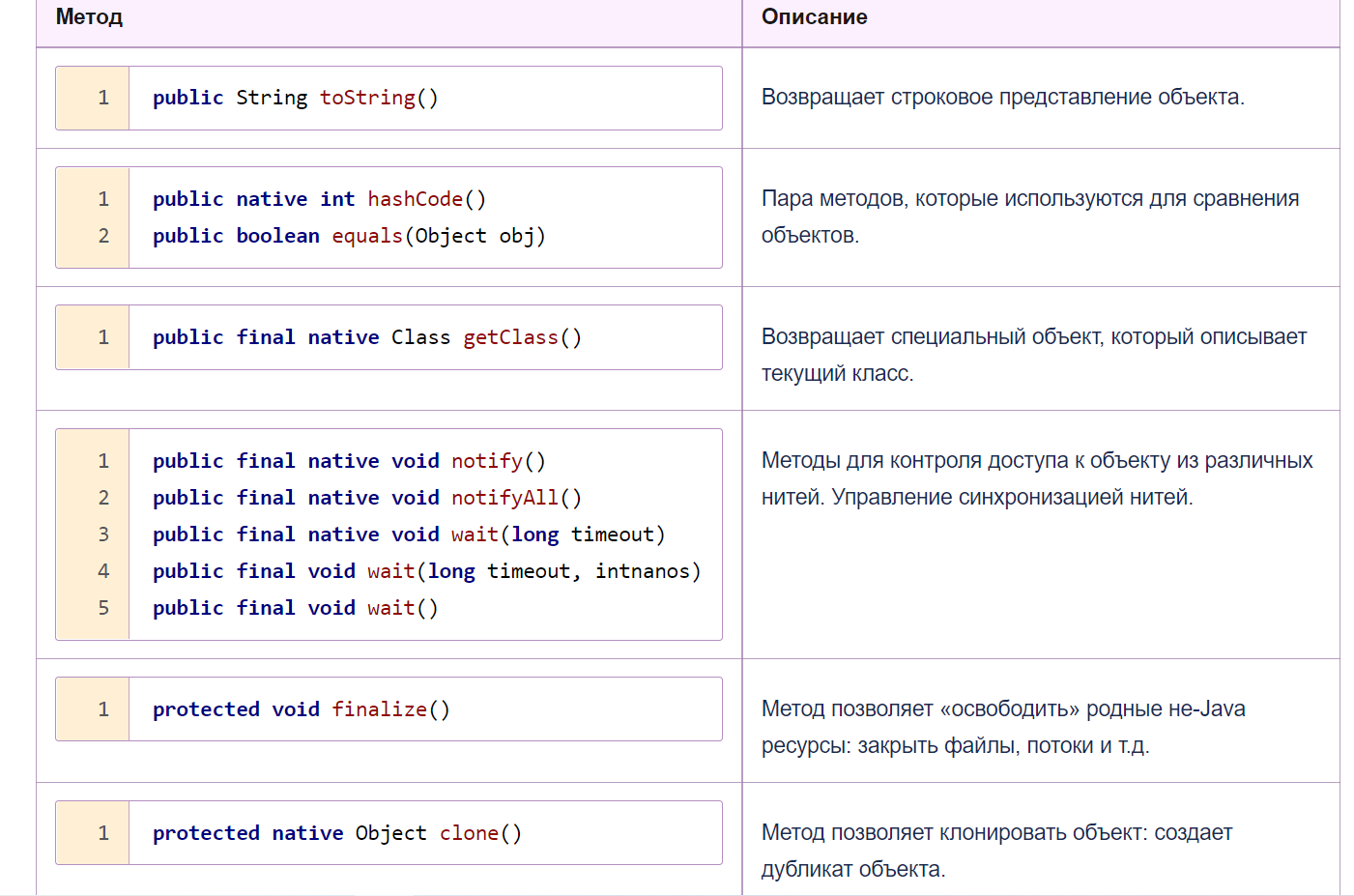
**Java Classes**

19. Базовый класс в java. Методы класса.

В Java определен один специальный класс, называемый Object. Все остальные классы являются подклассами, производными от этого класса, даже если в объявлении это явно не указано. В классе Object определен ряд методов, которые доступны всем классам языка Java.



1. OOP abstraction. Принципы ООП.

**Основные принципы ООП:**

* [Абстракция](https://javarush.ru/groups/posts/principy-oop#%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F-%D0%9E%D0%9E%D0%9F)
* [Инкапсуляция](https://javarush.ru/groups/posts/principy-oop#%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F)
* [Наследование](https://javarush.ru/groups/posts/principy-oop#%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Полиморфизм](https://javarush.ru/groups/posts/principy-oop#%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC)
* **Абстракция** –  означает выделение значимой информации и исключение из рассмотрения незначимой. С точки зрения программирования это правильное разделение программы на объекты. Абстракция позволяет отобрать главные характеристики и опустить второстепенные.

Пример: описание должностей в компании. Здесь название должности значимая информация, а описание обязанностей у каждой должности это второстепенная информация. К примеру главной характеристикой для “директор” будет то, что это должность чем-то управляет, а чем именно (директор по персоналу, финансовый директор, исполнительный директор) это уже второстепенная информация.

* **Инкапсуляция** – свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Для Java корректно будет говорить, что инкапсуляция это “сокрытие реализации”. Пример из жизни – пульт от телевизора. Мы нажимаем кнопочку “увеличить громкость” и она увеличивается, но в этот момент происходят десятки процессов, которые скрыты от нас. Для Java: можно создать класс с 10 методами, например вычисляющие площадь сложной фигуры, но сделать из них 9 private. 10й метод будет называться “вычислитьПлощадь()” и объявлен public, а в нем уже будут вызываться необходимые скрытые от пользователя методы. Именно его и будет вызывать пользователь.
* **Наследование** – свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.
* **Полиморфизм**– свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Пример (чуть переделанный) из Thinking in Java:

Есть общий интерфейс “Фигура” и две его реализации “Треугольник” и “Круг”. У каждого есть метод “нарисовать”. Благодаря полиморфизму нам нет нужды писать отдельный метод для каждой из множества фигур, чтобы вызвать метод “нарисовать”.  Вызов полиморфного метода позволяет одному типу выразить свое отличие от другого, сходного типа, хотя они и происходят от одного базового типа. Это отличие выражается различным действием методов, вызываемых через базовый класс (или интерфейс).  
Здесь приведен пример полиморфизма (также называемый динамическим связыванием, или поздним связыванием, или связыванием во время выполнения), в котором продемонстрировано как во время выполнения программы будет выполнен тот метод, который принадлежит передаваемому объекту.

21. Правила переопределения метода boolean equals(Object o).

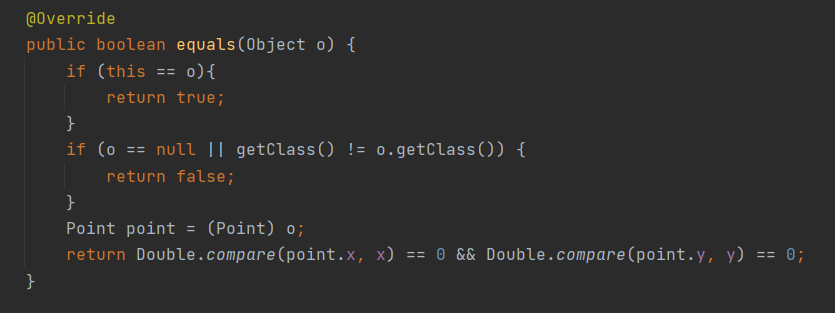
Рефлексивность: Объект должен равняться себе самому.

Симметричность: если a.equals(b) возвращает true, то b.equals(a) должен тоже вернуть true.

