жизни — это автомобиль. У нас нет прямого доступа к работе двигателя. Для нас работа заключается в том, чтобы вставить ключ в зажигание и включить двигатель. А какие уже процессы будут происходить под капотом — не наше дело.

Даже более того, наше вмешательство в эту деятельность может привести к непредсказуемой ситуации, из-за которой можно и машину сломать, и себе навредить. Ровно то же самое происходит и в программировании.

Хорошо описано в [википедии](#).
Статья об инкапсуляции есть и на [JavaRush](#).

4. Что такое полиморфизм?

Полиморфизм — это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта. Как говорится, один интерфейс — множество реализаций.

При помощи полиморфизма можно объединять и использовать разные типы объектов по их общему поведению.

Например, есть у нас класс Animal, у которого есть два наследника — Dog и Cat. У общего класса Animal есть общее поведение для всех — издавать звук. В случае, когда нужно собрать воедино всех наследников класса Animal и выполнить метод “издавать звук”, используем возможности полиморфизма. Вот как будет это выглядеть:

```java
List<Animal> animals = Arrays.asList(new Cat(), new Dog(), new Cat());
animals.forEach(animal -> animal.makeSound());
```

Таким образом, полиморфизм помогает нам. Причем это относится и к полиморфным (перегруженным) методам.

Практика использования полиморфизма

Вопросы на собеседовании — Java Syntax

5. Что такое конструктор в Java?

Следующие характеристики являются валидными:

1. Когда новый объект создается, программа использует для этого соответствующий конструктор.
2. Конструктор похож на метод. Его особенность заключается в том, что нет возвращающего элемента (в том числе и void), а его имя совпадает с именем класса.
3. Если не пишется никакого конструктора явно, пустой конструктор будет создан автоматически.
4. Конструктор может быть переопределен.
5. Если был создан конструктор с параметрами, а нужен еще и без параметров, его нужно писать отдельно, так как он не создается автоматически.

6. Какие два класса не наследуются от Object?

Не ведитесь на провокации, нет таких классов: все классы прямо или через предков наследуются от класса Object!

7. Что такое Local Variable?

Еще один из популярных вопросов на собеседовании Java-разработчика. Local variable — это переменная, которая определена внутри метода и существует вплоть до того момента, пока выполняется этот метод. Как только выполнение закончится, локальная переменная перестанет существовать.

Вот программа, которая использует локальную переменную helloMessage в методе main():

```java
public static void main(String[] args) {
   String helloMessage;
   helloMessage = "Hello, World!";
   System.out.println(helloMessage);
}
```

8. Что такое Instance Variable?

Instance Variable — переменная, которая определена внутри класса, и она существует вплоть до того момента, пока существует объект.

Пример — класс Bee, в котором есть две переменные nectarCapacity и maxNectarCapacity:

```java
public class Bee {

   /**
    * Current nectar capacity
    */
   private double nectarCapacity;

   /**
    * Maximal nectar that can take bee.
```

