Java.until

[Collection](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Collection.html)<E> - корневой элемент в иерархии коллекций

[Comparator](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Comparator.html)<T> - функция сравнения, которая накладывает порядок на некоторую коллекцию

[Deque](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Deque.html)<E> - линейная коллекция которая позволяет добавлять и удалять элементы с обоих концов

[Enumeration](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Enumeration.html)<E> - генерирует ряд элементов по одному.

[RandomAccess](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/RandomAccess.html) – маркер используемый List, который говорит о том, что возможен рандом аксес

Classes:

Arrays – содержит методы для сортировки и поиска элементов в массиве. (static methods for:

Лексикографического сравнения(compare()), бинарного поиска(binarySearch()), copyOf(original, length) – усекает или заполняет 0, copyOfRange(original, start, stop) – ошибка если индексы за пределами), deepEquals(Object[] a, Object[] b), deepHashCode(Object[] a) - , deepToString(Object[] a), equals() – сравнение one by one, fill(original, from, to, value) – Заполняет массив, hashCode(), mismatch(a, [from, to], b, [from, to]) – находит первый различный элемент, либо -1, sort()(для объектов вторым аргументом может идти компаратор), stream(), toString())

Comparator<Object > реализует один метод compare(object a, object b).

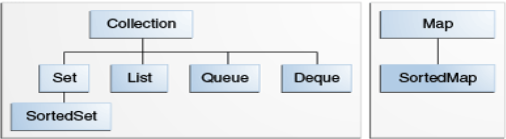
ArrayList – add: *amortized constant time. Значит что для добавления n элементов O(n)*

constant factor меньше чем у LinkedList

growth politics – не специфицирована, но такая, чтобы добавление n элементов происходило за O(n)

capacity – размер массива для хранения элементов, можно увеличивать перед добавлением

итераторы от ArrayList fail-fast, т.е вернут ошибку если List будет модифицирован как либо кроме iterator add and remove.



**HashMap – load factor – 0.75**

**Bucket – Отдельный слот в hashmap.**

**This implementation provides constant-time performance for the basic operations (get and put), полагая, что хэш-функция распределяет элементы равномерно**

**Итерация по коллекции требует времени, пропорционального "capacity" экземпляра HashMap (количеству сегментов) плюс его размеру (количеству сопоставлений ключ-значение). Таким образом, очень важно не устанавливать слишком высокую начальную емкость (или слишком низкий коэффициент загрузки), если важна производительность итерации.**

**capacity - количество сегментов в hash table.**

**initial capacity – емкость на момент создания.**

**load factor – показатель того насколько будет заполнена коллекция, прежде чем ее capacity будет увеличена.**

**После перестроения коллекция увеличивается примерно вдвое.**

**Как правило, начальный Load factor равен 0.75 – компромисс между затратами временными и по памяти. Более высокое значение уменьшает накладные расходы, но увеличивает время поиска. Ожидаемое количество записей и load factor должны учитываться при установке начальной емкости чтобы уменьшить количество операций rehash – пересчитывание хэша каждого элемента и сохранение в новом массиве даблсайз(при перестроении коллекции).**

**Если initial capacity > maximum number of entries / load factor, То операции rehash не будут выполняться**

**Порядок вставки элемента:**

1. **K is converted into a small integer (called its hash code) using a hash function.**
2. **The hash code is used to find an index (hashCode % arrSize) and the entire linked list at that index(Separate chaining) is first searched for the presence of the K already.**
3. **If found, it’s value is updated and if not, the K-V pair is stored as a new node in the list.**

**Iterator is fail-fast.**

**Разрешение коллизий:**

1. **Separate Chaining  - цепочка LinkedList**

**Плюсы**

**1) Простота реализации.**

**2) Хэш-таблица никогда не заполняется, мы всегда можем добавить больше элементов в цепочку.**

**3) Менее чувствительны к хеш-функции или load factor.**

**4) Он в основном используется, когда неизвестно, сколько и как часто ключи могут быть вставлены или удалены.**

**Минусы**

**1) Производительность кэша цепочки не очень хороша, так как ключи хранятся с использованием связанного списка. Открытая адресация обеспечивает лучшую производительность кэша, поскольку все данные хранятся в одной таблице.**

