**Java собеседование: вопросы по ООП**

**1. Какие особенности есть у Java?**

Ответ:

1. ООП концепты:
   1. объектная ориентированность;
   2. наследование;
   3. инкапсуляция;
   4. полиморфизм;
   5. абстракция.
2. Кроссплатформенность: программа на Java может быть запущена на любой платформе без каких-либо изменений. Единственное, что нужно — установленная JVM (java virtual machine).
3. Высокая производительность: JIT(Just In Time compiler) позволяет высокую производительность. JIT конвертирует байт-код в машинный код и потом JVM стартует выполнение.
4. Мультипоточность: поток выполнения, известный как Thread. JVM создает thread, который называется main thread. Программист может создать несколько потоков наследованием от класса Thread или реализуя интерфейс Runnable.

**2. Что такое наследование?**

Под наследованием подразумевается, что один класс может наследовать(“extends”) другой класс. Таким образом можно переиспользовать код с класса, от которого наследуются. Существующий класс известен как superclass, а создаваемый — subclass. Также еще говорят parent и child.

**public** **class** Animal {

**private** **int** age;

}

**public** **class** Dog **extends** Animal {

}

где Animal — это parent, а Dog — child.

**3. Что такое инкапсуляция?**

Инкапсуляция — это сокрытие реализации при помощи модификаторов доступа, при помощи геттеров и сеттеров. Это делается для того, чтобы закрыть доступ для внешнего использования в тех местах, где разработчики считают нужным. Доступный пример из жизни — это автомобиль. У нас нет прямого доступа к работе двигателя. Для нас работа заключается в том, чтобы вставить ключ в зажигание и включить двигатель. А какие уже процессы будут происходить под капотом — не наше дело.

**4. Что такое полиморфизм?**

Полиморфизм — это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта. Как говорится, один интерфейс — множество реализаций. При помощи полиморфизма можно объединять и использовать разные типы объектов по их общему поведению. Например, есть у нас класс Animal, у которого есть два наследника — Dog и Cat. У общего класса Animal есть общее поведение для всех — издавать звук. В случае, когда нужно собрать воедино всех наследников класса Animal и выполнить метод “издавать звук”, используем возможности полиморфизма. Вот как будет это выглядеть:

List<Animal> animals = Arrays.asList(**new** Cat(), **new** Dog(), **new** Cat());

animals.forEach(animal -> animal.makeSound());

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

**Вопросы на собеседовании — Java Syntax**

**5. Что такое конструктор в Java?**

Следующие характеристики являются валидными:

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

**6. Какие два класса не наследуются от Object?**

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

**7. Что такое Local Variable?**

Еще один из популярных вопросов на собеседовании Java-разработчика. Local variable — это переменная, которая определена внутри метода и существует вплоть до того момента, пока выполняется этот метод. Как только выполнение закончится, локальная переменная перестанет существовать. Вот программа, которая использует локальную переменную helloMessage в методе main():

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

String helloMessage;

helloMessage = "Hello, World!";

System.out.println(helloMessage);

}

**8. Что такое Instance Variable?**

Instance Variable — переменная, которая определена внутри класса, и она существует вплоть до того момента, пока существует объект. Пример — класс Bee, в котором есть две переменные nectarCapacity и maxNectarCapacity:

**public** **class** Bee {

/\*\*

\* Current nectar capacity

\*/

**private** **double** nectarCapacity;

/\*\*

\* Maximal nectar that can take bee.

\*/

**private** **double** maxNectarCapacity = 20.0;

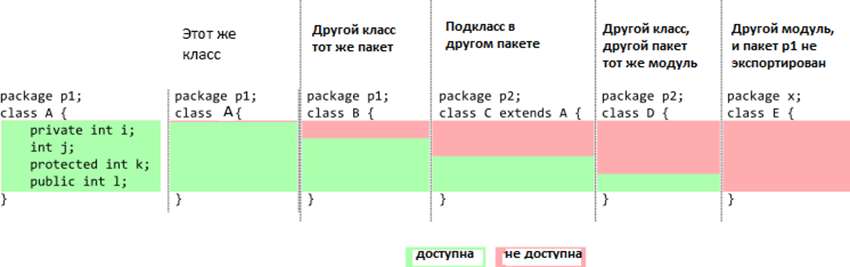
...

}

**9. Что такое модификаторы доступа?**

Модификаторы доступа — это инструмент, при помощи которого можно настроить доступ к классам, методам и переменным. Бывают следующие модификаторы, упорядоченные в порядке повышения доступа:

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

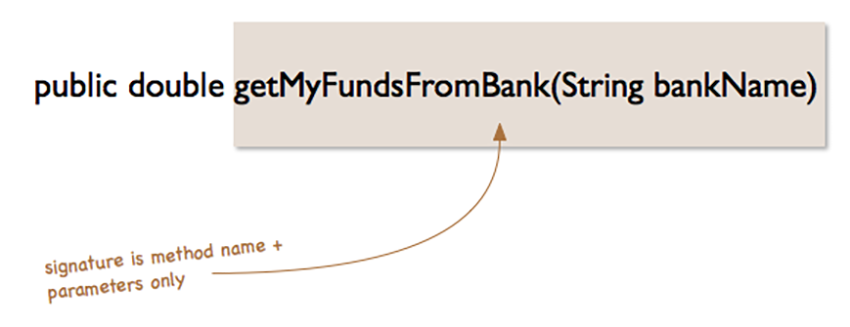


**10. Что такое переопределение (overriding) методов?**

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

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

**11. Что такое сигнатура метода?**

Сигнатура метода — это набор из названия метода и аргументов, какие принимает метод. Сигнатура метода является уникальным идентификатором для метода при перегрузке методов.

**12. Что такое перегрузка методов?**

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

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

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

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

**13. Что такое Interface?**

Множественное наследование не реализовано в джаве, поэтому чтобы преодолеть эту проблему, были добавлены интерфейсы в том виде, в котором мы их знаем ;) Долгое время у интерфейсов были только методы без их реализации. В рамках этого ответа поговорим именно о них. Например:

**public** **interface** Animal {

**void** makeSound();

**void** eat();

**void** sleep();

}

Из этого вытекают некоторые нюансы:

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

Вот так:

**public** **class** Cat **implements** Animal {

**public** **void** makeSound() {

// реализация метода

}

**public** **void** eat() {

// реализация

}

**public** **void** sleep() {

// реализация

}

}

**14. Что такое default method в Interface?**

Теперь поговорим о дефолтных методах. Для чего, для кого? Эти методы добавили, чтобы все сделать “и вашим, и нашим”. О чем это я? Да о том, что с одной стороны нужно было добавить новую функциональность: лямбды, Stream API, с другой стороны, нужно было оставить то, чем славится джава — обратную совместимость. Для этого нужно было ввести уже готовые решения в интерфейсы. Так к нам и пришли дефолтные методы. То есть, дефолтный метод — это реализованный метод в интерфейсе, у которого есть ключевое слово default. Например, всем известный метод stream() в интерфейсе Collection. Проверьте, этот интерфейс вовсе не так прост как кажется ;). Или также не менее известный метод forEach() из интерфейса Iterable. Его также не был до тех пор, пока не добавили дефолтные методы.

