**1.Шаблоны. 1.1 Порождающие**

Начнём с начала жизненного цикла объектов — с создания объектов. Порождающие шаблоны как раз и помогают создавать объекты удобнее, обеспечить гибкость этого процесса.

Одним из самых известных является "Строитель" (Builder). Данный паттерн позволяет создавать сложные объекты пошагово. В Java самый известный пример — StringBuilder:

Другим известным подходом к созданию объекта — вынос создания в отдельный метод. Такой метод как бы становится фабрикой объектов. Поэтому и шаблон называется "Фабричный метод" (Factory Method).

Но иногда одного фабричного метода мало. Иногда требуется создавать разные объекты так, чтобы они друг с другом сочетались. В этом нам поможет другой шаблон - "Абстрактная фабрика" (Abstract factory).

И тогда нам требуется создавать разные фабрики в одном месте. При этому плюсом является то, что нам не важны детали реализации, т.е. не важно, какую конкретно фабрику мы получим. Главное, чтобы она создавала правильные реализации.

То есть в зависимости от окружения (от операционной системы) мы получим определённую фабрику, которая создаст совместимые элементы.

Как альтернатива подхода к созданию через кого-то, мы можем воспользоваться паттерном "Прототип". Суть его проста — новые объекты создаются по образу и подобию уже существующих объектов, т.е. по их прототипу. В Java с этим паттерном сталкивался каждый — это использование интерфейса java.lang.Cloneable

Ну и самый последний в этом списке — паттерн "Одиночка" (Singleton).

Цель его проста — обеспечить единственный экземпляр объекта на всё приложение. Данный паттерн интересен тем, что на нём часто показывают проблемы многопоточности.

**1.2 Структурные**

Их цель — построение удобных в поддержке иерархий классов и их взаимосвязей.

Одним из первых и всем известных паттернов — "Заместитель" (Proxy).

Заместитель имеет тот же интерфейс, что и реальный объект, поэтому для клиента нет разницы — работать через заместителя или напрямую. Самым простым примером является java.lang.reflect.Proxy

Как мы видим, взаимодействие в паттерне идёт через интерфейсы. Но иногда заместителя недостаточно. И тогда может быть использован паттерн "Декоратор" (Decorator). Декоратор ещё называют обёрткой или враппером (Wrapper). Прокси и декоратор очень похожи, но если посмотреть на пример — будет видна разница.

В отличии от прокси, декоратор оборачивается вокруг чего-то, что передали на вход. Прокси же может как принимать то, что нужно проксировать, так и сам управлять жизнью проксируемого объекта (например, создавать проксируемый объект).

Есть ещё один интересный паттерн — "Адаптер" (adapter). Он похож на декоратор — на вход декоратор принимает один объект и возвращает обёртку над этим объектом. Отличие в том, что цель у этого не изменение ыфункционала, а адаптация одного интерфейса к другому.

Следующим интересным подходом в разработке структуры классов является паттерн "Компоновщик" (Сomposite). Интересен он тем, что некоторый набор элементов использующих один интерфейс выстраиваются в некоторую древовидную иерархию. Вызывая метод в родительском элементе мы получаем вызов этого метода по всем необходимым дочерним элементам.

Яркий пример этого паттерна — UI (будь то java.awt или JSF)

Ещё одним из интересных паттернов является паттерн "Мост" (Bridge). Называется он так, потому что описывает соединение или мост между двумя различными иерархиями классов.

Одну из этих иерархий считают абстракцией, а другую — реализацией. Так выделено потому что абстракция сама не выполняет действия, а делегирует это выполнение реализации. Такой паттерн часто применяют тогда, когда есть классы "управления" и несколько видов классов "платформ" (например, Windows, Linux и т.д.).

При таком подходе одна из этих иерархий (абстракция) получит ссылку на объекты другой иерархии (реализация) и будет делегировать им основную работу. Благодаря тому, что все реализации будут следовать общему интерфейсу, их можно будет взаимозаменять внутри абстракции.

В Java яркие пример этому — java.awt

Среди структурных паттернов так же хочется отметить паттерн "Фасад" (facade). Суть его в том, чтобы за удобным и лаконичным интерфейсом спрятать сложность использования библиотек/фрэймворков, стоящих за этим API. Например, как пример, можно привести JSF или EntityManager из JPA.

Так же есть другой паттерн, называемый "Легковес" (Flyweight). Его суть заключается в том, что если у разных объектов есть одинаковое состояние, то его можно обобщить и хранить не в каждом объекте, а в одном месте. И тогда каждый объект сможет ссылаться на общу часть, что позволит сократить расходы памяти на хранение.

**1.3 Поведенческие**

Итак, мы разобрались, как можно создать объекты и как можно организовать связь между классами. Осталось самое интересно — обеспечить гибкость в изменении поведения объектов. И в этом нам помогут поведенческие паттерны.

Одним из самых часто упоминаемых паттернов является паттерн "Стратегия".

При помощи паттерна "Стратегия" мы можем внутри объекта хранить то, каким образом мы будем выполнять действие, т.е. объект внутри хранит стратегию, которая может быть изменена в том числе во врем выполнения кода.

Этот паттерн мы часто используем, когда применяем компаратор

Интересно, что есть почти такой же паттерн с названием "Состояние" (State). Паттерн "Состояние" говорит, что если у нас есть у главного объекта некоторое поведение, зависимое от состояние этого объекта, то тогда можно описать само состояние в виде объекта и менять объект состояния. А вызовы из главного объекта делегировать состоянию.

Ещё один паттерн, известный нам с изучения самых основ языка Java — паттерн "Комманда". Этот паттерн проектирования говорит о том, что различные команды можно представлять в виде разных классов. Данный паттерн очень похож на паттерн "Стратегия". Но в паттерне "Стратегия" мы переопределяли то, как будет выполняться конкретное действие (например, сортировка в TreeSet). В паттерне "Комманда" же мы переопределяем то, какое вообще действие будет выполнено.