**2) Потеря пространства (Некоторые части хэш-таблицы никогда не используются)**

**3) Если цепочка становится длинной, то время поиска может стать O(n) в худшем случае.**

**4) Использует дополнительное пространство для ссылок.**

1. **Open Addressing - Все элементы хранятся в таблице**

**Линейное зондирование – установка в следующий пустой слот**

**Одной из проблем линейного зондирования является Первичная кластеризация, многие последовательные элементы образуют группы, и для поиска свободного слота или поиска элемента требуется время.**

**Двойное хэширование.**

**HashSet**

**Не гарантирует порядок итерации, допускает хранение null.**

**(add, remove, contains и size) – constant time.**

**Итерация по этому набору требует времени, пропорционального сумме размера экземпляра HashSet (количество элементов) плюс "емкость" резервного экземпляра HashMap (количество сегментов). Таким образом, очень важно не устанавливать слишком высокую начальную емкость (или слишком низкий коэффициент загрузки), если важна производительность итерации.**

**Hashtable<K, V> - любой ненулевой (в HashMap можно использовать нулевые элементы) элемент может использоваться как ключ. Если происходит hash коллизия, то сингл buckets хранит несколько элементов, поиск по которым выполняется последовательно.Методы Hashtable synchronized – поэтому производительность падает, но зато можно использовать в multithreading. При это в случае итерирования по коллекции требуется ручная синхронизация, т.е итератор -fail-fast.**

**LinkedHasMap<K, V> – Реализация Hashtable и LinkedList с предсказуемым порядком адресации.**

**Отличается от HasMap тем, что поддерживает связный список, проходящий через все записи.**

**Этот список определяет порядок итерации. Обычно, это порядок, в котором ключи были вставлены в Map.Повторная вставка ключа не меняет порядок итерации. Эта реализация избавляет от хаотичного упорядочивания, которое предоставляется HashMap и Hashtable без увеличения затрат связанных с TreeMap.**

**Можно создать копию Map, зафиксировав порядок.**

**Порядок итерации от самого последнего вставленного до самого первого.**

**Постоянное время для основных операций. Производительность немного ниже, чем у HashMap из-за дополнительных затрат на поддержание списка. Исключение: итерация по коллекции требует время пропорционального размеру Map независимо от capacity, Итерация по HashMap требует время пропорционального ее capacity.**

**initial capacity and load factor – определяются так же как у HashMap. LinkedHashMap(Map<>) -конструктор для фиксирования порядка.**

**TreeMap<K, V> - Red-Black tree имплементация NavigableMap. Map сортируется в соответствии с натруальным порядком либо по Comapator<E>.**

**Предоставляет гарантированное log(n) время для операций containsKey, get, put, remove. Имплементация не sуnchronized, для synchronized имплементации использовать** [**Collections.synchronizedSortedMap**](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedSortedMap-java.util.SortedMap-)**.**

**Iterator is fail-fast. Все Map.Entry пары представляют из себя snapshot map времени, когда он был сделан, и не поддерживают Entry.setValue метод.**

**LinkedHashSet<> - Итерация пропорциональная количеству элементов, тогда как итерация по HashSet пропорциональна capacity. Сохраняет порядок вставки. Обеспечивает константное время для операций add, contains, remove (полагая, что хэш функция распределяет элементы равномерно по сету). Iterator is fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a** [**ConcurrentModificationException**](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html).

**LinkedList<E> - double-linked list, может содержать null. Implementation is not synchronized, for synchronized Collections.synchronizedList(new LinkedList(...)). Все операции производятся в соответствии с double-linked list.**

SortedMap – Map которая обеспечивает порядок в своих ключах в соответсвии с ествественным порядком либо через Comparator.

Порядок отображается при entryKey, keySet, values.