**15. А как тогда наследовать два одинаковых дефолтных метода?**

Исходя из предыдущего ответа на то, что такое дефолтный метод, можно задать другой вопрос. Если можно реализовать методы в интерфейсах, то теоретически можно реализовать два интерфейса с одинаковым методом, и как такое делать? Есть два разных интерфейса с одинаковым методом:

**interface** A {

**default** **void** foo() {

System.out.println(“Foo A”);

}

}

**interface** B {

**default** **void** foo() {

System.out.println(“Foo B”);

}

}

И есть класс, который реализует эти два интерфейса. Но только как выбрать специфический метод интерфейса А или В? Для этого есть конструкция такого вида: A.super.foo():

**public** **class** C **implements** A, B {

**public** **void** fooA() {

A.**super**.foo();

}

**public** **void** fooB() {

B.**super**.foo();

}

}

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

**16. Что такое абстрактные методы и классы?**

В джава есть зарезервированное слово abstract, которое используется для обозначения абстрактных классов и методов. Для начала — определения. Абстрактным методом называется метод, который создан без реализации с ключевым словом abstract в абстрактном классе. То есть, это метод как в интерфейсе, только с добавкой ключевого слова, например:

**public** **abstract** **void** foo();

Абстрактным классом называется класс, который имеет также abstract слово:

**public** **abstract** **class** A {

}

У абстрактного класса есть несколько особенностей:

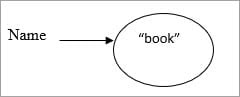
* на его основе нельзя создать объект;
* он может иметь абстрактные методы;
* он может и не иметь абстрактные методы.

Абстрактные классы нужны для обобщения какой-то абстракции, которой в реальной жизни нет, но она содержит множество общих поведений и состояний (то есть, методов и переменных). Примеров из жизни — хоть отбавляй. Это может быть “животное”, “машина”, “геометрическая фигура” и так далее.

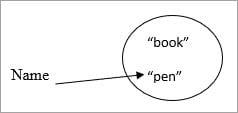
**17. Какая разница между String, String Builder и String Buffer?**

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

String name = "book";

...переменная будет ссылаться на стринг пул Constant string pool Если задать переменной name другое значение, получится следующее:

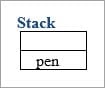
name = "pen";

Constant string pool Таким образом, эти два значения так и останутся там. String Buffer:

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

Пример:

StringBuffer name = “book”;

Как только значение name сменится, в стеке измениться значение: 

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

**18. Какая разница между абстрактным классом и интерфейсом?**

Абстрактный класс:

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

Интерфейс:

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

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

Для начала нужно уточнить, что O(1) — это обозначение [временной сложности алгоритма](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0), когда операция проходит за константное время. То есть это обозначение самого быстрого выполнения. Чтоб создать массив int, мы должны написать следующее:

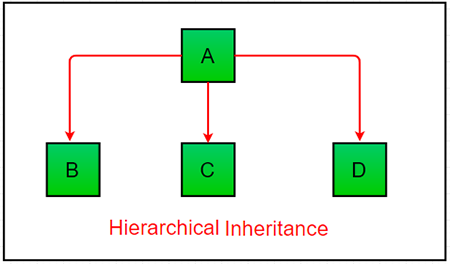
**int**[] intArray = **new** **int**[100];

Из этой записи можно сделать несколько выводов:

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

Из этого следует: чтобы понять, в какую ячейку записать, нужно просто вычислить, в какую область памяти записать. Для машины это проще простого. У машины есть начало выделенной памяти, количество элементов и размер одной ячейки. Из этого понятно, что место для записи будет равно начальному месту массива + размер ячейки, умноженный на ее размер.

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

Это вопрос сразу же идет за предыдущим. Ведь правда, когда мы работаем с массивом и там примитивы, то нам известно заранее, какой размер этого типа, при его создании. А что делать, если есть такая схема, как на картинке: 

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

List<A> list = **new** ArrayList();

list.add(**new** B());

list.add(**new** C());

list.add(**new** D());

list.add(**new** B());

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

**21. Автоупаковка (autoboxing) и Автораспаковка (unboxing)**

Автоупаковка (autoboxing) - процесс автоматического преобразования из примитивного типа в соответствующий класс обертку. Автораспаковка (unboxing) - делает ровно обратное к автоупаковке - преобразует класс обертку в примитив. А вот если окажется значение обертки null, то при распаковке будет выброшено исключение NullPointerException.

**Соответствие примитив - обертка**

|  |  |
| --- | --- |
| **Примитив** | **Класс обертка** |
| boolean | Boolean |
| int | Integer |
| byte | Byte |
| char | Character |
| float | Float |
| long | Long |
| short | Short |
| double | Double |

**Автоупаковка происходит:**

* когда присваивают примитиву ссылку на класс обертку:

ДО Java 5:

//ручная упаковка или как это было ДО Java 5.

**public** **void** boxingBeforeJava5() {

Boolean booleanBox = **new** Boolean(true);

Integer intBox = **new** Integer(3);

// и так далее к другим типам

}

после Java 5:

//автоматическая упаковка или как это стало в Java 5.

**public** **void** boxingJava5() {

Boolean booleanBox = true;

Integer intBox = 3;

// и так далее к другим типам

}

* когда передают примитив в аргумент метода, который ожидает обертку:
* **public** **void** exampleOfAutoboxing() {
* **long** age = 3;
* setAge(age);
* }
* **public** **void** setAge(Long age) {
* **this**.age = age;

}

**Автораспаковка происходит:**

* когда присваиваем классу обертке примитивную переменную:
* //до Java 5:
* **int** intValue = **new** Integer(4).intValue();
* **double** doubleValue = **new** Double(2.3).doubleValue();
* **char** c = **new** Character((**char**) 3).charValue();
* **boolean** b = Boolean.TRUE.booleanValue();
* //и после JDK 5:
* **int** intValue = **new** Integer(4);
* **double** doubleValue = **new** Double(2.3);
* **char** c = **new** Character((**char**) 3);

**boolean** b = Boolean.TRUE;

* В случаях с арифметическими операциями. Они применяются только к примитивным типам, для этого нужно делать распаковку к примитиву.
* // До Java 5
* Integer integerBox1 = **new** Integer(1);
* Integer integerBox2 = **new** Integer(2);
* // для сравнения нужно было делать так:
* integerBox1.intValue() > integerBox2.intValue()
* //в Java 5

integerBox1 > integerBox2

* когда передают в обертку в метод, который принимает соответствующий примитив:
* **public** **void** exampleOfAutoboxing() {
* Long age = **new** Long(3);
* setAge(age);
* }
* **public** **void** setAge(**long** age) {
* **this**.age = age;

}

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

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

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

Пробежали по верхам, теперь обсудим более подробно.

**final переменные**

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

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

Пример с использованием final переменной для этих случаев:

**public** **class** FinalExample {

//статическая переменная final, которая сразу же инициализируется:

**final** **static** String FINAL\_EXAMPLE\_NAME = "I'm likely final one";

//final переменная, которая не инициализирована, но работать будет только если

//инициализировать это в конструкторе:

**final** **long** creationTime;

**public** FinalExample() {

**this**.creationTime = System.currentTimeMillis();

}

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

FinalExample finalExample = **new** FinalExample();

System.out.println(finalExample.creationTime);

// final поле FinalExample.FINAL\_EXAMPLE\_NAME не может быть заасайнено

// FinalExample.FINAL\_EXAMPLE\_NAME = "Not you're not!";

// final поле Config.creationTime не может быть заасайнено

// finalExample.creationTime = 1L;

}

}

**Можно ли считать Final переменную константой?**

Поскольку у нас не получится присвоить новое значение для final переменной, кажется, что это переменные константы. Но это только на первый взгляд. Если тип данных, на который ссылается переменная — immutable, то да, это константа. А если тип данных mutable, то есть изменяемый, при помощи методов и переменных можно будет изменить значение объекта, на который ссылается final переменная, и в таком случае назвать ее константой нельзя. Так вот, на примере видно, что часть финальных переменных действительно константы, а часть — нет, и их можно изменить.