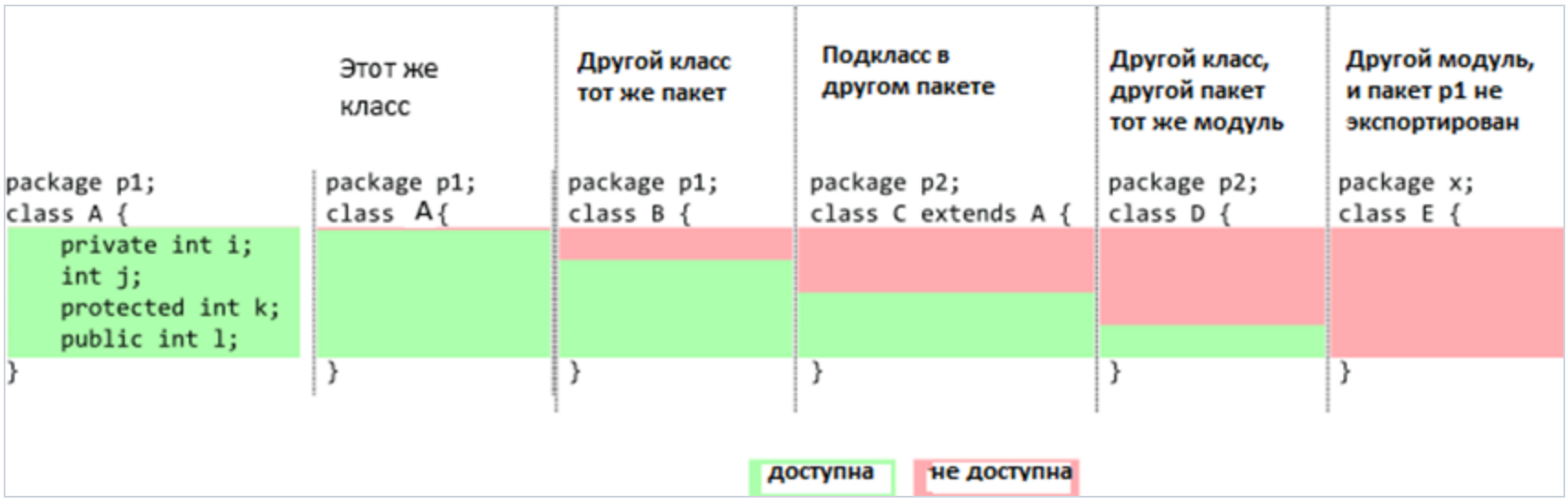
```
12
13      ...
14  }
```

9. Что такое модификаторы доступа?

Модификаторы доступа — это инструмент, при помощи которого можно настроить доступ к классам, методам и переменным.

Бывают следующие модификаторы, упорядоченные в порядке повышения доступа:

1. `private` — используется для методов, полей и конструкторов. Уровень доступа — только класс, внутри которого он объявлен.
2. `package-private(default)` — может использоваться для классов. Доступ только в конкретном пакете (package), в котором объявлен класс, метод, переменная, конструктор.
3. `protected` — такой же доступ, как и `package-private` + для тех классов, которые наследуются от класса с модификатором `protected`.
4. `public` — используется и для классов. Полноценный доступ во всем приложении.



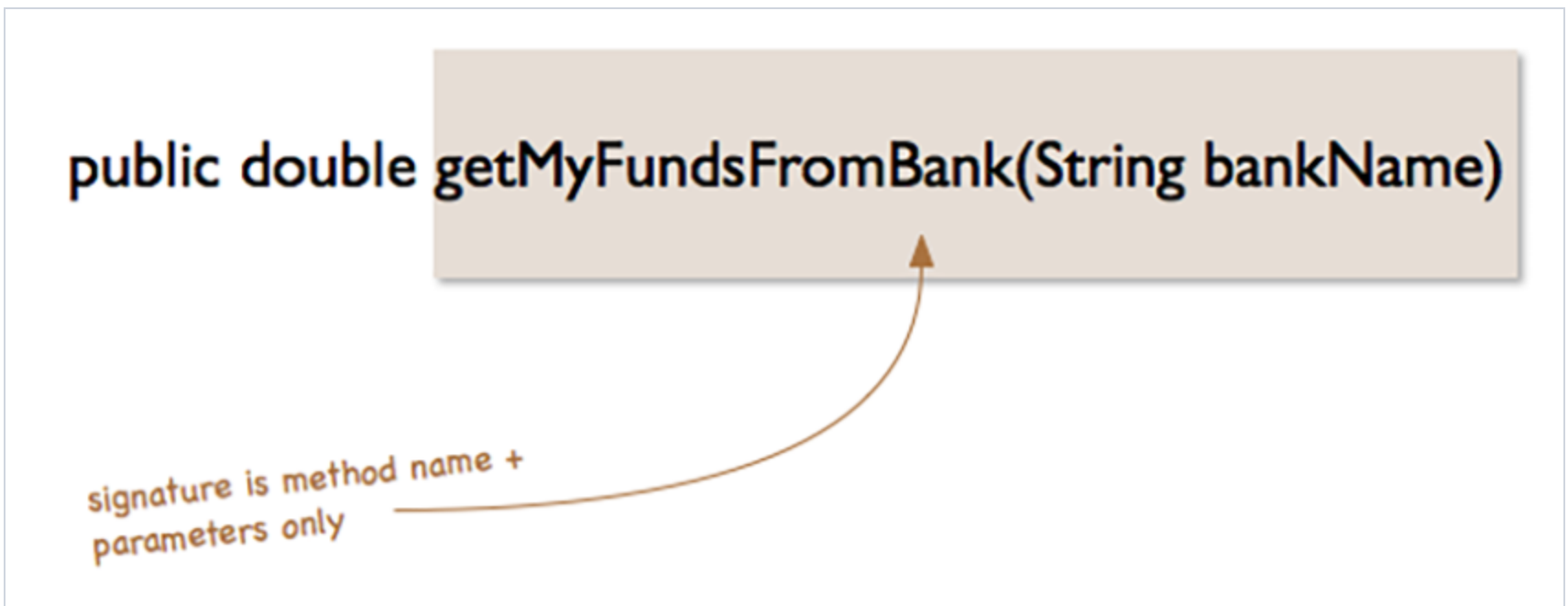
10. Что такое переопределение (overriding) методов?