Паттерн комманда с нами каждый день, когда мы используем потоки.

Как видим, command определяет действие или комманду, которая будет выполнена в новом потоке.

Так же стоит рассмотреть и паттерн "Цепочка обязанностей" (Chain of responsibility). Данный паттерн тоже очень просто. Этот паттерн говорит, что если что-то надо обработать, то можно собрать обработчики в цепочку.

Например, такой шаблон часто используется в веб-серверах. На входе сервер имеет некоторый запрос от пользователя. Дальше этот запрос проходит цепочку обработки. В этой цепочке обработчиков есть фильтры (например, не принимать запросы из чёрного списка IP-адресов), обработчики аутентификации (пускать только разрешённых пользователей), обработчик заголовков запроса, обработчик кэширования и т.д. Но есть в Java и более простой и понятный пример — java.util.logging

Ещё один паттерн, который мы видим каждый день "Итератор". Суть его заключается в том, чтобы разделить коллекцию объектов (т.к. класс, представляющий структуру данных. Например, List) и обход этой коллекции.

Как видно, итератор — не является частью коллекции, а представлен отдельным классом, который обходит коллекцию. Использующий итератор даже может не знать про то, по какой коллекции он итерируется, т.е. какую коллекцию он обходит.

Стоит рассмотреть и паттерн "Посетитель" (Visitor).

Паттерн посетитель очень похож на итератор. Данный паттерн помогает обходить структуру объектов и выполнять действия над этими объектами. Отличаются они скорее концепцией. Итератор обходит коллекцию так, что клиенту, использующему итератор, всё равно, что за коллекция внутри, важны лишь элементы из последовательности.

Посетитель же именно про то, что есть некоторая иерархия или структура объектов, которые мы посещаем. Например, мы можем использовать отдельную обработку каталогов и отдельную обработку файлов.

В Java "из коробки" есть реализация этого паттерна в виде java.nio.file.FileVisitor

Иногда возникает необходимость одним объектам реагировать на изменения в других объектах и тогда нам поможет паттерн "Наблюдатель" (Observer).

Cамый удобный способ — это обеспечить механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Данный паттерн часто применяется в различных Listener'ах и Observer'ах, реагирующих на разные события.

Есть ещё один полезный поведенческий шаблон — "Посредник" (Mediator). Полезен он тем, что в сложных системах помогает убрать связь между разными объектами и делегировать все взаимодействия между объектами некоторому объекту, который и является посредником. Часто можно увидеть в примерах так же java.util.Timer.

Стоит рассмотреть и паттерн "Шаблонный метод" (Template Method)

Понятный уже по названию шаблон. Суть заключается в том, что код написан так, что пользователям кода (разработчикам) предоставляется некоторый шаблон алгоритма, шаги в котором разрешается переопределять. Это позволяет пользователям кода не писать весь алгоритм, а думать только над тем, как правильно выполнить тот или иной шаг этого алгоритма.

Например, в Java есть абстрактный класс AbstractList, определяющий поведение итератора по List. Однако, сам итератор использует методы листа, такие как: get, set, remove. Поведение этих методов определяет разработчик наследников AbstractList. Таким образом итератор в AbstractList — является шаблоном алгоритма итерирования по листу. А разработчики конкретных реализаций AbstractList меняют поведение этого итерирования, определяя поведение конкретных шагов.

Последний из разбираемых нами паттернов — паттерн "Снимок" (Momento). Суть его заключается в сохранении некоторого состояния объекта с возможностью это состояние восстановить.

Самым узнаваемым примером из JDK является сериализация объекта, т.е. java.io.Serializable

Как мы увидели из обзора, паттернов существует огромное множество. Каждый из них решает свою задачу. И знание этих паттернов может помочь Вам вовремя понять, как написать Вашу систему так, чтобы она была гибкая, поддерживаемая и устойчивая к изменениям.

**2. Коллекции**

“Коллекции” — общее название для нескольких структур данных в Java. Данные можно хранить многими разными способами. Мы пока изучили только класс ArrayList, где данные хранятся в массиве. С остальными коллекциями мы познакомимся позднее. Сейчас достаточно понимать что класс Collections рассчитан на работу не только с ArrayList, но и с другими видами коллекций в Java (отсюда, собственно, и его название).

Итак, какие же задачи при работе с ArrayList позволяет решить класс Collections?

Первое и самое очевидное — сортировка. В лекции о массивах мы рассматривали пример с числами, а теперь рассмотрим пример со строками. Для сортировки содержимого коллекций в классе Collections реализован метод sort()

В классе String запрограммировано, как именно строки сравниваются между собой (как раз — по алфавиту). Для классов, которые ты будешь создавать сам, можешь реализовать свой механизм сравнения, но об этом мы поговорим в других лекциях.

Кроме того, класс Collections позволяет найти минимальный и максимальный элемент в ArrayList. Это делается с помощью методов min() и max():

Это, конечно, гораздо удобнее, чем вручную писать код для прохождения по всем элементам и поискам наибольшего/наименьшего элемента :)

Еще один крайне полезный метод — reverse(). Если бы нам нужно было “перевернуть” список, чтобы элементы шли в обратном порядке — как бы мы это делали? Вероятно, написать такой алгоритм самому было бы не так просто :)

К счастью, метод reverse() уже это умеет. Например, нам не нравится как метод sort() отсортировал наши планеты в алфавитном порядке, и мы хотим изменить порядок на обратный — от Я до А

Например, мы пытаемся реализовать механизм работы лотереи. Мы добавили в барабан 100 чисел, которые по одному должны появляться на экране. Кто из участников первым зачеркнет все числа на своем билете — побеждает.

Реализовать такой механизм очень легко с помощью метода shuffle():

Теперь представим другую ситуацию. Ранее мы создали список solarSystem с перечисленными в нем планетами.