**public** **class** FinalExample {

//неизменяемые финальные переменные:

**final** **static** String FINAL\_EXAMPLE\_NAME = "I'm likely final one";

**final** **static** Integer FINAL\_EXAMPLE\_COUNT = 10;

//изменяемые фильнаные переменные

**final** List<String> addresses = **new** ArrayList();

**final** StringBuilder finalStringBuilder = **new** StringBuilder("constant?");

}

**Local final переменные**

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

**public** **class** FinalExample {

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

// Вот так можно

**final** **int** minAgeForDriveCar = 18;

// а можно и так, в цикле foreach:

**for** (**final** String arg : args) {

System.out.println(arg);

}

}

}

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

// final local переиенная j не может быть назначена

**for** (**final** **int** i = 0; i < args.length; i ++) {

System.out.println(args[i]);

}

**Final класс**

Нельзя расширять класс, объявленный как final. Проще говоря, никакой класс не может наследоваться от данного. Прекрасным примером final класса в JDK является String. Первый шаг к созданию неизменяемого класса — пометить его как final, и тогда нельзя будет его расширить:

**public** **final** **class** FinalExample {

}

// Здесь будет ошибка компиляции

**class** WantsToInheritFinalClass **extends** FinalExample {

}

**Final методы**

Когда метод маркирован как final, его называют final метод (логично, правда?). Final метод нельзя переопределять у класса наследника. К слову, методы в классе Object — wait() и notify() — это final, поэтому у нас нет возможность их переопределять.

**public** **class** FinalExample {

**public** **final** String generateAddress() {

**return** "Some address";

}

}

**class** ChildOfFinalExample **extends** FinalExample {

// здесь будет ошибка компиляции

@Override

**public** String generateAddress() {

**return** "My OWN Address";

}

}

**Как и где использовать final в Java**

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

**23. Что такое mutable immutable?**

**Mutable**

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

**public** **class** MutableExample {

**private** String address;

**public** MutableExample(String address) {

**this**.address = address;

}

**public** String getAddress() {

**return** address;

}

// этот сеттер может изменить поле name

**public** **void** setAddress(String address) {

**this**.address = address;

}

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

MutableExample obj = **new** MutableExample("first address");

System.out.println(obj.getAddress());

// обновляем поле name, значит это mutable объект

obj.setAddress("Updated address");

System.out.println(obj.getAddress());

}

}

**Immutable**

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

// сделаем этот класс финальным, чтобы никто не мог его изменить

**public** **final** **class** ImmutableExample {

**private** String address;

ImmutableExample (String address) {

**this**.address = address;

}

**public** String getAddress() {

**return** address;

}

//удаляем сеттер

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

ImmutableExample obj = **new** ImmutableExample("old address");

System.out.println(obj.getAddress());

// Поэтому никак не изменить это поле, значит это immutable объект

// obj.setName("new address");

// System.out.println(obj.getName());

}

}

**24. Как написать immutable класс?**

После того, как выясните, что такое mutable и immutable объекты, следующий вопрос будет закономерный — как написать его? Чтоб написать immutable неизменяемый класс, нужно следовать простым пунктам:

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

Пример:

/\*\*

\* Пример по созданию immutable объекта.

\*/

**public** **final** **class** FinalClassExample {

**private** **final** **int** age;

**private** **final** String name;

**private** **final** HashMap<String, String> addresses;

**public** **int** getAge() {

**return** age;

}

**public** String getName() {

**return** name;

}

/\*\*

\* Клонируем объект перед тем, как вернуть его.

\*/

**public** HashMap<String, String> getAddresses() {

**return** (HashMap<String, String>) addresses.clone();

}

/\*\*

\* В конструкторе выполняем глубокое копирование для mutable объектов.

\*/

**public** FinalClassExample(**int** age, String name, HashMap<String, String> addresses) {

System.out.println("Выполняем глубокое копирование в конструкторе");

**this**.age = age;

**this**.name = name;

HashMap<String, String> temporaryMap = **new** HashMap<>();

String key;

Iterator<String> iterator = addresses.keySet().iterator();

**while** (iterator.hasNext()) {

key = iterator.next();

temporaryMap.put(key, addresses.get(key));

}

**this**.addresses = temporaryMap;

}

}

## Collections

### 25. Что имеется в виду под Collections в Java?

Collection — это фреймворк, который создан для сохранения и манипуляции объектами. Используется для выполнения следующих операций:

* поиск;
* сортировка;
* манипуляция;
* добавление;
* удаление.

Все классы и интерфейсы для Collection фреймворка находятся в java.util пакете.

### 26. Какие классы и интерфейсы доступны в Collection фреймворке?

Интерфейсы:

* Collection;
* List;
* Set;
* Map;
* Sorted Set;
* Sorted Map;
* Queue.

Классы:

* Lists:
  1. ArrayList;
  2. LinkedList;
  3. Vector(deprecated).
* Sets:
  1. HashSet;
  2. LinkedHashSet;
  3. TreeSet.
* Maps:
  1. HashMap
  2. TreeMap
  3. HashTable (deprecated)
  4. LinkedHashMap
* Queue
  1. Priority Queue.

### 27. Что подразумевается под sorted и ordered в коллекциях?

#### Ordered (упорядочивание):

Это означает, что элементы, которые хранятся в коллекции, основаны на значениях, добавленных в коллекцию. Таким образом, мы можем перебирать значения из коллекции в определенном порядке. Другими словами это значит, что у элементов коллекции есть свой специфический порядок, согласно которому они расположены. Для лучшего понимания, коллекция, которая не упорядочена (ordered), хранит элементы в произвольном порядке. Например, Set.

#### Sorted (отсортированный):

Это значит, что группа элементов отсортирована в коллекции на основе данных элемента коллекции. То есть не только коллекция упорядочена (ordered), но еще и порядок элементов зависит от их значений. Этот порядок может меняться, если отсортировать по другому значению элемента.

### 28. Какие есть коллекции с List интерфейсом? Как происходит работа с List?

Значения элементов в лист базируются на их индексе — они упорядочены по индексу. Повторения элементов разрешены (то есть можно добавить один и тот же объект в коллекцию несколько раз, и это будет нормально).

#### ArrayList:

Самая распространенная коллекция. По сути, это массив с динамически расширяемым размером. Работа по управлению размером массива лежит на коллекции. Для нас важно понять, что в большинстве случаев это то, что нам нужно использовать. Особенности:

* быстрый перебор и быстрый поиск по индексу;
* коллекция упорядочена по индексу, но не сортирована;
* реализует RandomAccess интерфейс;
* медленное добавление в середину списка.

Пример:

**public** **class** A {

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

ArrayList names = **new** ArrayList<>();

names.add("John");

names.add("John");

names.add("Roman");

names.add("Ivan");

}

}

>> output

[John, John, Roman, Ivan]

В выводе видно, что это повторяемые элементы. Они выведены в порядке, в котором их записали.

#### Linked List:

Это коллекция, в которой каждый элемент имеет ссылку на предыдущий и следующий элементы. По этим ссылкам можно переходить от одного элемента к другому. При добавлении элемента просто меняются ссылки на предыдущий и следующий элементы:

* элементы связаны друг с другом, то есть реализован двусвязный список;
* общая скорость работы заметно ниже, чем в ArrayList;
* отличный выбор для большого количества вставок и удалений в середину массива;
* реализует интерфейсы списков Queue и Deque, поэтому и имеет их методы для работы.

Пример:

LinkedList<String> linkedList = **new** LinkedList<>();

linkedList.add("One");

linkedList.add("Two");

linkedList.add("Three");

### 29. Расскажите о коллекции Map и ее реализациях?

Map — это коллекция ключ-значение (key-value). Есть уникальный ключ и значение, которое соответствует этому значению. Используется equals() и hashcode() методы для определения уникальности ключа.

#### HashMap:

* не отсортирован и не упорядочен;
* используют если не важны порядок и сортировка;
* поддерживает null ключ.