Транзитивность: если a.equals(b) возвращает true и b.equals(c) тоже возвращает true, то c.equals(a) тоже должен возвращать true.

Согласованность: повторный вызов метода equals() должен возвращать одно и тоже значение до тех пор, пока какое-либо значение свойств объекта не будет изменено. То есть, если два объекта равны в Java, то они будут равны пока их свойства остаются неизменными.

Сравнение null: объект должны быть проверен на null. Если объект равен null, то метод должен вернуть false, а не NullPointerException. Например, a.equals(null) должен вернуть false.



22. Зачем переопределять методы hashCode  и equals одновременно?

## Соглашение между equals и hashCode в Java.

1. Если объекты равны по результатам выполнения метода equals, тогда их hashcode должны быть одинаковыми.
2. Если объекты не равны по результатам выполнения метода equals, тогда их hashcode могут быть как одинаковыми, так и разными. Однако для повышения производительности, лучше, чтобы разные объекты возвращали разные коды.

Необходимо переопределять hashCode() в каждом классе, который переопределяет equals(). Несоблюдение этого требования приведет к нарушению общего договора для Object.hashCode(), что предотвратит правильное функционирование вашего класса в сочетании со всеми коллекциями на основе хешей, включая HashMap, HashSet и Hashtable.

1. Написать метод equals для класса, содержащего одно поле типа String или StringBuilder.

if (builder == user.builder){  
 boolean result = builder.toString().equals(user.builder.toString());  
}

login.equals(user.login) &&  
email.equals(user.email) &&  
icon.equals(user.icon)

1. Правила переопределения метода int hashCode(). Можно ли в качестве результата возвращать константу?

Хэш-код – это целое число, которое является уникальным идентификатором содержимого объекта. Отсюда вывод – у каждого объекта, с разными данными, свое уникальное число, с помощью которого мы можем судить о равенстве или неравенстве. За вычисление этого кода отвечает метод **hashCode(),**который переопределенный в наследниках **Object.**Для наших классов мы также должны переопределить его.

**Есть несколько правил по переопределению данного метода:**

1. Это не должна быть константа. Иначе все будет равным, даже если это не так.

2. Метод генерации должен быть хорошо продуман, иначе могут часто попадаться ситуации коллизии.

3. В генерации желательно использовать именно те поля, которые идентифицируют объект, его уникальность.

4. Использовать поля, сравниваемые equals().

Пример того как это все выглядит:



У нас есть 2 объекта с двумя полями. Поля одинаковые, значит их хэш-код должен совпадать и указывать на равенство. Создадим генерацию нашего кода на обычной сумме атрибутов:

**1 + 2 = 1 + 2 –**равенство верно.

Но что будет если у нас такие объекты:



Здесь поменялся только порядок значений для второго объекта. Эти объекты уже не могут быть равны, но…

**1 + 2 = 2 + 1 –**равенство осталось верно.

Это есть прямой пример **коллизии.**Два разных объекта имеют одинаковый хэш-код. В нашем случае, причиной этому есть плохо составленный алгоритм вычисления.

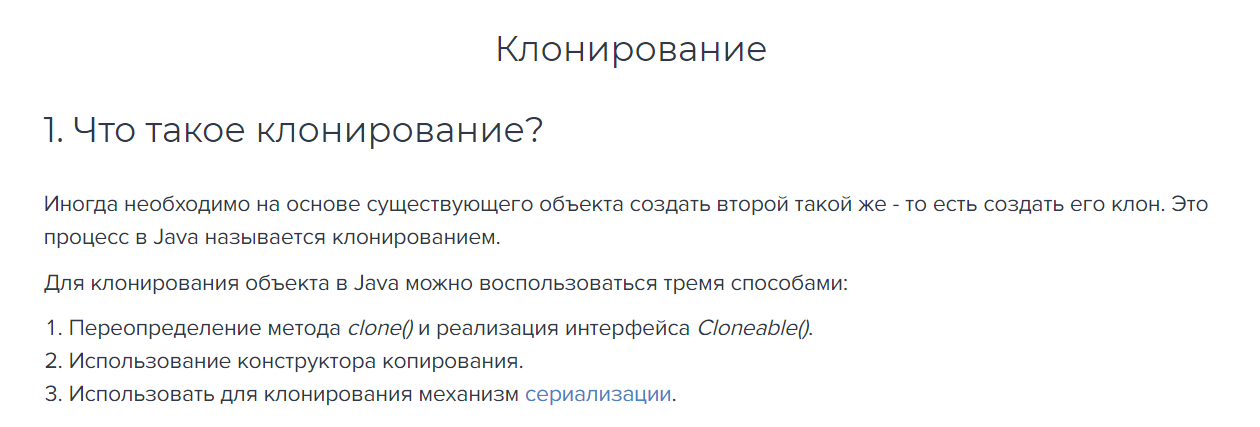
**Можем сделать вывод:**

Для одного и того же объекта хэш-код всегда один(если не изменять вычисляемые поля)

Если хэш-коды равны, то это не значит что и объекты равны.

Если хэш-коды не равны, то значит и объекты не могут быть равны.

1. Правила переопределения метода clone().

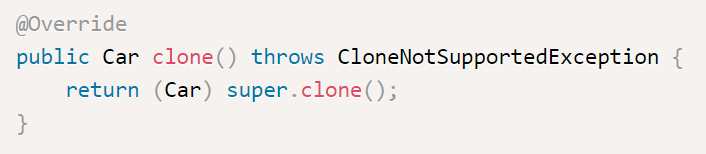


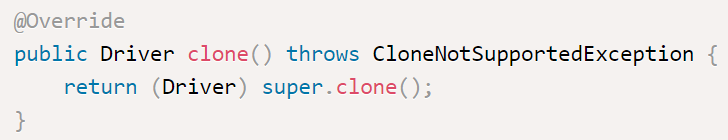
## **Переопределение метода clone().**

Класс Object определяет метод clone(), который создает копию объекта. Если вы хотите, чтобы экземпляр вашего класса можно было клонировать, необходимо переопределить этот метод и реализовать интерфейс Cloneable. Интерфейс Cloneable - это интерфейс маркер, он не содержит ни методов, ни переменных. Интерфейсы маркер просто определяют поведение классов.

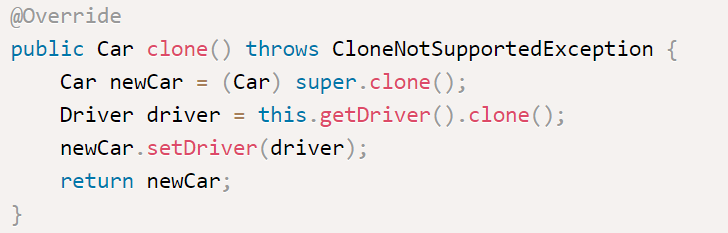
Object.clone() выбрасывает исключение CloneNotSupportedException при попытке клонировать объект не реализующий интерфейс Cloneable.

Метод clone() в родительском классе Object является protected, поэтому желательно переопределить его как public. Реализация по умолчанию метода Object.clone() выполняет неполное/поверхностное (shallow) копирование.





Глубокое клонирование:



1. Чем отличаются finally и finalize? Для чего используется ключевое слово final?

final - объявление константы.

finally - управляет исключениями. Блок finally необязательный и обеспечивает механизм очистки независимо от того, что произошло в try блоке. Используйте finally для закрытия файлов или для освобождения других системных ресурсов таких, как соединения с базой данных.

finalize() - метод, который помогает при сборке мусора. Метод, который вызывается перед тем, как объект будет уничтожен сборщиком мусора.Не должен быть использован для освобождения ресурсов вне оперативной памяти, потому что Java имеет лимитированное количество этих ресурсов.