И он нас, вроде как, всем устраивает, если бы не одно но: из него можно удалять элементы и добавлять новые! Это явно не то поведение, которое мы ждем: Солнечная система в нашей программе должна быть в неизменном состоянии.

В классе Collections есть очень интересный метод — unmodifiableList(). Он создает из переданного списка его неизменяемый вариант. В него нельзя будет ни добавить, ни удалить элемент.

Ошибка в solarSystem теперь нельзя ничего добавлять!

Единственное, на что в данном случае нужно обратить внимание — тип переменной должен быть List<>, а не ArrayList<> (данный метод возвращает объект именно такого типа, общего для всех видов списков).

Еще одна рядовая ситуация, которая может произойти во время работы — программист добавил элементы в неправильном порядке.

Если такое произошло, и Меркурий и Нептун неожиданно поменялись местами — исправить эту оплошность нам поможет метод swap():

В метод swap() мы передали наш список, а также индексы двух элементов, которые нужно поменять местами. Обрати внимание: метод работает именно с индексами, а не со ссылками. Поэтому здесь нам понадобился метод ArrayList.indexOf().

Напоследок познакомимся с очень интересным методом — disjoint().

Он проверяет, есть ли у двух коллекций пересечения, то есть хоть один одинаковый элемент. Если нет — возвращает true, если есть — false.

Как видишь, в наших двух списках элементы полностью разные, поэтому программа выводит true.

Вот такой интересный и очень полезный класс. Как и Arrays, он выполняет вместо нас много рутинной, черновой работы, позволяя сосредоточиться на других вещах.

**3. Потоки данных. Промежуточные и оконечные операции.**

Многопоточность в Java была заложена с самых первых дней. Поэтому давайте кратко ознакомимся с тем, про что это — многопоточность.

Что такое процесс?

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

При помощи процессов выполнение разных программ изолировано друг от друга: каждое приложение использует свою область памяти, не мешая другим программам.

Процесс не может существовать без потоков, поэтому если существует процесс, в нём существует хотя бы один поток.

Как же это происходит в Java? Когда мы запускаем Java программу, ее выполнение начинается с метода main. Мы как бы входим в программу, поэтому этот особый метод main называется точкой входа, или "entry point". Метод main всегда должен быть public static void, чтобы виртуальная машина Java (JVM) смогла начать выполнение нашей программы.

Java launcher выполняет определённый набор Java Native Interface (JNI) вызовов. JNI — это механизм, соединяющий мир виртуальной машины Java и мир C++. Получается, что launcher — это не JVM, а её загрузчик. Он знает, какие правильные команды нужно выполнить, чтобы запустилась JVM. Знает, как организовать всё необходимое окружение при помощи JNI вызовов.

В эту организацию окружения входит и создание главного потока, который обычно называется main.

Интересно, что каждый поток имеет свою обособленную область в памяти, выделенной для процесса. Эту структуру памяти называют стеком.

Стек состоит из фрэймов. Фрэйм — это точка вызова метода, execution point.

Если посмотреть на Java API и поискать там слово Thread, мы увидим, что есть класс java.lang.Thread.

Именно этот класс представляет в Java поток, и с ним нам и предстоит работать.

Поток в Java представлен в виде экземпляра класса java.lang.Thread. Стоит сразу понимать, что экземпляры класса Thread в Java сами по себе не являются потоками. Это лишь своего рода API для низкоуровневых потоков, которыми управляет JVM и операционная система. Когда при помощи java launcher'а мы запускаем JVM, она создает главный поток с именем main и ещё несколько служебных потоков.

Существует 2 типа потоков: демоны и не демоны. Демон-потоки — это фоновые потоки (служебные), выполняющие какую-то работу в фоне. Такой интересный термин — это отсылка к "демону Максвелла", о чём подробнее можно прочитать в википедии в статье про "демонов".

Как сказано в документации, JVM продолжает выполнение программы (процесса), до тех пор, пока:

Не вызван метод Runtime.exit

Все НЕ демон-потоки завершили свою работу (как без ошибок, так и с выбрасыванием исключений)

Отсюда и важная деталь: демон-потоки могут быть завершены на любой выполняемой команде. Поэтому целостность данных в них не гарантируется. Поэтому, демон потоки подходят для каких-то служебных задач. Например, в Java есть поток, который отвечает за обработку методов finalize или потоки, относящиеся к сборщику мусора (Garbage Collector, GC).

Каждый поток входит в какую-то группу (ThreadGroup).

А группы могут входит друг в друга, образовывая некоторую иерархию или структуру.

Группы позволяют упорядочить управление потоками и вести их учёт.

Помимо групп, у потоков есть свой обработчик исключений

Кроме того, у потока есть приоритет.

у нас 2 способа создать поток.

Первый — создать своего наследника.

запуск задачи выполняется в методе run, а запуск потока в методе start. Не стоит их путать, т.к. если мы запустим метод run напрямую, никакой новый поток не будет запущен. Именно метод start просит JVM создать новый поток.

Вариант с потомком от Thread плох уже тем, что мы в иерархию классов включаем Thread. Второй минус — мы начинаем нарушать принцип "Единственной ответственности" SOLID, т.к. наш класс становится одновременно ответственным и за управление потоком и за некоторую задачу, которая должна выполняться в этом потоке.

Потоки нужны, чтобы одновременно выполнять работу. Поэтому очень вероятно, что потоки будут как-то взаимодействовать между собой. Давайте разберёмся, как это происходит и какие базовые средства управления у нас есть.

Метод Thread.yield() загадочный и редко используемый. Существует много вариаций его описания в интернете. Вплоть до того, что некоторые пишут про какую-то очередь потоков, в которой поток переместится вниз с учётом их приоритетов. Кто-то пишет, что поток изменит статус с running на runnable (хотя разделения на эти статусы нет, и Java их не различает). Но на самом деле всё куда неизвестнее и в каком-то смысле проще.