Все ключи должны быть сравнимы друг с другом

Все реализации должны содержать 4 коснтруктора с параметрами  
Comparator

Map

SortedMap

Void

NavigableMap -

TreeMap –

**I Comparator<T> - compare(T o1, T o2)**

**Math – все методы статичные, Math.min(a, b), Math.max(a, b), e.t.c**

**Java.util.Collections –**

**Collections.revers(List ) – изменяет порядок элементов в List**

**Collections.min/max – возвращает минимальный максимальный элемент.**

**Collections.singleton[List, Map](T o) – возвращает immutable Set[List, Map], содержащий только один элемент o**

**Collections.sort(List, [Comparator]) – сортирует коллекцию**

**Collections.binarySearch (List, Key) – бинарный поиск**

**Collections.synchronizedCollection[List e.t.c]**

**Collections.addAll(Collection, T … element)**

**Generics**

**Objects**

**compare(a,b,Comaprator) – 0 если идентичны comparator.compare иначе**

|  |
| --- |
| [**deepEquals**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#deepEquals-java.lang.Object-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**a,** [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**b)**  **Returns true if the arguments are deeply equal to each other and false otherwise.** |
| [**equals**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#equals-java.lang.Object-java.lang.Object-)  **Returns true if the arguments are equal to each other and false otherwise.** |
| [**hash**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hash-java.lang.Object...-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**... values)**  **Generates a hash code for a sequence of input values.** |
| [**hashCode**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hashCode-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**o)**  **Returns the hash code of a non-null argument and 0 for a null argument.** |
| [**isNull**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#isNull-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**obj)**  **Returns true if the provided reference is null otherwise returns false.** |
| [**nonNull**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#nonNull-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**obj)**  **Returns true if the provided reference is non-null otherwise returns false.** |
| [**requireNonNull**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#requireNonNull-T-)**(T obj)**  **Checks that the specified object reference is not null.** |
| [**requireNonNull**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#requireNonNull-T-java.lang.String-)**(T obj,** [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)**message)**  **Checks that the specified object reference is not null and throws a customized** [**NullPointerException**](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) **if it is.** |
| [**requireNonNull**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#requireNonNull-T-java.util.function.Supplier-)**(T obj,** [**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)**<**[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)**> messageSupplier)**  **Checks that the specified object reference is not null and throws a customized** [**NullPointerException**](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) **if it is.** |
| [**toString**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#toString-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**o)**  **Returns the result of calling toString for a non-null argument and "null" for a null argument.** |
| [**toString**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#toString-java.lang.Object-java.lang.String-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**o,** [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)**nullDefault)**  **Returns the result of calling toString on the first argument if the first argument is not null and returns the second argument otherwise.** |

**Object -**

**Clone() – создает копию объекта. Класс должен реализовывать интерфейс Cloneable, иначе CloneNotSupportedException. Not deep copy.**

**Equals() – если не переопределять, то (==).**

**Запись по все примитивы кроме long, double происходит атомарно.**

**С long и double это не работает, потому что некоторые машины не могу обработать 64-бита сразу.**

**Если установить переменную как volataile, то запись будет атомарна.**

**Java happens-before:**

* **Освобождение мьютекса *happens before* происходит раньше захвата этого же монитора другим потоком.**
* **Метод Thread.start() *happens before* Thread.run().**
* **Завершение метода run() *happens before* выход из метода join().**
* **Запись в volatile переменную *happens-before* чтению из той же переменной.**

**Можно получить синхронизацию над объектом Class выполнив статический synchronized метод.**

**Finalize () – вызывается сборщиком мусора. Когда будет вызван непонятно. В новых версиях деприкейтид.**

**getClass () – переопределять нельзя. Возвращает объект Class**

**hashCode () – если не переопределять, то возвращает memory address объекта в виде hexadecimal числа. По определению, если 2 объекта equals, то их hashCode обязаны быть одинаковыми. Т.е переопределять вместе. Обратно неверно.**

**toString() -**

**Java.nio**

**Buffers – контейнер для данных**

**Charsets – decoders and encoders переводящие символы из байтов в Unicode символы и charset’ы**

**Channels – различных типов, представляющие соединение с сущностями, способными выполнять операции ввода/вывода**

**Selector’ы и selection key – вместе с каналами осуществляют неблокирующий ввод/вывод**

**Buffer – контейнер для фиксированного объема данных примитивного типа.**

**Содержит**

**position – индекс элемента, который должен быть прочитан или записан**

**limit – индекс элемента, который не должен быть записан.**

**mark – индекс элемента, на который будет сброшена Position при reset.**

**ByteBuffer, CharBuffer e.t.c Get/put**

**MappedByteBuffer – маппирует регион файла в буфер.**

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

**Определение интерфейса == определение нового референс типа. 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 определен и это не метод экземпляра то он игнорируется – такое возникает тогда и только тогда , когда у интерфейсов общий предок с дефолтным методом, в противном случае ошибка компиляции.**

