

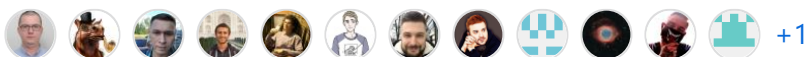


 enhorse / java-interview Public[Code](#) [Issues 9](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...

[java-interview](#) / [core.md](#)

ivan100kg Fix answer about static modifier

 History 13 contributors 1551 lines (1104 sloc) | 171 KB

...

[Вопросы для собеседования](#)

Java Core

- Чем различаются JRE, JVM и JDK?
- Какие существуют модификаторы доступа?
- О чем говорит ключевое слово `final` ?
- Какими значениями инициализируются переменные по умолчанию?
- Что вы знаете о функции `main()` ?
- Какие логические операции и операторы вы знаете?
- Что такое тернарный оператор выбора?
- Какие побитовые операции вы знаете?
- Где и для чего используется модификатор `abstract` ?
- Дайте определение понятию «интерфейс». Какие модификаторы по умолчанию имеют поля и методы интерфейсов?
- Чем абстрактный класс отличается от интерфейса? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?
- Почему в некоторых интерфейсах вообще не определяют методов?
- Почему нельзя объявить метод интерфейса с модификатором `final` ?

- Что имеет более высокий уровень абстракции - *класс*, *абстрактный класс* или *интерфейс*?
- Может ли объект получить доступ к члену класса, объявленному как `private` ? Если да, то каким образом?
- Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?
- Зачем нужны и какие бывают блоки инициализации?
- К каким конструкциям Java применим модификатор `static` ?
- Для чего в Java используются статические блоки инициализации?
- Что произойдёт, если в блоке инициализации возникнет исключительная ситуация?
- Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?
- Может ли статический метод быть переопределён или перегружен?
- Могут ли нестатические методы перегрузить статические?
- Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?
- Возможно ли при переопределении метода изменить: модификатор доступа, возвращаемый тип, тип аргумента или их количество, имена аргументов или их порядок; убирать, добавлять, изменять порядок следования элементов секции `throws` ?
- Как получить доступ к переопределённым методам родительского класса?
- Можно ли объявить метод абстрактным и статическим одновременно?
- В чем разница между членом экземпляра класса и статическим членом класса?
- Где разрешена инициализация статических/нестатических полей?
- Какие типы классов бывают в java?
- Расскажите про вложенные классы. В каких случаях они применяются?
- Что такое «*статический класс*»?
- Какие существуют особенности использования вложенных классов: статических и внутренних? В чем заключается разница между ними?
- Что такое «*локальный класс*»? Каковы его особенности?
- Что такое «*анонимные классы*»? Где они применяются?
- Каким образом из вложенного класса получить доступ к полю внешнего класса?
- Для чего используется оператор `assert` ?

- Что такое *Heap* и *Stack* память в Java? Какая разница между ними?
- Верно ли утверждение, что примитивные типы данных всегда хранятся в стеке, а экземпляры ссылочных типов данных в куче?
- Каким образом передаются переменные в методы, по значению или по ссылке?
- Для чего нужен сборщик мусора?
- Как работает сборщик мусора?
- Какие разновидности сборщиков мусора реализованы в виртуальной машине HotSpot?
- Опишите алгоритм работы какого-нибудь сборщика мусора, реализованного в виртуальной машине HotSpot.
- Что такое «пул строк»?
- Что такое `finalize()` ? Зачем он нужен?
- Что произойдет со сборщиком мусора, если выполнение метода `finalize()` требует ощутимо много времени, или в процессе выполнения будет выброшено исключение?
- Чем отличаются `final` , `finally` и `finalize()` ?
- Расскажите про приведение типов. Что такое понижение и повышение типа?
- Когда в приложении может быть выброшено исключение `ClassCastException` ?
- Что такое литералы?
- Что такое *autoboxing* («автоупаковка») в Java и каковы правила упаковки примитивных типов в классы-обертки?
- Какие есть особенности класса `String` ?
- Почему `String` неизменяемый и финализированный класс?
- Почему `char[]` предпочтительнее `String` для хранения пароля?
- Почему строка является популярным ключом в `HashMap` в Java?
- Что делает метод `intern()` в классе `String` ?
- Можно ли использовать строки в конструкции `switch` ?
- Какая основная разница между `String` , `StringBuffer` , `StringBuilder` ?
- Что такое класс `Object` ? Какие в нем есть методы?
- Дайте определение понятию «конструктор».
- Что такое «конструктор по умолчанию»?
- Чем отличаются конструктор по-умолчанию, конструктор копирования и конструктор с параметрами?

- Где и как вы можете использовать приватный конструктор?
- Расскажите про классы-загрузчики и про динамическую загрузку классов.
- Что такое *Reflection*?
- Зачем нужен `equals()` . Чем он отличается от операции `==` ?
- Если вы хотите переопределить `equals()` , какие условия должны выполняться?
- Какими свойствами обладает порождаемое `equals()` отношение эквивалентности?
- Правила переопределения метода `Object.equals()` .
- Какая связь между `hashCode()` и `equals()` ?
- Если `equals()` переопределен, есть ли какие-либо другие методы, которые следует переопределить?
- Что будет, если переопределить `equals()` не переопределяя `hashCode()` ? Какие могут возникнуть проблемы?
- Каким образом реализованы методы `hashCode()` и `equals()` в классе `Object` ?
- Для чего нужен метод `hashCode()` ?
- Каковы правила переопределения метода `Object.hashCode()` ?
- Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете `hashCode()` ?
- Могут ли у разных объектов быть одинаковые `hashCode()` ?
- Если у класса `Point{int x, y;}` реализовать метод `equals(Object that)` `{(return this.x == that.x && this.y == that.y)}` , но сделать хэш код в виде `int hashCode() {return x;}` , то будут ли корректно такие точки помещаться и извлекаться из `HashSet` ?
- Могут ли у разных объектов `(ref0 != ref1)` быть `ref0.equals(ref1) == true` ?
- Могут ли у разных ссылок на один объект `(ref0 == ref1)` быть `ref0.equals(ref1) == false` ?
- Можно ли так реализовать метод `equals(Object that) {return this.hashCode() == that.hashCode();}` ?
- В `equals()` требуется проверять, что аргумент `equals(Object that)` такого же типа что и сам объект. В чем разница между `this.getClass() == that.getClass()` и `that instanceof MyClass` ?
- Можно ли реализовать метод `equals()` класса `MyClass` вот так: `class MyClass {public boolean equals(MyClass that) {return this == that;}}` ?
- Есть класс `Point{int x, y;}` . Почему хэш код в виде `31 * x + y`

предпочтительнее чем $x + y$?

- Расскажите про клонирование объектов.
- В чем отличие между *поверхностным* и *глубоким* клонированием?
- Какой способ клонирования предпочтительней?
- Почему метод `clone()` объявлен в классе `Object`, а не в интерфейсе `Cloneable` ?
- Опишите иерархию исключений.
- Какие виды исключений в Java вы знаете, чем они отличаются?
- Что такое *checked* и *unchecked exception*?
- Какой оператор позволяет принудительно выбросить исключение?
- О чем говорит ключевое слово `throws` ?
- Как написать собственное («пользовательское») исключение?
- Какие существуют *unchecked exception*?
- Что представляет из себя ошибки класса `Error` ?
- Что вы знаете о `OutOfMemoryError` ?
- Опишите работу блока *try-catch-finally*.
- Что такое механизм *try-with-resources*?
- Возможно ли использование блока *try-finally* (без `catch`)?
- Может ли один блок `catch` отлавливать сразу несколько исключений?
- Всегда ли выполняется блок `finally` ?
- Существуют ли ситуации, когда блок `finally` не будет выполнен?
- Может ли метод `main()` выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?
- Предположим, есть метод, который может выбросить `IOException` и `FileNotFoundException` в какой последовательности должны идти блоки `catch` ? Сколько блоков `catch` будет выполнено?
- Что такое *generics*?
- Что такое «интернационализация», «локализация»?

Чем различаются JRE, JVM и JDK?

JVM, Java Virtual Machine (Виртуальная машина Java) — основная часть среды времени исполнения Java (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java. JVM может также использоваться для выполнения программ, написанных на других языках программирования.

JRE, Java Runtime Environment (Среда времени выполнения Java) - минимально-необходимая реализация виртуальной машины для исполнения Java-приложений. Состоит из JVM и стандартного набора библиотек классов Java.

JDK, Java Development Kit (Комплект разработки на Java) - JRE и набор инструментов разработчика приложений на языке Java, включающий в себя компилятор Java, стандартные библиотеки классов Java, примеры, документацию, различные утилиты.

Коротко: **JDK** - среда для разработки программ на Java, включающая в себя **JRE** - среду для обеспечения запуска Java программ, которая в свою очередь содержит **JVM** - интерпретатор кода Java программ.

[к оглавлению](#)

Какие существуют модификаторы доступа?

private (приватный): члены класса доступны только внутри класса. Для обозначения используется служебное слово `private`.

default, **package-private**, **package level** (доступ на уровне пакета): видимость класса/членов класса только внутри пакета. Является модификатором доступа по умолчанию - специальное обозначение не требуется.

protected (защищённый): члены класса доступны внутри пакета и в наследниках. Для обозначения используется служебное слово `protected`.

public (публичный): класс/члены класса доступны всем. Для обозначения используется служебное слово `public`.

Последовательность модификаторов по возрастанию уровня закрытости: `public`, `protected`, `default`, `private`.

Во время наследования возможно изменения модификаторов доступа в сторону большей видимости (для поддержания соответствия **принципу подстановки Барбары Лисков**).

[к оглавлению](#)

О чем говорит ключевое слово `final`?

Модификатор `final` может применяться к переменным, параметрам методов, полям и методам класса или самим классам.

- Класс не может иметь наследников;
- Метод не может быть переопределен в классах наследниках;
- Поле не может изменить свое значение после инициализации;
- Параметры методов не могут изменять своё значение внутри метода;
- Локальные переменные не могут быть изменены после присвоения им значения.

[к оглавлению](#)

Какими значениями инициализируются переменные по умолчанию?

- Числа инициализируются `0` или `0.0`;
- `char` — `\u0000`;
- `boolean` — `false`;
- Объекты (в том числе `String`) — `null`.

[к оглавлению](#)

Что вы знаете о функции `main()`?

Метод `main()` — точка входа в программу. В приложении может быть несколько таких методов. Если метод отсутствует, то компиляция возможна, но при запуске будет получена ошибка ``Error: Main method not found``.

```
public static void main(String[] args) {}
```

[к оглавлению](#)

Какие логические операции и операторы вы знаете?