Sleep - Засыпание потока

Поток в процессе своего выполнения может засыпать. Это самой простой тип взаимодействия с другими потоками.

В операционной системе, на которой установлена виртуальная Java машина, где выполняется Java код, есть свой планировщик потоков, называемый Thread Scheduler. Именно он решает, какой поток когда запускать.

Программист не может взаимодействовать с этим планировщиком напрямую из Java кода, но он может через JVM попросить планировщик на какое-то время поставить поток на паузу, "усыпить" его.

Прерывание потока или Thread.interrupt

Всё дело в том, что пока поток ожидает во сне, кто-то может захотеть прервать это ожидание. На этот случай мы обрабатываем такое исключение. Сделано это было после того, как метод Thread.stop объявили Deprecated, т.е. устаревшим и нежелательным к использованию. Причиной тому было то, что при вызове метода stop поток просто "убивался", что было очень непредсказуемо. Мы не могли знать, когда поток будет остановлен, не могли гарантировать консистентность данных.

Представте, что вы пишете данные в файл и тут поток уничтожают. Поэтому, решили, что логичнее будет поток не убивать, а информировать его о том, что ему следует прерваться. Как на это реагировать — дело самого потока.

Join — Ожидание завершения другого потока

Самым простым типом ожидания является ожидание завершения другого потока.

Понятие Монитор

В многопоточности есть такое понятие, как Monitor. Вообще, слово монитор с латинского переводится как "надзиратель" или "надсмотрщик".

для целей синхронизации между потоками Java использует некий механизм, который называется "Монитор". С каждым объектом ассоциирован некоторый монитор, а потоки могут его заблокировать "lock" или разблокировать "unlock".

Итак, при помощи ключевого слова synchronized текущий поток (в котором выполняются эти строки кода) пытается использовать монитор, ассоциированный с объектом object и "получить лок" или "захватить монитор" (второй вариант даже предпочтетельнее). Если за монитор нет соперничества (т.е. никто больше не хочет выполнить synchronized по такому же объекту), Java может попытаться выполнить оптимизацию, называемую "biased locking". В заголовке объекта в Mark Word выставится соответствующий тэг и запись о том, к какому потоку привязан монитор. Это позволяет сократить накладные расходы при захватывании монитора.

Если монитор уже ранее был привязан к другому потоку, тогда такой блокировки недостаточно. JVM переключается на следующий тип блокировки — basic locking. Она использует compare-and-swap (CAS) операции. При этом в заголовке в Mark Word уже хранится не сам Mark Word, а ссылка на его хранение + изменяется тэг, чтобы JVM поняла, что у нас используется базовая блокировка.

Если же возникает соперничество (contention) за монитор нескольких потоков (один захватил монитор, а второй ждёт освобождение монитора), тогда тэг в Mark Word меняется, и в Mark Word начинает храниться ссылка уже на монитор как объект — некоторую внутреннюю сущность JVM. Как сказано в JEP, в таком случае требуется место в Native Heap области памяти на хранение этой сущности. Ссылка на место хранения этой внутренней сущности и будет находиться в Mark Word объекта.

Таким образом, как мы видим, монитор — это действительно механизм обеспечения синхронизации доступа нескольких потоков к общим ресурсам. Существует несколько реализаций этого механизма, между которыми переключается JVM. Поэтому для простоты, говоря про монитор, мы говорим на самом деле про локи.

С понятием монитора, как мы ранее видели, тесно связано понятие "блок синхронизации" (или как ещё называют — критическая секция).Здесь главный поток сначала отправляет задачу task в новый поток, а потом сразу же "захватывает" лок и выполняет с ним долгую операцию (N секунд). Всё это время task не может для своего выполнения зайти в блок synchronized, т.к. лок уже занят.

Если поток не может получить лок, он будет ждать этого у монитора. Как только получит — продолжит выполнение. Когда поток выходит из-под монитора, он освобождает лок.

У Thread есть ещё один метод ожидания, который при этом связан с монитором. В отличие от sleep и join, его нельзя просто так вызвать. И зовут его wait.

Выполняется метод wait на объекте, на мониторе которого мы хотим выполнить ожидание.

Существует возможность отправить уведомление только одному из потоков (notify) или сразу всем потокам из очереди (notifyAll).

**4. Лямбда-вырежния**

Среди новшеств, которые были привнесены в язык Java с выходом JDK 8, особняком стоят лямбда-выражения. Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

Рассмотрим пример:

public class LambdaApp {

public static void main(String[] args) {

Operationable operation;

operation = (x,y)->x+y;

int result = operation.calculate(10, 20);

System.out.println(result); //30

}

}

interface Operationable{

int calculate(int x, int y);

}

В роли функционального интерфейса выступает интерфейс Operationable, в котором определен один метод без реализации - метод calculate. Данный метод принимает два параметра - целых числа, и возвращает некоторое целое число.

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java. В частности, предыдущий пример мы можем переписать следующим образом:

public class LambdaApp {

public static void main(String[] args) {

Operationable op = new Operationable(){

public int calculate(int x, int y){

return x + y;

}

};

int z = op.calculate(20, 10);

System.out.println(z); // 30

}

}

interface Operationable{

int calculate(int x, int y);

}

Чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

Определение ссылки на функциональный интерфейс:

Operationable operation;

Создание лямбда-выражения:

operation = (x,y)->x+y;

Причем параметры лямбда-выражения соответствуют параметрам единственного метода интерфейса Operationable, а результат соответствует возвращаемому результату метода интерфейса. При этом нам не надо использовать ключевое слово return для возврата результата из лямбда-выражения.

Так, в методе интерфейса оба параметра представляют тип int, значит, в теле лямбда-выражения мы можем применить к ним сложение. Результат сложения также представляет тип int, объект которого возвращается методом интерфейса.