**Если 2 или более определенных независимо методов, или дефолтный и абстрактный методы 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:**

**Метод ребенка и метод суперкласса – overriding**

**Статический метод ребенка и статический метод суперкласса – hiding**

**Иначе ошибка компиляции.**

**Классы**

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

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

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

**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 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.**

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

**Нельзя наследоваться от интерфейса внутри блока, потому что все интерфейсы наследуются статически.**

**Все интерфейсы – статические классы.**

**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**

**Java.lang.String – представляет строку в формате UTF-16. Является immutable объектом. Константные выражения с типом String всегда интернированы чтобы разделять уникальное значение между собой. Объект String можно интернировать использую String.intern().**

**Строковый литерал – ссылка на объект String(интернированный объект String).**

**Строки которые были concat(“+”) на стадии compile – интернированы, тогда как строки concat(“+”) на стадии run time – нет.**

**String(char[] str[byte[] b, StringBuilder, StringBuffer, String]) – конструкторы**

**Contains(CharSequence s) –**

**getBytes**

**indexOf**

**intern()**

**replace**

**split**

**substring**

**trim**

**String.valueOf**

**getChars(begin, end, [] buffer, beginBuffer)**

**Java.lang.StringBuilder – mutable последовательность символов, без гарантии synchronization, быстрее чем StringBuffer потому что отсутствует синхронизация.**

**Append() -**

**Insert() - sb.append(“x”) has same effect as sb.insert(sb.lenght(), “x”)**

**Delete()**

**getChars(begin, end, [] buffer, beginBuffer)**

**reverse() –**

**length() –**

**toString –**

**java.lang.StringBuffer – thread-safe аналог StringBuilder**

**Java.lang.Thread –**

**Потоки выполняются пока:**

* **не был вызван метод exit из класса Runtime.**
* **Все потоки не демоны, умерли, вернулись из run, или выбросили исключение.**

**Daemon thread – низкоприоритетный поток, который поставляет некоторый сервис для User thread’ам, например: garbage collector – daemon thread.**

**Join() – ждет пока поток умрет**

**Sleep() – усыпляет поток**

**Start() –**

**Yield() – освобождает процессор. Планировщик запускает следующий поток, если таких нет, то поток который предпоследним сделал yield.**

**Interrupt() – прерывает поток.**

**Java.util.Concurrency –**

**Executor**

**Java.io -**

**Metaspace – (stored in native heap) Область в памяти предназначенная для хранения загруженных классов, интернированных строк и других метаданных. Сборка мусора вызывается, когда достигнуто определенное значение, заданное при запуске jvm.**