1. JavaBeans: основные требования к классам Bean-компонентов, соглашения об именах.

**JavaBeans** — классы в языке [Java](https://ru.wikipedia.org/wiki/Java" \o "Java), написанные по определённым правилам. Они используются для объединения нескольких [объектов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в один для удобной передачи данных.

# Основные требования к классам Bean-компонентов

**Bean-компонент**— это основанный на спецификации JavaBeans элемент программного обеспечения, который предназначен для повторного использования и которым можно управлять визуально в интегрированной среде разработки.

Если класс Java соответствует спецификации JavaBeans, он является классом компонента.

**Класс Bean-компонента должен удовлетворять следующим требованиям.**

1.Способность к инициализации нового экземпляра (объекта). Компоненты JavaBeans нельзя создавать на основе интерфейсов и абстрактных классов.

2.Наличие принимаемого по умолчанию конструктора. Например, если класс называется MyBean,

необходимо, чтобы он имел конструктор без параметров MyBean().

3.Возможность сериализации. В компоненте должен быть реализован интерфейс Serializable (или Extemalizable), который позволяет преобразовывать содержимое компонента в поток байтов.

4.Использование модели делегирования событий. При использовании событий необходимо обрабатывать или генерировать их согласно новой модели, предложенной в JDK 1.1.

5.Совместимость со стандартами проектирования компонентов JavaBeans. Разработчик Bean-компонента должен указывать, какие из его свойств, событий и методов должны быть предоставлены среде визуального проектирования. Существует два способа:

•Используются простые соглашения об именах.

•Создаётся дополнительный класс Beanlnfo, который явно поставляет эту информацию.

# Соглашения об именах

Соглашения об именах позволяют определить функциональные возможности компонента.

## •Соглашения об именах для свойств

Каждый Bean-компонент обладает набором свойств. Свойства определяют состояние компонента.

Для определения свойств Bean визуальная среда использует интроспекцию (**Интроспекция**— процесс анализа компонента компоновщиком с целью определения его возможностей. Она позволяет визуальной среде предоставить разработчику информацию о компонентах (их свойствах, методах, событиях) для настройки и связывания компонентов.

В JavaBeans интроспекция поддерживается механизмом отражения (рефлексии), который обеспечивается пакетом java.lang.reflect и элементами класса java.lang.Class).

Таблицы свойств предоставляются разработчику, и он может изменять свойства (осуществлять настройку компонента).

Свойство (**property**) — это именованный атрибут Bean, идентифицируемый в результате интроспекции. Значения, назначенные свойствам, определяют поведение и внешний вид компонента.

Различают три разновидности свойств: простые, логические (булевы) и индексированные.

Так называемые простые свойства могут иметь простой (int, float и т.п.) или объектный

(например: Color, Font, String) тип. Свойства идентифицируются следующими методами: **public type getPropertyName()**

**public void setPropertyName(type value)**

где PropertyName — имя свойства, a type — его тип.

Свойство использует эти методы для доступа к своим значениям. Свойство только для чтения использует только метод getter, а свойство только для записи — метод setter.

He все среды разработки допускают использование индексированных свойств.

## •Соглашения об именах для событий

Компоненты JavaBeans используют модель делегирования событий. События, которые компонент

может генерировать и посылать другим компонентам, идентифицируются следующими методами:

**public void addTypeListener(TypeListener eventListener)**

**public void removeTypeListener(TypeListener eventListener)**

где Type — тип события.

Если, например, в результате интроспекции выясняется наличие у компонента методов addActionListener() и removeActionListener(), то компоновщик предполагает, что компонент может генерировать событие типа ActionEvent.

Первый из указанных выше методов используется для регистрации блока прослушивания событий, когда нужно подтвердить заинтересованность в обработке событий определённого типа. Второй — для удаления блока прослушивания, когда больше нет необходимости принимать уведомления о событиях определённого типа от компонента.

## •Соглашения об именах для методов

Все public-методы компонента JavaBeans доступны для использования средами визуального проектирования. Protected- и privite-методы скрываются.

1. Как работает Garbage Collector. Какие самые распространенные алгоритмы? Можно ли самому указать сборщику мусора, какой объект удалить из памяти.

## Garbage Collector

Сборщик мусора Garbage Collector выполняет всего две задачи, связанные с поиском мусора и его очисткой. Для обнаружения мусора существует два подхода :

* Reference counting – учет ссылок;
* Tracing – трассировка.

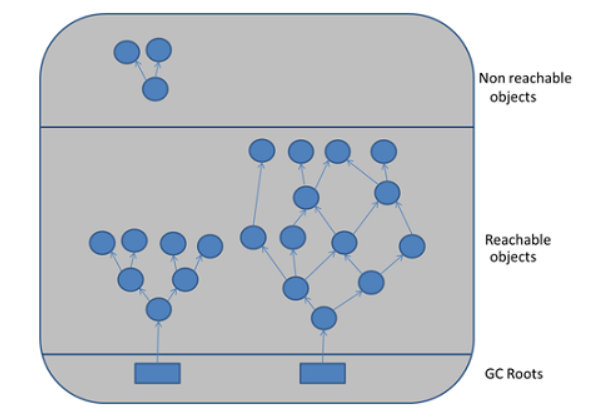
#### Reference counting

Суть подхода «Reference counting» связана с тем, что каждый объект имеет счетчик, который хранит информацию о количестве указывающих на него ссылок. При уничтожении ссылки счетчик уменьшается. При нулевом значении счетчика объект можно считать мусором.

Главным недостатком данного подхода является сложность обеспечения точности счетчика и «невозможность» выявлять циклические зависимости. Так, например, два объекта могут ссылаться друг на друга, но ни на один из них нет внешней ссылки. Это сопровождается утечками памяти. В этой связи данный подход не получил распространения.

#### Tracing

Главная идея «Tracing» связана с тем, что до «живого» объекта можно добраться из корневых точек (GC Root). Всё, что доступно из «живого» объекта, также является «живым». Если представить все объекты и ссылки между ними как дерево, то необходимо пройти от корневых узлов GC Roots по всем узлам. При этом узлы, до которых нельзя добраться, являются мусором.



Данный подход, обеспечивающий выявление циклических ссылок, используется в виртуальной машине HotSpot VM. Теперь, осталось понять, а что представляет из себя корневая точка (GC Root)? «Источники» говорят, что существуют следующие типы корневых точек :

* Основной Java поток.
* Локальные переменные в основном методе.
* Статические переменные основного класса.

Таким образом, простое java-приложение будет иметь следующие корневые точки:

* Параметры main метода и локальные переменные внутри main метода.
* Поток, который выполняет main.
* Статические переменные основного класса, внутри которого находится main метод.

### Очистка памяти

Имеется несколько подходов к очистке памяти, которые в совокупности определяют принцип функционирования Garbage Collection.

#### Copying collectors

При использовании «Copying collectors» область памяти делится на две части : в одной части размещаются объекты, а вторая часть остается чистой. На время очистки мусора приложение останавливает работу и запускается сборщик мусора, который находит в первой области объекты со ссылками и переносит их во вторую (чистую) область. После этого, первая область очищается от оставшихся там объектов без ссылок, и области меняются местами.

Главным достоинством данного подхода является плотное заполнение памяти. Недостатком «Copying collectors» является необходимость остановки приложения и размеры двух частей памяти должны быть одинаковыми на случай, когда все объекты остаются «живыми».

Данный подход в чистом виде в HotSpot VM не используется.

#### Mark-and-sweep

При использовании «mark-and-sweep» все объекты размещаются в одном сегменте памяти. Сборка мусора также приостанавливает приложение, и Garbage Collection проходит по дереву объектов, помечая занятые ими области памяти, как «живые». После этого, все не помеченные участки памяти сохраняются в «free list», в которой будут, после завершения сборки мусора, размещаться новые объекты.

