1. История развития языков программирования. ЯП как инструмент. Специализированные ЯП. Примеры.

20-е гг. XIX в. – изобретена выч машина (Бэббидж), перфокарты, Ада Лавлейс разработала приемы и конструкции прогр-я.

Дж. Моучли – система кодирования команд в виде мат формул

Г.Хоппер – разработаны понятия подпрограммы, отладка, компилятор

1954 – Фортран (FORTRAN — от FORmula TRANshlor - переводчик формул)

В конце 1950-х гг. плодом международного сотрудничества в области программирования явился Алгол (ALGOL — от ALGOrithmic Language — алгоритмический язык). Алгол предназначен для записи алгоритмов, которые строятся в виде последовательности процедур, применяемых для решения поставленных задач

В 1958 г. появился компилятор FLOW-MATIC. В отличие от Фортрана — языка для научных приложений — FL0W-MAT1C был первым языком для задач обработки коммерческих данных. Работы в этом направлении привели к созданию в г. языка Кобол (COBOL — Common Business Oriented Language) — общего зыка, ориентированного на деловые задачи.

В середине 1960-х гг. сотрудники математического факультета Дартмутского кол-Томас Курц и Джон Кемени создали специализированный язык программирования, который состоял из простых слов английского языка. Новый язык звали «универсальным символическим кодом для начинающих» (Beginner's All-urpose Symbolic Instruction Code, или сокращенно, BASIC). Годом рождения нового языка можно считать 1964 г.

В 1960-е гг. были предприняты попытки преодолеть эту «разносорт-иосицу» путем создания универсального языка программирования. Первым детищем этого направления стал PL/I (Programm Language One), созданный в 1967 г. В дальнейшем на эту роль претендовал АЛГОЛ-68 (1968 г.). Предполагалось, что подобные языки будут развиваться и совершенствоваться и вытеснят все остальные. Однако и одна из этих попыток на сегодняшний день не увенчалась успехом

В начале 1970-х гг. языка Паскаль швейцарским ученым Никлаусом Виртом. Язык Паскаль первоначально разрабатывался как учебный

70-е гг. также подарили нам универсальный язык С. Его авторами были Кен Томпсон и Денис Ритчи. Язык пользовался повышенной популярностью у системных программистов, первое ядро ОС UNIX было разработано именно на нем.

В 1982 году стандарт С поступил в разработку в ANSI, получившийся вариант был принят в 1990 году. На основе этого языка были разработаны современные языки Java и C++.

90-е – появление скриптовых языков Perl, Python, PHP, Ruby.

В настоящее время языки программирования применяются в самых различных областях человеческой деятельности, таких как:

научные вычисления (языки C++, FORTRAN, Java);

системное программирование (языки C++, Java);

обработка информации (языки C++, COBOL, Java);

искусственный интеллект (LISP, Prolog);

издательская деятельность (Postscript, TeX);

удаленная обработка информации (Perl, PHP, Java, C++);

описание документов (HTML, XML).

Специализированные языки программирования

Специализированные языки предназначены для решения задач одного, максимум нескольких, видов задач (например, работы с базами данных, web-программирования или написание скриптов для администрирования операционных систем).

В их число входит язык RPG, используемый для генерации деловых отчетов, язык APT, созданный для управления программируемыми устройствами, язык GPSS, разработанный для моделирования систем.

Специализированные языки программирования в системах управления базами данных:

В локальных и файл – серверных СУБД: Microsoft Visual FoxPro (в одноименной СУБД), Visual Basic for Application (СУБД Access).

Клиент – серверных промышленных СУБД: PL-SQL (СУБД Oracle), Transact – SQL (СУБД Microsoft SQL Server).

К универсальным языкам можно отнести языки Visual Basic, Visual C++, Visual C++.Net, Java, Delphi, Borland C#, Borland C++ Builder.

2. Парадигмальный подход в прог-и. Сравнение и анализ основных парадигм.

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Структурное - методология разработки ПО, в основе лежит представление программы в виде иерархической структуры. Вся программа разбивается на отдельные блоки, элементарными блоками являются: последовательность, ветвление, циклический блок. Языки: Fortran, C, Pascal, Basic.

Преимущества: Повторное использование ранее написанных блоков кода. Высокая степень независимости программы от типа вычислительной машины. Повышение эффективности труда разработчиков, в том числе и за счет абстрагирования от конкретных деталей аппаратного обеспечения.

Недостатки: Некоторая потеря в скорости вычислений.

Применение: Создание операционных систем и системных программ. Разработка небольших пользовательских приложений. Научные расчеты.

Функциональное - в которой любой процесс вычисления трактуется как вычисление значения функции в ее математическом понимании. Вычисление ведется от исходных данных, которые в свою очередь могут являться результатами работы предыдущих функций. Функция будет запущена тогда и только тогда, когда на ее входе будет иметься полный набор исходных данных. Порядок выполнения функции определяется порядком прихода аргументов. Языки: LISP, Scheme, Haskell.

Преимущества: Автоматическое динамическое распределение памяти компьютера для хранения данных. Программист получает возможность абстрагироваться от представления данных и других рутинных операциях и сосредоточиться на предметной области.

Недостатки: Нелинейная структура программы, следовательно, такое программирование сложно для понимания. Относительно невысокая эффективность вычислений.