Пример:

**public** **class** CollectionExample {

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

HashMap positions = **new** HashMap<>();

positions.put("junior", "Ivan");

positions.put("middle", "Roman");

positions.put("senior", "Vasily");

positions.put("team lead", "Anton");

positions.put("arthitect", "Andrew");

positions.put("senior", "John");

System.out.println(positions);

}

}

// вывод в консоль

// {junior=Ivan, middle=Roman, senior=John, team lead=Anton, arthitect=Andrew}

Ключи всегда уникальны, поэтому записан только один senior.

#### LinkedHashMap:

* поддерживает порядок вставки;
* медленнее, чем HashMap;
* ожидается, что итерация быстрее, чем в HashMap.

Пример:

**public** **class** CollectionExample {

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

LinkedHashMap<String, String> positions = **new** LinkedHashMap<>();

positions.put("junior", "Ivan");

positions.put("middle", "Roman");

positions.put("senior", "Vasily");

positions.put("team lead", "Anton");

positions.put("arthitect", "Andrew");

positions.put("senior", "John");

System.out.println(positions);

}

}

// вывод в консоль

// {junior=Ivan, middle=Roman, senior=John, team lead=Anton, arthitect=Andrew}

#### TreeMap:

Реализация мапы, которая сохраняет записи отсортированными в соответствии с естественным порядком их ключей или, что еще лучше, с использованием компаратора, если он предоставляется в конструкторе при создании мапы. Пример:

1. Без компаратора
2. **public** **class** CollectionExample {
3. **public** **static** **void** main(String[] args) {
4. TreeMap<Integer, String> positions = **new** TreeMap<>();
5. positions.put(1, "Ivan");
6. positions.put(3, "Roman");
7. positions.put(2, "Vasily");
8. positions.put(10, "Anton");
9. positions.put(7, "Andrew");
10. positions.put(1, "John");
11. System.out.println(positions);
12. }
13. }
14. // вывод в консоль

// {1=John, 2=Vasily, 3=Roman, 7=Andrew, 10=Anton}

1. С компаратором
2. **public** **class** CollectionExample {
3. **public** **static** **void** main(String[] args) {
4. //используем реализацию Strategy Pattern'a и добавим компаратор:
5. TreeMap<Integer, String> positions = **new** TreeMap<>(Comparator.reverseOrder());
6. positions.put(1, "Ivan");
7. positions.put(3, "Roman");
8. positions.put(2, "Vasily");
9. positions.put(10, "Anton");
10. positions.put(7, "Andrew");
11. positions.put(1, "John");
12. System.out.println(positions);
13. }
14. }
15. // вывод в консоль

// {10=Anton, 7=Andrew, 3=Roman, 2=Vasily, 1=John}

Видим, что стандартно реализована сортировка в порядке возрастания, но это можно изменить, добавив компаратор в конструктор. Хорошо описана TreeMap [здесь](https://www.codeflow.site/ru/article/java-treemap).

### 30. Расскажите о коллекции Set и ее реализациях?

Set — это множество уникальных элементов, и это ее главная особенность. То есть Set не допускает повторения одних и тех же элементов. Здесь важно, чтобы у объектов, которые добавляются, был реализован метод equals.

#### HashSet:

* не отсортирован и не упорядочен. Под капотом там HashMap с заглушкой для значения. Посмотрите сами ;)
* использует hashCode для добавления объектов;
* стоит использовать, когда нужно иметь уникальные объекты и их порядок не важен.

Пример:

**public** **class** CollectionExample {

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

HashSet<String> positions = **new** HashSet<>();

positions.add("junior");

positions.add("junior");

positions.add("middle");

positions.add("senior");

positions.add("team lead");

positions.add("architect");

System.out.println(positions);

}

}

// вывод в консоль

// [senior, middle, team lead, architect, junior]

Здесь видно, что элемент “junior”, который дважды добавлен, присутствует только в единичном экземпляре. И порядок не такой же, как при добавлении.

#### LinkedHashSet:

* упорядоченная версия HashSet;
* поддерживает двусвязный список для всех элементов;
* использовать его, когда требуется упорядоченность при итерации.

Пример:

**public** **class** CollectionExample {

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

LinkedHashSet<String> positions = **new** LinkedHashSet<>();

positions.add("junior");

positions.add("junior");

positions.add("middle");

positions.add("senior");

positions.add("team lead");

positions.add("architect");

System.out.println(positions);

}

}

// вывод в консоль

// [senior, middle, team lead, architect, junior]

#### TreeSet:

* одна из двух сортированных коллекций;
* использует структуру [красно-черного дерева](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) и гарантирует, что элементы будут в возрастающем порядке;
* под капотом это TreeMap с заглушкой на значениях. А элементами TreeSet являются ключи к TreeMap (также посмотрите ;)).

Пример:

**public** **class** CollectionExample {

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

TreeSet<String> positions = **new** TreeSet<>();

positions.add("junior");

positions.add("junior");

positions.add("middle");

positions.add("senior");

positions.add("team lead");

positions.add("architect");

System.out.println(positions);

}

}

// вывод в консоль

// [architect, junior, middle, senior, team lead]

## Exceptions

### 31. Что такое Exception?

**Exception** — это проблема, которая может возникнуть в runtime. Это **исключительная** ситуация, которая возникает из-за каких-то причин. Диаграмма наследования исключений выглядит так

На диаграмме видно, что в целом все исключения делятся на две группы — **exceptions** и **error.** **Error** — используются JVM для отображения ошибок, после которых работа приложения уже не имеет смысла. Например StackOverFlowError, которая говорит, что стек заполнен и программа уже не может работать. **Exception** — исключения, которые генерируются программно в коде. Есть разные исключения, проверяемые и непроверяемые, но главное, что они есть, и их можно перехватить и продолжить работу приложения. Exceptions, в свою очередь, еще делятся на тех, кто наследуется от RuntimeException и других наследников Exception.

### 32. Как JVM обрабатывает исключения?

Как это работает? Как только где-то создается исключение, runtime создает Exception Object (обозначим как ExcObj). В нем хранится вся необходимая для работы информация — само исключение, которое вызывалось и место, где это произошло. Создание ExcObj и передача в runtime есть ничто иное как “выбрасывание исключения”. ExcObj содержит методы, по которым можно дойти до место создания исключения. Это множество методов называется Call Stack. Далее, runtime система ищет метод в Call Stack, который сможет обработать наше исключение. Если он таки находит соответствующий обработчик, то есть тип исключения совпадает с типом в обработчике, все хорошо. Если не находит, то runtime передает всё в default exception handler, который подготавливает ответ и завершает работу. Как это выглядит наглядно:

/\*\*

\* Пример, в котором показываются две опции — когда находится обработчик для исключения и когда нет.

\*/

**class** ThrowerExceptionExample {

**public** **static** **void** main(String[] args) **throws** IllegalAccessException {

ThrowerExceptionExample example = **new** ThrowerExceptionExample();

System.out.println(example.populateString());

}

/\*\*

\* Здесь происходит перехват одного из возможных исключений — {@link IOException}.

\* А вот второй будет пробрасываться дальше вверх по вызову.

\*/

**private** String populateString() **throws** IllegalAccessException {

**try** {

**return** randomThrower();

} **catch** (IOException e) {

**return** "Caught IOException";

}

}

/\*\*

\* Здесь две опции: или бросается {@link IOException} или {@link IllegalAccessException}.

\* Выбирается случайным образом.