К недостаткам данного подхода следует отнести необходимость приостановки приложения. Кроме этого, время сборки мусора, как и время приостановки приложения, зависит от размера памяти. Память становится «решетчатой», и, если не применить «уплотнение», то память будет использоваться неэффективно.

Данный подход также в чистом виде в HotSpot VM не используется.

#### Generational Garbage Collection

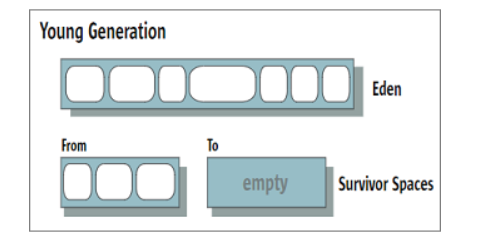
JVM HotSpot использует алгоритм сборки мусора типа «Generational Garbage Collection», который позволяет применять разные модули для разных этапов сборки мусора. Всего в HotSpot реализовано четыре сборщика мусора :

* Serial Garbage Collection
* Parallel Garbage Collection
* CMS Garbage Collection
* G1 Garbage Collection

**Serial Garbage Collection** относится к одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложение приостанавливается и возобновляет работу только после прекращения сборки мусора. В *Serial Garbage Collection* область памяти делится на две части («young generation» и «old generation»), для которых выполняются два типа сборки мусора :

* minor GC – частый и быстрый c областью памяти «young generation»;
* mark-sweep-compact – редкий и более длительный c областью памяти «old generation».

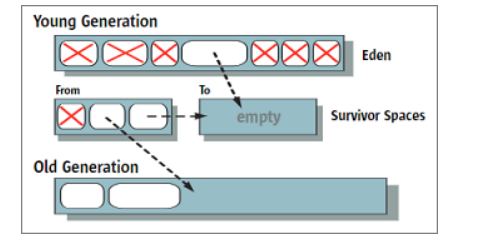
Область памяти «young generation», представленная на следующем рисунке, разделена на две части, одна из которых Survior также разделена на 2 части (From, To).



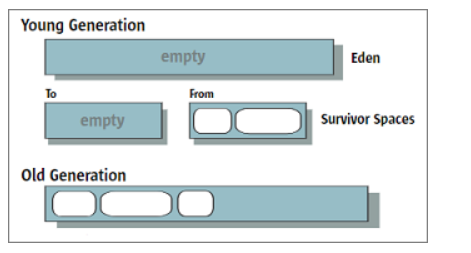
### Алгоритм работы minor GC

Алгоритм работы *minor GC* очень похож на описанный выше «Copying collectors». Отличие связано с дополнительным использованием области памяти «Eden». Очистка мусора выполняется в несколько шагов :

* приложение приостанавливается на начало сборки мусора;
* «живые» объекты из Eden перемещаются в область памяти «To»;
* «живые» объекты из «From» перемещаются в «To» или в «old generation», если они достаточно «старые»;
* Eden и «From» очищаются от мусора;
* «To» и «From» меняются местами;
* приложение возобновляет работу.



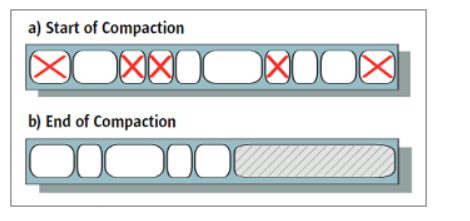
В результате сборки мусора картинка области памяти изменится и будет выглядеть следующим образом :



Некоторые объекты, пережившие несколько сборок мусора в области From, переносятся в «old generation». Следует, также отметить, что и «большие живые» объекты могут также сразу же пеместиться из области Eden в «old generation» (на картинке не показаны).

### Алгоритм работы mark-sweep-compact

Алгоритм «mark-sweep-compact» связяан с очисткой и уплотнением области памяти «old generation».



Принцип работы «mark-sweep-compact» похож на описанный выше «Mark-and-sweep», но добавляется процедура «уплотнения», позволяющая более эффективно использовать память. В процедуре живые объекты перемещаются в начало. Таким образом, мусор остается в конце памяти.

При работе с областью памяти используется механизм «bump-the-pointer», определяющий указатель на начало свободной памяти, в которой размещается создаваемый объект, после чего указатель смещается. В многопоточном приложении используется механизм TLAB (Thread-Local Allocation Buffers), который для каждого потока выделяет определенную область памяти.

Сборщик Мусора — сложная и эффективная составляющая Java. Многие его части работают эвристически – на основе алгоритмов-догадок. Поэтому он часто «не слушается» пользователя.

Объект GC (Garbage Collector – Сборщик Мусора) можно вызвать с помощью метода System.gc().

Также можно принудительно инициировать вызов finalize-методов удаляемых объектов, посредством System.runFinalization(). Но дело в том, что по документации Java, это не гарантирует ни начало сборки мусора, ни вызов методов finalize(). Garbage Collector сам решает, что и когда ему вызывать.

1. В каких областях памяти хранятся значения и объекты, массивы?
2. Какие идентификаторы по умолчанию имеют поля интрефейса?

Интерфейс может содержать поля, но они автоматически являются статическими (static) и неизменными (final). Все методы и переменные неявно объявляются как public.

1. Чем отличается абстрактный класс от интерфейса

Абстрактный класс — это класс, у которого не реализован один или больше методов (некоторые языки требуют такие методы помечать специальными ключевыми словами).

Интерфейс — это абстрактный класс, у которого **ни один** метод не реализован, все они публичные и нет переменных класса.

Интерфейс нужен обычно когда описывается только интерфейс (тавтология). Например, один класс хочет дать другому возможность доступа к некоторым своим методам, но не хочет себя «раскрывать». Поэтому он просто реализует интерфейс.

Абстрактный класс нужен, когда нужно семейство классов, у которых есть много общего. Конечно, можно применить и интерфейс, но тогда нужно будет писать много идентичного кода.

* **Интерфейс описывает только поведение. У него нет состояния. А у абстрактного класса состояние есть: он описывает и то, и другое.**
* **Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.**
* **Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса.**

1. Когда применять интерфейс логичнее, а когда абстрактный класс?

Поскольку в опубликованный интерфейс практически невозможно добавить новый метод, в случае потенциальной необходимости доработки лучше использовать абстрактный класс. Развивать абстрактные классы в Java проще, чем интерфейсы. Аналогично, если в интерфейсе слишком много методов и реализация их всех становится настоящей головной болью, лучше создать абстрактный класс для реализации по умолчанию. Этому паттерну следуют и в пакете коллекций Java, абстрактный класс AbstractList обеспечивает реализацию по умолчанию для интерфейса List.

**Используйте абстрактные классы, если:**

* Вы хотите поделиться кодом между несколькими тесно связанными классами.
* Вы ожидаете, что классы, которые расширяют ваш абстрактный класс, имеют много общих методов или полей, или требуют других модификаторов доступа, кроме public (например, protected и private).
* Вы хотите объявить нестатические или не-final поля. Это позволяет вам определять методы, которые могут получить доступ и изменить состояние объекта, которому они принадлежат.

**Используйте интерфейсы, если:**

* Вы ожидаете, что несвязанные классы будут реализовывать ваш интерфейс. Например,интерфейсы Comparable и Cloneable реализуются многими несвязанными классами.
* Вы хотите определить поведение конкретного типа данных, но вам не важно, кто его реализует.
* Вы хотите использовать множественное наследование типа.

1. Бывают ли интерфейсы без методов. Для чего?

Интерфейсы без каких-либо методов действуют как маркеры. Они лишь говорят компилятору, что объекты классов, которые имплементят такой интерфейс без методов, должен иметь отличительные черты, восприниматься по-другому. Например, java.io.Serializable, java.lang.Cloneable, java.util.EventListener. Маркерные интерфейсы также известных как "теги", поскольку они добавляют общий тег ко всем унаследованым классам, объединяя их в одну категорию.