Применение: Обработка рекурсивных структур данных. Обработка символьной информации.

Логическое программирование - парадигма, основанная на автоматическом доказательстве теорем на базе заданных фактов и правил вывода. Оно основано на аппарате математической логики. Язык: Prolog.

Преимущества: Возможность откатов, т.е. возвращения к предыдущей подцели при отрицательном результате одного из вариантов в процессе поиска решения. Это избавляет от

необходимости поиска решения путем полного перебора вариантов и увеличивает эффективность реализации.

Недостатки: Узкий класс решаемых задач.

Применение: Эмуляция искусственного интеллекта. Разработка экспертных систем.

Автоматное программирование - при использовании которой программа или ее фрагмент определяется как модель формального автомата. В зависимости от сложности задачи в автоматном программирование могут использоваться конечные автоматы или автоматы более сложной структуры.

Объектно-ориентированное программирование - основные компоненты: объект и класс, а программа представляет собой процесс взаимодействия объектов, функционирующих на основе имеющихся у них методов, обладающих поведением и возможностью обмена сообщениями. На базе ООП может быть построено прототипное программирование. В этом случае отсутствует понятие класса, а наследование определяется как механизм клонирования или создания прототипов, то есть полной копии уже существующего в программе объекта. Языки: Java, C++, Ruby.

Преимущества: Смысловая близость к предметной области любой структуры и назначения. Механизм наследования свойств и методов позволяет строить производные понятия на основе базовых, создавая тем самым модели предметной области. Использование ранее созданных библиотек классов позволяет сэкономить время при разработке новых программных продуктов. Полиморфизм, заложенный в ООП, обеспечивает гибкость и универсальность программного обеспечения. Удобство разработки ПО группой лиц.

Недостатки: Сложность полной формализации реального мира создает в дальнейшем трудности тестирования созданного ПО.

Применение: Разработка больших пользовательских приложений.

Событийно-ориентированное программирование - при которой выполнение программы определяется срабатыванием некоторых событий, генерируемых пользователем либо другими элементами системы. События представляют собой потоки, организующие взаимодействие между компонентами программы с помощью исполнительных механизмов.

Агентно-ориентированное программирование - парадигма, в которой основная концепция - работа параллельных независимых агентов. Для каждого агента определяется поведение, которое зависит от среды, в которую агент помещается в текущий момент времени. Агент воспринимает среду через набор датчиков и регулирует собственное поведение в зависимости от полученное информации за счет собственных исполнительных организмов. Для разработки на языке Java используются системы JATLite, Agent Builder, JAFMAS.

Языки сценариев - Программа представляет собой совокупность возможных сценариев обработки данных. Выбор конкретного сценария зависит от наступления того или иного события.

Преимущества: Основные достоинства данного класса языков программирования унаследованы от объектно-ориентированных языков. Легкость использования с инструментальными средствами автоматизированного проектирования и быстрого создания ПО.

Недостатки: Сложность тестирования. Большое количество вариантов, которые требуется предусмотреть. Большая вероятность побочных эффектов.

Применение: Интернет технологии

Примеры языков: JavaScript, Python, PHP.

3. Императивное и декларативное прог-е. Их преимущества и недостатки. Языки функционального прогр-я.

Императи́вное программи́рование — это парадигма программирования, для которого характерно следующее: в исходном коде программы записываются инструкции (команды); инструкции должны выполняться последовательно; при выполнении инструкции данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти; данные, полученные при выполнении инструкции, могут записываться в память.

При императивном подходе к составлению кода широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе.

Основные черты императивных языков: использование именованных переменных; использование оператора присваивания; использование составных выражений; использование подпрограмм; использование циклов.

+: легкость написания, выше скорость работы,

-: ошибки, связанные с присваиванием.

Декларативные (функциональные и логические) языки

Программный код на декларативном языке программирования представляет собой описание действий, которые можно осуществлять, а не последовательный набор команд.

Преимущества: Легче формализуется математическими средствами. Как следствие, программы проще тестировать, т.е. проверять на наличие ошибок. Высокая степень абстракции.

Недостатки: Снижение скорости работы программы.

Применение: Доказательство теорем. Возможность обработки разнородных данных.

Наиболее известными языками функционального программирования являются:

Lisp — и множество его диалектов, наиболее современные из которых:

Scheme, Clojure, Common Lisp

Erlang — функциональный язык с поддержкой процессов.

Elixir

APL — предшественник современных научных вычислительных сред, таких как MATLAB.

ML (из ныне используемых диалектов известны Standard ML и Objective CAML).

F# — функциональный язык семейства ML для платформы .NET

Scala

Miranda (который впоследствии дал развитие языку Haskell).

Nemerle — гибридный функционально/императивный язык.

XSLT и XQuery

Haskell — чистый функциональный.

Ещё не полностью функциональные изначальные версии и Лиспа, и APL внесли особый вклад в создание и развитие функционального программирования. Более поздние версии Lisp, такие как Scheme, а также различные варианты APL поддерживали все свойства и концепции функционального языка. Как правило, интерес к функциональным языкам программирования, особенно чисто функциональным, был скорее научный, нежели коммерческий. Однако, такие примечательные языки как Erlang, OCaml, Haskell, Scheme (после 1986) а также специфические R (статистика), Wolfram

(символьная математика), J и K (финансовый анализ), и XSLT (XML) находили применение в индустрии коммерческого программирования.