Использование лямбда-выражения в виде вызова метода интерфейса:

int result = operation.calculate(10, 20);

Так как в лямбда-выражении определена операция сложения параметров, результатом метода будет сумма чисел 10 и 20.

При этом для одного функционального интерфейса мы можем определить множество лямбда-выражений. Например:

Operationable operation1 = (int x, int y)-> x + y;

Operationable operation2 = (int x, int y)-> x - y;

Operationable operation3 = (int x, int y)-> x \* y;

System.out.println(operation1.calculate(20, 10)); //30

System.out.println(operation2.calculate(20, 10)); //10

System.out.println(operation3.calculate(20, 10)); //200

Отложенное выполнение

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (deferred execution). То есть мы определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

Выполнение кода отдельном потоке

Выполнение одного и того же кода несколько раз

Выполнение кода в результате какого-то события

Выполнение кода только в том случае, когда он действительно необходим и если он необходим

Передача параметров в лямбда-выражение

Параметры лямбда-выражения должны соответствовать по типу параметрам метода из функционального интерфейса. При написании самого лямбда-выражения тип параметров писать необязательно, хотя в принципе это можно сделать, например:

1

operation = (int x, int y)->x+y;

Если метод не принимает никаких параметров, то пишутся пустые скобки, например:

()-> 30 + 20;

Если метод принимает только один параметр, то скобки можно опустить:

n-> n \* n;

Терминальные лямбда-выражения

Выше мы рассмотрели лямбда-выражения, которые возвращают определенное значение. Но также могут быть и терминальные лямбды, которые не возвращают никакого значения. Например:

interface Printable{

void print(String s);

}

public class LambdaApp {

public static void main(String[] args) {

Printable printer = s->System.out.println(s);

printer.print("Hello Java!");

}

}

Лямбды и локальные переменные

Лямбда-выражение может использовать переменные, которые объявлены во вне в более общей области видимости - на уровне класса или метода, в котором лямбда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямбдах. Рассмотрим первый пример - использования переменных уровня класса:

public class LambdaApp {

static int x = 10;

static int y = 20;

public static void main(String[] args) {

Operation op = ()->{

x=30;

return x+y;

};

System.out.println(op.calculate()); // 50

System.out.println(x); // 30 - значение x изменилось

}

}

interface Operation{

int calculate();

}

Переменные x и y объявлены на уровне класса, и в лямбда-выражении мы их можем получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной x.

Теперь рассмотрим другой пример - локальные переменные на уровне метода:

public static void main(String[] args) {

int n=70;

int m=30;

Operation op = ()->{

//n=100; - так нельзя сделать

return m+n;

};

// n=100; - так тоже нельзя

System.out.println(op.calculate()); // 100

}

Локальные переменные уровня метода мы также можем использовать в лямбдах, но изменять их значение нельзя. Если мы попробуем это сделать, то среда разработки (Netbeans) может нам высветить ошибку и то, что такую переменную надо пометить с помощью ключевого слова final, то есть сделать константой: final int n=70;. Однако это необязательно.

Более того, мы не сможем изменить значение переменной, которая используется в лямбда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути она является константой.

Блоки кода в лямбда-выражениях

Существуют два типа лямбда-выражений: однострочное выражение и блок кода. Примеры однострочных выражений демонстрировались выше. Блочные выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

Operationable operation = (int x, int y)-> {

if(y==0)

return 0;

else

return x/y;

};

System.out.println(operation.calculate(20, 10)); //2

System.out.println(operation.calculate(20, 0)); //0

Обобщенный функциональный интерфейс

Функциональный интерфейс может быть обобщенным, однако в лямбда-выражении использование обобщений не допускается. В этом случае нам надо типизировать объект интерфейса определенным типом, который потом будет применяться в лямбда-выражении. Например:

public class LambdaApp {

public static void main(String[] args) {

Operationable<Integer> operation1 = (x, y)-> x + y;

Operationable<String> operation2 = (x, y) -> x + y;

System.out.println(operation1.calculate(20, 10)); //30

System.out.println(operation2.calculate("20", "10")); //2010

}

}

interface Operationable<T>{

T calculate(T x, T y);

}

Таким образом, при объявлении лямбда-выражения ему уже известно, какой тип параметры будут представлять и какой тип они будут возвращать.

**5. Обработка текстовой информации, кодировки. Регулярные выражения**

Регулярные выражения

Большинство современных языков программирования поддерживают РВ, Java не является исключением.

Механизм

Существует две базовые технологии, на основе которых строятся механизмы РВ:

Недетерминированный конечный автомат (НКА) — «механизм, управляемый регулярным выражением»

Детерминированный конечный автомат (ДКА) — «механизм, управляемый текстом»

НКА — механизм, в котором управление внутри РВ передается от компонента к компоненту. НКА просматривает РВ по одному компоненту и проверяет, совпадает ли компонент с текстом. Если совпадает — проверятся следующий компонент. Процедура повторяется до тех пор, пока не будет найдено совпадение для всех компонентов РВ (пока не получим общее совпадение).

ДКА — механизм, который анализирует строку и следит за всеми «возможными совпадениями». Его работа зависит от каждого просканированного символа текста (то есть ДКА «управляется текстом»). Даний механизм сканирует символ текста, обновляет «потенциальное совпадение» и резервирует его. Если следующий символ аннулирует «потенциальное совпадение», то ДКА возвращается к резерву. Нет резерва — нет совпадений.

Логично, что ДКА должен работать быстрее чем НКА (ДКА проверяет каждый символ текста не более одного раза, НКА — сколько угодно раз пока не закончит разбор РВ). Но НКА предоставляет возможность определять ход дальнейших событий. Мы можем в значительной степени управлять процессом за счет правильного написания РВ.

Регулярные выражения в Java используют механизм НКА.