1. Перегрузка и переопределение. Можно ли менять модификатор доступа метода, если да, то каким образом?

**Переопределение метода** (англ. Method overriding) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

**Модификатор доступа у переопределенного метода не может отличаться от «оригинального» (Исключение: расширение package -> protected -> public).**

**Private метод нельзя переопределить.**

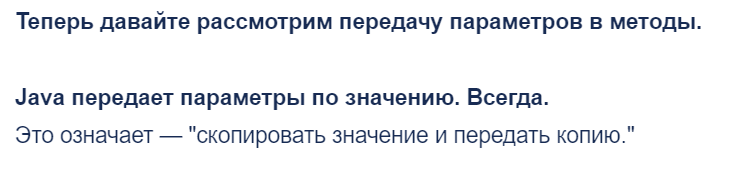
1. Перегрузка и переопределение. Можно ли менять возвращаемый тип метода, если да, то как? Можно ли менять тип передаваемых параметров?

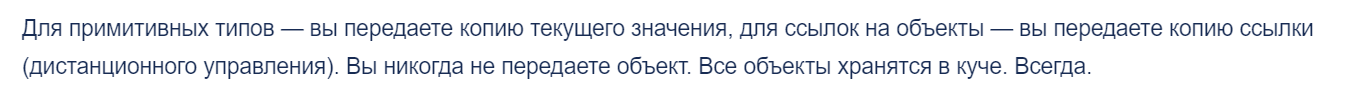
Возвращаемый тип должен быть тот же (Исключение: случай, когда выполняется Downcasting (понижающее преобразование, т.е. возвращаемый тип переопределенного метода в классе наследнике должен быть не шире возвращаемого типа в родительском классе : Object -> Number -> Integer).

Если менять тип передаваемых параметров, то получим перегрузку (параметрический полиморфизм).

**Перегрузка методов** - это **возможность создавать несколько методов с одинаковым названием, но разными параметрами**.

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



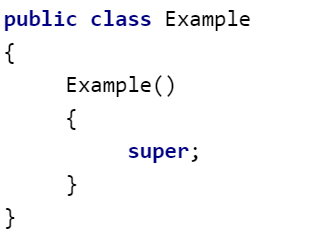


1. Что такое конструктор по умолчанию?

Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст конструктор по умолчанию (default constructor). Этот конструктор пустой (без параметров) и не делает ничего, кроме вызова конструктора суперкласса. Т.е. если написать:



то это эквивалентно написанию:



В данном случае явно класса предка не указано, а по умолчанию все классы Java наследуют класс **Object** поэтому вызывается конструктор класса **Object**.

Если в классе определен конструктор с параметрами, а перегруженного конструктора без параметров нет, то вызов конструктора без параметров является ошибкой. Тем не менее, в Java, начиная с версии 1.5, можно использовать конструкторы с аргументами переменной длины. И если есть конструктор, имеющий аргумент переменной длины, то вызов конструктора по умолчанию ошибкой не будет. Не будет потому, что аргумент переменной длины может быть пустым.

Однако, в распространенном случае, когда в классе вообще не определено ни одного конструктора, вызов конструктора по умолчанию (без параметров) будет обязательным явлением, поскольку подстановка конструктора по умолчанию происходит автоматически.

1. Свойства конструктора. Способы его вызова.

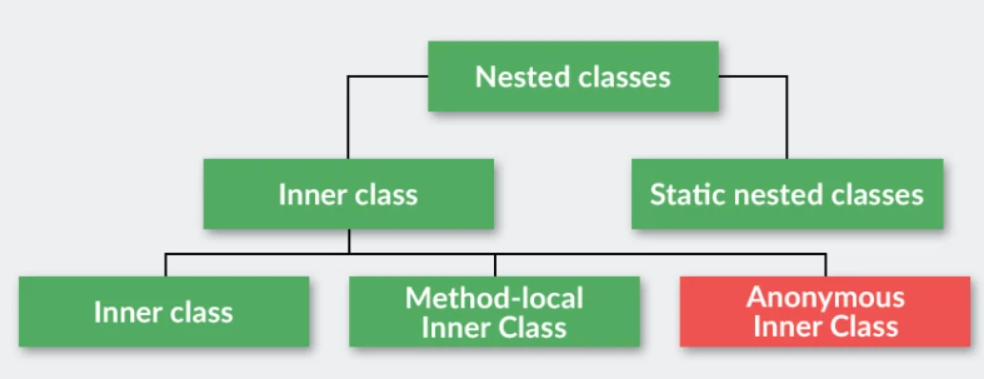
Назначение конструктора – создать экземпляр класса.

Свойства:

* Использование аргументов переменной длины.
* Имя конструктора должно совпадать с именем класса (по договоренности, первая буква — заглавная, обычно имя существительное);
* Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст конструктор по умолчанию (default constructor), который будет пустым и не делает ничего, кроме вызова конструктора суперкласса.
* Конструктор похож на метод, но не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе;
* Конструкторы не наследуются;
* Конструкторов может быть несколько в классе. В этом случае конструкторы называют перегруженными;
* Если в классе не описан конструктор, компилятор автоматически добавляет в код конструктор без параметров;
* Конструктор не имеет возвращаемого типа, им не может быть даже тип **void**, если возвращается тип **void**, то это уже не конструктор а метод, несмотря на совпадение с именем класса.
* В конструкторе допускается оператор **return**, но только пустой, без всякого возвращаемого значения;
* В конструкторе допускается применение модификаторов доступа, можно задать один из модификаторов: **public**, **protected**, **private** или без модификатора.
* Конструктор не может иметь модификаторов **abstract**, **final**, **native**, **static** или **synchronized**;
* Ключевое слово **this** cсылается на другой конструктор в этом же классе. Если используется, то обращение должно к нему быть первой строкой конструктора;
* Ключевое слово **super** вызывает конструктор родительского класса. Если используется, должно обращение к нему быть первой строкой конструктора;
* Если конструктор не делает вызов конструктора **super** класса-предка (с аргументами или без аргументов), компилятор автоматически добавляет код вызова конструктора класса-предка без аргументов;

Конструктор всегда вызывается оператором **new**. При вызове конструктора оператором **new**, конструктор всегда формирует ссылку на новый объект. Заставить конструктор сформировать вместо ссылки на новый объект ссылку на уже существующий объект нельзя, кроме подстановки десериализируемого объекта. А с оператором new сформировать вместо ссылки на новый объект ссылку на уже существующий объект нельзя.

1. Mutable и Immutable классы. Привести примеры из java core. Как создать класс, который будет immutable. Класс record.
2. static - что такое? Что будет, если значение атрибута изменить через объект класса? Всегда ли static поле содержит одинаковые значения для всех его объектов?
3. Внутренние классы, какие бывают и для каких целей используются. Области видимости данных при определенных ситуациях.



Вложенный класс – это класс, который объявлен внутри объявления другого класса. Вложенные классы делятся на статические и нестатические (non-static). Собственно нестатические вложенные классы имеют и другое название – внутренние классы (inner classes).