- `&` : Логическое *AND* (И);
- `&&` : Сокращённое *AND*;
- `|` : Логическое *OR* (ИЛИ);
- `||` : Сокращённое *OR*;
- `^` : Логическое *XOR* (исключающее *OR* (ИЛИ));
- `!` : Логическое унарное *NOT* (НЕ);
- `&=` : *AND* с присваиванием;
- `|=` : *OR* с присваиванием;
- `^=` : *XOR* с присваиванием;
- `==` : Равно;
- `!=` : Не равно;
- `?:` : Тернарный (троичный) условный оператор.

[к оглавлению](#)

Что такое тернарный оператор выбора?

Тернарный условный оператор `?:` - оператор, которым можно заменить некоторые конструкции операторов `if-then-else`.

Выражение записывается в следующей форме:

`условие ? выражение1 : выражение2`

Если `условие` выполняется, то вычисляется `выражение1` и его результат становится результатом выполнения всего оператора. Если же `условие` равно `false`, то вычисляется `выражение2` и его значение становится результатом работы оператора. Оба операнда `выражение1` и `выражение2` должны возвращать значение одинакового (или совместимого) типа.

[к оглавлению](#)

Какие побитовые операции вы знаете?

- `~` : Побитовый унарный оператор *NOT*;
- `&` : Побитовый *AND*;
- `&=` : Побитовый *AND* с присваиванием;
- `|` : Побитовый *OR*;

- |= : Побитовый OR с присваиванием;
- ^ : Побитовый исключающее XOR;
- ^= : Побитовый исключающее XOR с присваиванием;
- >> : Сдвиг вправо (деление на 2 в степени сдвига);
- >>= : Сдвиг вправо с присваиванием;
- >>> : Сдвиг вправо без учёта знака;
- >>>= : Сдвиг вправо без учёта знака с присваиванием;
- << : Сдвиг влево (умножение на 2 в степени сдвига);
- <<= : Сдвиг влево с присваиванием.

[к оглавлению](#)

Где и для чего используется модификатор `abstract` ?

Класс, помеченный модификатором `abstract`, называется абстрактным классом. Такие классы могут выступать только предками для других классов. Создавать экземпляры самого абстрактного класса не разрешается. При этом наследниками абстрактного класса могут быть как другие абстрактные классы, так и классы, допускающие создание объектов.

Метод, помеченный ключевым словом `abstract` - абстрактный метод, т.е. метод, который не имеет реализации. Если в классе присутствует хотя бы один абстрактный метод, то весь класс должен быть объявлен абстрактным.

Использование абстрактных классов и методов позволяет описать некий шаблон объекта, который должен быть реализован в других классах. В них же самих описывается лишь некое общее для всех потомков поведение.

[к оглавлению](#)

Дайте определение понятию «интерфейс». Какие модификаторы по умолчанию имеют поля и методы интерфейсов?

Ключевое слово `interface` используется для создания полностью абстрактных классов. Основное предназначение интерфейса - определять каким образом мы можем использовать класс, который его реализует. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не реализует их поведение. Все методы неявно объявляются как `public`.

Начиная с Java 8 в интерфейсах разрешается размещать реализацию методов по умолчанию `default` и статических `static` методов.

Интерфейс также может содержать и поля. В этом случае они автоматически являются публичными `public`, статическими `static` и неизменяемыми `final`.

[к оглавлению](#)

Чем абстрактный класс отличается от интерфейса? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?

- В Java класс может одновременно реализовать несколько интерфейсов, но наследоваться только от одного класса.
- Абстрактные классы используются только тогда, когда присутствует тип отношений «is a» (является). Интерфейсы могут реализоваться классами, которые не связаны друг с другом.
- Абстрактный класс - средство, позволяющее избежать написания повторяющегося кода, инструмент для частичной реализации поведения. Интерфейс - это средство выражения семантики класса, контракт, описывающий возможности. Все методы интерфейса неявно объявляются как `public abstract` или (начиная с Java 8) `default` - методами с реализацией по умолчанию, а поля - `public static final`.
- Интерфейсы позволяют создавать структуры типов без иерархии.
- Наследуясь от абстрактного, класс «растворяет» собственную индивидуальность. Реализуя интерфейс, он расширяет собственную функциональность.

Абстрактные классы содержат частичную реализацию, которая дополняется или расширяется в подклассах. При этом все подклассы схожи между собой в части реализации, унаследованной от абстрактного класса, и отличаются лишь в части собственной реализации абстрактных методов родителя. Поэтому абстрактные классы применяются в случае построения иерархии однопоточных, очень похожих друг на друга классов. В этом случае наследование от абстрактного класса, реализующего поведение объекта по умолчанию может быть полезно, так как позволяет избежать написания повторяющегося кода. Во всех остальных случаях лучше использовать интерфейсы.

[к оглавлению](#)

Почему в некоторых интерфейсах вообще не определяют методов?

Это так называемые *маркерные интерфейсы*. Они просто указывают что класс относится к определенному типу. Примером может послужить интерфейс `Cloneable`, который указывает на то, что класс поддерживает механизм клонирования.

[к оглавлению](#)

Почему нельзя объявить метод интерфейса с модификатором `final`?

В случае интерфейсов указание модификатора `final` бессмысленно, т.к. все методы интерфейсов неявно объявляются как абстрактные, т.е. их невозможно выполнить, не реализовав где-то еще, а этого нельзя будет сделать, если у метода идентификатор `final`.

[к оглавлению](#)

Что имеет более высокий уровень абстракции - класс, абстрактный класс или интерфейс?

Интерфейс.

[к оглавлению](#)

Может ли объект получить доступ к члену класса, объявленному как `private`? Если да, то каким образом?

- Внутри класса доступ к приватной переменной открыт без ограничений;
- Вложенный класс имеет полный доступ ко всем (в том числе и приватным) членам содержащего его класса;
- Доступ к приватным переменным извне может быть организован через отличные от приватных методов, которые предоставлены разработчиком класса. Например: `getX()` и `setX()`.
- Через механизм рефлексии (Reflection API):

```
class Victim {  
    private int field = 42;  
}  
//...  
Victim victim = new Victim();  
Field field = Victim.class.getDeclaredField("field");  
field.setAccessible(true);  
int fieldValue = (int) field.get(victim);  
//...
```

[к оглавлению](#)

Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?

Сначала вызываются все статические блоки в очередности от первого статического блока корневого предка и выше по цепочке иерархии до статических блоков самого класса.

Затем вызываются нестатические блоки инициализации корневого предка, конструктор корневого предка и так далее вплоть до нестатических блоков и конструктора самого класса.

Parent static block(s) → Child static block(s) → Grandchild static block(s)

→ Parent non-static block(s) → Parent constructor →

→ Child non-static block(s) → Child constructor →

→ Grandchild non-static block(s) → Grandchild constructor

Пример 1:

```
public class MainClass {

    public static void main(String args[]) {
        System.out.println(TestClass.v);
        new TestClass().a();
    }

}

public class TestClass {

    public static String v = "Some val";

    {
        System.out.println("!!! Non-static initializer");
    }

    static {
        System.out.println("!!! Static initializer");
    }

    public void a() {
        System.out.println("!!! a() called");
    }

}
```

Результат выполнения:

```
!!! Static initializer
Some val
!!! Non-static initializer
!!! a() called
```

Пример 2:

```
public class MainClass {

    public static void main(String args[]) {
```

```
        new TestClass().a();
    }

}

public class TestClass {

    public static String v = "Some val";

    {
        System.out.println("!!! Non-static initializer");
    }

    static {
        System.out.println("!!! Static initializer");
    }

    public void a() {
        System.out.println("!!! a() called");
    }

}
```

Результат выполнения:

```
!!! Static initializer
!!! Non-static initializer
!!! a() called
```

[к оглавлению](#)

Зачем нужны и какие бывают блоки инициализации?

Блоки инициализации представляют собой код, заключенный в фигурные скобки и размещаемый внутри класса вне объявления методов или конструкторов.

- Существуют статические и нестатические блоки инициализации.
- Блок инициализации выполняется перед инициализацией класса загрузчиком классов или созданием объекта класса с помощью конструктора.
- Несколько блоков инициализации выполняются в порядке следования в коде класса.

- Блок инициализации способен генерировать исключения, если их объявления перечислены в `throws` всех конструкторов класса.
- Блок инициализации возможно создать и в анонимном классе.

[к оглавлению](#)

К каким конструкциям Java применим модификатор `static`?

- полям;
- методам;
- вложенным классам;
- блокам инициализации;
- членам секции `import`.

[к оглавлению](#)

Для чего в Java используются статические блоки инициализации?

Статические блоки инициализации используются для выполнения кода, который должен выполняться один раз при инициализации класса загрузчиком классов, в момент, предшествующий созданию объектов этого класса при помощи конструктора. Такой блок (в отличие от нестатических, принадлежащих конкретному объекту класса) принадлежит только самому классу (объекту метакласса `Class`).

[к оглавлению](#)

Что произойдёт, если в блоке инициализации возникнет исключительная ситуация?

Для нестатических блоков инициализации, если выбрасывание исключения прописано явным образом требуется, чтобы объявления этих исключений были перечислены в `throws` всех конструкторов класса. Иначе будет ошибка компиляции. Для статического блока выбрасывание исключения в явном виде, приводит к ошибке компиляции.

В остальных случаях, взаимодействие с исключениями будет проходить так же, как и в любом другом месте. Класс не будет инициализирован, если ошибка происходит в статическом блоке и объект класса не будет создан, если ошибка возникает в нестатическом блоке.

[к оглавлению](#)

Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?

Если возникшее исключение - наследник `RuntimeException`:

- для статических блоков инициализации будет выброшено `java.lang.ExceptionInInitializerError`;
- для нестатических будет выброшено исключение-источник.

Если возникшее исключение - наследник `Error`, то в обоих случаях будет выброшено `java.lang.Error`. Исключение: `java.lang.ThreadDeath` - смерть потока. В этом случае никакое исключение выброшено не будет.

[к оглавлению](#)

Может ли статический метод быть переопределён или перегружен?

Перегружен - да. Всё работает точно так же, как и с обычными методами - 2 статических метода могут иметь одинаковое имя, если количество их параметров или типов различается.

Переопределён - нет. Выбор вызываемого статического метода происходит при раннем связывании (на этапе компиляции, а не выполнения) и выполняться всегда будет родительский метод, хотя синтаксически переопределение статического метода - это вполне корректная языковая конструкция.

В целом, к статическим полям и методам рекомендуется обращаться через имя класса, а не объект.

[к оглавлению](#)

Могут ли нестатические методы перегрузить статические?

Да. В итоге получится два разных метода. Статический будет принадлежать классу и будет доступен через его имя, а нестатический будет принадлежать конкретному объекту и доступен через вызов метода этого объекта.

[к оглавлению](#)

Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?