Переопределение методов происходит, когда child хочет изменить поведение parent класса. Если нужно, чтоб выполнилось-таки то, что есть в методе parent, можно использовать в child конструкцию вида super.methodName(), что выполнит работу parent метода, а уже потом добавить логику.

Требования, которые нужно соблюдать:

- сигнатура метода должна быть такая же;
- возвращаемое значение должно быть таким же.

11. Что такое сигнатура метода?



Сигнатура метода — это набор из названия метода и аргументов, какие принимает метод.

Сигнатура метода является уникальным идентификатором для метода при перегрузке методов

12. Что такое перегрузка методов?

Перегрузка методов — это свойство полиморфизма, в котором при помощи изменения сигнатуры метода можно создать разные методы для одних действий:

- одно и то же имя метода;
- разные аргументы;
- может быть разный возвращаемый тип.

Например, один и тот же `add()` из `ArrayList` может быть перегружен следующим образом и будет выполнять добавление разным способом, в зависимости от входящих аргументов:

- `add(Object o)` — просто добавляет объект;
- `add(int index, Object o)` — добавляет объект в определенный индекс;
- `add(Collection<Object> c)` — добавляет список объектов;
- `add(int index, Collection<Object> c)` — добавляет список объектов, начиная с определенного индекса.

13. Что такое Interface?

Множественное наследование не реализовано в джаве, поэтому чтобы преодолеть эту проблему, были добавлены интерфейсы в том виде, в котором мы их знаем ;)

Долгое время у интерфейсов были только методы без их реализации. В рамках этого ответа поговорим именно о них.

Например:

```
1  public interface Animal {
2      void makeSound();
3      void eat();
4      void sleep();
5  }
```

Из этого вытекают некоторые нюансы:

- все методы в интерфейсе — публичные и абстрактные;
- все переменные — public static final;
- классы не наследуют их (extends), реализовывают (implements). Причем реализовывать можно сколь угодно много интерфейсов.
- классы, которые реализуют интерфейс, должны предоставить реализацию всех методов, которые есть в интерфейсе.

Вот так:

```
1  public class Cat implements Animal {
2      public void makeSound() {
3          // реализация метода
4      }
5
6      public void eat() {
7          // реализация
8      }
9
10     public void sleep() {
11         // реализация
12     }
13 }
```

Теперь поговорим о дефолтных методах. Для чего, для кого? Эти методы добавили, чтобы все сделать “и вашим, и нашим”.

О чем это я? Да о том, что с одной стороны нужно было добавить новую функциональность: лямбды, Stream API, с другой стороны, нужно было оставить то, чем славится джава — обратную совместимость. Для этого нужно было ввести уже готовые решения в интерфейсы. Так к нам и пришли дефолтные методы.

То есть, дефолтный метод — это реализованный метод в интерфейсе, у которого есть ключевое слово `default`.

Например, всем известный метод `stream()` в интерфейсе `Collection`. Проверьте, этот интерфейс вовсе не так прост как кажется ;).

Или также не менее известный метод `forEach()` из интерфейса `Iterable`. Его также не был до тех пор, пока не добавили дефолтные методы.

Кстати, еще можно почитать на JavaRush об этом.

15. А как тогда наследовать два одинаковых дефолтных метода?

Исходя из предыдущего ответа на то, что такое дефолтный метод, можно задать другой вопрос.

Если можно реализовать методы в интерфейсах, то теоретически можно реализовать два интерфейса с одинаковым методом, и как такое делать?

Есть два разных интерфейса с одинаковым методом:

```
1  interface A {
2     default void foo() {
3         System.out.println(“Foo A”);
4     }
5  }
6
7  interface B {
8     default void foo() {
9         System.out.println(“Foo B”);
10    }
11 }
```

И есть класс, который реализует эти два интерфейса. Но только как выбрать специфический метод интерфейса A или B?

Для этого есть конструкция такого вида: `A.super.foo()`:

```
1  public class C implements A, B {
2     public void fooA() {
3         A.super.foo();
4     }
5
6     public void fooB() {
7         B.super.foo();
8     }
9  }
```

Таким образом, метод `fooA()` будет использовать дефолтный метод `foo()` из интерфейса `A`, а метод `fooB()`, соответственно, метод `foo()` из интерфейса `B`.