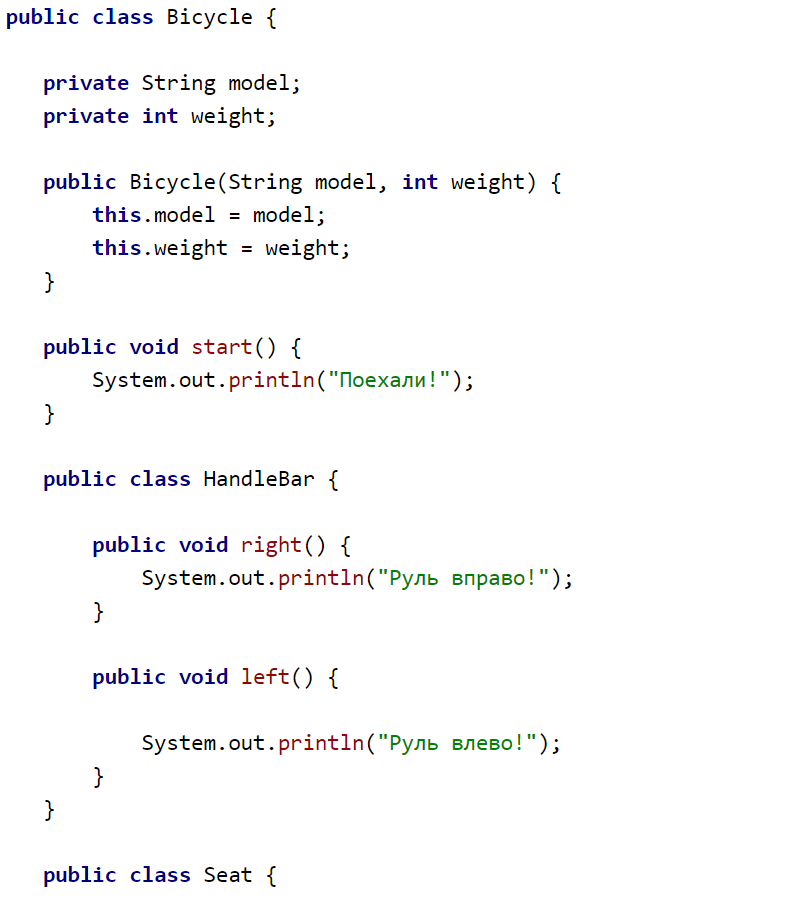
Внутренние классы в Java делятся на такие три вида:

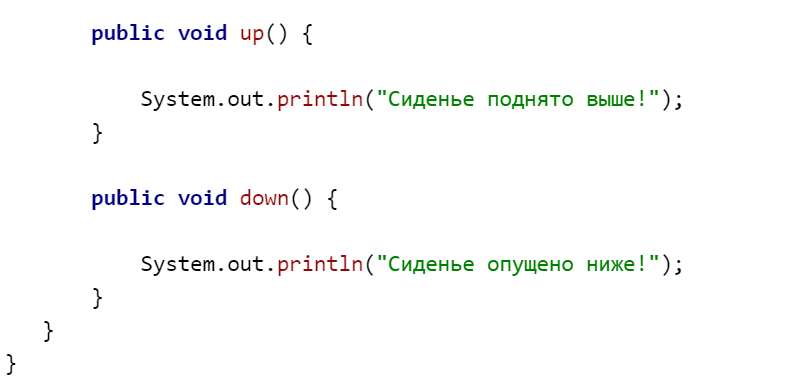
* внутренние классы-члены (member inner classes);

Внутренние классы-члены ассоциируются не с самим внешним классом, а с его экземпляром. При этом они имеют доступ ко всем его полям и методам.

Inner class не может иметь статических объявлений. Вместо этого можно объявить статические методы у обрамляющего класса. Кроме этого, внутри таких классов нельзя объявлять перечисления.

Еще одна важная особенность - интерфейсы в Java не могут быть инстанциированы, соответственно объявить нестатический внутренний интерфейс нельзя, так как элементарно не будет объекта для ассоциации с экземпляром внешнего класса.





* локальные классы (local classes);

Локальные классы (local classes) определяются в блоке Java кода. На практике чаще всего объявление происходит в методе некоторого другого класса. Хотя объявлять локальный класс можно внутри статических и нестатических блоков инициализации.

Как и member классы, локальные классы ассоциируются с экземпляром обрамляющего класса и имеют доступ к его полям и методам. Кроме этого, локальный класс может обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.

У локальных классов есть множество ограничений:

* они видны только в пределах блока, в котором объявлены;
* они не могут быть объявлены как private, public, protected или static;
* они не могут иметь внутри себя статических объявлений (полей, методов, классов); исключением являются константы (static final);

Кстати, интерфейсы тоже нельзя объявлять локально по тем же причинам, по каким их нельзя объявлять внутренними.

Пример:



* анонимные классы (anonymous classes).

Анонимный класс – это локальный класс без имени.

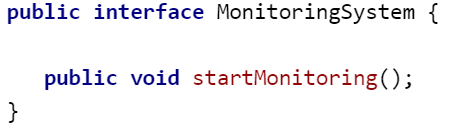
Основное ограничение при использовании анонимных классов - это невозможность описания конструктора, так как класс не имеет имени. Аргументы, указанные в скобках, автоматически используются для вызова конструктора базового класса с теми же параметрами.

Так как анонимный класс является локальным классом, он имеет все те же ограничения, что и локальный класс.

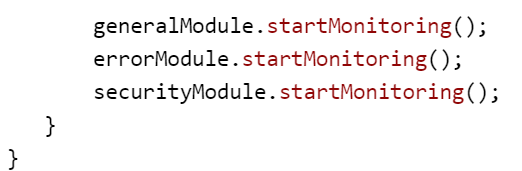
Использование анонимных классов оправдано во многих случаях, в частности когда:

* тело класса является очень коротким;
* нужен только один экземпляр класса;
* класс используется в месте его создания или сразу после него;
* имя класса не важно и не облегчает понимание кода.

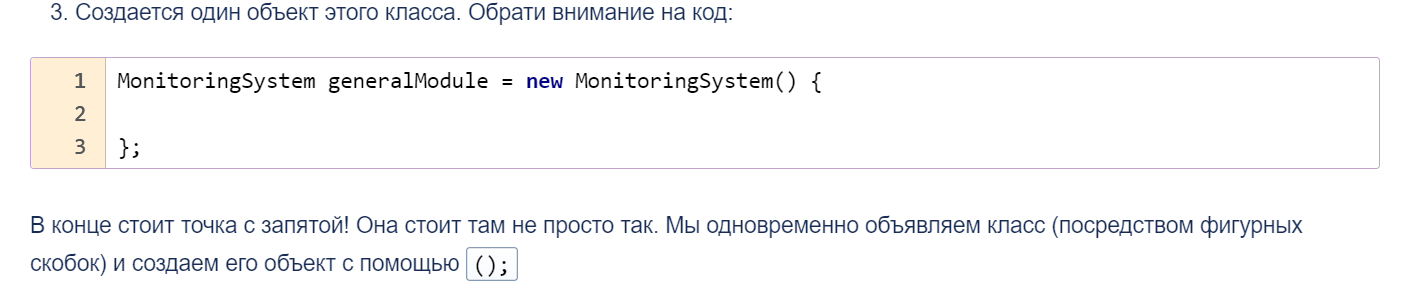
Пример:











Использование вложенных классов всегда приводит к некоторому нарушению инкапсуляции — вложенный класс может обращаться к закрытым членам внешнего класса (но не наоборот!). Если это обстоятельство учитывается в архитектуре вашего приложения, не стоит уделять ему особого внимания, поскольку внутренний класс всего лишь является специализированным членом внешнего класса.

1. Анонимные классы. Практическое применение.

Инфа про анонимные классы в предыдущем вопросе.

1. Generics. Что это такое и для чего применяются. Во что превращается во время компиляции и выполнения? Использование wildcards.
2. Что такое enum? Область применения. Какое использование перечисления некорректно? Привести примеры.

Кроме отдельных примитивных типов данных и классов в Java есть такой тип как **enum** или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую.

Enum — это настоящий класс со всеми вытекающими из этого возможностями (можно добавить переменные, конструкторы и методы).