- При переопределении метода нельзя сузить модификатор доступа к методу (например с public в MainClass до private в Class extends MainClass).
- Изменить тип возвращаемого значения при переопределении метода нельзя, будет ошибка attempting to use incompatible return type.
- Можно сузить возвращаемое значение, если они совместимы.

Например:

```
public class Animal {

    public Animal eat() {
        System.out.println("animal eat");
        return null;
    }

    public Long calc() {
        return null;
    }

}

public class Dog extends Animal {

    public Dog eat() {
        return new Dog();
    }

    /*attempting to use incompatible return type
    public Integer calc() {
        return null;
    }

}
```

```
*/  
}
```

Возможно ли при переопределении метода изменить: модификатор доступа, возвращаемый тип, тип аргумента или их количество, имена аргументов или их порядок; убирать, добавлять, изменять порядок следования элементов секции `throws` ?

При переопределении метода сужать модификатор доступа не разрешается, т.к. это приведёт к нарушению принципа подстановки Барбары Лисков. Расширение уровня доступа возможно.

Можно изменять все, что не мешает компилятору понять какой метод родительского класса имеется в виду:

- Изменять тип возвращаемого значения при переопределении метода разрешено только в сторону сужения типа (вместо родительского класса - наследника).
- При изменении типа, количества, порядка следования аргументов вместо переопределения будет происходить *overloading* (перегрузка) метода.
- Секцию `throws` метода можно не указывать, но стоит помнить, что она остаётся действительной, если уже определена у метода родительского класса. Так же, возможно добавлять новые исключения, являющиеся наследниками от уже объявленных или исключения `RuntimeException`. Порядок следования таких элементов при переопределении значения не имеет.

[к оглавлению](#)

Как получить доступ к переопределенным методам родительского класса?

С помощью ключевого слова `super` мы можем обратиться к любому члену родительского класса - методу или полю, если они не определены с модификатором `private`.

```
super.method();
```

[к оглавлению](#)

Можно ли объявить метод абстрактным и статическим одновременно?

Нет. В таком случае компилятор выдаст ошибку: *"Illegal combination of modifiers: 'abstract' and 'static'"*. Модификатор `abstract` говорит, что метод будет реализован в другом классе, а `static` наоборот указывает, что этот метод будет доступен по имени класса.

[к оглавлению](#)

В чем разница между членом экземпляра класса и статическим членом класса?

Модификатор `static` говорит о том, что данный метод или поле принадлежат самому классу и доступ к ним возможен даже без создания экземпляра класса. Поля, помеченные `static` инициализируются при инициализации класса. На методы, объявленные как `static`, накладывается ряд ограничений:

- Они могут вызывать только другие статические методы.
- Они должны осуществлять доступ только к статическим переменным.
- Они не могут ссылаться на члены типа `this` или `super`.

В отличие от статических, поля экземпляра класса принадлежат конкретному объекту и могут иметь разные значения для каждого. Вызов метода экземпляра возможен только после предварительного создания объекта класса.

Пример:

```
public class MainClass {  
  
    public static void main(String args[]) {  
        System.out.println(TestClass.v);  
        new TestClass().a();  
        System.out.println(TestClass.v);  
    }  
  
}
```

```
public class TestClass {  
  
    public static String v = "Initial val";  
  
    {  
        System.out.println("!!! Non-static initializer");  
        v = "Val from non-static";  
    }  
  
    static {  
        System.out.println("!!! Static initializer");  
        v = "Some val";  
    }  
  
    public void a() {  
        System.out.println("!!! a() called");  
    }  
  
}
```

Результат:

```
!!! Static initializer  
Some val  
!!! Non-static initializer  
!!! a() called  
Val from non-static
```

[к оглавлению](#)

Где разрешена инициализация статических/нестатических полей?

- Статические поля можно инициализировать при объявлении, в статическом или нестатическом блоке инициализации.
- Нестатические поля можно инициализировать при объявлении, в нестатическом блоке инициализации или в конструкторе.

[к оглавлению](#)

Какие типы классов бывают в java?

- **Top level class** (Обычный класс):
 - **Abstract class** (Абстрактный класс);
 - **Final class** (Финализированный класс).
- **Interfaces** (Интерфейс).
- **Enum** (Перечисление).
- **Nested class** (Вложенный класс):
 - **Static nested class** (Статический вложенный класс);
 - **Member inner class** (Простой внутренний класс);
 - **Local inner class** (Локальный класс);
 - **Anonymous inner class** (Анонимный класс).

[к оглавлению](#)

Расскажите про вложенные классы. В каких случаях они применяются?

Класс называется **вложенным (Nested class)**, если он определен внутри другого класса. **Вложенный класс** должен создаваться только для того, чтобы обслуживать **обрамляющий его класс**. Если вложенный класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня. **Вложенные классы** имеют доступ ко всем (в том числе приватным) полям и методам **внешнего класса**, но не наоборот. Из-за этого разрешения использование **вложенных классов** приводит к некоторому нарушению **инкапсуляции**.

Существуют четыре категории **вложенных классов**:

- **Static nested class** (Статический вложенный класс);
- **Member inner class** (Простой внутренний класс);
- **Local inner class** (Локальный класс);
- **Anonymous inner class** (Анонимный класс).

Такие категории классов, за исключением первого, также называют **внутренними (Inner class)**. **Внутренние классы** ассоциируются не с **внешним классом**, а с **экземпляром внешнего**.

Каждая из категорий имеет рекомендации по своему применению. Если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах одного метода и если каждому экземпляру такого класса необходима ссылка на включающий его экземпляр, то используется **нестатический внутренний класс**. В случае, если ссылка на обрамляющий класс не требуется - лучше сделать такой класс **статическим**. Если класс необходим только внутри какого-то метода и требуется создавать экземпляры этого класса только в этом методе, то используется **локальный класс**. А, если к тому же применение класса сводится к использованию лишь в одном месте и уже существует тип, характеризующий этот класс, то рекомендуется делать его **анонимным классом**.

[к оглавлению](#)

Что такое «статический класс»?

Это вложенный класс, объявленный с использованием ключевого слова `static`. К классам верхнего уровня модификатор `static` неприменим.

[к оглавлению](#)

Какие существуют особенности использования вложенных классов: статических и внутренних? В чем заключается разница между ними?

- Вложенные классы могут обращаться ко всем членам обрамляющего класса, в том числе и приватным.
- Для создания объекта статического вложенного класса объект внешнего класса не требуется.
- Из объекта **статического вложенного** класса нельзя обращаться к не статическим членам обрамляющего класса напрямую, а только через ссылку на экземпляр внешнего класса.
- Обычные вложенные классы не могут содержать статических методов, блоков инициализации и классов. Статические вложенные классы - могут.
- В объекте обычного вложенного класса хранится ссылка на объект **внешнего класса**. Внутри статической такой ссылки нет. Доступ к экземпляру обрамляющего класса осуществляется через указание `.this` после его имени. Например: `Outer.this`.

[к оглавлению](#)

Что такое «локальный класс»? Каковы его особенности?

Local inner class (Локальный класс) - это **вложенный класс**, который может быть **декларирован в любом блоке**, в котором разрешается декларировать переменные. Как и простые внутренние классы (**Member inner class**) **локальные классы** имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр только тогда, когда применяются в нестатическом контексте.

Локальные классы имеют следующие особенности:

- Видны только в пределах блока, в котором объявлены;
- Не могут быть объявлены как `private` / `public` / `protected` или `static` ;
- Не могут иметь внутри себя статических объявлений методов и классов, но могут иметь финальные статические поля, проинициализированные константой;
- Имеют доступ к полям и методам **обрамляющего класса**;
- Могут обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором `final` .

[к оглавлению](#)

Что такое «анонимные классы»? Где они применяются?

Это **вложенный локальный класс без имени**, который разрешено декларировать в любом месте **обрамляющего класса**, разрешающем размещение выражений. Создание экземпляра **анонимного класса** происходит одновременно с его объявлением. В зависимости от местоположения **анонимный класс** ведет себя как **статический** либо как **нестатический вложенный класс** - в нестатическом контексте появляется окружающий его экземпляр.

Анонимные классы имеют несколько ограничений:

- Их использование разрешено только в одном месте программы - месте его создания;

- Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;
- Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

- создания объекта функции (*function object*), например, реализация интерфейса `Comparator` ;
- создания объекта процесса (*process object*), такого как экземпляры классов `Thread` , `Runnable` и подобных;
- в статическом методе генерации;
- инициализации открытого статического поля `final` , которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

[к оглавлению](#)

Каким образом из вложенного класса получить доступ к полю внешнего класса?

Статический вложенный класс имеет прямой доступ только к статическим полям обрамляющего класса.

Простой внутренний класс, может обратиться к любому полю внешнего класса напрямую. В случае, если у вложенного класса уже существует поле с таким же литералом, то обращаться к такому полю следует через ссылку на его экземпляр. Например: `Outer.this.field` .

[к оглавлению](#)

Для чего используется оператор `assert` ?

Assert (Утверждение) — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Утверждение может автоматически сигнализировать об обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.

Утверждения существенно упрощают локализацию ошибок в коде. Даже проверка результатов выполнения очевидного кода может оказаться полезной при последующем рефакторинге, после которого код может стать не настолько очевидным и в него может закрасться ошибка.

Обычно утверждения оставляют включенными во время разработки и тестирования программ, но отключают в релиз-версиях программ.

Т.к. утверждения могут быть удалены на этапе компиляции либо во время исполнения программы, они не должны менять поведение программы. Если в результате удаления утверждения поведение программы может измениться, то это явный признак неправильного использования *assert*. Таким образом, внутри *assert* нельзя вызывать методы, изменяющие состояние программы, либо внешнего окружения программы.

В Java проверка утверждений реализована с помощью оператора `assert`, который имеет форму:

```
assert [Выражение типа boolean]; или assert [Выражение типа boolean] :  
[Выражение любого типа, кроме void];
```

Во время выполнения программы в том случае, если проверка утверждений включена, вычисляется значение булевского выражения, и если его результат `false`, то генерируется исключение `java.lang.AssertionError`. В случае использования второй формы оператора `assert` выражение после двоеточия задаёт детальное сообщение о произошедшей ошибке (вычисленное выражение будет преобразовано в строку и передано конструктору `AssertionError`).

[к оглавлению](#)

Что такое *Heap* и *Stack* память в Java? Какая разница между ними?

Heap (куча) используется **Java Runtime** для выделения памяти под объекты и классы. Создание нового объекта также происходит в куче. Это же является областью работы сборщика мусора. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться из любой части приложения.