\*/

**private** String randomThrower() **throws** IOException, IllegalAccessException {

**if** (**new** Random().nextBoolean()) {

**throw** **new** IOException();

} **else** {

**throw** **new** IllegalAccessException();

}

}

}

В нашем случае CallStack схематично будет иметь вид:

randomThrower() => populateString() => main(String[] args)

Есть две опции: случайным образом будет выброшено одно или другое исключение. Для IOException все ок, если будет сгенерировано оно, то результатом работы будет: "Caught IOException". А вот если будет второе исключение, обработчика на которого нет, программа будет остановлена с таким выводом:

Exception in thread "main" java.lang.IllegalAccessException

at ThrowerExceptionExample.randomThrower(CollectionExample.java:38)

at ThrowerExceptionExample.populateString(CollectionExample.java:24)

at ThrowerExceptionExample.main(CollectionExample.java:15)

### 33. Как программистам обрабатывать исключения?

В вопросах выше уже использовались те или иные ключевые слова для работы с исключениями, теперь нужно поговорить о них более подробно. Какие есть ключевые слова?

* try
* catch
* throw
* throws
* finally

Важно отметить, что catch, throw и throws можно использовать только с java.lang.Throwable. С другими типами работать это не будет. Сейчас обсудим именно try, catch и finally.

* try-catch-finally — это конструкция, при помощи которой можно правильным образом перехватить и обработать исключение.
* try — может быть только один раз, в нем и происходит логика;
* catch — блок, который принимает какой-то тип исключения, их может быть множество. Например, в блоке try будет генерироваться несколько исключений, которые никак друг с другом не связаны;
* finally — “наконец-то” и этот блок. Это блок, который выполнится в любом случае, независимо от того, что делается в try, catch.

Вот как это выглядит:

**try** {

// сюда передают тот код, который может вызвать исключение.

} **catch** (IOException e) {

// первый catch блок, который принимает IOException и все его подтипы(потомки).

// Например, нет файла при чтении, выпадает FileNotFoundException, и мы уже соответствующе

// обрабатываем это.

} **catch** (IllegalAccessException e) {

// если нужно, можно добавить больше одного catch блока, в этом нет проблем.

} **catch** (OneException | TwoException e) {

// можно даже объединять несколько в один блок

} **catch** (Throwable t) {

// а можно сказать, что мы принимаем ВСЁ))))

} **finally** {

// этот блок может быть, а может и не быть.

// и он точно выполнится.

}

Внимательно вчитайтесь в описание примера и будет всё ясно)

### 34. throw и throws в Java

#### throw

throw используют в случае, когда нужно явно создать новое исключение. Применяют его для создания и выбрасывания пользовательских исключений. Например, исключения, связанные с валидацией. Обычно для валидации наследуются от RuntimeException. Пример:

// пример пробрасывания исключения

**throw** **new** RuntimeException("because I can :D");

Важно, что использовать эту конструкцию можно только тем, что наследуется от Throwable. То есть, нельзя сказать так:

**throw** **new** String("как тебе такое, Илон Маск?");

Далее, работа потока обрывается и начинается поиск обработчика, который смог бы обработать его. Когда не находит, идет к методу, который вызвал его, и так поиск будет идти наверх по строке вызовов пока либо не найдет соответствующий обработчик, либо оставит работу приложения. Смотрим:

// Пример, который демонстрирует работу throw

**class** ThrowExample {

**void** willThrow() **throws** IOException {

**throw** **new** IOException("Because I Can!");

}

**void** doSomething() {

System.out.println("Doing something");

**try** {

willThrow();

} **catch** (IOException e) {

System.out.println("IOException was successfully handled.");

}

}

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

ThrowExample throwExample = **new** ThrowExample();

throwExample.doSomething();

}

}

Если запустить программу, получим такой результат:

Doing something

IOException was successfully handled.

#### throws

throws — механизм, при помощи которого метод может выбрасывать одно или более исключений. Добавляются они через запятую. Смотрим как это легко и просто:

**private** Object willThrow() **throws** RuntimeException, IOException, FileNotFoundException

Причем важно отметить, что могут быть как проверяемые так и непроверяемые исключения. Разумеется, что непроверяемые исключения можно и не добавлять в throws, но правила хорошего тона говорят об обратном. Если это проверяемые, то используя метод, который их генерирует, нужно как-то его обработать. Есть два варианта:

1. Написать try-catch с соответствующим и выше по наследованию исключением.
2. Использовать throws точно так же, чтобы эта проблема была уже у кого-то другого :D

### 35. Checked и Unchecked исключения в Java

B Java есть два типа исключений — checked и unchecked.

#### Checked исключения:

Это исключения, которые проверяются во время компиляции. Если какой-то код в методе во время исключения выдает checked исключение, метод обязан либо обработать его при помощи try-catch, либо пробросить его дальше На примере, который считывает картинку из пути "/users/romankh3/image.png", обновляет ее каким-то образом(для нас это не важно) и сохраняет ее обратно.

**class** CheckedImageExample {

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

File imageFile = **new** File("/users/romankh3/image.png");

BufferedImage image = ImageIO.read(imageFile);

updateAndSaveImage(image, imageFile);

}

**private** **static** **void** updateAndSaveImage(BufferedImage image, File imageFile) {

ImageIO.write(image, "png", imageFile);

}

}

Такой код компилироваться не будет, так как статические методы ImageIO.read() и ImageIO.write() выбрасывают исключение IOException, которое является checked (проверяемым) и должно соответственно быть обработанным. Здесь две опции, которые мы уже обсудили выше: или использовать try-catch, или пробросить дальше. Для лучшего усвоения сделаем и так, и эдак. То есть в методе updateAndSave просто пробросим, а уже в главном методе воспользуемся try-catch:

**class** CheckedImageExample {

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

File imageFile = **new** File("/users/romankh3/image.png");

**try** {

BufferedImage image = ImageIO.read(imageFile);

updateAndSaveImage(image, imageFile);

} **catch** (IOException e) {

e.printStackTrace();

}

}

**private** **static** **void** updateAndSaveImage(BufferedImage image, File imageFile) **throws** IOException {

ImageIO.write(image, "png", imageFile);

}

}

#### Unchecked исключения:

Это те исключения, которые на этапе компиляции не проверяются. То есть метод может генерировать RuntimeException, а компилятор не напомнит каким-то образом это обработать. Как показано ниже, у нас все, кто наследуются от RuntimeException и Error являются непроверяемыми. Рассмотрим следующую Java-программу. Код прекрасно компилируется, но при запуске выдает исключение ArrayIndexOutOfBoundsException. Компилятор позволяет его компилировать, потому что ArrayIndexOutOfBoundsException является непроверенным исключением. Обычная ситуация с массивом, которая может быть:

**class** CheckedImageExample {

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

**int**[] array = **new** **int**[3];

array[5] = 12;

}

}

Результатом будет:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at main(CheckedImageExample.java:12)

### 36. Что такое try-with-resources?

Это механизм, при помощи которого нужно правильно закрывать все ресурсы. Для начала, что такое ресурс... Ресурс — это объект, после работы с которым нужно закрыть его, то есть вызвать метод close(). Ресурсом называются все объекты, которые реализуют интерфейс AutoClosable, который, в свою очередь реализует интерфейс Closeable. Для нас важно понять, что все InputStream, OutpuStream являются ресурсами и их нужно правильно и успешно высвобождать. Вот как раз для этого и нужно нам использовать try-with-resource конструкцию. Вот как она выглядит:

**private** **void** unzipFile(File zipFile) **throws** IOException {

**try**(ZipInputStream zipOutputStream = **new** ZipInputStream(**new** FileInputStream(zipFile))) {

ZipEntry zipEntry = zipOutputStream.getNextEntry();

**while** (zipEntry != **null**) {

}

}

}

**private** **void** saveZipEntry(ZipEntry zipEntry) {

// логика сохранения

}

Вот этом примере ресурс — это ZipInputStream, после работы с которым нужно будет закрыть его. И чтоб не думать о том, что нужно вызвать метод close(), мы просто определяем эту переменную в блоке try, как показано в примере и в рамках этого блока выполняем все необходимое. Что делает пример? Он разархивирует zip архив. Для этого нужно воспользоваться InputStream’ом. Определять можно больше одной переменной, разделяют их точкой с запятой. А в чем проблема? Но ведь можно использовать finally блок, — возможно, скажете вы.