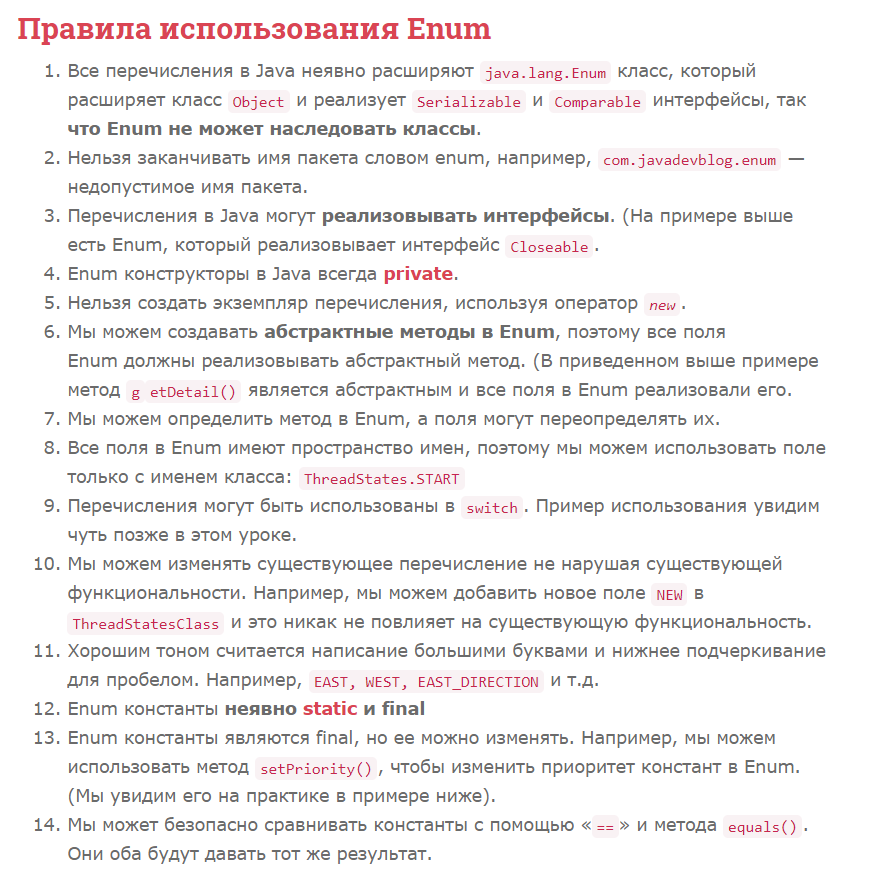
По сравнению с обычными классами, на Enum наложили одно серьезное ограничение — от него невозможно наследоваться.

Кроме того, у перечислений есть характерные только для них методы:

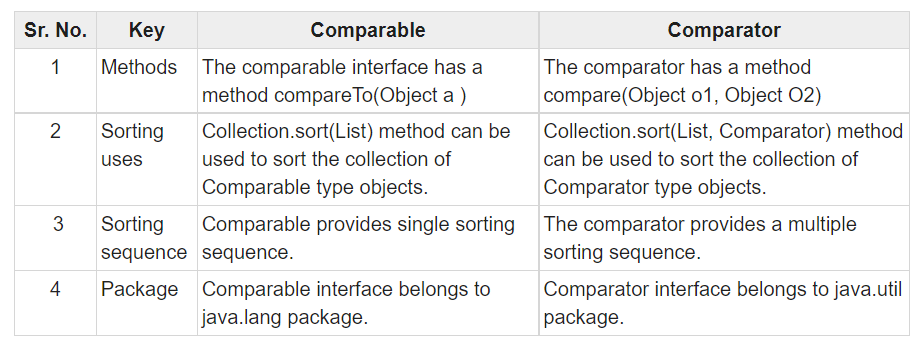
* **values()**: возвращает массив из всех хранящихся в Enum значений
* **ordinal()**: возвращает порядковый номер константы. Отсчет начинается с нуля
* **valueOf()**: возвращает объект Enum, соответствующий переданному имени

Enum можно наделить следующими качествами и свойствами, в зависимости от точки обзора:

* Перечисление и данные
* Каркас для утилитного класса
* Каркас для синглтона класса + антипатерн прилагается
* Каркас для фабрики



1. Отличия в применении интерфейсов Comparator и Comparable?

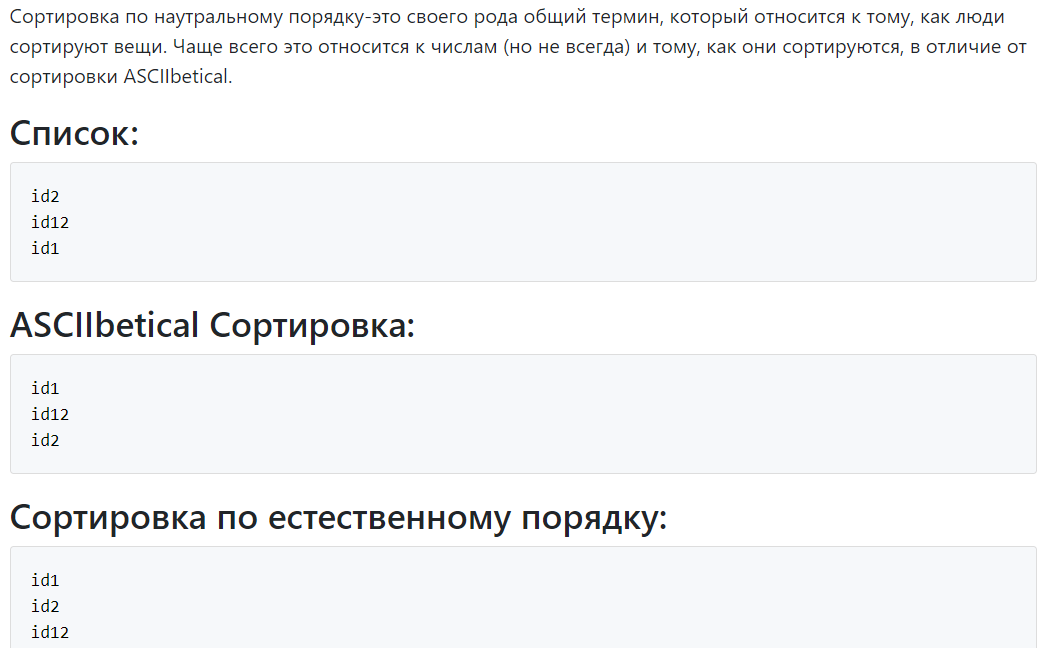


5. Comparable **affects the original class**, i.e., the actual class is modified.

Comparator **doesn't affect the original class**, i.e., the actual class is not modified.

1. Annotation: Comparable - сортировка на основе одного элемента, Comparator – сортировка на основе нескольких элементов.

**Если сортировка объектов должна основываться на естественном порядке, то используйте Comparable, тогда как если сортировка должна быть выполнена по атрибутам разных объектов, то используйте Comparator в Java.**

****

Вывод:

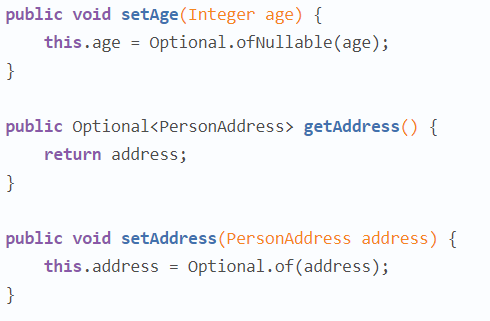
В большинстве случаев аргументы за использование Comparator перевешивают аргументы за использование Comparable.

1. Класс Optional. Как помогает бороться с null?