В джава есть зарезервированное слово `abstract`, которое используется для обозначения абстрактных классов и методов. Для начала — определения.

Абстрактным методом называется метод, который создан без реализации с ключевым словом `abstract` в абстрактном классе. То есть, это метод как в интерфейсе, только с добавкой ключевого слова, например:

```
1  public abstract void foo();
```

Абстрактным классом называется класс, который имеет также `abstract` слово:

```
1  public abstract class A {
2
3  }
```

У абстрактного класса есть несколько особенностей:

- на его основе нельзя создать объект;
- он может иметь абстрактные методы;
- он может и не иметь абстрактные методы.

Абстрактные классы нужны для обобщения какой-то абстракции (сорян за тавтологию), которой в реальной жизни нет, но она содержит множество общих поведений и состояний (то есть, методов и переменных).

Примеров из жизни — хоть отбавляй. Всё вокруг нас. Это может быть “животное”, “машина”, “геометрическая фигура” и так далее.

17. Какая разница между String, String Builder и String Buffer?

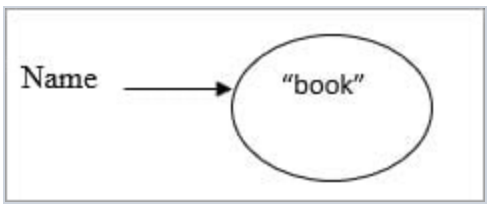
Значения `String` хранятся в пуле стрингов (constant string pool). Как только будет создана строка, она появится в этом пуле. И удалить ее будет нельзя.

Например:

```
1  String name = "book";
```

...переменная будет ссылаться на стринг пул

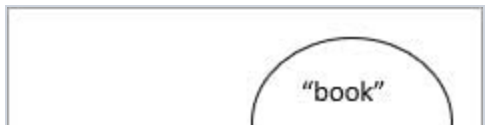
Constant string pool



Если задать переменной name другое значение, получится следующее:

```
1  name = "pen";
```

Constant string pool



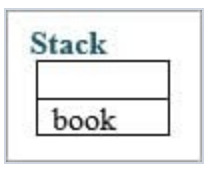
Таким образом, эти два значения так и останутся там.

String Buffer:

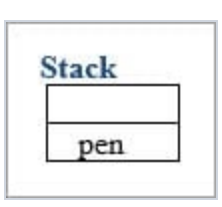
- значения `String` хранятся в стеке(Stack). Если значение изменено, значит новое значение будет заменено на старое;
- `String Buffer` синхронизирован, и поэтому он потокобезопасный;
- из-за потокобезопасности скорость работы оставляет желать лучшего.

Пример:

```
1  StringBuffer name = “book”;
```



Как только значение name сменится, в стеке измениться значение:



StringBuilder

Точно такой же, как и `StringBuffer`, только он не потокобезопасный. Поэтому скорость его явно выше, чем в `StringBuffer`.

18. Какая разница между абстрактным классом и интерфейсом?

Абстрактный класс:

- абстрактные классы имеют дефолтный конструктор; он вызывается каждый раз, когда создается потомок этого абстрактного класса;
- содержит как абстрактные методы, так и не абстрактные. По большому счету может и не содержать абстрактных методов, но все равно быть абстрактным классом;
- класс, который наследуется от абстрактного, должен реализовать только абстрактные методы;
- абстрактный класс может содержать Instance Variable(смотри вопрос №5).

Интерфейс:

- не имеет никакого конструктора и не может быть инициализирован;
- только абстрактные методы должны быть добавлены (не считая default methods);
- классы, реализующие интерфейс, должны реализовать все методы (не считая default methods);
- интерфейсы могут содержать только константы.

19. Почему доступ по элементу в массиве происходит за O(1)?

Это вопрос буквально с последнего собеседования. Как я узнал позже, это вопрос задается для того, чтобы увидеть, как человек мыслит. Ясно, что практического смысла в этих знаниях немного: хватает только лишь знания этого факта.

Для начала нужно уточнить, что O(1) — это обозначение временной сложности алгоритма, когда операция проходит за константное время. То есть это обозначение самого быстрого выполнения.

Чтобы ответить на этот вопрос, нужно понять, что мы знаем о массивах?