**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.**

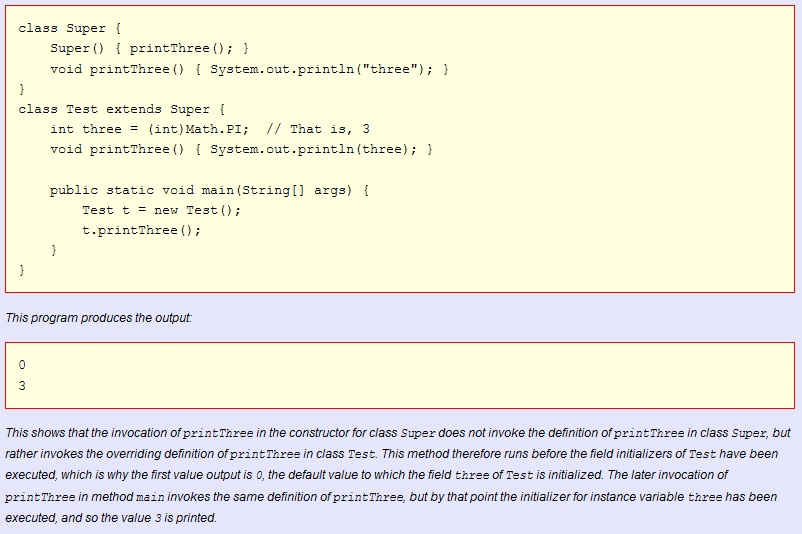
**Implicity может создаться при:**

* **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.**



**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.**

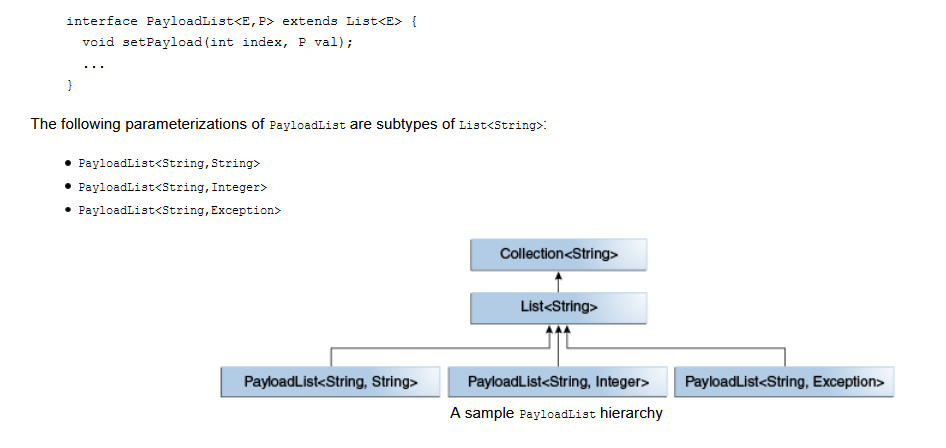
**Объект Class содержит ссылку на объект опрделяющего загрузчика, т.е того который его загрузил, загрузчик в свою очередь содержит ссылку на загруженный Class.**

**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**

**Generics –**

**Raw type – дженерик без спецификации параметра. Использование raw type ведет к unchecked warning. Кроме классов дженериками могут быть методы. Type inference – использование дженериков без спецификации типов в angel backets. Дженерики могут быть bounded <U extends Number> или <I implements Runnable>. Так же возможна multiple bound <T extends U1 & U2 & U3> - T должен быть subtype of all types, если одна из bounds’ов класс, то он должен идти первым. !!!!Операторы >, < используются только для примитивных типов, т.е в дженерик методе с типом T: <T> Boolean fun(T a, T b){ return a < b;} //compile error , т.к T – объект. Пофиксить можно используя интерфейс Comparable<T> и функцию comapreTo.**

**Если А является подтипом B, но MyClass<A> не является подтипом MyClass<B>. MyClass<A> и MyClass<B> являются подтипами Object. Дженерики являются подтипами в том случае если FirstClass<E> extends SecondClass<E>, тогда FirstClass<A> будет подтипом SecondClass<B> - пример Collection -> List -> ArrayList.**

**Еще пример:** 

**Type inference – способность компилятора сопоставлять объявление и вызов метода, чтобы определять типы аргументов. Алгоритм определяет тип аргументов и пытается найти наиболее специфичное значение, удовлетворяющее всем аргументам**

**SQL.**

**Чтобы исполнить любое SQL выражение, нужно выполнить действия:**

* **Устанавливается соединение с Data Source**
* **Создать Statement:**
  + **Statement – простые выражения без параметров**
  + **PreparedStatement – если использовать объект statement’а много раз, то PreparedStatement сокращает время выполнения. Это происходит потому что при создании preparedStatement’у присваивается SQL statement, который отправляется в СУБД и там компилируется, это значит, что при каждом новом запросе бд может запустить уже скомпилированный стейтмент. Можно использовать как с параметрами так и без. Одно из главных приемуществ – preparedStatement позволяют избежать sql injection, т.к интерпретируют входящие параметры как содержимое параметра, а не запрос.**
  + **CallableStatement – используется для выполнения хранимых процедур. Содержащих как входные так и выходные параметры.**

**Lambda expressions -**