Stack (стек) это область хранения данных также находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. **Размер стековой памяти намного меньше объема памяти в куче.** Стек в Java работает по схеме *LIFO* (Последний-зашел-Первый-вышел)

Различия между *Heap* и *Stack* памятью:

- Куча используется всеми частями приложения, в то время как стек используется только одним потоком исполнения программы.
- Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится лишь ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче.
- Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков.
- Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы.
- Если память стека полностью занята, то Java Runtime бросает исключение `java.lang.StackOverflowError`. Если заполнена память кучи, то бросается исключение `java.lang.OutOfMemoryError: Java Heap Space`.
- Размер памяти стека намного меньше памяти в куче.
- Из-за простоты распределения памяти, стековая память работает намного быстрее кучи.

Для определения начального и максимального размера памяти в куче используются `-Xms` и `-Xmx` опции JVM. Для стека определить размер памяти можно с помощью опции `-Xss`.

[к оглавлению](#)

Верно ли утверждение, что примитивные типы данных всегда хранятся в стеке, а экземпляры ссылочных типов данных в куче?

Не совсем. Примитивное поле экземпляра класса хранится не в стеке, а в куче. Любой объект (всё, что явно или неявно создаётся при помощи оператора `new`) хранится в куче.

[к оглавлению](#)

Каким образом передаются переменные в методы, по значению или по ссылке?

В Java параметры всегда передаются только по значению, что определяется как «скопировать значение и передать копию». С примитивами это будет копия содержимого. Со ссылками - тоже копия содержимого, т.е. копия ссылки. При этом внутренние члены ссылочных типов через такую копию изменить возможно, а вот саму ссылку, указывающую на экземпляр - нет.

[к оглавлению](#)

Для чего нужен сборщик мусора?

Сборщик мусора (Garbage Collector) должен делать всего две вещи:

- **Находить мусор** - неиспользуемые объекты. (Объект считается неиспользуемым, если ни одна из сущностей в коде, выполняемом в данный момент, не содержит ссылок на него, либо цепочка ссылок, которая могла бы связать объект с некоторой сущностью приложения, обрывается);
- **Освобождать память от мусора.**

Существует два подхода к обнаружению мусора:

- *Reference counting*;
- *Tracing*

Reference counting (подсчёт ссылок). Суть этого подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Когда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором. Главным минусом такого подхода является сложность обеспечения точности счетчика. Также при таком подходе сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается), что приводит к утечкам памяти.

Главная идея подхода **Tracing** (трассировка) состоит в утверждении, что живыми могут считаться только те объекты, до которых мы можем добраться из **корневых точек (GC Root)** и те объекты, которые доступны с живого объекта. Всё остальное - мусор.

Существует 4 типа **корневых точки**:

- Локальные переменные и параметры методов;
- Потоки;
- Статические переменные;
- Ссылки из JNI.

Самое простое java приложение будет иметь корневые точки:

- Локальные переменные внутри `main()` метода и параметры `main()` метода;
- Поток который выполняет `main()` ;
- Статические переменные класса, внутри которого находится `main()` метод.

Таким образом, если мы представим все объекты и ссылки между ними как дерево, то нам нужно будет пройти с корневых узлов (точек) по всем рёбрам. При этом узлы, до которых мы сможем добраться - не мусор, все остальные - мусор.

При таком подходе **циклические зависимости** легко выявляются. **HotSpot VM** использует именно такой подход.

Для **очистки памяти от мусора** существуют два основных метода:

- **Copying collectors**
- **Mark-and-sweep**

При **copying collectors** подходе память делится на две части **«from-space»** и **«to-space»**, при этом сам принцип работы такой:

- Объекты создаются в **«from-space»**;
- Когда **«from-space»** заполняется, приложение приостанавливается;
- Запускается сборщик мусора. Находятся живые объекты в **«from-space»** и копируются в **«to-space»**;
- Когда все объекты скопированы **«from-space»** полностью очищается;
- **«to-space»** и **«from-space»** меняются местами.

Главный плюс такого подхода в том, что объекты плотно забивают память. Минусы подхода:

1. Приложение должно быть остановлено на время, необходимое для полного прохождения цикла сборки мусора;
2. В худшем случае (когда все объекты живые) «form-space» и «to-space» будут обязаны быть одинакового размера.

Алгоритм работы mark-and-sweep можно описать так:

- Объекты создаются в памяти;
- В момент, когда нужно запустить сборщик мусора приложение приостанавливается;
- Сборщик проходит по дереву объектов, помечая живые объекты;
- Сборщик проходит по всей памяти, находя все не отмеченные куски памяти и сохраняя их в «free list»;
- Когда новые объекты начинают создаваться они создаются в памяти доступной во «free list».

Минусы этого способа:

1. Приложение не работает пока происходит сборка мусора;
2. Время остановки напрямую зависит от размеров памяти и количества объектов;
3. Если не использовать «compacting» память будет использоваться не эффективно.

Сборщики мусора HotSpot VM используют комбинированный подход **Generational Garbage Collection**, который позволяет использовать разные алгоритмы для разных этапов сборки мусора. Этот подход опирается на том, что:

- большинство создаваемых объектов быстро становятся мусором;
- существует мало связей между объектами, которые были созданы в прошлом и только что созданными объектами.

[к оглавлению](#)

Как работает сборщик мусора?

Механизм сборки мусора - это процесс освобождения места в куче, для возможности добавления новых объектов.

Объекты создаются посредством оператора `new`, тем самым присваивая объекту ссылку. Для окончания работы с объектом достаточно просто перестать на него ссылаться, например, присвоив переменной ссылку на другой объект или значение `null`; прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом. Объекты, ссылки на которые отсутствуют, принято называть мусором (*garbage*), который будет удален.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые недостижимы из исполняемого кода, ввиду отсутствия ссылок на них, удаляются с высвобождением отведенной для них памяти. Точнее говоря, объект не попадает в сферу действия процесса сборки мусора, если он достижим посредством цепочки ссылок, начиная с корневой (*GC Root*) ссылки, т.е. ссылки, непосредственно существующей в выполняемом коде.

Память освобождается сборщиком мусора по его собственному «усмотрению». Программа может успешно завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте и поэтому ей так и не потребуются «услуги» сборщика мусора.

Мусор собирается системой автоматически, без вмешательства пользователя или программиста, но это не значит, что этот процесс не требует внимания вовсе. Необходимость создания и удаления большого количества объектов существенным образом сказывается на производительности приложений и, если быстродействие программы является важным фактором, следует тщательно обдумывать решения, связанные с созданием объектов, — это, в свою очередь, уменьшит и объем мусора, подлежащего утилизации.

[к оглавлению](#)

Какие разновидности сборщиков мусора реализованы в виртуальной машине HotSpot?

Java HotSpot VM предоставляет разработчикам на выбор четыре различных сборщика мусора:

- **Serial (последовательный)** — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. На данный

момент используется сравнительно редко, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию. Использование Serial GC включается опцией `-XX:+UseSerialGC`.

- **Parallel (параллельный)** — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности. Параллельный сборщик включается опцией `-XX:+UseParallelGC`.
- **Concurrent Mark Sweep (CMS)** — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти. Использование CMS GC включается опцией `-XX:+UseConcMarkSweepGC`.
- **Garbage-First (G1)** — создан для замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных. G1 включается опцией Java `-XX:+UseG1GC`.

[к оглавлению](#)

Опишите алгоритм работы какого-нибудь сборщика мусора, реализованного в виртуальной машине HotSpot.

Serial Garbage Collector (Последовательный сборщик мусора) был одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложения приостанавливается и продолжает работать только после прекращения сборки мусора.

Память приложения делится на три пространства:

- *Young generation*. Объекты создаются именно в этом участке памяти.
- *Old generation*. В этот участок памяти перемещаются объекты, которые переживают «minor garbage collection».
- *Permanent generation*. Тут хранятся метаданные об объектах, *Class data sharing (CDS)*, *пул строк (String pool)*. Permanent область делится на две: только для чтения и для чтения-записи. Очевидно, что в этом случае область только для чтения не чистится сборщиком мусора никогда.

Область памяти Young generation состоит из трёх областей: *Eden* и двух меньших по размеру *Survivor spaces* - *To space* и *From space*. Большинство объектов создаются в области *Eden*, за исключением очень больших объектов, которые не могут быть размещены в ней и поэтому сразу размещаются в *Old generation*. В *Survivor spaces* перемещаются объекты, которые пережили по крайней мере одну сборку мусора, но ещё не достигли порога «старости» (*tenuring threshold*), чтобы быть перемещёнными в *Old generation*.

Когда Young generation заполняется, то в этой области запускается процесс лёгкой сборки (*minor collection*), в отличие от процесса сборки, проводимого над всей кучей (*full collection*). Он происходит следующим образом: в начале работы одно из *Survivor spaces* - *To space*, является пустым, а другое - *From space*, содержит объекты, пережившие предыдущие сборки. Сборщик мусора ищет живые объекты в *Eden* и копирует их в *To space*, а затем копирует туда же и живые «молодые» (то есть не пережившие еще заданное число сборок мусора) объекты из *From space*. Старые объекты из *From space* перемещаются в *Old generation*. После лёгкой сборки *From space* и *To space* меняются ролями, область *Eden* становится пустой, а число объектов в *Old generation* увеличивается.

Если в процессе копирования живых объектов *To space* переполняется, то оставшиеся живые объекты из *Eden* и *From space*, которым не хватило места в *To space*, будут перемещены в *Old generation*, независимо от того, сколько сборок мусора они пережили.

Поскольку при использовании этого алгоритма сборщик мусора просто копирует все живые объекты из одной области памяти в другую, то такой сборщик мусора называется *copying* (копирующий). Очевидно, что для работы копирующего сборщика мусора у приложения всегда должна быть свободная область памяти, в которую будут копироваться живые объекты, и такой алгоритм может применяться для областей памяти сравнительно небольших по отношению к общему размеру памяти приложения. Young generation как раз удовлетворяет этому условию (по умолчанию на машинах клиентского типа эта область занимает около 10% кучи (значение может варьироваться в зависимости от платформы)).

Однако, для сборки мусора в *Old generation*, занимающем большую часть всей памяти, используется другой алгоритм.

В Old generation сборка мусора происходит с использованием алгоритма *mark-sweep-compact*, который состоит из трёх фаз. В фазе *Mark* (пометка) сборщик мусора помечает все живые объекты, затем, в фазе *Sweep* (очистка) все не помеченные объекты удаляются, а в фазе *Compact* (уплотнение) все живые объекты перемещаются в начало Old generation, в результате чего свободная память после очистки представляет собой непрерывную область. Фаза уплотнения выполняется для того, чтобы избежать фрагментации и упростить процесс выделения памяти в Old generation.

Когда свободная память представляет собой непрерывную область, то для выделения памяти под создаваемый объект можно использовать очень быстрый (около десятка машинных инструкций) алгоритм *bump-the-pointer*: адрес начала свободной памяти хранится в специальном указателе, и когда поступает запрос на создание нового объекта, код проверяет, что для нового объекта достаточно места, и, если это так, то просто увеличивает указатель на размер объекта.