4. Компилируемые и интерпретируемые языки. Преимущества и недостатки. Использование виртуальных машин.

Компилируемые языки

Программа на компилируемом языке при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в исполняемый файл, который может быть запущен на выполнение как отдельная программа. Другими словами, компилятор переводит программу с языка высокого уровня на низкоуровневый язык, понятный процессору сразу и целиком, создавая при этом отдельную программу

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем при каждом изменении текста программы требуется ее перекомпиляция, что создает трудности при разработке. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция.

Компилируемые языки обычно позволяют получить более быструю и, возможно, более компактную программу, и поэтому применяются для создания часто используемых программ.

Примерами компилируемых языков являются Pascal, C, C++, Erlang, Haskell, Rust, Go, Ada.

Интерпретируемые языки

Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ее текст без предварительного перевода. При этом программа остается на исходном языке и не может быть запущена без интерпретатора. Интерпретатор переводит на машинный язык прямо во время исполнения программы.

Программы на интерпретируемых языках можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий. Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без дополнительной программы-интерпретатора.

Примерами интерпретируемых языков являются PHP, Perl, Ruby, Python, JavaScript. К интерпретируемым языкам также можно отнести все скриптовые языки.

Многие языки в наши дни имеют как компилируемые, так и интерпретируемые реализации, сводя разницу между ними к минимуму. Некоторые языки, например, Java и C#, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету». Для Java байт-код исполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# — Common Language Runtime.

5. Автоматическая генерация документа. Javadoc.

Генератор документации — программа или пакет программ, позволяющая получать документацию, предназначенную для программистов (документация на API) и/или для конечных пользователей системы, по особым образом комментированному исходному коду и, в некоторых случаях, по исполняемым модулям (полученным на выходе компилятора).

Обычно генератор анализирует исходный код программы, выделяя синтаксические конструкции, соответствующие значимым объектам программы (типам, классам и их членам/свойствам/методам, процедурам/функциям и т. п.). В ходе анализа также используется мета-информация об объектах программы, представленная в виде документирующих комментариев. На основе всей собранной информации формируется готовая документация, как правило, в одном из общепринятых форматов — HTML, HTMLHelp, PDF, RTF и других.

javadoc — это генератор документации в HTML-формате из комментариев исходного кода Java и определяет стандарт для документирования классов Java. Для создания доклетов и тэглетов, которые позволяют программисту анализировать структуру Java-приложения, javadoc также предоставляет API. В каждом случае комментарий должен находиться перед документируемым элементом.

/** комментирование документации */

С помощью утилиты javadoc, входящей в состав JDK, комментарий документации можно извлекать и помещать в HTML файл. Утилита javadoc позволяет вставлять HTML тэги и использовать специальные ярлыки (дескрипторы) документирования. HTML тэги заголовков не используют, чтобы не нарушать стиль файла, сформированного утилитой.

Дескрипторы javadoc, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ * игнорируется). Дескрипторы, начинающиеся с фигурной скобки, например {@code}, называются встроенными и могут применяться внутри описания.

Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом.

javadoc дескрипторы : @author, @version, @since, @see, @param, @return

Дескриптор	Применение	Описание
@author	Класс, интерфейс	Автор
@version	Класс, интерфейс	Версия. Не более одного дескриптора на класс
@since	Класс, интерфейс, поле, метод	Указывает, с какой версии доступно
@see	Класс, интерфейс, поле, метод	Ссылка на другое место в документации
@param	Метод	Входной параметр метода
@return	Метод	Описание возвращаемого значения
@exception имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@throws имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@deprecated	Класс, интерфейс,	Описание устаревших блоков кода

	поле, метод	
{@link reference}	Класс, интерфейс, поле, метод	Ссылка
{@value}	Статичное поле	Описание значения переменной

Форма документирования кода

Документирование класса, метода или переменной начинается с комбинации символов /** , после которого следует тело комментариев; заканчивается комбинацией символов */.

В тело комментариев можно вставлять различные дескрипторы. Каждый дескриптор, начинающийся с символа '@' должен стоять первым в строке. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Встроенные дескрипторы (начинаются с фигурной скобки) можно помещать внутри любого описания.

```
/**
 * Класс продукции со свойствами <b>maker</b> и <b>price</b>.
 * @autor Киса Воробьянинов
 * @version 2.1
*/
class Product
{
   /** Поле производитель */
   private String maker;

   /** Поле цена */
   public double price;

   /**
    * Конструктор - создание нового объекта
    * @see Product#Product(String, double)
    */
   Product()
   {
      setMaker("");
      price=0;
   }

   /**
    * Конструктор - создание нового объекта с определенными значениями
    * @param maker - производитель
    * @param price - цена
    * @see Product#Product()
    */
   Product(String maker,double price){
      this.setMaker(maker);
      this.price=price;
   }

   /**
    * Функция получения значения поля {@link Product#maker}
    * @return возвращает название производителя
    */
   public String getMaker() {
      return maker;
   }
```

```
    /**
     * Процедура определения производителя {@link Product#maker}
     * @param maker - производитель
     */
    public void setMaker(String maker) {
        this.maker = maker;
    }
}
```

6. Рефакторинг программного кода. Средства автоматизации рефакторинга. Оптимизация и обфускация программного кода.