Чтоб создать массив `int`, мы должны написать следующее:

```
1  int[] intArray = new int[100];
```

Из этой записи можно сделать несколько выводов:

1. При создании массива известен его тип.Если известен тип, то понятно, какого размера будет каждая ячейка массива.
2. Известно, какого размера будет массив.

Из этого следует: чтобы понять, в какую ячейку записать, нужно просто вычислить, в какую область памяти записать.

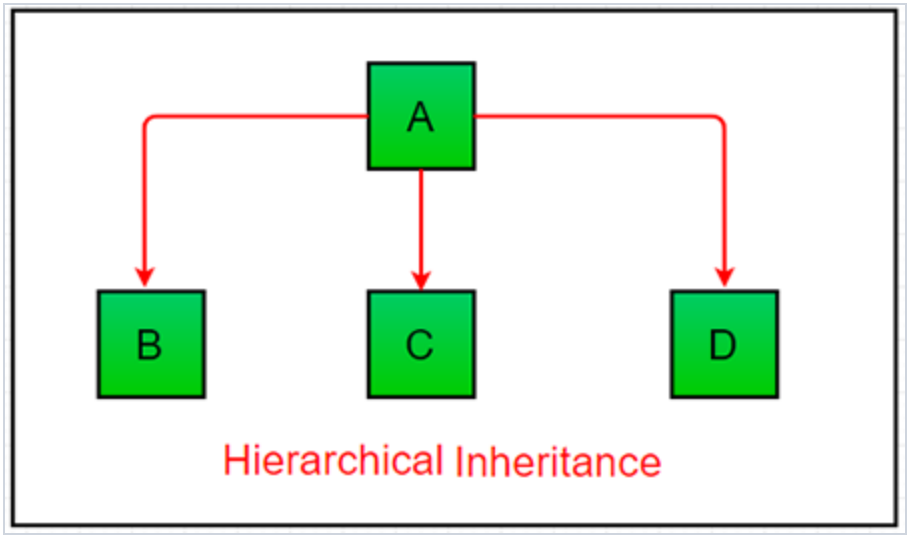
Для машины это проще простого. У машины есть начало выделенной памяти, количество элементов и размер одной ячейки.

Из этого понятно, что место для записи будет равно начальному месту массива + размер ячейки, умноженный на ее размер.

А как получается O(1) в доступе к объектам в ArrayList?

Это вопрос сразу же идет за предыдущим. Ведь правда, когда мы работаем с массивом и там примитивы, то нам известно заранее, какой размер этого типа, при его создании.

А что делать, если есть такая схема, как на картинке:



и мы хотим создать коллекцию с элементами, у которых тип A, и добавить разные реализации — B, C, D:

```
List<A> list = new ArrayList();
list.add(new B());
list.add(new C());
list.add(new D());
list.add(new B());
```

Как в этой ситуации понять, какой будет размер у каждой ячейки, ведь каждый объект будет разным и может иметь разные дополнительные поля (или быть полностью различными). Что делать?

Здесь вопрос ставится так, чтобы запутать и сбить с толку.

Мы же знаем, что на самом деле в коллекции хранятся не объекты, а лишь ссылки на эти объекты. А у всех ссылок размер один и тот же, и он известен. Поэтому здесь работает подсчет места так же, как и в предыдущем вопросе.

21. Автоупаковка (autoboxing) и Автораспаковка (unboxing)

Историческая справка: автоупаковка и автораспаковка - одно из главных нововведений JDK 5.

Автоупаковка (autoboxing) - процесс автоматического преобразования из примитивного типа в соответствующий класс обертку.

Автораспаковка (unboxing) - делает ровно обратное к автоупаковке - преобразует класс обертку в примитив. А вот если окажется значение обертки `null`, то при распаковке будет выброшено исключение `NullPointerException`.

Соответствие примитив - обертка

boolean	Boolean
int	Integer
byte	Byte
char	Character
float	Float
long	Long
short	Short
double	Double

Автоупаковка происходит:

- когда присваивают примитиву ссылку на класс обертку:

ДО Java 5:

```
1  //ручная упаковка или как это было ДО Java 5.
2  public void boxingBeforeJava5() {
3     Boolean booleanBox = new Boolean(true);
4     Integer intBox = new Integer(3);
5     // и так далее к другим типам
6  }
7
8  после Java 5:
9  //автоматическая упаковка или как это стало в Java 5.
10 public void boxingJava5() {
11    Boolean booleanBox = true;
12    Integer intBox = 3;
13    // и так далее к другим типам
14 }
```