Последовательный сборщик мусора отлично подходит для большинства приложений, использующих до 200 мегабайт кучи, работающих на машинах клиентского типа и не предъявляющих жёстких требований к величине пауз, затрачиваемых на сборку мусора. В то же время модель «stop-the-world» может вызвать длительные паузы в работе приложения при использовании больших объёмов памяти. Кроме того, последовательный алгоритм работы не позволяет оптимально использовать вычислительные ресурсы компьютера, и последовательный сборщик мусора может стать узким местом при работе приложения на многопроцессорных машинах.

[к оглавлению](#)

Что такое «пул строк»?

Пул строк – это набор строк, хранящийся в *Heap*.

- Пул строк возможен благодаря неизменяемости строк в Java и реализации идеи интернирования строк;
- Пул строк помогает экономить память, но по этой же причине создание строки занимает больше времени;
- Когда для создания строки используются " ", то сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка на неё;
- При использовании оператора `new` создаётся новый объект `String`. Затем

при помощи метода `intern()` эту строку можно поместить в пул или же получить из пула ссылку на другой объект `String` с таким же значением;

- Пул строк является примером паттерна «*Приспособленец*» (*Flyweight*).

[к оглавлению](#)

Что такое `finalize()` ? Зачем он нужен?

Через вызов метода `finalize()` (который наследуется от `Java.lang.Object`) JVM реализуется функциональность аналогичная функциональности деструкторов в C++, используемых для очистки памяти перед возвращением управления операционной системе. Данный метод вызывается при уничтожении объекта сборщиком мусора (*garbage collector*) и переопределяя `finalize()` можно запрограммировать действия необходимые для корректного удаления экземпляра класса - например, закрытие сетевых соединений, соединений с базой данных, снятие блокировок на файлы и т.д.

После выполнения этого метода объект должен быть повторно собран сборщиком мусора (и это считается серьезной проблемой метода `finalize()` т.к. он мешает сборщику мусора освобождать память). Вызов этого метода не гарантируется, т.к. приложение может быть завершено до того, как будет запущена сборка мусора.

Объект не обязательно будет доступен для сборки сразу же - метод `finalize()` может сохранить куда-нибудь ссылку на объект. Подобная ситуация называется «возрождением» объекта и считается антипаттерном. Главная проблема такого трюка - в том, что «возродить» объект можно только 1 раз.

Пример:

```
public class MainClass {  
  
    public static void main(String args[]) {  
        TestClass a = new TestClass();  
        a.a();  
        a = null;  
        a = new TestClass();  
        a.a();  
        System.out.println("!!! done");  
    }  
}
```

```
public class TestClass {  
  
    public void a() {  
        System.out.println("!!! a() called");  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        System.out.println("!!! finalize() called");  
        super.finalize();  
    }  
}
```

Так как в данном случае сборщик мусора может и не быть вызван (в силу простоты приложения), то результат выполнения программы с большой вероятностью будет следующий:

```
!!! a() called  
!!! a() called  
!!! done
```

Теперь несколько усложним программу, добавив принудительный вызов Garbage Collector:

```
public class MainClass {  
  
    public static void main(String args[]) {  
        TestClass a = new TestClass();  
        a.a();  
        a = null;  
        System.gc(); // Принудительно зовём сборщик мусора  
        a = new TestClass();  
        a.a();  
        System.out.println("!!! done");  
    }  
}
```

Как и было сказано ранее, Garbage Collector может в разное время отработать, поэтому результат выполнения может разниться от запуска к запуску: Вариант а:

```
!!! a() called
```

```
!!! a() called
!!! done
!!! finalize() called
```

Вариант б:

```
!!! a() called
!!! a() called
!!! finalize() called
!!! done
```

[к оглавлению](#)

Что произойдет со сборщиком мусора, если выполнение метода `finalize()` требует ощутимо много времени, или в процессе выполнения будет выброшено исключение?

Непосредственно вызов `finalize()` происходит в отдельном потоке *Finalizer* (`java.lang.ref.Finalizer.FinalizerThread`), который создаётся при запуске виртуальной машины (в статической секции при загрузке класса *Finalizer*). Методы `finalize()` вызываются последовательно в том порядке, в котором были добавлены в список сборщиком мусора. Соответственно, если какой-то `finalize()` зависнет, он подвесит поток *Finalizer*, но не сборщик мусора. Это в частности означает, что объекты, не имеющие метода `finalize()` , будут исправно удаляться, а вот имеющие будут добавляться в очередь, пока поток *Finalizer* не освободится, не завершится приложение или не кончится память.

То же самое применимо и выброшенным в процессе `finalize()` исключениям: метод `runFinalizer()` у потока *Finalizer* игнорирует все исключения выброшенные в момент выполнения `finalize()` . Таким образом возникновение исключительной ситуации никак не скажется на работоспособности сборщика мусора.

[к оглавлению](#)

Чем отличаются `final`, `finally` и `finalize()` ?

Модификатор `final` :

- Класс не может иметь наследников;
- Метод не может быть переопределен в классах наследниках;
- Поле не может изменить свое значение после инициализации;
- Локальные переменные не могут быть изменены после присвоения им значения;
- Параметры методов не могут изменять своё значение внутри метода.

Оператор `finally` гарантирует, что определенный в нём участок кода будет выполнен независимо от того, какие исключения были возбуждены и перехвачены в блоке `try-catch`.

Метод `finalize()` вызывается перед тем как сборщик мусора будет проводить удаление объекта.

Пример:

```
public class MainClass {

    public static void main(String args[]) {
        TestClass a = new TestClass();
        System.out.println("result of a.a() is " + a.a());
        a = null;
        System.gc(); // Принудительно зовём сборщик мусора
        a = new TestClass();
        System.out.println("result of a.a() is " + a.a());
        System.out.println("!!! done");
    }

}

public class TestClass {

    public int a() {
        try {
            System.out.println("!!! a() called");
            throw new Exception("");
        } catch (Exception e) {
            System.out.println("!!! Exception in a()");
            return 2;
        } finally {
            System.out.println("!!! finally in a() ");
        }
    }

}
```

```
@Override
protected void finalize() throws Throwable {
    System.out.println("!!! finalize() called");
    super.finalize();
}
}
```

Результат выполнения:

```
!!! a() called
!!! Exception in a()
!!! finally in a()
result of a.a() is 2
!!! a() called
!!! Exception in a()
!!! finally in a()
!!! finalize() called
result of a.a() is 2
!!! done
```

[к оглавлению](#)

Расскажите про приведение типов. Что такое понижение и повышение типа?

Java является строго типизированным языком программирования, а это означает, то что каждое выражение и каждая переменная имеет строго определенный тип уже на момент компиляции. Однако определен механизм *приведения типов* (*casting*) - способ преобразования значения переменной одного типа в значение другого типа.

В Java существуют несколько разновидностей приведения:

- **Тождественное (identity).** Преобразование выражения любого типа к точно такому же типу всегда допустимо и происходит автоматически.
- **Расширение (повышение, upcasting) примитивного типа (widening primitive).** Означает, что осуществляется переход от менее емкого типа к более ёмкому. Например, от типа `byte` (длина 1 байт) к типу `int` (длина 4 байта). Такие преобразование безопасны в том смысле, что новый тип всегда гарантировано вмещает в себя все данные, которые хранились в старом типе и

таким образом не происходит потери данных. Этот тип приведения всегда допустим и происходит автоматически.

- **Сужение (понижение, downcasting) примитивного типа (narrowing primitive).** Означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа `int` было больше `127`, то при приведении его к `byte` значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, при этом все старшие биты, не уместящиеся в новом типе, просто отбрасываются - никакого округления или других действий для получения более корректного результата не производится.
- **Расширение объектного типа (widening reference).** Означает неявное восходящее приведение типов или переход от более конкретного типа к менее конкретному, т.е. переход от потомка к предку. Разрешено всегда и происходит автоматически.
- **Сужение объектного типа (narrowing reference).** Означает нисходящее приведение, то есть приведение от предка к потомку (подтипу). Возможно только если исходная переменная является подтипом приводимого типа. При несоответствии типов в момент выполнения выбрасывается исключение `ClassCastException`. Требуется явное указание типа.
- **Преобразование к строке (to String).** Любой тип может быть приведен к строке, т.е. к экземпляру класса `String`.
- **Запрещенные преобразования (forbidden).** Не все приведения между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся приведения от любого ссылочного типа к примитивному и наоборот (кроме преобразования к строке). Кроме того, невозможно привести друг к другу классы, находящиеся на разных ветвях дерева наследования и т.п.

При приведении ссылочных типов с самим объектом ничего не происходит, - меняется лишь тип ссылки, через которую происходит обращение к объекту.

Для проверки возможности приведения нужно воспользоваться оператором `instanceof`:

```
Parent parent = new Child();
if (parent instanceof Child) {
    Child child = (Child) parent;
}
```

[к оглавлению](#)

Когда в приложении может быть выброшено исключение `ClassCastException`?

`ClassCastException` (потомок `RuntimeException`) - исключение, которое будет выброшено при ошибке приведения типа.

[к оглавлению](#)

Что такое литералы?