Рефа́кторинг — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Рефакторинг улучшает композицию программного обеспечения, облегчает понимание программного обеспечения, помогает найти ошибки, позволяет быстрее писать программы.

Рефакторинг следует отличать от оптимизации производительности. Как и рефакторинг, оптимизация обычно не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу[3].

С другой стороны, нужно отличать рефакторинг от реинжиниринга, который осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют реинжиниринг.

Приемы рефакторинга

Изменение сигнатуры метода (Change Method Signature)

Инкапсуляция поля (Encapsulate Field)

Выделение класса (Extract Class)

Выделение интерфейса (Extract Interface)

Выделение локальной переменной (Extract Local Variable)

Выделение метода (Extract Method)

Генерализация типа (Generalize Type)

Встраивание (Inline)

Введение параметра (Introduce Parameter)

Подъём метода (Pull Up Method)

Спуск метода (Push Down Method)

Переименование метода (Rename Method)

Перемещение метода (Move Method)

Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)

Замена наследования делегированием (Replace Inheritance with Delegation)

Замена кода типа подклассами (Replace Type Code with Subclasses)

В данный момент, IDE практически все крупных производителей имеют в своем арсенале базовый набор методов рефакторинга. Наиболее распространенными классами автоматизированных методов являются следующие:

Изменение имени и физической организации кода - переименование полей, переменных, классов, интерфейсов, перенос пакетов и классов;

Изменение логической организации кода на уровне класса – преобразование класса вложенный<->верхнего уровня, преобразование класс<->интерфейс, перемещение методов в подкласс или суперкласс;

Изменение кода внутри класса – перемещение переменных из методов в класс, преобразование участка кода в метод.

И так, рынок Java IDE с поддержкой рефакторинга представляют следующие:

Eclipse Foundation с Eclipse 3.1.2;

JetBrains с IntelliJ Idea 5.1;

Sun с NetBeans 5.0;

Oracle с JDeveloper 10.1.3;

Borland с JBuilder 2006.

Сторонними разработчиками предлагаются несколько плагинов для вышеперечисленных средств, причем большинство из них могут интегрироваться не с одной, а со многими средами.

Вот некоторые из них:

RefactorIt;

JFactor;

JRefactory.

Обфуска́ция или запутывание кода — приведение исходного текста или исполняемого кода программы к виду, сохраняющему её функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции.

«Запутывание» кода может осуществляться на уровне алгоритма, исходного текста и/или ассемблерного текста. Для создания запутанного ассемблерного текста могут использоваться специализированные компиляторы, использующие неочевидные или недокументированные возможности среды исполнения программы. Существуют также специальные программы, производящие обфускацию, называемые обфускаторами.

Цели обфускации

Затруднение декомпиляции/отладки и изучения программ с целью обнаружения функциональности.

Затруднение декомпиляции проприетарных программ с целью предотвращения обратной разработки или обхода DRM и систем проверки лицензий.

Оптимизация программы с целью уменьшения размера работающего кода и (если используется некомпилируемый язык) ускорения работы.

Демонстрация неочевидных возможностей языка и квалификации программиста (если производится вручную, а не инструментальными средствами).

Оптимизация программного кода — это модификация программ с целью улучшения их характеристик, таких как производительности или компактности, — без изменения функциональности.

На практике используется весьма широкий набор машинно-независимых оптимизирующих преобразований, что связано с большим разнообразием неоптимальностей. К ним относятся:

• разгрузка участков повторяемости - это такой способ оптимизации, который состоит в вынесении вычислений из многократно исполняемых участков программы на участки программы, редко исполняемые. К этому виду преобразования относятся различные чистки зон, тел циклов и тел рекурсивных процедур, когда инвариантные по результату выполнения выражения, исполняемые при каждом прохождении участка повторяемости, выносятся из него. Если размещение осуществляется перед входом в участок повторяемости, то эту ситуацию называют чисткой вверх, если же за выходом из участка повторяемости, то чисткой вниз

• упрощение действий - этот способ оптимизации ориентирован на улучшение программы за счет замены групп (как правило, удаленных друг от друга) вычислений на группу вычислений, дающий тот же результат с точки зрения всей программы, но имеющих меньшую сложность.

• чистка программы - данный способ повышает качество программы за счет удаления из нее ненужных объектов и конструкций. Набор преобразований этого типа включает в себя следующие оптимизации: удаление идентичных операторов, удаление из программы операторов, недостижимых по управлению от начального, удаление несущественных операторов, то есть операторов не влияющих на результат программы, удаление процедур, к которым нет обращений, удаление неиспользуемых переменных и другие.

• экономия памяти и оптимизация работы с памятью - улучшения быстродействия возможно за счет уменьшения объема памяти, отводимой под информационные объекты программы в каждом ее исполнении.

• реализация действий - это способ повышения быстродействия программы за счет выполнения определенных ее вычислений на этапе трансляции.

• сокращение программы и другие методы.

Системы контроля версий (СКВ). Subversion и Git

Общие положения

Система контроля версиями — программное обеспечение для облегчения работы с изменяющейся информацией.

Широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. А вообще, везде, где ведется работа с большим кол-вом непрерывно изменяющихся электронных документов.

Основные функции:

- Хранение **нескольких версий** одного и того же документа
- Возвращение к более ранним версиям
- Создание **разных вариантов** одного документа, т. н. **ветки**, с общей историей изменений до точки ветвления и с разными — после неё.
- Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле.

- **Журнал изменений**, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии.
- **Контроль прав доступа** пользователей: разрешение/запрещение чтения или изменения данных, в зависимости от того, кто запрашивает это действие.

Словарь определений:

- **blame** Понять, кто внёс изменение.
- **branch** Ветвь — направление разработки, независимое от других. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные — после неё.
- **changeset, changelist, activity** Набор изменений. Представляет собой поименованный набор правок, сделанных в локальной копии для какой-то общей цели.
- **check-in, commit, submit** Создание новой версии, фиксация изменений.
- **check-out, clone** Извлечение документа из хранилища и создание рабочей копии.
- **head** Основная версия — самая свежая версия для ветви/ствола, находящаяся в хранилище. Сколько ветвей, столько основных версий.
- **merge, integration** Слияние — объединение независимых изменений в единую версию документа. Осуществляется, когда два человека изменили один и тот же файл или при переносе изменений из одной ветки в другую.
- **pull, update** Получить новые версии из хранилища.
- **push** Залить новые версии в хранилище. Многие распределённые СКВ (Git, Mercurial) предполагают, что commit надо давать каждый раз, когда программист выполнил какую-то законченную функцию. А залить — когда есть интернет и другие хотят ваши изменения. Commit обычно не требует ввода имени и пароля, а push — требует.
- **repository, depot** Хранилище документов — место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.
- **revision** Версия документа. Системы управления версиями различают версии по номерам, которые назначаются автоматически.
- **tag, label** Метка, которую можно присвоить определённой версии документа. Метка представляет собой символическое имя для группы документов, причём метка описывает не только набор имён файлов, но и версию каждого файла.
- **trunk, mainline, master** Ствол — основная ветвь разработки проекта. Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол; если требуется серьёзное изменение, способное привести к нестабильности, создаётся ветвь, которая сливается со стволом, когда нововведение будет в достаточной мере испытано.
- **update, sync, switch** Синхронизация рабочей копии до некоторого заданного состояния хранилища. Можно и до самой последней версии, и до какой-лиюо предыдущей.
- **working copy** Рабочая (локальная) копия документов.

Централизованная модель

Имеется **единое хранилище документов**, управляемое специальным сервером, который и выполняет большую часть функций по управлению версиями.

Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, т. н. «**рабочая копия**».

Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище.

В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время.

Subversion

Subversion – свободная централизованная система управления версиями.

Возможности:

- Хранение **полной истории изменений,** отслеживаемых в централизованном хранилище.
- Поддержка переноса изменений между копиями объектов, в том числе полного слияния копий (в рабочей копии; без объединения истории)
- Эмуляция ветвей и меток (просто копирование репозитория)
- Слияние ветвей с автоматическим определением и переносом изменений
- История изменений и копии объектов хранятся в виде связанных **разностных копий**
- Поддержка многопользовательской работы с хранилищем
- Фиксации изменений в хранилище – атомарные операции
- Сервер и клиент обмениваются **только различиями** между локальной копией и хранилищем
- Хорошо работает и с **текстовыми**, и **двоичными** файлами
- Вывод клиента командной строки одинаково удобен и для чтения, и для разбора программами

Рабочий цикл:

- Обновление рабочей копии из хранилища (`svn update`) или её создание (`svn checkout`).
- Изменение рабочей копии. Изменения директорий и информации о файлах производится средствами Subversion, в изменении же содержимого файлов Subversion никак не задействован — изменения производятся программами, предназначенными для этого (текстовые редакторы, средства разработки и т. п.):
    - новые файлы и директории нужно добавить (команда `svn add`), то есть передать под управление версиями;
    - если файл или директорию в рабочей копии нужно удалить, переименовать, переместить или скопировать, необходимо использовать средства Subversion (`svn mkdir, svn delete, svn move, svn copy`*);*
    - просмотр состояния рабочей копии и локальных (ещё не зафиксированных) изменений (`svn info, svn status, svn diff`);
    - любые локальные изменения, если они признаны неудачными, можно откатить (`svn revert`).
- При необходимости — дополнительное обновление, для получения изменений, зафиксированных в хранилище другими пользователями и слияния этих изменений со своими (`svn update`).
- Фиксация своих изменений (и/или результатов слияния) в хранилище (`svn commit`).

Большой косяк Subversion: информация, однажды помещённая в хранилище Subversion, остаётся там навсегда: файл можно удалить в текущей ревизии, но всегда есть возможность получить из хранилища одну из предыдущих ревизий, в которых файл существовал. Хотя сохранность прошлых ревизий и является, собственно, целью использования систем управления версиями, иногда бывает необходимо полностью удалить из хранилища информацию, попавшую туда по ошибке. В Subversion не предусмотрено для этого никакого штатного способа.