Подход к обработке

В языках программирования существует три подхода к обработке РВ:

интегрированный

процедурный

объектно-ориентированный

Интегрированный подход — встраивание РВ в низкоуровневый синтаксис языка. Этот подход скрывает всю механику, настройку и, как следствие, упрощает работу программиста.

Функциональность РВ при процедурном и объектно-ориентированном подходе обеспечивают функции и методы соответственно. Вместо специальных конструкций языка, функции и методы принимают в качестве параметров строки и интерпретируют их как РВ.

Для работы с регулярными выражениями в Java представлен пакет java.util.regex. Пакет был добавлен в версии 1.4 и уже тогда содержал мощный и современный прикладной интерфейс для работы с регулярными выражениями. Обеспечивает хорошую гибкость из-за использования объектов, реализующих интерефейс CharSequence.

Класс Pattern представляет собой скомпилированное представление РВ. Класс не имеет публичных конструкторов, поэтому для создания объекта данного класса необходимо вызвать статический метод compile и передать в качестве первого аргумента строку с РВ:

// XML тэг в формате <xxx></xxx>

Pattern pattern = Pattern.compile("^<([a-z]+)([^>]+)\*(?:>(.\*)<\\/\\1>|\\s+\\/>)$");

Также в качестве второго параметра в метод compile можно передать флаг в виде статической константы класса Pattern, например:

// email адрес в формате xxx@xxx.xxx (регистр букв игнорируется)

Pattern pattern = Pattern.compile("^([a-z0-9\_\\.-]+)@([a-z0-9\_\\.-]+)\\.([a-z\\.]{2,6})$", Patt

Иногда нам необходимо просто проверить есть ли в строке подстрока, что удовлетворяет заданному РВ. Для этого используют статический метод matches, например:

// это hex код цвета?

if (Pattern.matches("^#?([a-f0-9]{6}|[a-f0-9]{3})$", "#8b2323")) { // вернет true

// делаем что-то

}

Также иногда возникает необходимость разбить строку на массив подстрок используя РВ. В этом нам поможет метод split:

Pattern pattern = Pattern.compile(":|;");

String[] animals = pattern.split("cat:dog;bird:cow");

Arrays.asList(animals).forEach(animal -> System.out.print(animal + " "));

// cat dog bird cow

Matcher — класс, который представляет строку, реализует механизм согласования (matching) с РВ и хранит результаты этого согласования (используя реализацию методов интерфейса MatchResult). Не имеет публичных конструкторов, поэтому для создания объекта этого класса нужно использовать метод matcher класса Pattern:

// будем искать URL

String regexp = "^(https?:\\/\\/)?([\\da-z\\.-]+)\\.([a-z\\.]{2,6})([\\/\\w \\.-]\*)\*\\/?$";

String url = "http://habrahabr.ru/post/260767/";

Pattern pattern = Pattern.compile(regexp);

Matcher matcher = pattern.matcher(url);

Но результатов у нас еще нет. Чтобы их получить нужно воспользоваться методом find. Можно использовать matches — этот метод вернет true только тогда, когда вся строка соответствует заданному РВ, в отличии от find, который пытается найти подстроку, которая удовлетворяет РВ. Для более детальной информации о результатах согласования можно использовать реализацию методов интерфейса MatchResult, например:

// IP адрес

String regexp = "(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)";

// для сравнения работы find() и matches()

String goodIp = "192.168.0.3";

String badIp = "192.168.0.3g";

Pattern pattern = Pattern.compile(regexp);

Matcher matcher = pattern.matcher(goodIp);

// matches() - true, find() - true

matcher = pattern.matcher(badIp);

// matches() - false, find() - true

// а теперь получим дополнительную информацию

System.out.println(matcher.find() ?

"I found '"+matcher.group()+"' starting at index "+matcher.start()+" and ending at index "+matcher.end()+"." :

"I found nothing!");

// I found the text '192.168.0.3' starting at index 0 and ending at index 11.

Также можно начинать поиск с нужной позиции используя find(int start). Стоит отметить что существует еще один способ поиска — метод lookingAt. Он начинает проверку совпадений РВ с начала строки, но не требует полного соответствия, в отличии от matches.

Класс предоставляет методы для замены текста в указанной строке:

appendReplacement(StringBuffer sb, String replacement) Реализует механизм «добавление-и-замена» (append-and-replace). Формирует обьект StringBuffer (получен как параметр) добавляя replacement в нужные места. Устанавливает позицию, которая соответствует end() последнего результата поиска. После этой позиции ничего не добавляет.

appendTail(StringBuffer sb) Используется после одного или нескольких вызовов appendReplacement и служит для добавления оставшейся части строки в объект класса StringBuffer, полученного как параметр.

replaceFirst(String replacement) Заменяет первую последовательность, которая соответствует РВ, на replacement. Использует вызовы методов appendReplacement и appendTail.

replaceAll(String replacement) Заменяет каждую последовательность, которая соответствует РВ, на replacement. Также использует методы appendReplacement и appendTail.

quoteReplacement(String s) Возвращает строку, в которой коса черта (' \ ') и знак доллара (' $ ') будут лишены особого смысла.

PatternSyntaxException

Неконтролируемое (unchecked) исключение, возникает при синтаксической ошибке в регулярном выражении. В таблице ниже приведены все методы и их описание.

getDescription() Возвращает описание ошибки.

getIndex() Возвращает индекс строки, где была найдена ошибка в РВ

getPattern() Возвращает ошибочное РВ.

getMessage() getDescription() + getIndex() + getPattern()

**6. Локализация, интернационализация.**

Интернационализация (i18n) - это процесс разработки приложения такой структуры, при которой дополнение нового языка не требует перестройки и перекомпиляции (сборки) всего приложения. Достигается это за счет отдельного хранения данных интернационализации в виде файлов свойств, загружаемых приложением динамически в процессе работы.

Локализация (l10n) – это адаптация приложения к конкретному языку и региону путем перевода выводимых пользователю текстовых элементов и документации, а также определения данных времени, валют и др., согласно специфике данного региона.