**Multithreading**

**37. Как создать в Java новый тред (поток)?**

Так или иначе создание происходит через использование класса Thread. Но здесь могут быть варианты…

1. Наследуемся от java.lang.Thread
2. Имплементируем интерфейс java.lang.Runnable, объект которого принимает к себе к конструктор Thread класс

Поговорим о каждом из них.

**Наследуемся от Thread класса**

Чтобы это заработало, в нашем классе наследуемся от java.lang.Thread. В нем есть метом run(), он как раз нам и нужен. Вся жизнь и логика нового потока будет в этом методе. Это своего рода main метод для нового потока. После этого останется только создать объект нашего класса и выполнить метод start(), который создаст новый поток и запустит записанную в нем логику. Смотрим:

/\*\*

\* Пример того, как создавать треды путем наследования {@link Thread} класса.

\*/

**class** ThreadInheritance **extends** Thread {

@Override

**public** **void** run() {

System.out.println(Thread.currentThread().getName());

}

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

ThreadInheritance threadInheritance1 = **new** ThreadInheritance();

ThreadInheritance threadInheritance2 = **new** ThreadInheritance();

ThreadInheritance threadInheritance3 = **new** ThreadInheritance();

threadInheritance1.start();

threadInheritance2.start();

threadInheritance3.start();

}

}

Вывод в консоль будет такой:

Thread-1

Thread-0

Thread-2

То есть даже здесь мы видим, что выполняются потоки не по очереди, а как JVM рассудила)

**Реализуем интерфейс Runnable**

Если вы противник наследований и/или уже наследуете какой-то из других классов, можно воспользоваться интерфейсом java.lang.Runnable. Здесь мы в нашем классе реализуем этот интерфейс и имплементируем метод run(), как это было и в том примере. Только нужно будет еще создать объекты Thread. Казалось бы, больше строк и это хуже. Но мы то знаем как пагубно наследование и что его лучше избегать всеми способами ;) Смотрим:

/\*\*

\* Пример того, как создавать треды из интерфейса {@link Runnable}.

\* Здесь проще простого - реализуем этот интерфейс и потом передаем в конструктор

\* экземпляр реализуемого объекта.

\*/

**class** ThreadInheritance **implements** Runnable {

@Override

**public** **void** run() {

System.out.println(Thread.currentThread().getName());

}

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

ThreadInheritance runnable1 = **new** ThreadInheritance();

ThreadInheritance runnable2 = **new** ThreadInheritance();

ThreadInheritance runnable3 = **new** ThreadInheritance();

Thread threadRunnable1 = **new** Thread(runnable1);

Thread threadRunnable2 = **new** Thread(runnable2);

Thread threadRunnable3 = **new** Thread(runnable3);

threadRunnable1.start();

threadRunnable2.start();

threadRunnable3.start();

}

}

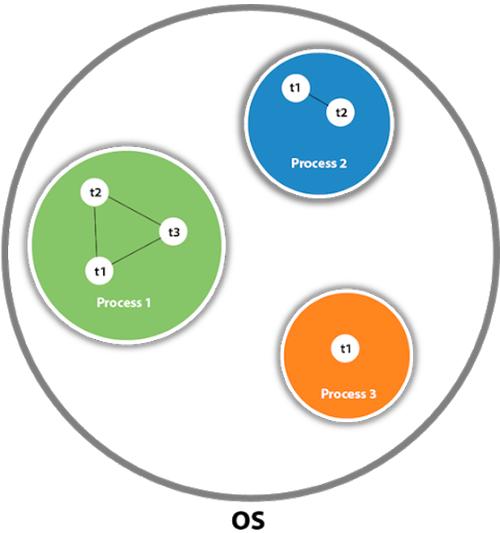
И результат выполнения:

Thread-0

Thread-1

Thread-2

**38. Какая разница между процессом и потоком?**

Существуют следующие различия между процессом и потоком:

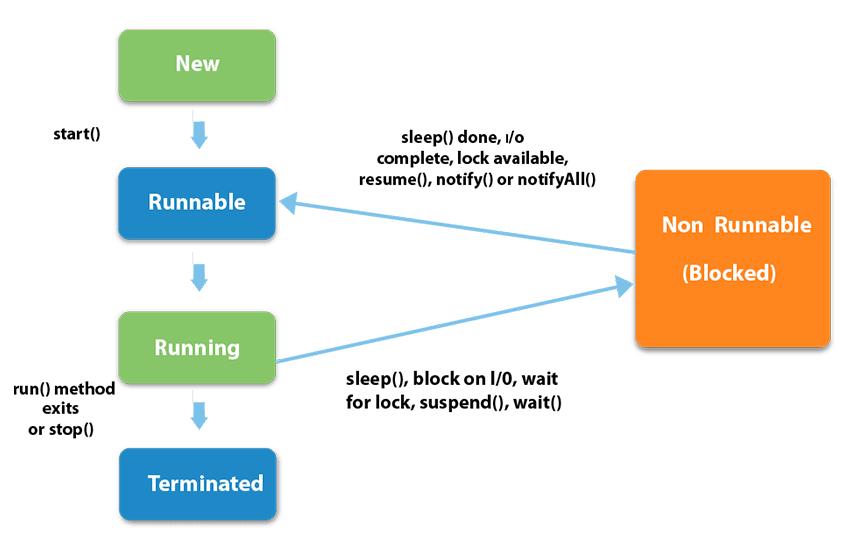
1. Программа в исполнении называется процессом, тогда как Поток является подмножеством процесса.
2. Процессы независимы, тогда как потоки являются подмножеством процесса.
3. Процессы имеют различное адресное пространство в памяти, в то время как потоки содержат общее адресное пространство.
4. Переключение контекста происходит быстрее между потоками по сравнению с процессами.
5. Межпроцессное взаимодействие медленнее и дороже, чем межпотоковое взаимодействие.
6. Любые изменения в родительском процессе не влияют на дочерний процесс, тогда как изменения в родительском потоке могут влиять на дочерний поток.

**39. Какие преимущества есть у многопоточности?**



1. Многопоточность позволяет приложению / программе всегда реагировать на ввод, даже если она уже выполняется с некоторыми фоновыми задачами;
2. Многопоточность позволяет быстрее выполнять задачи, поскольку потоки выполняются независимо;
3. Многопоточность обеспечивает лучшее использование кэш-памяти, поскольку потоки разделяют общие ресурсы памяти;
4. Многопоточность уменьшает количество требуемого сервера, поскольку один сервер может одновременно выполнять несколько потоков.

**40. Каковы состояния в жизненном цикле потока?**



1. **New:** В этом состоянии объект класса Thread создается с использованием оператора new, но поток не существует. Поток не запускается, пока мы не вызовем метод start().
2. **Runnable:** В этом состоянии поток готов к запуску после вызова метода start(). Однако он еще не выбран планировщиком потока.
3. **Running:** В этом состоянии планировщик потока выбирает поток из состояния готовности, и тот работает.
4. **Waiting/Blocked:** в этом состоянии поток не работает, но все еще жив или ожидает завершения другого потока.
5. **Dead/Terminated:** при выходе из метода run() поток находится в завершенном или мертвом состоянии.

**41. Можно ли запустить тред дважды?**

Нет, мы не можем перезапустить поток, так как после запуска и выполнения потока он переходит в состояние Dead. Поэтому, если мы попытаемся запустить поток дважды, он выдаст исключение runtimeException "*java.lang.IllegalThreadStateException*". Смотрим:

**class** DoubleStartThreadExample **extends** Thread {

/\*\*

\* Имитируем работу треда

\*/

**public** **void** run() {

// что-то происходит. Для нас не существенно на этом этапе

}

/\*\*

\* Запускаем тред дважды

\*/

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

DoubleStartThreadExample doubleStartThreadExample = **new** DoubleStartThreadExample();

doubleStartThreadExample.start();

doubleStartThreadExample.start();

}

}

Как только работа дойдет до выполнения второго старта одного и того же треда - тогда и будет исключение. Попробуйте сами ;) лучше один раз увидеть, чем сто раз услышать.