Распределенная модель

Нет нужды в централизованном хранилище: вся история изменения документов хранится на каждом компьютере, в локальном хранилище, и при необходимости отдельные фрагменты истории локального хранилища синхронизируются с аналогичным хранилищем на другом компьютере.

Когда пользователь такой системы выполняет обычные действия, такие как извлечение определённой версии документа, создание новой версии и тому подобное, он работает со своей локальной копией хранилища. По мере внесения изменений, хранилища, принадлежащие разным разработчикам, начинают различаться, и возникает необходимость в их синхронизации. Такая синхронизация может осуществляться с помощью обмена **патчами** между пользователями.

Описанная модель логически близка созданию отдельной ветки для каждого разработчика в классической системе управления версиями. Отличие состоит в том, что до момента синхронизации другие разработчики этой ветви не видят. Пока разработчик изменяет только свою ветвь, его работа не влияет на других участников проекта и наоборот. По завершении обособленной части работы, внесённые в ветви изменения сливают с основной (общей) ветвью. Как при слиянии ветвей, так и при синхронизации разных хранилищ возможны конфликты версий. На этот случай во всех системах предусмотрены те или иные методы обнаружения и разрешения конфликтов слияния.

Git

Git – свободная распределённая система управления версиями.

Ядро Git представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в **текстовых файлах конфигурации**. Как следствие – легкая портируемость на другие платформы.

Для каждого объекта в репозитории вычисляется **SHA-1-хеш**, и именно он становится именем файла.

Рабочий цикл:

- Обновление рабочей копии из удаленного репозитория (`git pull`) или её создание (`git clone`).
- Изменение рабочей копии:
  - переключение на другую ветку (`git checkout`)
  - новые файлы и директории нужно добавить (команда `git add`), то есть передать под управление версиями;
  - если файл или директорию в рабочей копии нужно удалить, переименовать, переместить или скопировать, необходимо использовать средства Subversion (`git mv, git rm, git mkdir`);
  - просмотр состояния рабочей копии и локальных (ещё не зафиксированных) изменений (`git info, git status, git diff`);
  - фиксация изменений (`git commit`)
  - любые локальные изменения, если они признаны неудачными, можно откатить (`git revert`).
- Отправление зафиксированных изменений в удаленный репозиторий (`git push`).

Практически все обычные операции с системой контроля версий, такие, как коммит и слияние, производятся только с **локальным репозиторием**. Удаленный репозиторий можно только синхронизировать с локальным как «вверх» (`push`), так и «вниз» (`pull`).

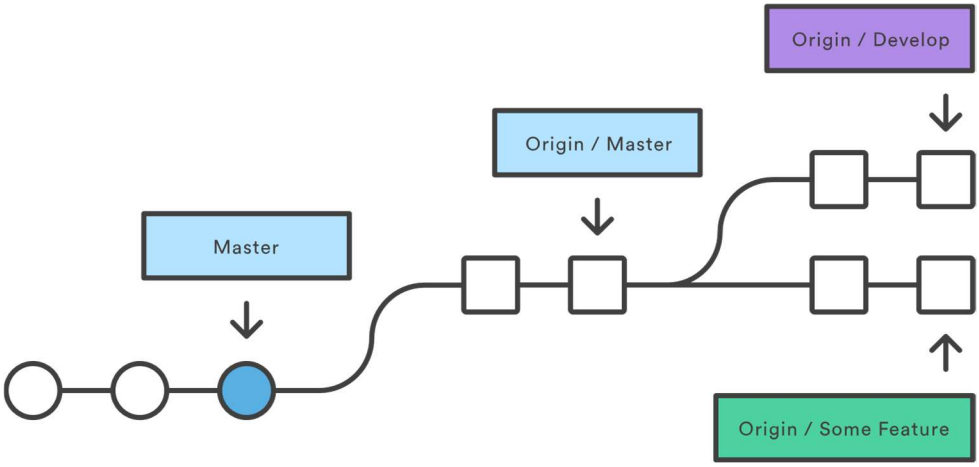
Типы объектов в репозитории Git:

- Файл – какая-то версия пользовательского файла

- Дерево – совокупность файлов из разных поддиректорий
- Коммит – дерево и некая дополнительная информация (например, родительский(е) коммит(ы), а также комментарий). Другими словами, какое-то действие

История изменений в Git хранится в виде **списка коммитов**, каждый коммит указывает на предыдущий. При создании нового коммита указатель текущей ветки перемещается на него.

Ветки, в отличие от Subversion, реализованы не полным копированием репозитория, а в виде отдельной цепочки коммитов. Каждая ветка определяется указателем на определенный коммит. При переключении на другую ветку происходит проход по цепочке коммитов до места разветвления, а потом выполняется обратный проход по другой цепочке до указателя на другую ветку.



Кружочки и квадратики – коммиты. Origin/Master, Origin/Develop, Origin/Some Feature – ветки (указатели на коммиты).

Автоматизация типовых операций. Apache Ant

Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует **специальные make-файлы** (обычно называются `Makefile`), в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Использование

```
$ make [ -f make-файл ] [ цель ] ...
```

make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной **цели**.

Стандартные цели для сборки:

- all — выполнить сборку пакета;
- install — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные каталоги);
- uninstall — удалить пакет (производит удаление исполняемых файлов и библиотек из системных каталогов);
- clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы, созданные в процессе компиляции);
- distclean — очистить все созданные при компиляции файлы и все вспомогательные файлы, созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива.