- когда передают примитив в аргумент метода, который ожидает обертку:

```
1  public void exampleOfAutoboxing() {
2     long age = 3;
3     setAge(age);
4  }
5
6  public void setAge(Long age) {
7     this.age = age;
8  }
```

Автораспаковка происходит:

- когда присваиваем классу обертке примитивную переменную:

```
1  //до Java 5:
2  int intValue = new Integer(4).intValue();
3  double doubleValue = new Double(2.3).doubleValue();
4  char c = new Character((char) 3).charValue();
```

```
7  //и после JDK 5:
8  int intValue = new Integer(4);
9  double doubleValue = new Double(2.3);
10 char c = new Character((char) 3);
11 boolean b = Boolean.TRUE;
```

- В случаях с арифметическими операциями. Они применяются только к примитивным типам, для этого нужно делать распаковку к примитиву.

```
1  // До Java 5
2  Integer integerBox1 = new Integer(1);
3  Integer integerBox2 = new Integer(2);
4
5  // для сравнения нужно было делать так:
6  integerBox1.intValue() > integerBox2.intValue()
7
8  //в Java 5
9  integerBox1 > integerBox2
```

- когда передают в обертку в метод, который принимает соответствующий примитив:

```
1  public void exampleOfAutoboxing() {
2      Long age = new Long(3);
3      setAge(age);
4  }
5
6  public void setAge(long age) {
7      this.age = age;
8  }
```

22. Что такое ключевое слово final и где его использовать?

Ключевое слово `final` можно использовать для переменных, методов и классов.

1. final переменную нельзя переназначить на другой объект.
2. final класс бесплоден)) у него не может быть наследников.
3. final метод не может быть переопределен у предка.

Пробежали по верхам, теперь обсудим более подробно.

final переменные

;Java дает нам два способа создать переменную и присвоить ей некоторое значение:

1. Можно объявить переменную и инициализировать ее позже.
2. Можно объявить переменную и сразу же назначить ее.

Пример с использованием final переменной для этих случаев:

```
1  public class FinalExample {
2
3      //статическая переменная final, которая сразу же инициализируется:
4      final static String FINAL_EXAMPLE_NAME = "I'm likely final one";
5
6      //final переменная, которая не инициализирована, но работать будет только если
7      //инициализировать это в конструкторе:
8      final long creationTime;
```

```
        this.creationTime = System.currentTimeMillis();
    }

    public static void main(String[] args) {
        FinalExample finalExample = new FinalExample();
        System.out.println(finalExample.creationTime);

        // final поле FinalExample.FINAL_EXAMPLE_NAME не может быть заасайнено
//    FinalExample.FINAL_EXAMPLE_NAME = "Not you're not!";

        // final поле Config.creationTime не может быть заасайнено
//    finalExample.creationTime = 1L;
    }
}
```

Можно ли считать Final переменную константой?

Поскольку у нас не получится присвоить новое значение для final переменной, кажется, что это переменные константы. Но это только на первый взгляд.

Если тип данных, на который ссылается переменная — `immutable`, то да, это константа.

А если тип данных `mutable`, то есть изменяемый, при помощи методов и переменных можно будет изменить значение объекта, на который ссылается `final` переменная, и в таком случае назвать ее константой нельзя.

Так вот, на примере видно, что часть финальных переменных действительно константы, а часть — нет, и их можно изменить.

```
public class FinalExample {

    //неизменяемые финальные переменные:
    final static String FINAL_EXAMPLE_NAME = "I'm likely final one";
    final static Integer FINAL_EXAMPLE_COUNT  = 10;

    //изменяемые фильнаные переменные
    final List<String> addresses = new ArrayList();
    final StringBuilder finalStringBuilder = new StringBuilder("constant?");
}
```

Local final переменные

Когда `final` переменная создается внутри метода, ее называют `local final` переменная:

```
public class FinalExample {

    public static void main(String[] args) {
        // Вот так можно
        final int minAgeForDriveCar = 18;

        // а можно и так, в цикле foreach:
        for (final String arg : args) {
            System.out.println(arg);
        }
    }

```

Мы можем использовать ключевое слово `final` в расширенном цикле `for`, потому что после завершения итерации цикла `for` каждый раз создается новая переменная. Только это все не относится к нормальному циклу for, поэтому приведенный ниже код выдаст ошибку времени компиляции.

```
// final local переиенная j не может быть назначена
for (final int i = 0; i < args.length; i ++) {
   System.out.println(args[i]);
}
```

Final класс

Нельзя расширять класс, объявленный как `final`. Проще говоря, никакой класс не может наследоваться от данного. Прекрасным примером `final` класса в JDK является `String`.