Класс Locale

Класс Java java.util.Locale позволяет учесть особенности региональных представлений алфавита, символов, чисел и дат. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять.

Для некоторых стран региональные параметры устанавливаются с помощью констант, например: Locale.US, Locale.FRANCE.

Для всех остальных объект класса Locale нужно создавать с помощью конструктора, например: Locale rus = new Locale("ru", "RU");

Существует константа Locale.ROOT. Она представляет локаль, для которой язык, страна равны пустой строке(""). Эта локаль является базовой для всех остальных локалей. Используется для написания приложений, не зависящих от локали.

Определить текущий вариант региональных параметров можно следующим образом: Locale current = Locale.getDefault();

А можно и изменить для текущего экземпляра (instance) JVM: Locale.setDefault(Locale.CANADA)

Рассмотрим пример использования класса Locale:

import java.util.Locale;

public class LocaleDemo1 {

public static void main(String[] args) {

Locale fr = Locale.FRANCE;

Locale us = Locale.US;

Locale uk = new Locale("uk", "UA");

Locale.setDefault(Locale.CANADA);

Locale current = Locale.getDefault();

getLocaleInfo(current);

getLocaleInfo(fr);

getLocaleInfo(us);

getLocaleInfo(uk);

}

private static void getLocaleInfo(Locale current) {

System.out.println("Код региона: " + current.getCountry());

System.out.println("Название региона: " + current.getDisplayCountry());

System.out.println("Код языка региона: " + current.getLanguage());

System.out.println("Название языка региона: "

+ current.getDisplayLanguage());

System.out.println();

}

}

Класс ResourceBundle

Класс ResourceBundle вместе с классом Locale лежит в основе процесса интернационализации в языке Java.

Класс ResourceBundle предназначен для чтения данных из текстовых файлов свойств (расширение - properties). Каждый отдельный файл с расширением properties содержит информацию для отдельной локали.

В файлах свойств (\*.properties) информация должна быть организована по принципу:

#Комментарий

group1.key1 = value1

group1.key2 = value2

group2.key1 = value3…

Например:

label.button = submit

label.field = login

message.welcome = Welcome!

или

label.button = принять

label.field = логин

message.welcome = Добро пожаловать!

Рассмотрим правила выбора имени для properties файлов. Выбираем базовое имя для группы properties файлов. Например text. Добавляем к базовому имени через пробел код языка (uk) и код страны (UA).

Следующий список показывает возможный набор соответствующих ресурсов с базовым именем:

text.properties

text\_ru\_RU.properties

text\_uk\_UA.properties

text\_fr\_CA.properties

Чтобы выбрать определенный объект ResourceBundle, следует вызвать один из статических перегруженных методов getBundle(параметры).

Следующий фрагмент выбирает text объекта ResourceBundle для объекта Locale, который соответствует французскому языку и стране Канада:

Locale locale = new Locale("fr", "CA");

ResourceBundle rb = ResourceBundle.getBundle("text", locale);

Если объект ResourceBundle для заданного объекта Locale не существует, то метод getBundle() извлечет наиболее общий.

Если общее определение файла ресурсов не задано, то метод getBundle() генерирует исключительную ситуацию MissingResourceException. Чтобы это не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов, а именно: text.properties в дополнение к частным случаям вида:

text\_en\_US.properties

text\_ru\_RU.properties.

Пусть базовое имя ресурса baseName, и ресурс нам нужен для fr, CA. Locale по умолчанию определена через uk, UA. Мы вызываем ResourceBundle.getBundle(baseName, canadaLocale). В каком порядке будет организован поиск в properties файлах? Формируется следующий список строк (так называемые кандидаты в имена пакетов):

baseName + "\_"+ fr + "\_" + CA

baseName + "\_"+ fr

baseName + "\_"+ uk + "\_" + UA

baseName + "\_"+ uk

baseName

Рассмотрим пример использования класса ResourceBundle:

import java.io.UnsupportedEncodingException;

import java.util.Locale;

import java.util.ResourceBundle;

public class ResourceBundleDemo1 {

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

printInfo("", "");

printInfo("en", "US");

printInfo("uk", "UA");

}

private static void printInfo(String language, String country)

throws UnsupportedEncodingException {

Locale current = new Locale(language, country);

ResourceBundle rb = ResourceBundle.getBundle("text", current);

String s1 = rb.getString("str1");

System.out.println(s1);

String s2 = rb.getString("str2");

System.out.println(s2);

System.out.println();

}

}

В классе ResourceBundle определен ряд полезных методов:

getKeys() - возвращает объект Enumeration, который применяется для последовательного обращения к элементам.

keySet() – возвращает множество Set всех ключей.

getString(String key) - извлекается конкретное значение по конкретному ключу.

boolean containsKey(String key) - проверить наличие ключа в файле.

В следующем примере используется метод keySet() класса ResourceBundle:

public class ResourceBundleDemo2 {

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

printInfo("", "");

printInfo("en", "US");

printInfo("uk", "UA");

}

private static void printInfo(String language, String country)

throws UnsupportedEncodingException {

Locale current = new Locale(language, country);

ResourceBundle rb = ResourceBundle.getBundle("text", current);

for (String key : rb.keySet()) {

String value = rb.getString(key);

System.out.println(value);

}

System.out.println();

}

}

Класс NumberFormat

Класс NumberFormat языка Java используется для форматирования чисел.