Makefile

make-файл состоит из **правил** и **переменных**.

Правила имеют следующий синтаксис:

```
цель1 цель2 ...: реквизит1 реквизит2 ...
        команда1
        команда2
        ...
```

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизитов. Строки, в которых записаны команды, должны начинаться с символа **табуляции**.

Синтаксис для определения переменных:

```
переменная = значение
```

Значением может являться произвольная последовательность символов, включая пробелы и обращения к значениям других переменных.

Обращение к переменной:

```
$(var_name)
```

Пример простого make-файла для программы из одного файла main.c:

```
OBJ = main.o
SRC = main.c
program: $(OBJ)
        cc -o program $(OBJ)
$(OBJ):
        cc -c $(SRC)
```

При вызове утилиты make будут выполнены следующие команды:

```
cc -c main.c
cc -o program main.o
```

Результат выполнения: исполняемый файл program.

Apache Ant

Утилита Ant полностью независима от платформы, требуется лишь рабочей среды Java — **JRE**. Отказ от использования команд операционной системы и формат XML обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством XML-сценария, (Build-файлом). В первую очередь этот файл содержит определение проекта (**property**), состоящего из отдельных **целей** (Targets). Цели содержат вызовы **команд-заданий** (Tasks). Каждое задание – **атомарная команда**, выполняющая некоторое элементарное действие.

Между целями могут быть определены зависимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится).

Конкретный набор целей и их взаимосвязи зависят от специфики проекта.

Ant позволяет определять собственные типы заданий путём создания Java-классов, реализующих определённые интерфейсы.

Часто применяемые задания (Tasks)

Код	Действие
javac	компиляция Java-кода
copy	копирование файлов
delete	удаление файлов и директорий
move	перемещение файлов и директорий
replace	замещение фрагментов текста в файлах
junit	автоматический запуск юнит-тестов
exec	выполнение внешней команды
zip	создание архива в формате Zip
cvs	выполнение CVS-команды
mail	отправка электронной почты
xslt	наложение XSLT-преобразования

Пример сценария

Думаю все делали

```
<?xml version="1.0"?>
<project default="build" basedir=".">
```

```
    <property name="name" value="AntBuildJar"/>
    <property name="src.dir" location="${basedir}/src"/>
    <property name="build" location="${basedir}/build"/>
    <property name="build.classes" location="${build}/classes"/>
    <path id="libs.dir">
        <fileset dir="lib" includes="**/*.jar"/>
    </path>
    <!-- Сборка приложения -->
    <target name="build" depends="clean" description="Builds the application">
        <!-- Создание директорий -->
        <mkdir dir="${build.classes}"/>

        <!-- Компиляция исходных файлов -->
        <javac srcdir="${src.dir}"
               destdir="${build.classes}"
               debug="false"
               deprecation="true"
               optimize="true" >
            <classpath refid="libs.dir"/>
        </javac>

        <!-- Копирование необходимых файлов -->
        <copy todir="${build.classes}">
            <fileset dir="${src.dir}" includes="**/*.*" excludes="**/*.java"/>
        </copy>

        <!-- Создание JAR-файла -->
        <jar jarfile="${build}/${name}.jar">
            <fileset dir="${build.classes}"/>
        </jar>
    </target>

    <!-- Очистка -->
    <target name="clean" description="Removes all temporary files">
        <!-- Удаление файлов -->
        <delete dir="${build.classes}"/>
    </target>
</project>
```

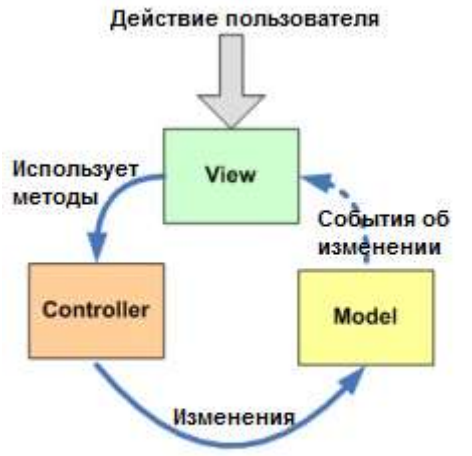
Альтернативы

- Apache Maven (Java)
- Gradle (Java)
- qmake (C, C++)
- Cmake (C, C++)

Технология MVC. Анализ, преимущества, недостатки

Model-View-Controller (MVC, «Модель-Представление-Контроллер») – **схема разделения данных** приложения, пользовательского интерфейса и управляющей логики на **три отдельных компонента**: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

Компоненты



Model-View-Controller

Модель

Под Моделью, обычно понимается часть, содержащая в себе **бизнес-логику приложения**. Модель должна быть **полностью независима** от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

Модель обладает следующими признаками:

- Модель — это бизнес-логика приложения;
- Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях;
- Для некоторых проектов модель — это просто слой данных (база данных, XML-файл);
- Для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения;

Представление

В обязанности Представления входит **отображение данных** полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает **доступом «только на чтение»** к данным.

Представление обладает следующими признаками:

- В представлении реализуется отображение данных, которые получаются от модели любым способом;
- В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

Примеры представления: HTML-страница, WPF форма, Windows Form.