Литералы — это явно заданные значения в коде программы — константы определенного типа, которые находятся в коде в момент запуска.

```
class Test {  
    int a = 0b1101010110;  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

В этом классе "Hello world!" — литерал.

Переменная `a` - тоже литерал.

Литералы бывают разных типов, которые определяются их назначением и способом написания.

[к оглавлению](#)

Что такое *autoboxing* («автоупаковка») в Java и каковы правила упаковки примитивных типов в классы-обертки?

Автоупаковка - это механизм неявной инициализации объектов классов-оберток (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) значениями соответствующих им исходных примитивных типов (`byte`, `short`, `int` ...), без явного использования конструктора класса.

- Автоупаковка происходит при прямом присваивании примитива классу-обертке (с помощью оператора `=`), либо при передаче примитива в параметры метода (типа класса-обертки).
- Автоупаковке в классы-обертки могут быть подвергнуты как переменные примитивных типов, так и константы времени компиляции (литералы и `final` - примитивы). При этом литералы должны быть синтаксически корректными для инициализации переменной исходного примитивного типа.
- Автоупаковка переменных примитивных типов требует точного соответствия типа исходного примитива типу класса-обертки. Например, попытка упаковать переменную типа `byte` в `Short`, без предварительного явного приведения `byte` в `short` вызовет ошибку компиляции.
- Автоупаковка констант примитивных типов допускает более широкие границы соответствия. В этом случае компилятор способен предварительно осуществлять неявное расширение/сужение типа примитивов:
 - i. неявное расширение/сужение исходного типа примитива до типа примитива, соответствующего классу-обертке (для преобразования `int` в `Byte`, сначала компилятор самостоятельно неявно сужает `int` к `byte`)
 - ii. автоупаковку примитива в соответствующий класс-обертку. Однако, в этом случае существуют два дополнительных ограничения: а) присвоение примитива обертке может производиться только оператором `=` (нельзя передать такой примитив в параметры метода без явного приведения типов) б) тип левого операнда не должен быть старше чем `Character`, тип правого не должен старше, чем `int`: допустимо расширение/сужение `byte` в/из `short`, `byte` в/из `char`, `short` в/из `char` и только сужение `byte` из `int`, `short` из `int`, `char` из `int`. Все остальные варианты требуют явного приведения типов).

Дополнительной особенностью целочисленных классов-обертки, созданных автоупаковкой констант в диапазоне `-128 ... +127` является то, что они кэшируются JVM. Поэтому такие обертки с одинаковыми значениями будут являться ссылками на один объект.

[к оглавлению](#)

Какие есть особенности класса `String`?

- Это неизменяемый (`immutable`) и финализированный тип данных;

- Все объекты класса `String` JVM хранит в пуле строк;
- Объект класса `String` можно получить, используя двойные кавычки;
- Можно использовать оператор `+` для конкатенации строк;
- Начиная с Java 7 строки можно использовать в конструкции `switch`.

[к оглавлению](#)

Почему `String` неизменяемый и финализированный класс?

Есть несколько преимуществ в неизменности строк:

- Пул строк возможен только потому, что строка неизменяемая, таким образом виртуальная машина сохраняет больше свободного места в *Heap*, поскольку разные строковые переменные указывают на одну и ту же переменную в пуле. Если бы строка была изменяемой, то интернирование строк не было бы возможным, потому что изменение значения одной переменной отразилось бы также и на остальных переменных, ссылающихся на эту строку.
- Если строка будет изменяемой, тогда это станет серьезной угрозой безопасности приложения. Например, имя пользователя базы данных и пароль передаются строкой для получения соединения с базой данных и в программировании сокетов реквизиты хоста и порта передаются строкой. Так как строка неизменяемая, её значение не может быть изменено, в противном случае злоумышленник может изменить значение ссылки и вызвать проблемы в безопасности приложения.
- Неизменяемость позволяет избежать синхронизации: строки безопасны для многопоточности и один экземпляр строки может быть совместно использован различными потоками.
- Строки используются *classloader* и неизменность обеспечивает правильность загрузки класса.
- Поскольку строка неизменяемая, её `hashCode()` кэшируется в момент создания и нет необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в `HashMap` т.к. его обработка происходит быстрее.

[к оглавлению](#)

Почему `char[]` предпочтительнее `String` для

хранения пароля?

С момента создания строка остаётся в пуле, до тех пор, пока не будет удалена сборщиком мусора. Поэтому, даже после окончания использования пароля, он некоторое время продолжает оставаться доступным в памяти и способа избежать этого не существует. Это представляет определённый риск для безопасности, поскольку кто-либо, имеющий доступ к памяти сможет найти пароль в виде текста. В случае использования массива символов для хранения пароля имеется возможность очистить его сразу по окончании работы с паролем, позволяя избежать риска безопасности, свойственного строке.

[к оглавлению](#)

Почему строка является популярным ключом в HashMap в Java?

Поскольку строки неизменяемы, их хэш код вычисляется и кэшируется в момент создания, не требуя повторного пересчета при дальнейшем использовании. Поэтому в качестве ключа `HashMap` они будут обрабатываться быстрее.

[к оглавлению](#)

Что делает метод `intern()` в классе `String`?

Метод `intern()` используется для сохранения строки в пуле строк или получения ссылки, если такая строка уже находится в пуле.

[к оглавлению](#)

Можно ли использовать строки в конструкции `switch`?

Да, начиная с Java 7 в операторе `switch` можно использовать строки, ранние версии Java не поддерживают этого. При этом:

- участвующие строки чувствительны к регистру;
- используется метод `equals()` для сравнения полученного значения со значениями `case`, поэтому во избежание `NullPointerException` стоит предусмотреть проверку на `null`.

- согласно документации, Java 7 для строк в `switch`, компилятор Java формирует более эффективный байткод для строк в конструкции `switch`, чем для сцепленных условий `if - else`.

[к оглавлению](#)

Какая основная разница между `String`, `StringBuffer`, `StringBuilder`?

Класс `String` является неизменяемым (*immutable*) - модифицировать объект такого класса нельзя, можно лишь заменить его созданием нового экземпляра.

Класс `StringBuffer` изменяемый - использовать `StringBuffer` следует тогда, когда необходимо часто модифицировать содержимое.

Класс `StringBuilder` был добавлен в Java 5 и он во всем идентичен классу `StringBuffer` за исключением того, что он не синхронизирован и поэтому его методы выполняются значительно быстрее.

[к оглавлению](#)

Что такое класс `Object`? Какие в нем есть методы?

`Object` это базовый класс для всех остальных объектов в Java. Любой класс наследуется от `Object` и, соответственно, наследуют его методы:

`public boolean equals(Object obj)` – служит для сравнения объектов по значению;

`int hashCode()` – возвращает hash код для объекта;

`String toString()` – возвращает строковое представление объекта;

`Class getClass()` – возвращает класс объекта во время выполнения;

`protected Object clone()` – создает и возвращает копию объекта;

`void notify()` – возобновляет поток, ожидающий монитор;

`void notifyAll()` – возобновляет все потоки, ожидающие монитор;

`void wait()` – остановка вызвавшего метод потока до момента пока другой поток не вызовет метод `notify()` или `notifyAll()` для этого объекта;

`void wait(long timeout)` – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод `notify()` или `notifyAll()` для этого объекта;

`void wait(long timeout, int nanos)` – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод `notify()` или `notifyAll()` для этого объекта;

`protected void finalize()` – может вызываться сборщиком мусора в момент удаления объекта при сборке мусора.

[к оглавлению](#)

Дайте определение понятию «конструктор».

Конструктор — это специальный метод, у которого отсутствует возвращаемый тип и который имеет то же имя, что и класс, в котором он используется. Конструктор вызывается при создании нового объекта класса и определяет действия необходимые для его инициализации.

[к оглавлению](#)

Что такое «конструктор по умолчанию»?

Если у какого-либо класса не определить конструктор, то компилятор сгенерирует конструктор без аргументов - так называемый «конструктор по умолчанию».