**42. Что если вызвать напрямую метод run(), не вызывая метод start()?**

Да, вызвать метод run() конечно можно, но это никак не создаст новый поток и не выполнит его как отдельный. В этом случае, это простой объект, который вызывает простой метод. Если мы говорим о методе start(), то там другое дело. Запуская этот метод, runtime запускает новый потом и он уже, в свою очередь, дергает наш метод ;) Не верите — вот, попробуйте:

**class** ThreadCallRunExample **extends** Thread {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

System.out.print(i);

}

}

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

ThreadCallRunExample runExample1 = **new** ThreadCallRunExample();

ThreadCallRunExample runExample2 = **new** ThreadCallRunExample();

// просто будут вызваны в потоке main два метода, один за другим.

runExample1.run();

runExample2.run();

}

}

И вывод в консоль будет такой:

0123401234

Видно, что никакой нити не было создано. Все сработало как обычный класс. Вначале отработал метод первого класса, затем второй.

**43. Что такое daemon тред?**

**Daemon thread** (далее — демон-тред) — это тред, который выполняет задачи в фоне по отношению к другому потоку. То есть, его работа заключается в том, чтоб выполнять задачи вспомогательные, которые нужно делать только в привязке другому (основному) потоку. Есть много потоков демонов, работающих автоматически, например Garbage Collector, finalizer и т. д.

**Почему Java закрывает демон-поток?**

Единственная цель потока демона состоит в том, что он предоставляет сервисы потоку пользователя для фоновой задачи поддержки. Поэтому если основной поток завершился, то runtime закрывает автоматически и все его демон-потоки.

**Методы для работы в Thread классе**

Класс java.lang.Thread предоставляет два метода для работы с демоном-потоком:

1. public void setDaemon(boolean status) — указывает, что это будет демон-поток. По умолчанию стоит false, что значит, что будут создаваться не демон-потоки, если не указать это отдельно.
2. public boolean isDaemon() — по сути это геттер для переменной daemon, который мы устанавливаем предыдущим методом.

Пример:

**class** DaemonThreadExample **extends** Thread {

**public** **void** run() {

// Проверяет, демон ли этот поток или нет

**if** (Thread.currentThread().isDaemon()) {

System.out.println("daemon thread");

} **else** {

System.out.println("user thread");

}

}

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

DaemonThreadExample thread1 = **new** DaemonThreadExample();

DaemonThreadExample thread2 = **new** DaemonThreadExample();

DaemonThreadExample thread3 = **new** DaemonThreadExample();

// теперь thread1 - поток-демон.

thread1.setDaemon(true);

System.out.println("демон?.. " + thread1.isDaemon());

System.out.println("демон?.. " + thread2.isDaemon());

System.out.println("демон?.. " + thread3.isDaemon());

thread1.start();

thread2.start();

thread3.start();

}

}

Вывод в консоль:

демон?.. true

демон?.. false

демон?.. false

daemon thread

user thread

user thread

Из вывода мы видим, что внутри самого потока при помощи статического currentThread() метода можно узнать какой это поток с одной стороны, с другой стороны, если у нас есть ссылка на объект этого потока, мы можем узнать и непосредственно у него. Это дает ту необходимую гибкость в настройке.

**44. Можно ли сделать поток демоном уже после его создания?**

Нет. Если вы сделаете это, он выдаст исключение IllegalThreadStateException. Следовательно, мы можем создать поток демона только до его запуска. Пример:

**class** SetDaemonAfterStartExample **extends** Thread {

**public** **void** run() {

System.out.println("Working...");

}

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

SetDaemonAfterStartExample afterStartExample = **new** SetDaemonAfterStartExample();

afterStartExample.start();

// здесь будет выброшено исключение

afterStartExample.setDaemon(true);

}

}

Вывод в консоль:

Working...

Exception in thread "main" java.lang.IllegalThreadStateException

at java.lang.Thread.setDaemon(Thread.java:1359)

at SetDaemonAfterStartExample.main(SetDaemonAfterStartExample.java:14)

**45. Что такое shutdownhook?**

**Shutdownhook** — это поток, который неявно вызывается до завершения работы JVM(виртуальная машина Java). Таким образом, мы можем использовать его для очистки ресурса или сохранения состояния, когда виртуальная машина Java выключается нормально или внезапно. Мы можем добавить shutdown hook, используя следующий метод:

Runtime.getRuntime().addShutdownHook(**new** ShutdownHookThreadExample());

Как показано в примере:

/\*\*

\* Программа, которая показывает как запустить shutdown hook тред,

\* который выполнится аккурат до окончания работы JVM

\*/

**class** ShutdownHookThreadExample **extends** Thread {

**public** **void** run() {

System.out.println("shutdown hook задачу выполнил");

}

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

Runtime.getRuntime().addShutdownHook(**new** ShutdownHookThreadExample());

System.out.println("Теперь программа засыпает, нажмите ctrl+c чтоб завершить ее.");

**try** {

Thread.sleep(60000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

Вывод в консоль:

Теперь программа засыпает, нажмите ctrl+c чтоб завершить ее.

shutdown hook задачу выполнил

**46. Что такое синхронизация (synchronization)?**

Синхронизация (Synchronization) в Java — это возможность контролировать доступ нескольких потоков к любому общему ресурсу. Когда несколько потоков пытаются выполнить одну и ту же задачу, существует вероятность ошибочного результата, поэтому для устранения этой проблемы Java использует синхронизацию, благодаря которой будет только один тред сможет работать в один момент. Синхронизация может быть достигнута тремя способами:

* Синхронизируя метод
* Синхронизируя определенный блок
* Статической синхронизацией

**Синхронизация метода**

Синхронизированный метод используется для блокировки объекта для любого общего ресурса. Когда поток вызывает синхронизированный метод, он автоматически получает блокировку для этого объекта и снимает ее, когда поток завершает свою задачу. Чтоб заработало, нужно добавить ключевое слово synchronized. На примере увидим, как это работает:

/\*\*

\* Пример, где мы синхронизируем метод. То есть добавляем ему слово synchronized.

\* Есть два писателя, которые хотят использовать один принтер. Они подготовили свои поэмы

\* И конечно же не хотят, чтоб их поэмы перемешались, а хотят, чтоб работа была сделана по \* \* \* очереди для каждого из них

\*/

**class** Printer {

**synchronized** **void** print(List<String> wordsToPrint) {

wordsToPrint.forEach(System.out::print);

System.out.println();

}

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

// один объект для двух тредов

Printer printer = **new** Printer();

// создаем два треда

Writer1 writer1 = **new** Writer1(printer);

Writer2 writer2 = **new** Writer2(printer);

// запускаем их

writer1.start();

writer2.start();

}

}

/\*\*

\* Писатель номер 1, который пишет свою поэму.

\*/

**class** Writer1 **extends** Thread {

Printer printer;

Writer1(Printer printer) {

**this**.printer = printer;

}

**public** **void** run() {

List<string> poem = Arrays.asList("Я ", **this**.getName(), " Пишу", " Письмо");

printer.print(poem);

}

}

/\*\*

\* Писатель номер 2, который пишет свою поэму.

