**Interfaces – могут содержать только константы, сигнатуры методов, дефолтные методы, статические методы, абстрактные методы (по сути обычные методы). Интерфейсы могут наследоваться(extend) больше, чем от одного.**

**Определение интерфейса == определение нового референс типа. Интерфейсы всегда abstract.**

**Nested интерфейсы всегда static, т.е static public interface Interface{/\*BODY\*/}.**

**При наследовании возможно переопределение любого метода, кроме статического.**

**В случае если происходит конфликт в наследовании интерфейсов, т.е 2 интерфейса имею одинаковые дефолтные методы, то он разрешается по правилам:**

* **Методы экземпляра предпочтительней дефолтных**
* **Если метод уже переопределен и это не метод экземпляра, т.е метод интерфейса или класса, то он игнорируется – такое возникает тогда и только тогда, когда у интерфейсов общий предок с дефолтным методом, в противном случае ошибка компиляции.**

**P.s: Если 2 или более определенных независимо (не общий предок) методов, или дефолтный и абстрактный метод класса, или дефолтный метод интерфейса и статический метод класса конфликтуют, то ошибка компиляции.**

**Для разрешения конфликтов использовать ИмяИнтерфейса.Супер.Метод**

**Статические методы интерфейса никогда не наследуются.**

**Что происходит если определить метод с такой же сигнатурой, как и у суперкласса:**

* **Если это обычный метод ребенка и обычный метод суперкласса – overriding.**
* **Статический метод ребенка и статический метод суперкласса – hiding.**
* **Иначе ошибка компиляции.**

**Классы**

**Если модификатор члена класса не задан, то package-private**

**Protected – доступ в пакете и так же доступ у всех наследников.**

**Ошибка компиляции при попытке изменить инстанс метод суперкласса на статичный метод класса**

**Статическая инициализация происходит при первом вызове класса, либо принудительно Class.forName(String name). См инициализация классов.**

**Доступ к методам и полям суперкласса через super.**

**Если метод помечен как final, значит он не переопределяется в наследниках. Конструкторы всегда final. Так-же можно объявить полностью класс final – такой класс не имеет наследников.**

**Top level класс не может быть статическим.**

**Абстрактные классы**

**Классы с идентификатором abstract, не могут быть инстанцированы, но имеют наследников. Могу иметь как абстрактные методы, так и нет, но если класс содержит хоть один абстрактный метод, то он обязан быть абстрактным. Абстрактные методы должны переопределяться в наследниках, если они не абстрактные классы. С помощью абстрактных классов можно определять поля, которые константами, а также использовать модификаторы доступа. Extends только для одного класса. Абстрактный класс не должен, но может имплементить методы, наследуемого интерфейса. Статические методы принадлежат классу.**

**Статический блок инициализации – блок со словом static, может встречаться сколько угодно раз и где угодно в программе. Вызываются в том порядке, в котором идут в коде.**

**Нестатические блоки инициализации – компилятор копирует такие блоки в каждый конструктор, такой подход можно использовать если надо разделить блок кода м/у конструкторами без копирования.**

**Так же можно выполнять инициализацию через final методы, т.е private varType variable = initialize(), где initialize == final initialize(){/\*Инициализация\*/}**

**Nested class – Static-nested и inner class**

**Статический класс связан со своим внешним классом, не может напрямую взаимодействовать с инстанцироваными полями и методами кроме как через ссылку на объект.**

**Для создания объекта static-nested класса надо использовать enclosing class name, т.е OuterClass.StaticNestedClass nestedObject =**

**new OuterClass.StaticNestedClass();**

**Inner class – !!!не может определять никакие static поля и методы. Имеет доступ к полям и методам Enclosing класса. Ассоциирован с инстансом enclosing класса.**

**И может существовать только с ним. Создание Inner класса:**

**OuterClass.InnerClass innerObject = outerObject.new InnerClass();**

**Существуют также специальные Inner классы: Local и Anonymous.**

**Local class – определяется внутри блока!!! Имеет доступ к членам enclosing class’a. Локальный класс имеет доступ к локальным переменным, которые объявлены как final. Локальный класс не имеет модификатор доступа.**

**Local class как и Inner class не могу определять статические члены. Локальный класс, определенный в статическом методе, может обращаться только к enclosing static members.**

**!!!Единственный статический член, который может иметь Local class – константы.**

**Shadowing – если локальная переменная в классе имеет то же имя что и у enclosing class, то enclosing переменная будет заменена на локальную.**

**Effectively final – не меняются после инициализации.**

**Anonymous class – использовать если нужен локальный класс только однажды.**

**Анонимный класс – expression => должен быть частью statement.**

**Анонимный класс имеет доступ к members of enclosing class.**

**Не имеет доступа к локальный не final переменным.**

**Используется shadowing.**

**Lambda – доступ так же как и у nested классов, но нет shadowing**

**Функциональный интерфейс – содержит только один абстрактный метод, но может так же содержать static и default методы. НО ФУНКЦИОНАЛЬНЫЙ ИНТЕРФЕЙС ЯВЛЯЕТСЯ НАСЛЕДНИКОМ OBJECT А ПОЭТОМУ СОДЕРЖИТ ЕЩЕ АБСТРАКТНЫЕ МЕТОДЫ которые есть у Object.**

**Преобразование типов:**

**Автоматические преобразования**

**Расширяющие:**

* **byte -> short -> int -> long**
* **int -> double**
* **short -> float -> double**
* **char -> int**