Чтобы получить объект класса для форматирования в национальном стандарте по умолчанию, используются следующие методы:

NumberFormat.getlnstance()

NumberFormat.getNumberlnstance() - идентичен getInstance()

NumberFormat.getCurrencylnstance()

NumberFormat.getPercentlnstance()

Чтобы получить объект класса для форматирования в других национальных стандартах используются следующие методы:

NumberFormat.getlnstance(Locale locale)

NumberFormat.getNumberlnstance(Locale locale) - идентичен getInstance(Locale locale)

NumberFormat.getCurrencylnstance(Locale locale)

NumberFormat.getPercentlnstance(Locale locale)

Рассмотрим пример использования класса NumberFormat:

import java.text.NumberFormat;

import java.util.Locale;

public class NumberFormatDemo1 {

public static void main(String[] args) {

double number = 123.4567;

Locale locFR = new Locale("fr");

NumberFormat numberFormat1 = NumberFormat.getInstance();

System.out.println("Число в текущей локали: " + numberFormat1.format(number));

NumberFormat numberFormat2 = NumberFormat.getInstance(locFR);

System.out.println("Число во французской локали: " + numberFormat2.format(number));

NumberFormat numberFormat3 = NumberFormat.getCurrencyInstance();

System.out.println("Денежная единица в текущей локали: " + numberFormat3.format(number));

NumberFormat numberFormat4 = NumberFormat.getCurrencyInstance(locFR);

System.out.println("Денежная единица во французской локали: " + numberFormat4.format(number));

NumberFormat numberFormat5 = NumberFormat.getPercentInstance();

System.out.println("Процент в текущей локали: " + numberFormat5.format(number));

NumberFormat numberFormat6 = NumberFormat.getPercentInstance(locFR);

System.out.println("Процент во французской локали: " + numberFormat6.format(number));

}

}

Другие полезные методы NumberFormat класса:

setMaximumFractionDigits(int digits) - устанавливает максимальное количество цифр после десятичной точки в форматируемом объекте. Последняя отображаемая цифра округляется.

setMaximumlntegerDigits(int digits) - устанавливает максимальное количество цифр перед десятичной точкой в форматируемом объекте. Используйте этот метод с предельной осторожностью. Если вы зададите слишком мало цифр, число будет просто усечено, и результат станет совершенно неправильным!

setMinimumFractionDigits(int digits) - устанавливает минимальное количество цифр после десятичной точки в форматируемом объекте. Если количество цифр в дробной части числа меньше минимального, то на экран выводятся замыкающие нули.

setMinimumlntegerDigits(int digits) - устанавливает минимальное количество цифр перед десятичной точкой в форматируемом объекте. Если количество цифр в целой части числа меньше минимального, то на экран выводятся ведущие нули.

getMaximumFractionDigits() - возвращает максимальное количество цифр после десятичной точки в форматируемом объекте.

getMinimumFractionDigits() - возвращает минимальное количество цифр после десятичной точки в форматируемом объекте.

Метод parse() класса NumberFormat преобразует строку к числу. Если перед вызовом метода parse() вызвать метод setParseIntegerOnly(true), как показано в следующем примере, то преобразовываться будет только целая часть числа.

**7. Потоки ввода-вывода. Java NIO.**

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета java.io.

Ключевым понятием здесь является понятие потока. Хотя понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы будем говорить о потоке (stream), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. У нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-ввывод в консоли мы будем решать в Java с помощью потоков.

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, - потоком вывода. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: InputStream (представляющий потоки ввода) и OutputStream (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы Reader (для чтения потоков символов) и Writer (для записи потоков символов).

Потоки байтов

Класс InputStream

Класс InputStream является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его основные методы:

int available(): возвращает количество байтов, доступных для чтения в потоке

void close(): закрывает поток

int read(): возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1

int read(byte[] buffer): считывает байты из потока в массив buffer. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число -1

int read(byte[] buffer, int offset, int length): считывает некоторое количество байтов, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента buffer[offset]. Метод возвращает число успешно прочитанных байтов.

long skip(long number): пропускает в потоке при чтении некоторое количество байт, которое равно number

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

void close(): закрывает поток

void flush(): очищает буфер вывода, записывая все его содержимое

void write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b

void write(byte[] buffer): записывает в выходной поток массив байтов buffer.

void write(byte[] buffer, int offset, int length): записывает в выходной поток некоторое число байтов, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

Абстрактные классы Reader и Writer

Абстрактный класс Reader предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

absract void close(): закрывает поток ввода

int read(): возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1

int read(char[] buffer): считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

int read(CharBuffer buffer): считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

absract int read(char[] buffer, int offset, int count): считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count

long skip(long count): пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Класс Writer определяет функционал для всех символьных потоков вывода. Его основные методы:

Writer append(char c): добавляет в конец выходного потока символ c. Возвращает объект Writer

Writer append(CharSequence chars): добавляет в конец выходного потока набор символов chars. Возвращает объект Writer

abstract void close(): закрывает поток

abstract void flush(): очищает буферы потока

void write(int c): записывает в поток один символ, который имеет целочисленное представление

void write(char[] buffer): записывает в поток массив символов

absract void write(char[] buffer, int off, int len) : записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off

void write(String str): записывает в поток строку

void write(String str, int off, int len): записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off

Функционал, описанный классами Reader и Writer, наследуется непосредственно классами символьных потоков, в частности классами FileReader и FileWriter соответственно, предназначенными для работы с текстовыми файлами.

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.

Через конструктор класса FileOutputStream задается файл, в который производится запись. Класс поддерживает несколько конструкторов:

FileOutputStream(String filePath)

FileOutputStream(File fileObj)

FileOutputStream(String filePath, boolean append)

FileOutputStream(File fileObj, boolean append)

Файл задается либо через строковый путь, либо через объект File. Второй параметр - append задает способ записи: eсли он равен true, то данные дозаписываются в конец файла, а при false - файл полностью перезаписывается

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Для автоматического закрытия файла и освобождения ресурса объект FileOutputStream создается с помощью конструктции try...catch.

Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс FileInputStream, который является наследником класса InputStream и поэтому реализует все его методы.

Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

FileInputStream(String fileName) throws FileNotFoundException

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение FileNotFoundException.

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов, то есть для записи и чтения байтов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.