```
public class ClassName() {}
```

Если у класса уже определен какой-либо конструктор, то конструктор по умолчанию создан не будет и, если он необходим, его нужно описывать явно.

[к оглавлению](#)

Чем отличаются конструктор по-умолчанию, конструктор копирования и конструктор с параметрами?

У конструктора по умолчанию отсутствуют какие-либо аргументы. Конструктор копирования принимает в качестве аргумента уже существующий объект класса для последующего создания его клона. Конструктор с параметрами имеет в своей сигнатуре аргументы (обычно необходимые для инициализации полей класса).

[к оглавлению](#)

Где и как вы можете использовать приватный конструктор?

Приватный (помеченный ключевым словом `private`, скрытый) конструктор может использоваться публичным статическим методом генерации объектов данного класса. Также доступ к нему разрешён вложенным классам и может использоваться для их нужд.

[к оглавлению](#)

Расскажите про классы-загрузчики и про динамическую загрузку классов.

Основа работы с классами в Java — классы-загрузчики, обычные Java-объекты, предоставляющие интерфейс для поиска и создания объекта класса по его имени во время работы приложения.

В начале работы программы создается 3 основных загрузчика классов:

- **базовый загрузчик (bootstrap/primordial)**. Загружает основные системные и внутренние классы JDK (*Core API* - пакеты `java.*` (`rt.jar` и `i18n.jar`)). Важно заметить, что базовый загрузчик является «*Изначальным*» или «*Корневым*» и частью JVM, вследствие чего его нельзя создать внутри кода программы.
- **загрузчик расширений (extention)**. Загружает различные пакеты расширений, которые располагаются в директории `<JAVA_HOME>/lib/ext` или другой директории, описанной в системном параметре `java.ext.dirs`. Это позволяет обновлять и добавлять новые расширения без необходимости модифицировать настройки используемых приложений. Загрузчик расширений реализован классом `sun.misc.Launcher$ExtClassLoader`.
- **системный загрузчик (system/application)**. Загружает классы, пути к которым указаны в переменной окружения `CLASSPATH` или пути, которые указаны в командной строке запуска JVM после ключей `-classpath` или `-cp`. Системный

загрузчик реализован классом `sun.misc.Launcher$AppClassLoader` .

Загрузчики классов являются иерархическими: каждый из них (кроме базового) имеет родительский загрузчик и в большинстве случаев, перед тем как попробовать загрузить класс самостоятельно, он посылает вначале запрос родительскому загрузчику загрузить указанный класс. Такое делегирование позволяет загружать классы тем загрузчиком, который находится ближе всего к базовому в иерархии делегирования. Как следствие поиск классов будет происходить в источниках в порядке их доверия: сначала в библиотеке *Core API*, потом в папке расширений, потом в локальных файлах `CLASSPATH` .

Процесс загрузки класса состоит из трех частей:

- *Loading* – на этой фазе происходит поиск и физическая загрузка файла класса в определенном источнике (в зависимости от загрузчика). Этот процесс определяет базовое представление класса в памяти. На этом этапе такие понятия как «методы», «поля» и т.д. пока не известны.
- *Linking* – процесс, который может быть разбит на 3 части:
 - *Bytecode verification* – проверка байт-кода на соответствие требованиям, определенным в спецификации JVM.
 - *Class preparation* – создание и инициализация необходимых структур, используемых для представления полей, методов, реализованных интерфейсов и т.п., определенных в загружаемом классе.
 - *Resolving* – загрузка набора классов, на которые ссылается загружаемый класс.
- *Initialization* – вызов статических блоков инициализации и присваивание полям класса значений по умолчанию.

Динамическая загрузка классов в Java имеет ряд особенностей:

- *отложенная (lazy) загрузка и связывание классов*. Загрузка классов производится только при необходимости, что позволяет экономить ресурсы и распределять нагрузку.
- *проверка корректности загружаемого кода (type safeness)*. Все действия связанные с контролем использования типов производятся только во время загрузки класса, позволяя избежать дополнительной нагрузки во время выполнения кода.
- *программируемая загрузка*. Пользовательский загрузчик полностью контролирует процесс получения запрошенного класса — самому ли искать байт-код и создавать класс или делегировать создание другому загрузчику.

Дополнительно существует возможность выставять различные атрибуты безопасности для загружаемых классов, позволяя таким образом работать с кодом из ненадежных источников.

- *множественные пространства имен*. Каждый загрузчик имеет своё пространство имён для создаваемых классов. Соответственно, классы, загруженные двумя различными загрузчиками на основе общего байт-кода, в системе будут различаться.

Существует несколько способов инициировать загрузку требуемого класса:

- явный: вызов `ClassLoader.loadClass()` или `Class.forName()` (по умолчанию используется загрузчик, создавший текущий класс, но есть возможность и явного указания загрузчика);
- неявный: когда для дальнейшей работы приложения требуется ранее не использованный класс, JVM иницирует его загрузку.

[к оглавлению](#)

Что такое *Reflection*?

Рефлексия (Reflection) - это механизм получения данных о программе во время её выполнения (runtime). В Java *Reflection* осуществляется с помощью *Java Reflection API*, состоящего из классов пакетов `java.lang` и `java.lang.reflect`.

Возможности Java Reflection API:

- Определение класса объекта;
- Получение информации о модификаторах класса, полях, методах, конструкторах и суперклассах;
- Определение интерфейсов, реализуемых классом;
- Создание экземпляра класса;
- Получение и установка значений полей объекта;
- Вызов методов объекта;
- Создание нового массива.

[к оглавлению](#)

Зачем нужен `equals()`. Чем он отличается от операции `==`?

Метод `equals()` - определяет отношение эквивалентности объектов.

При сравнении объектов с помощью `==` сравнение происходит лишь между ссылками. При сравнении по переопределённому разработчиком `equals()` - по внутреннему состоянию объектов.

[к оглавлению](#)

Если вы хотите переопределить `equals()`, какие условия должны выполняться?

Какими свойствами обладает порождаемое `equals()` отношение эквивалентности?

- *Рефлексивность*: для любой ссылки на значение `x`, `x.equals(x)` вернет `true`;
- *Симметричность*: для любых ссылок на значения `x` и `y`, `x.equals(y)` должно вернуть `true`, тогда и только тогда, когда `y.equals(x)` возвращает `true`.
- *Транзитивность*: для любых ссылок на значения `x`, `y` и `z`, если `x.equals(y)` и `y.equals(z)` возвращают `true`, тогда и `x.equals(z)` вернёт `true`;
- *Непротиворечивость*: для любых ссылок на значения `x` и `y`, если несколько раз вызвать `x.equals(y)`, постоянно будет возвращаться значение `true` либо постоянно будет возвращаться значение `false` при условии, что никакая информация, используемая при сравнении объектов, не поменялась.

Для любой ненулевой ссылки на значение `x` выражение `x.equals(null)` должно возвращать `false`.

[к оглавлению](#)

Правила переопределения метода `Object.equals()`.

1. Использование оператора `==` для проверки, является ли аргумент ссылкой на указанный объект. Если является, возвращается `true`. Если сравниваемый объект `== null`, должно вернуться `false`.
2. Использование вызова метода `getClass()` для проверки, имеет ли аргумент правильный тип. Если не имеет, возвращается `false`.
3. Приведение аргумента к правильному типу. Поскольку эта операция следует за

проверкой `instanceof` она гарантированно будет выполнена.

4. Обход всех значимых полей класса и проверка того, что значение поля в текущем объекте и значение того же поля в проверяемом на эквивалентность аргументе соответствуют друг другу. Если проверки для всех полей прошли успешно, возвращается результат `true`, в противном случае - `false`.

По окончании переопределения метода `equals()` следует проверить: является ли порождаемое отношение эквивалентности рефлексивным, симметричным, транзитивным и непротиворечивым? Если ответ отрицательный, метод подлежит соответствующей правке.

[к оглавлению](#)

Какая связь между `hashCode()` и `equals()`?

Если `equals()` переопределен, есть ли какие-либо другие методы, которые следует переопределить?

Равные объекты должны возвращать одинаковые хэш коды. При переопределении `equals()` нужно обязательно переопределять и метод `hashCode()`.

[к оглавлению](#)

Что будет, если переопределить `equals()` не переопределяя `hashCode()`? Какие могут возникнуть проблемы?

Классы и методы, которые используют правила этого контракта могут работать некорректно. Так для `HashMap` это может привести к тому, что пара «ключ-значение», которая была в неё помещена при использовании нового экземпляра ключа не будет в ней найдена.

[к оглавлению](#)

Каким образом реализованы методы `hashCode()` и `equals()` в классе `Object`?

Реализация метода `Object.equals()` сводится к проверке на равенство двух ссылок:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Реализация метода `Object.hashCode()` описана как `native`, т.е. определенной не с помощью Java кода и обычно возвращает адрес объекта в памяти:

```
public native int hashCode();
```

[к оглавлению](#)

Для чего нужен метод `hashCode()` ?

Метод `hashCode()` необходим для вычисления хэш кода переданного в качестве входного параметра объекта. В Java это целое число, в более широком смысле - битовая строка фиксированной длины, полученная из массива произвольной длины. Этот метод реализован таким образом, что для одного и того же входного объекта, хэш код всегда будет одинаковым. Следует понимать, что в Java множество возможных хэш кодов ограничено типом `int`, а множество объектов ничем не ограничено. Из-за этого, вполне возможна ситуация, что хэш коды разных объектов могут совпасть:

- если хэш коды разные, то и объекты гарантированно разные;
- если хэш коды равны, то объекты не обязательно равны(могут быть разные).

[к оглавлению](#)

Каковы правила переопределения метода `Object.hashCode()` ?

Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете `hashCode()` ?

Общий совет: выбирать поля, которые с большой долей вероятности будут различаться. Для этого необходимо использовать уникальные, лучше всего примитивные поля, например, такие как `id`, `uuid`. При этом нужно следовать правилу, если поля задействованы при вычислении `hashCode()`, то они должны быть задействованы и при выполнении `equals()`.

[к оглавлению](#)

Могут ли у разных объектов быть одинаковые `hashCode()` ?

Да, могут. Метод `hashCode()` не гарантирует уникальность возвращаемого значения. Ситуация, когда у разных объектов одинаковые хэш коды называется *коллизией*. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хэш кода.

[к оглавлению](#)

Если у класса `Point{int x, y;}` реализовать метод `equals(Object that) {(return this.x == that.x && this.y == that.y)}`, но сделать хэш код в виде `int hashCode() {return x;}`, то будут ли корректно такие точки помещаться и извлекаться из `HashSet` ?

`HashSet` использует `HashMap` для хранения элементов. При добавлении элемента в `HashMap` вычисляется хэш код, по которому определяется позиция в массиве, куда будет вставлен новый элемент. У всех экземпляров класса `Point` хэш код будет одинаковым для всех объектов с одинаковым `x`, что приведёт к вырождению хэш таблицы в список.

При возникновении коллизии в `HashMap` осуществляется проверка на наличие элемента в списке: `e.hash == hash && ((k = e.key) == key || key.equals(k))`. Если элемент найден, то его значение перезаписывается. В нашем случае для разных объектов метод `equals()` будет возвращать `false`. Соответственно новый элемент будет успешно добавлен в `HashSet`. Извлечение элемента также будет осуществляться успешно. Но производительность такого кода будет невысокой и преимущества хэш таблиц использоваться не будут.

[к оглавлению](#)

Могут ли у разных объектов (`ref0 != ref1`) быть `ref0.equals(ref1) == true`?

Да, могут. Для этого в классе этих объектов должен быть переопределен метод `equals()`.

Если используется метод `Object.equals()`, то для двух ссылок `x` и `y` метод вернет `true` тогда и только тогда, когда обе ссылки указывают на один и тот же объект (т.е. `x == y` возвращает `true`).

[к оглавлению](#)

Могут ли у разных ссылок на один объект (`ref0 == ref1`) быть `ref0.equals(ref1) == false`?

В общем случае - могут, если метод `equals()` реализован некорректно и не выполняет свойство рефлексивности: для любых ненулевых ссылок `x` метод `x.equals(x)` должен возвращать `true`.

[к оглавлению](#)

Можно ли так реализовать метод `equals(Object that) {return this.hashCode() == that.hashCode();}`?

Строго говоря нельзя, поскольку метод `hashCode()` не гарантирует уникальность значения для каждого объекта. Однако для сравнения экземпляров класса `Object` такой код допустим, т.к. метод `hashCode()` в классе `Object` возвращает уникальные значения для разных объектов (его вычисление основано на использовании адреса объекта в памяти).

[к оглавлению](#)

В `equals()` требуется проверять, что аргумент `equals(Object that)` такого же типа что и сам

объект. В чем разница между `this.getClass() == that.getClass()` и `that instanceof MyClass`?

Оператор `instanceof` сравнивает объект и указанный тип. Его можно использовать для проверки является ли данный объект экземпляром некоторого класса, либо экземпляром его дочернего класса, либо экземпляром класса, который реализует указанный интерфейс.

`this.getClass() == that.getClass()` проверяет два класса на идентичность, поэтому для корректной реализации контракта метода `equals()` необходимо использовать точное сравнение с помощью метода `getClass()`.

[к оглавлению](#)

Можно ли реализовать метод `equals()` класса `MyClass` вот так: `class MyClass {public boolean equals(MyClass that) {return this == that;}}`?

Реализовать можно, но данный метод не переопределяет метод `equals()` класса `Object`, а перегружает его.

[к оглавлению](#)

Есть класс `Point{int x, y;}`. Почему хэш код в виде `31 * x + y` предпочтительнее чем `x + y`?

Множитель создает зависимость значения хэш кода от очередности обработки полей, что в итоге порождает лучшую хэш функцию.

[к оглавлению](#)

Расскажите про клонирование объектов.

Использование оператора присваивания не создает нового объекта, а лишь копирует ссылку на объект. Таким образом, две ссылки указывают на одну и ту же область памяти, на один и тот же объект. Для создания нового объекта с таким же состоянием используется клонирование объекта.

Класс `Object` содержит `protected` метод `clone()`, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод `clone()` как `public` для обеспечения возможности его вызова. В переопределенном методе следует вызвать базовую версию метода `super.clone()`, которая и выполняет собственно клонирование.

Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс `Cloneable`. Интерфейс `Cloneable` не содержит методов относится к маркерным интерфейсам, а его реализация гарантирует, что метод `clone()` класса `Object` возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение `CloneNotSupportedException`. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора.

Это решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их обёрток или неизменяемых (`immutable`) объектных типов. Если же поле клонируемого типа является изменяемым ссылочным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и сам объект поля класса.

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс `Cloneable` и переопределяет метод `clone()`. Так как, если это будет иначе вызов метода невозможен из-за его недоступности. Отсюда следует, что если класс имеет суперкласс, то для реализации механизма клонирования текущего класса-потомка необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений `final` для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

Помимо встроенного механизма клонирования в Java для клонирования объекта можно использовать:

- **Специализированный конструктор копирования** - в классе описывается конструктор, который принимает объект этого же класса и инициализирует поля создаваемого объекта значениями полей переданного.
- **Фабричный метод** - (Factory method), который представляет собой статический метод, возвращающий экземпляр своего класса.
- **Механизм сериализации** - сохранение и последующее восстановление объекта в/из потока байтов.

[к оглавлению](#)

В чем отличие между *поверхностным* и *глубоким* клонированием?

Поверхностное копирование копирует настолько малую часть информации об объекте, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т.е. класс `Object` не знает о структуре класса, которого он копирует. Клонирование такого типа осуществляется JVM по следующим правилам:

- Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.
- Если класс помимо членов примитивных типов содержит члены ссылочных типов, то тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.

Глубокое копирование дублирует абсолютно всю информацию объекта:

- Нет необходимости копировать отдельно примитивные данные;
- Все члены ссылочного типа в оригинальном классе должны поддерживать клонирование. Для каждого такого члена при переопределении метода `clone()` должен вызываться `super.clone()` ;
- Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

[к оглавлению](#)

Какой способ клонирования предпочтительней?

Наиболее безопасным и, следовательно, предпочтительным способом клонирования является использование специализированного конструктора копирования:

- Отсутствие ошибок наследования (не нужно беспокоиться, что у наследников появятся новые поля, которые не будут скопированы через метод `clone()`);
- Поля для клонирования указываются явно;
- Возможность клонировать даже `final` поля.

[к оглавлению](#)

Почему метод `clone()` объявлен в классе `Object`, а не в интерфейсе `Cloneable`?

Метод `clone()` объявлен в классе `Object` с указанием модификатора `native`, чтобы обеспечить доступ к стандартному механизму поверхностного копирования объектов. Одновременно он объявлен и как `protected`, чтобы нельзя было вызвать этот метод у не переопределивших его объектов. Непосредственно интерфейс `Cloneable` является маркерным (не содержит объявлений методов) и нужен только для обозначения самого факта, что данный объект готов к тому, чтобы быть клонированным. Вызов переопределённого метода `clone()` у не `Cloneable` объекта вызовет выбрасывание `CloneNotSupportedException`.

[к оглавлению](#)

Опишите иерархию исключений.

Исключения делятся на несколько классов, но все они имеют общего предка — класс `Throwable`, потомками которого являются классы `Exception` и `Error`.

Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине.

Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы, предсказуемы и последствия которых возможно устранить внутри программы. Например, произошло деление целого числа на ноль.

[к оглавлению](#)

Какие виды исключений в Java вы знаете, чем они отличаются?

Что такое *checked* и *unchecked exception*?

В Java все исключения делятся на два типа:

- **checked (контролируемые/проверяемые исключения)** должны обрабатываться блоком `catch` или описываться в заголовке метода (например, `throws IOException`). Наличие такого обработчика/модификатора в заголовке метода проверяется на этапе компиляции;
- **unchecked (неконтролируемые/непроверяемые исключения)**, к которым относятся ошибки `Error` (например, `OutOfMemoryError`), обрабатывать которые не рекомендуется и исключения времени выполнения, представленные классом `RuntimeException` и его наследниками (например, `NullPointerException`), которые могут не обрабатываться блоком `catch` и не быть описанными в заголовке метода.

[к оглавлению](#)

Какой оператор позволяет принудительно выбросить исключение?

Это оператор `throw`:

```
throw new Exception();
```

[к оглавлению](#)

О чем говорит ключевое слово `throws` ?

Модификатор `throws` прописывается в заголовке метода и указывает на то, что метод потенциально может выбросить исключение с указанным типом.

[к оглавлению](#)

Как написать собственное («пользовательское») исключение?

Необходимо унаследоваться от базового класса требуемого типа исключений (например, от `Exception` или `RuntimeException`).

```
class CustomException extends Exception {  
    public CustomException() {  
        super();  
    }  
}
```

```
    }

    public CustomException(final String string) {
        super(string + " is invalid");
    }

    public CustomException(final Throwable cause) {
        super(cause);
    }
}
```

[к оглавлению](#)

Какие существуют *unchecked exception*?

Наиболее часто встречающиеся: `ArithmeticException` , `ClassCastException` , `ConcurrentModificationException` , `IllegalArgumentException` , `IllegalStateException` , `IndexOutOfBoundsException` , `NoSuchElementException` , `NullPointerException` , `UnsupportedOperationException` .

[к оглавлению](#)

Что представляет из себя ошибки класса `Error` ?

Ошибки класса `Error` представляют собой наиболее серьёзные проблемы уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине. Обработать такие ошибки не запрещается, но делать этого не рекомендуется.

[к оглавлению](#)

Что вы знаете о `OutOfMemoryError` ?

`OutOfMemoryError` выбрасывается, когда виртуальная машина Java не может создать (разместить) объект из-за нехватки памяти, а сборщик мусора не может высвободить достаточное её количество.

Область памяти, занимаемая java процессом, состоит из нескольких частей. Тип `OutOfMemoryError` зависит от того, в какой из них не хватило места:

- `java.lang.OutOfMemoryError: Java heap space` : Не хватает места в куче, а

именно, в области памяти в которую помещаются объекты, создаваемые в приложении программно. Обычно проблема кроется в утечке памяти. Размер задается параметрами `-Xms` и `-Xmx`.

- `java.lang.OutOfMemoryError: PermGen space` : (до версии Java 8) Данная ошибка возникает при нехватке места в *Permanent* области, размер которой задается параметрами `-XX:PermSize` и `-XX:MaxPermSize`.
- `java.lang.OutOfMemoryError: GC overhead limit exceeded` : Данная ошибка может возникнуть как при переполнении первой, так и второй областей. Связана она с тем, что памяти осталось мало и сборщик мусора постоянно работает, пытаясь высвободить немного места. Данную ошибку можно отключить с помощью параметра `-XX:-UseGCOverheadLimit`.
- `java.lang.OutOfMemoryError: unable to create new native thread` : Выбрасывается, когда нет возможности создавать новые потоки.

[к оглавлению](#)

Опишите работу блока *try-catch-finally*.

`try` — данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке. `catch` — ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений в случае их возникновения. `finally` — ключевое слово для отметки начала блока кода, который является дополнительным. Этот блок помещается после последнего блока `catch`. Управление передаётся в блок `finally` в любом случае, было выброшено исключение или нет.

Общий вид конструкции для обработки исключительной ситуации выглядит следующим образом:

```
try {  
    //код, который потенциально может привести к исключительной ситуации  
}  
catch(SomeException e) { //в скобках указывается класс конкретной ожидаемой ошиб  
    //код обработки исключительной ситуации  
}  
finally {  
    //необязательный блок, код которого выполняется в любом случае  
}
```

[к оглавлению](#)

Что такое механизм *try-with-resources*?

Данная конструкция, которая появилась в Java 7, позволяет использовать блок *try-catch* не заботясь о закрытии ресурсов, используемых в данном сегменте кода. Ресурсы объявляются в скобках сразу после *try*, а компилятор уже сам неявно создаёт секцию *finally*, в которой и происходит освобождение занятых в блоке ресурсов. Под ресурсами подразумеваются сущности, реализующие интерфейс `java.lang.AutoCloseable`.

Общий вид конструкции:

```
try(/*объявление ресурсов*/) {  
    //...  
} catch(Exception ex) {  
    //...  
} finally {  
    //...  
}
```

Стоит заметить, что блоки *catch* и явный *finally* выполняются уже после того, как закрываются ресурсы в неявном *finally*.

[к оглавлению](#)

Возможно ли использование блока *try-finally* (без *catch*)?

Такая запись допустима, но смысла в такой записи не так много, всё же лучше иметь блок *catch*, в котором будет обрабатываться необходимое исключение.

[к оглавлению](#)

Может ли один блок *catch* отлавливать сразу несколько исключений?

В Java 7 стала доступна новая языковая конструкция, с помощью которой можно перехватывать несколько исключений одним блоком *catch*:

```
try {  
    //...
```

```
    } catch(IOException | SQLException ex) {  
        //...  
    }
```

[к оглавлению](#)

Всегда ли выполняется блок `finally`?

Код в блоке `finally` будет выполнен всегда, независимо от того, выброшено исключение или нет.

[к оглавлению](#)

Существуют ли ситуации, когда блок `finally` не будет выполнен?

Например, когда JVM «умирает» - в такой ситуации `finally` недостижим и не будет выполнен, так как происходит принудительный системный выход из программы:

```
try {  
    System.exit(0);  
} catch(Exception e) {  
    e.printStackTrace();  
} finally { }
```

[к оглавлению](#)

Может ли метод `main()` выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?

Может и оно будет передано в виртуальную машину Java (JVM).

[к оглавлению](#)

Предположим, есть метод, который может выбросить `IOException` и `FileNotFoundException` в какой

последовательности должны идти блоки `catch` ?

Сколько блоков `catch` будет выполнено?

Общее правило: обрабатывать исключения нужно от «младшего» к старшему. Т.е. нельзя поставить в первый блок `catch(Exception ex) {}`, иначе все дальнейшие блоки `catch()` уже ничего не смогут обработать, т.к. любое исключение будет соответствовать обработчику `catch(Exception ex)`.

Таким образом, исходя из факта, что `FileNotFoundException` extends `IOException` сначала нужно обработать `FileNotFoundException`, а затем уже `IOException`:

```
void method() {  
    try {  
        //...  
    } catch (FileNotFoundException ex) {  
        //...  
    } catch (IOException ex) {  
        //...  
    }  
}
```

[к оглавлению](#)

Что такое *generics*?

Generics - это технический термин, обозначающий набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры.

Примером использования обобщенных типов может служить *Java Collection Framework*. Так, класс `LinkedList<E>` - типичный обобщенный тип. Он содержит параметр `E`, который представляет тип элементов, которые будут храниться в коллекции. Создание объектов обобщенных типов происходит посредством замены параметризованных типов реальными типами данных. Вместо того, чтобы просто использовать `LinkedList`, ничего не говоря о типе элемента в списке, предлагается использовать точное указание типа `LinkedList<String>`, `LinkedList<Integer>` и т.п.

[к оглавлению](#)

Что такое «интернационализация», «локализация»?

Интернационализация (internationalization) - способ создания приложений, при котором их можно легко адаптировать для разных аудиторий, говорящих на разных языках.

Локализация (localization) - адаптация интерфейса приложения под несколько языков. Добавление нового языка может внести определенные сложности в локализацию интерфейса.

[к оглавлению](#)

Источники

- [Quizful](#)
- [JavaStudy.ru](#)
- [ggenikus.github.io](#)
- [Санкт-Петербургская группа тестирования JVM](#)
- [Объектно-ориентированное программирование](#)
- [JavaRush](#)

[Вопросы для собеседования](#)