Первый шаг к созданию неизменяемого класса — пометить его как `final`, и тогда нельзя будет его расширить:

```
public final class FinalExample {
}

// Здесь будет ошибка компиляции
class WantsToInheritFinalClass extends FinalExample {
}
```

Final методы

Когда метод маркирован как final, его называют final метод (логично, правда?). Final метод нельзя переопределять у класса наследника.

К слову, методы в классе Object — wait() и notify() — это final, поэтому у нас нет возможность их переопределять.

```
public class FinalExample {
   public final String generateAddress() {
       return "Some address";
   }
}

class ChildOfFinalExample extends FinalExample {

   // здесь будет ошибка компиляции
   @Override
   public String generateAddress() {
       return "My OWN Address";
   }
}
```

Как и где использовать final в Java

- использовать ключевое слово final, чтобы определить некоторые константы уровня класса;
- создавать final переменные для объектов, когда вы не хотите, чтобы они были изменены. Например, специфичные для объекта свойства, которые мы можем использовать для целей логирования;
- если не нужно, чтобы класс был расширен, отметить его как окончательный;
- если нужно создать immutable< класс, нужно сделать его финальным;
- если нужно, чтоб реализация метода не менялась в наследниках, обозначить метод как `final`. Это очень важно, чтобы быть уверенным, что реализация не изменится.

23. Что такое mutable immutable?

Mutable

Mutable называются объекты, чьи состояния и переменные можно изменить после создания. Например такие классы, как StringBuilder, StringBuffer.

Пример:

```
public class MutableExample {

   private String address;

   public MutableExample(String address) {
       this.address = address;
   }

   public String getAddress() {
       return address;
   }

   // этот сеттер может изменить поле name
   public void setAddress(String address) {
       this.address = address;
   }

   public static void main(String[] args) {

       MutableExample obj = new MutableExample("first address");
       System.out.println(obj.getAddress());

       // обновляем поле name, значит это mutable объект
       obj.setAddress("Updated address");
       System.out.println(obj.getAddress());
   }
}
```

Immutable

Immutable называются объекты, состояния и переменные которых нельзя изменить после создания объекта. Чем не отличный ключ для HashMap, да?) Например, String, Integer, Double и так далее.

Пример:

```
// сделаем этот класс финальным, чтобы никто не мог его изменить
public final class ImmutableExample {

```

```
    ImmutableExample (String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    //удаляем сеттер

    public static void main(String[] args) {

        ImmutableExample obj = new ImmutableExample("old address");
        System.out.println(obj.getAddress());

        // Поэтому никак не изменить это поле, значит это immutable объект
        // obj.setName("new address");
        // System.out.println(obj.getName());

    }
}
```

24. Как написать immutable класс?

После того, как выясните, что такое mutable и immutable объекты, следующий вопрос будет закономерный — как написать его?

Чтоб написать immutable неизменяемый класс, нужно следовать простым пунктам:

- сделать класс финальным.
- сделать все поля приватными и создать только геттеры к ним. Сеттеры, разумеется, не нужно.
- Сделать все mutable поля final, чтобы установить значение можно было только один раз.
- инициализировать все поля через конструктор, выполняя глубокое копирование (то есть, копируя и сам объект, и его переменные, и переменные переменных, и так далее)
- клонировать объекты mutable переменных в геттерах, чтобы возвращать только копии значений, а не ссылки на актуальные объекты.

Пример:

```
/**
* Пример по созданию immutable объекта.
*/
public final class FinalClassExample {

   private final int age;

   private final String name;

   private final HashMap<String, String> addresses;

   public int getAge() {
       return age;
   }


   public String getName() {
```

```
20
21      /**
22       * Клонируем объект перед тем, как вернуть его.
23       */
24      public HashMap<String, String> getAddresses() {
25          return (HashMap<String, String>) addresses.clone();
26      }
27
28      /**
29       * В конструкторе выполняем глубокое копирование для mutable объектов.
30       */
31      public FinalClassExample(int age, String name, HashMap<String, String> addresses) {
32          System.out.println("Выполняем глубокое копирование в конструкторе");
33          this.age = age;
34          this.name = name;
35          HashMap<String, String> temporaryMap = new HashMap<>();
36          String key;
37          Iterator<String> iterator = addresses.keySet().iterator();
38          while (iterator.hasNext()) {
39              key = iterator.next();
40              temporaryMap.put(key, addresses.get(key));
41          }
42          this.addresses = temporaryMap;
43      }
44  }
```

- Мой профиль на GitHub
- Топ-50 Java Core вопросов и ответов на собеседовании. Часть 2
- Топ-50 Java Core вопросов и ответов на собеседовании. Часть 3

Roman Beekeeper

автор тг-канала в t.me/romankh3

"7. Что такое Local Variable?
Local variable — это переменная, которая определена внутри метода и существует вплоть до того момента, пока выполняется этот метод. Как только выполнение закончится, локальная переменная перестанет существовать."