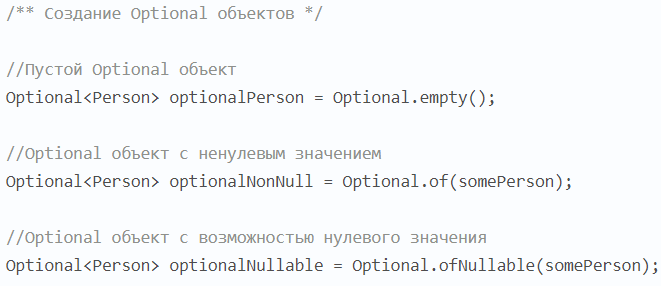
В релизе Java 8 появился новый класс Optional призванный помочь разработчикам в обработке NullPointerException.  
  
С NullPointerException встречались многие и во многих случаях, это очень неприятное исключение заставляет дебажить код, дабы понять, в каком месте, кто-то из твоих предшественников(а возможно и ты), не поставили пресловутую проверку на null.  
  
А что если вообще запретить назначать тем или иным полям класса значения равные null? Java естественно не запрещает нам делать этого, но с Optional это становится немного удобнее и нагляднее.

Пример:

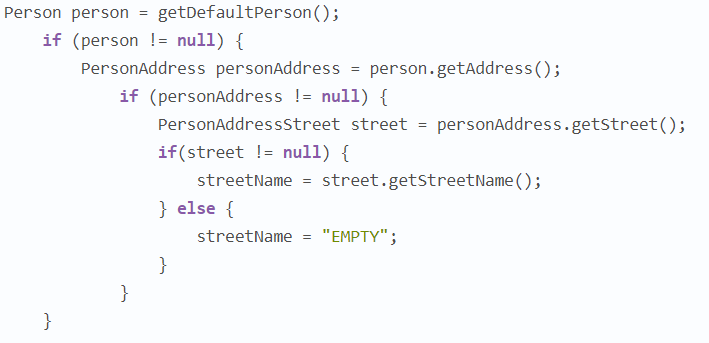


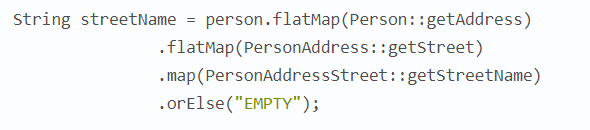


Используем статические методы класса Optional - **of(), ofNullable().**

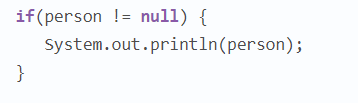


Использование Optional для устранения избыточного кода:



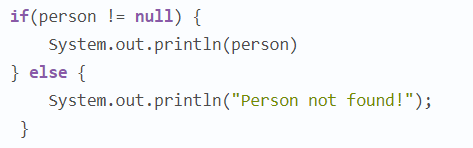


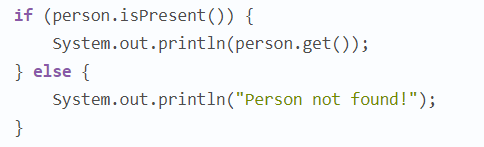
**ifPresent():**



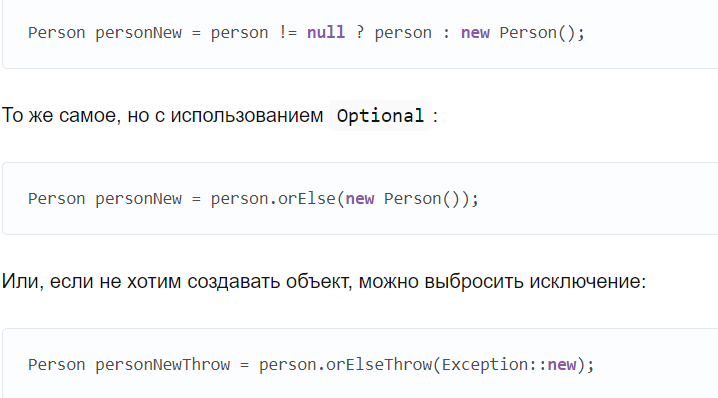


**isPresent():**



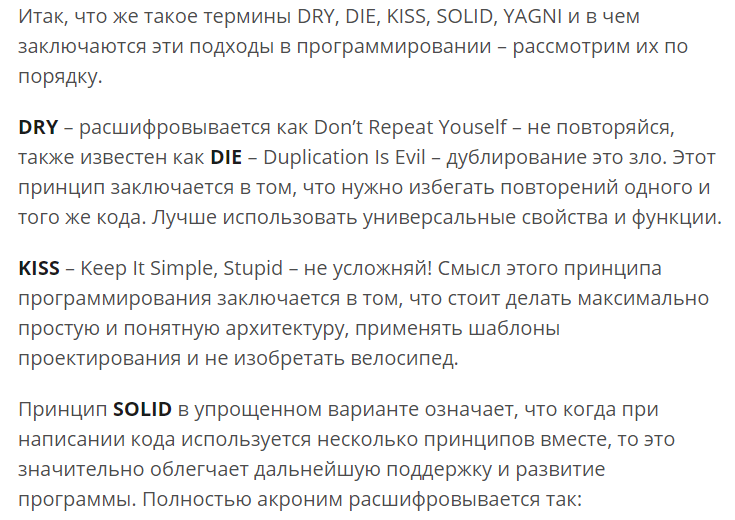


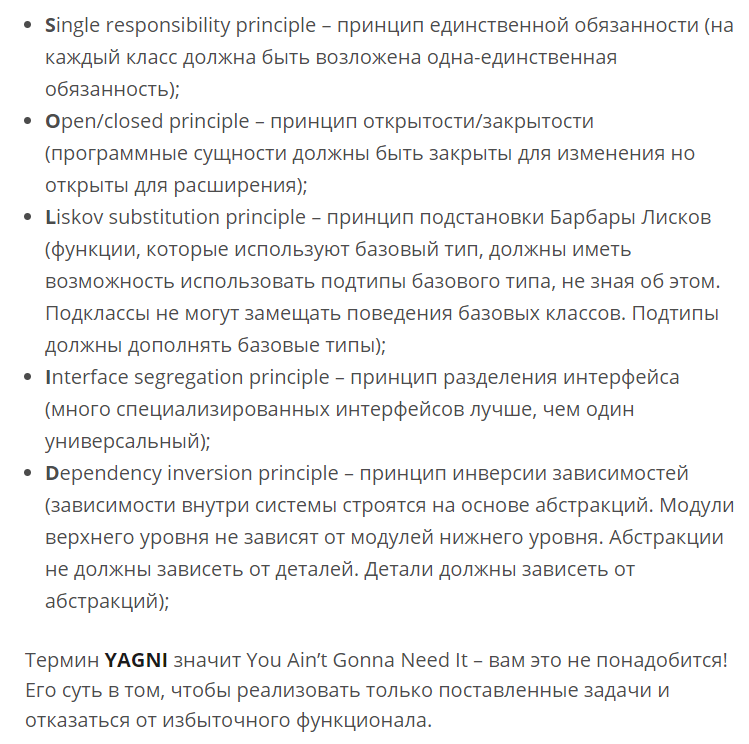
##### **orElse(), orElseThrow():**



Optional помогает придать большую информативность коду, повысить его читабельность.

1. Принципы SOLID, Yagni, Kiss, Dry.





1. Если Interf – это интерфейс, а Сlazz – это класс, который реализует интерфейс и в нем объявлен метод (которого нет в Interf), например, method(). Корректно ли выражение:

Interf a = new Сlazz();

a.method();

То же самое, если Interf – не интерфейс, а абстрактный класс.

Некорректно, т.к. интерфейсная ссылка позволяет получить доступ только к методам данного интерфейса, а не к методам его реализации.

Аналогично про абстрактный класс.