\*/

**class** Writer2 **extends** Thread {

Printer printer;

Writer2(Printer printer) {

**this**.printer = printer;

}

**public** **void** run() {

List<String> poem = Arrays.asList("Не Я ", **this**.getName(), " Не пишу", " Не Письмо");

printer.print(poem);

}

}

И вывод в консоль:

Я Thread-0 Пишу Письмо

Не Я Thread-1 Не пишу Не Письмо

**Блок синхронизации**

Синхронизированный блок может быть использован для выполнения синхронизации на любом конкретном ресурсе метода. Допустим, что в большом методе(да да, такие писать нельзя, но иногда бывает) нужно синхронизировать только небольшую часть, по каким-то причинам. Если вы поместите все коды метода в синхронизированный блок, он будет работать так же, как синхронизированный метод. Синтаксис выглядит так:

**synchronized** (“объект для блокировки”) {

// сам код, который нужно защитить

}

Для того, чтоб не повторять пример предыдущий, создадим треды через анонимные классы - то есть сразу реализуя Runnable интерфейс.

/\*\*

\* Вот как добавляется блок синхронизации.

\* Внутри нужно указать у кого будет взят мьютекс для блокировки.

\*/

**class** Printer {

**void** print(List<String> wordsToPrint) {

**synchronized** (**this**) {

wordsToPrint.forEach(System.out::print);

}

System.out.println();

}

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

// один объект для двух тредов

Printer printer = **new** Printer();

// создаем два треда

Thread writer1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

List<String> poem = Arrays.asList("Я ", "Writer1", " Пишу", " Письмо");

printer.print(poem);

}

});

Thread writer2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

List<String> poem = Arrays.asList("Не Я ", "Writer2", " Не пишу", " Не Письмо");

printer.print(poem);

}

});

// запускаем их

writer1.start();

writer2.start();

}

}

}

и вывод в консоль

Я Writer1 Пишу Письмо

Не Я Writer2 Не пишу Не Письмо

**Статическая синхронизация**

Если сделать статический метод синхронизированным, то блокировка будет на классе, а не на объекте. В этом примере мы применяем ключевое слово synchronized к статическому методу для выполнения статической синхронизации:

/\*\*

\* Вот как добавляется блок синхронизации.

\* Внутри нужно указать у кого будет взят мьютекс для блокировки.

\*/

**class** Printer {

**static** **synchronized** **void** print(List<String> wordsToPrint) {

wordsToPrint.forEach(System.out::print);

System.out.println();

}

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

// создаем два треда

Thread writer1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

List<String> poem = Arrays.asList("Я ", "Writer1", " Пишу", " Письмо");

Printer.print(poem);

}

});

Thread writer2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

List<String> poem = Arrays.asList("Не Я ", "Writer2", " Не пишу", " Не Письмо");

Printer.print(poem);

}

});

// запускаем их

writer1.start();

writer2.start();

}

}

и вывод в консоль:

Не Я Writer2 Не пишу Не Письмо

Я Writer1 Пишу Письмо

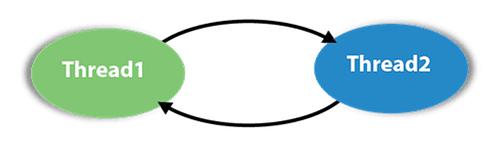
**47. Что такое volatile переменная?**

Ключевое слово volatile используется в многопоточном программировании для обеспечения безопасности потока, поскольку модификация одной изменяемой переменной видна всем другим потокам, поэтому одна переменная может использоваться одним потоком за раз. При помощи ключевого слова volatile можно гарантировать, что переменная будет потокобезопасна и будет храниться в общей памяти, и потоки не будут ее брать себе в свой кеш. Как это выглядит?

**private** **volatile** AtomicInteger count;

Просто добавляем к переменной volatile. Но это не говорит о полной потокобезопасности… Ведь операции могут быть не атомарны над переменной. Но можно использовать Atomic классы, которые делают операцию атомарно, то есть за одно выполнение процессором. Таких классов много можно найти в пакете java.util.concurrent.atomic.

**48. Что такое deadlock**

Deadlock в Java является частью многопоточности. Взаимная блокировка может возникнуть в ситуации, когда поток ожидает блокировки объекта, полученной другим потоком, а второй поток ожидает блокировки объекта, полученной первым потоком. Таким образом эти два потока ждут друг друга и не будут дальше выполнять свой код. Рассмотрим Пример, в котором есть класс имплементирующий Runnable. Принимает в конструкторе он два ресурса. Внутри метода run() он по-очереди берет блокировку для них, так вот если создать два объекта этого класса, а ресурсы передать в разном порядке, то легко можно нарваться на блокировку:

**class** DeadLock {

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

**final** Integer r1 = 10;

**final** Integer r2 = 15;

DeadlockThread threadR1R2 = **new** DeadlockThread(r1, r2);

DeadlockThread threadR2R1 = **new** DeadlockThread(r2, r1);

**new** Thread(threadR1R2).start();

**new** Thread(threadR2R1).start();

}

}

/\*\*

\* Класс, который принимает два ресурса.

\*/

**class** DeadlockThread **implements** Runnable {

**private** **final** Integer r1;

**private** **final** Integer r2;

**public** DeadlockThread(Integer r1, Integer r2) {

**this**.r1 = r1;

**this**.r2 = r2;

}

@Override

**public** **void** run() {

**synchronized** (r1) {

System.out.println(Thread.currentThread().getName() + " захватил ресурс: " + r1);

**try** {

Thread.sleep(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**synchronized** (r2) {

System.out.println(Thread.currentThread().getName() + " захватил ресурс: " + r2);

}

}

}

}

Вывод в консоль:

Первый тред захватил первый ресурс

Второй тред захватывает второй ресурс

**49. Как избежать deadlock?**

Исходя из того, что мы знаем как дедлок возникает, то можно сделать некоторые выводы…

* Как показано в примере выше, дедлок был из-за того, что была вложенность блокировок. То есть внутри одной блокировки находится еще одна или более. Избежать это можно следующим образом - вместо вложенности нужно добавить новую абстракцию поверх и дать блокировку на более высокий уровень, а вложенные блокировки убрать.
* Чем больше блокировок, тем больше шансов, что будет дедлок. Поэтому каждый раз добавляя блокировку нужно думать, а точно ли она нужна и можно ли избежать добавление новой.
* Использования Thread.join(). Дедлок можно сделать также при ожидании одного треда другим. Чтобы избежать этой проблемы, можно подумать над тем, чтобы выставлять ограниченное время на join() метод.
* Если у нас один поток - дедлока не будет ;)

**50. Что такое состояние гонки?**

Если в реальных гонках выступают машины, то в гонках терминологии многопоточности в гонках выступают треды. Но почему? Есть же два треда, которые работают и которые могут иметь доступ к одному и тому же объекту. И они могут попытаться обновить состояние в одно и тоже время. Пока все ясно, да? Так работа тредов происходит или реально параллельно(если есть больше одного ядра в процессоре) или условно параллельно, когда процессор выделяет по небольшому промежутку времени. И управлять этими процессами мы не можем, поэтому мы не можем гарантировать, что когда один тред прочитает данные из объекта, он успеет их изменить ДО того, как это сделает какой-то другой тред. Такие проблемы бывают, когда проходит такая комбинация “проверь-и-действуй”. Что это значит? Например у нас есть if выражение, в теле которого изменяется само условие, то есть:

**int** z = 0;

// проверь

**if** (z < 5) {

//действуй

z = z + 5;

}

Так вот может быть ситуация, когда два треда одновременно зайдут в этот блок кода в момент, когда z еще равно нулю и в двоем изменят это значение. И в итоге мы получим не ожидаемое значение 5, а уже 10. Как это избежать? Нужно поставить блокировку до начала выполнения и после. То есть, чтобы первый тред зашел в блок if, выполнил все действия, изменил z и уже потом дал возможность сделать это следующему треду. А вот уже следующий тред не зайдет в блок if, так как z уже будет равно 5:

// получить блокировку для z

**if** (z < 5) {

z = z + 5;

}

// выпустить из блокировки z

===================================================