не верно
в любимом цикле for (int i = 0 ; :) переменная i перестанет существовать сразу после цикла и за долго до окончания метода. Так же, как try with resources и прочее.

https://proglang.su/java/variable-types
* Локальные переменные объявляются в методах, конструкторах или блоках.
* Локальные переменные создаются, когда метод, конструктор или блок запускается и уничтожаются после того, как завершиться метод, конструктор или блок.
* Модификаторы доступа нельзя использовать для локальных переменных.
* Они являются видимыми только в пределах объявленного метода, конструктора или блока.
* Локальные переменные реализуются на уровне стека внутри.
* В Java не существует для локальных переменных значения по умолчанию, так что они должны быть объявлены и начальное значение должны быть присвоено перед первым использованием.

Ответить 0

Макс Дудин Уровень 40, Калининград, Россия 20 июля, 10:43

всё потом...

Ответить 0

Алексей Уровень 22, Минск, Беларусь 25 марта, 17:48

Вопрос 12.
Ответ некорректный.
Перегрузка метода - это возможность иметь несколько методов с одинаковым именем, но разными наборами аргументов. И всё.
Полиморфизм при перегрузке НЕ ЗАДЕЙСТВУЕТСЯ.
Перегруженный метод - это всего лишь совершенно другой метод с таким же именем. С наследованием или полиморфизмом это не имеет ничего общего.
Правила корректной перегрузки:
1. Типы возвращаемого значения (ТВЗ) могут быть разными.
2. Список аргументов обязательно должен отличаться. Если мы изменим только ТВЗ без изменения аргументов, компилятор расценит это как некорректную попытку переопределения и выдаст ошибку компиляции.
3. уровнем доступа можно оперировать как угодно. При перегрузке не выполняется полиморфический контракт - это по сути новый метод, поэтому менять уровень доступа можно как душа пожелает.

Ответить +3

Алексей Уровень 22, Минск, Беларусь 25 марта, 17:31

Вопрос 10.
К обязательным требованиям при переопределении следует так же отнести:
- модификатор доступа переопределенного метода должен быть таким же или менее строгим.

А то программный поток будет сильно удивлён когда при попытке вызвать как он наивно полагает public метод, JVM захлопнет дверь у него перед носом, потому что этот метод переопределен как private.

Ответить 0

Vadyxa715 Уровень 40, Нижний Новгород, Россия 28 февраля, 20:35

Актуальные ссылки в связи с изменением домена.
Топ-50 Java Core вопросов и ответов на собеседовании. Часть 2
Топ-50 Java Core вопросов и ответов на собеседовании. Часть 3

Ответить 0

Вячеслав Валерьевич Мищенко Уровень 6, Россия 18 октября 2021, 18:43

У вас в статье:
"Что такое переопределение (overriding) методов?
....
....
....
 Требования, которые нужно соблюдать:
сигнатура метода должна быть такая же;
возвращаемое значение должно быть таким же."

Точнее сказать: "Тип возвращаемого значения должен быть таким же".

Ответить 0

Andrew Savich Уровень 3, Беларусь 4 октября 2021, 00:40

6. Какие два класса не наследуются от Object?
Есть только 1 класс, который не наследуется от Object - это сам Object

Ответить +6

НАЧАТЬ ОБУЧЕНИЕ

основе существующего. Дальше читать желания не появилось. Но следующий пункт чекнул. "Инкапсуляция — это сокрытие реализации". Ага, а паттерны, наверное, знаешь синглтон? =)) Как с такими ответами проходить собеседование, когда я, не работавший, понимаю что это полная хероборина. Это хорошо, что я знаю, что это лажа, готовиться по этому - ну нахер.

Ответить +1

Anton Stezhkin Уровень 41 31 августа 2021, 15:12

"final метод не может быть переопределен у предка" - вы же имеете в виду "у потомка", да?

Ответить +2

Показать еще комментарии

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.