**С потерей точности:**

* **int -> float**
* **long -> float**
* **long -> double**

**В остальных случаях доступны любые явные преобразования, даже сужающие.**

**Преобразования при операциях:**

* **если один из операндов double, то к double**
* **если один из операндов float, то к float**
* **если один из операндов long, то к long**
* **если один из операндов string, то к string**
* **иначе к int**

**В Java ленивая загрузка классов, это значит, что классы ссылочных полей не будут загружены, до того пока не будет явного обращения к ним в программе.**

**Загрузка происходит по следующим принципам:**

* **Класс должен быть полностью загружен, прежде чем слинкован**
* **Класс должен быть полностью проверен и подготовлен, прежде чем проинициализирован**
* **Ошибки разрешения ссылок происходят на этапе выполнения программы, даже если обнаружены на этапе линковки.**

**Инициализация классов и интерфейсов: Инициализация класса или интерфейса происходит перед первым:**

* **Созданием инстанса класса (для классов)**
* **Вызова static метода (для класса и интерфейса)**
* **Присваивание статическому полю значения**
* **Использование статического поля, не являющегося константой**

**Перед инициализацией класса инициализируется его суперклассы (если они не были инициализированы), а также все интерфейсы (объявляют методы по-умолчанию, если они не были инициализированы). Инициализация интерфейса сама по себе не вызывает инициализацию суперинтерфейсов. Ссылка на статическое поле вызывают инициализацию только класса или интерфейса даже если на него можно ссылаться через дочерний класс.**

**Статические инициализаторы и инициализаторы переменных класса (статических переменных) выполняются по порядку и не могут ссылаться на переменные класса, которые идут после данной.**

**Т.к Java-multithreading надо заботиться о том, чтобы инициализация выполнялась одним потоком, и не было рекурсий.**

**Полагается, что объект Class уже верифицирован и подготовлен к загрузке, тогда объект Class находится в одном из состояний:**

* **Верифицирован и подготавливается, но не инициализируется**
* **Верифицируется определенным потоком**
* **Объект инициализирован и готов к использованию**
* **Инициализация не удалась, ошибка**

**Для каждого класса C существует initialization lock - LC. Процедура инициализации выглядит так:**

* **Текущий поток приобретает блокировку LC.**
* **Если инициализация C уже выполняется, то блокировка освобождается и поток блокируется, до того момента пока не поступит сообщение, что инициализация завершена, после чего повторяется шаг.**
* **Если класс C помечен как загружаемый данным потоком, то освободить LC и продолжить выполнение. Это возможно при рекурсивной загрузке.**
* **Если C помечен как Initialized, то освободить LC**
* **Если C находится в ошибочном состоянии, то отпустить блокировку и пробросить NoClassDefFoundError**
* **Иначе, записать что текущий поток начал инициализировать C и отпустить LC.**
* **Идет инициализация констант**
* **Если C – класс, а не интерфейс и SC его суперкласс, SI1..SIn его суперинтерфейсы, которые декларируют хотя бы один default метод, то для каждого S из [SC, SI1,…,SIn] – выполняется инициализация, с верификацией и подготовкой перед этим, если нужно.**
* **Затем выполняется инициализация статических переменных и статических блоков по порядку.**
* **Если инициализация завершается успешно, то LC освобождается и notify all waiting threads.**

**Создание Class instance**

**Происходит при вызове new.**

**Implicity может создаться при:**

* **При загрузке класса или интерфейса, содержится string литерал. Он будет интернирован.**
* **Boxing операции**
* **Использую операцию конкатенации строк “+”, не являющуюся частью constant expression’a**

**Когда создается новый экземпляр класса для него выделяется место в памяти для всех переменных, определенных как члены класса, всех переменных суперкласса, включая тех, которые могут впоследствии быть скрыты.**

**Перед тем как ссылка на новый объект будет возвращена выполняются следующие процедуры:**

* **Присвоение аргументов конструктора переменным**
* **Если конструктор начинается с явного вызова суперконструктора или конструктора в этом классе (super, this), аналогичным образом(проходя те же пункты вызовется конструктор суперкласса).**
* **Если констурктор не начинается с явного вызова конструктора суперкласса и этот конструктор не Object, то вычислятся аргументы и будет вызван конструктор суперкласса.**
* **Выполняются instance initializer и instance variable initializer.**
* **Выполнение остальной части конструктора.**

Изображение выглядит как текст

Автоматически созданное описание

**Классы могу быть выгружены из памяти если их определяющий загрузчик будет reclaimed garbage collector’ом. Классы загруженные bootstrap loader’ом никогда не будут выгружены.**

**Class Loader – Объект ответственный за загрузку классов. Типичная стратегия: заданное имя трансформировать в file name и read classfile from file system.**

**Объект Class содержит ссылку на объект определяющего загрузчика, т.е того который его загрузил, загрузчик в свою очередь содержит ссылку на загруженный Class.**

**Каждый ClassLoader имеет ассоциированный с ним родительский ClassLoader. Прежде чем самому искать требуемый файл ClassLoader делегирует обязанность загрузки своему родителю. Java имеет встроенные загрузчики BootstrapClassLoader – не имеет parent, обычно null; PlatfromClassLoader – platform classes(Java SE Api’s, JDK); SystemClassLoader – родитель PlatformClassLoader, application class loader. Используется чтобы загружать классы из class path, module path, JDK specific tools**