Контроллер

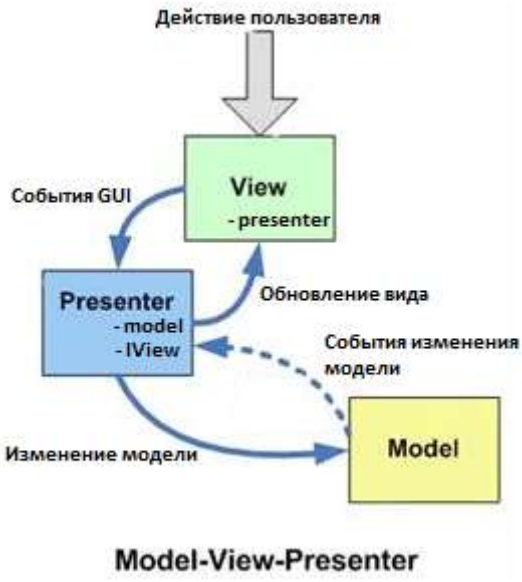
Основная идея этого паттерна в том, что и контроллер, и представление **зависят от модели**, но модель никак **не зависит** от этих двух компонент.

Контроллер обладает следующими признаками:

- Контроллер определяет, какое представление должно быть отображено в данный момент;
- События представления могут повлиять только на контроллер. Контроллер может повлиять на модель и определить другое представление.
- Возможно несколько представлений только для одного контроллера;

Вариации

MVP (Model-View-Presenter)



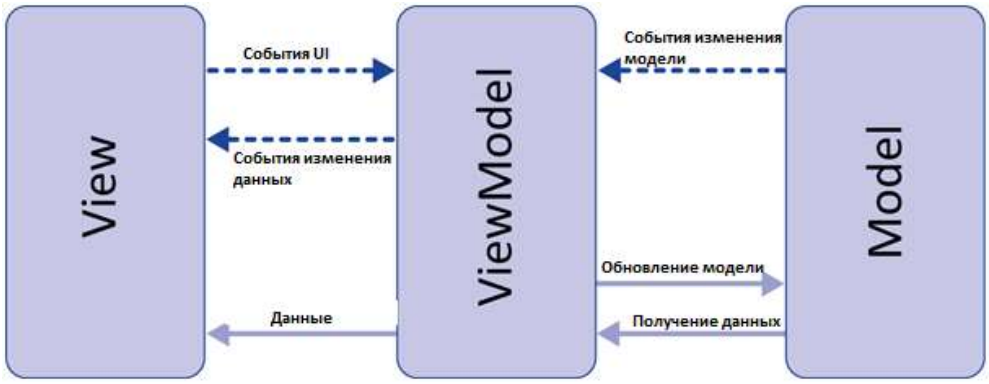
Model-View-Presenter

Данный подход позволяет создавать **абстракцию представления**. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу **изменяет модель**.

Признаки презентера:

- Двухсторонняя коммуникация с представлением;
- Представление взаимодействует напрямую с презентером, путем вызова соответствующих функций или событий экземпляра презентера;
- Презентер взаимодействует с View путем использования специального интерфейса, реализованного представлением;
- *Один экземпляр презентера связан с одним отображением.*

MVVM (Model-View-ViewModel)



Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

Признаки View-модели:

- Двухсторонняя коммуникация с представлением;
- View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели / модели
- View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings)
- Один экземпляр View-модели связан с одним отображением.

Критика

Недостатки

1. Необходимость использования большего количества ресурсов. Сложность обусловлена тем, что все три фундаментальных блока являются абсолютно независимыми и взаимодействуют между собой исключительно путем передачи данных.
2. Усложнен механизм разделения программы на модули. В концепции MVC наличие трех блоков (Model, View, Controller) прописано жестко. Соответственно каждый функциональный модуль должен состоять из трех блоков, что в свою очередь, несколько усложняет архитектуру функциональных модулей программы.
3. Усложнен процесс расширения функционала. Проблема очень схожа с вышеописанной. Недостаточно просто написать функциональный модуль и подключить его в одном месте программы. Каждый функциональный модуль должен состоять из трех частей, и каждая из этих частей должна быть подключена в соответствующем блоке.

Преимущества

1. Единая концепция системы. Несомненным плюсом MVC является единая глобальная архитектура приложения. Даже в сложных системах, разработчики могут легко ориентироваться в программных блоках. Например, если возникла ошибка в логике обработки данных, разработчик сразу отбрасывает 2 блока программы (controller и view) и занимается исследованием 3-го (model).
2. Упрощен механизм отладки приложения. Т. к. весь механизм визуализации теперь сконцентрирован в одном программном блоке, упростились механизмы опционального вывода графических элементов.

Регулярные выражения. Поддержка в Java и Python. Реализация на нижнем уровне

Регулярные выражения — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. Для поиска используется строка-образец (*pattern*), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.

Обычные символы

Большинство символов в регулярном выражении представляют сами себя за исключением специальных символов `[ ] \ / ^ $ . | ? * + ( ) { }`, которые могут быть экранированы символом `\` для представления самих себя в качестве символов текста.

Символьные классы

Символьный класс – набор символов в квадратных скобках `[ ]`. На данном месте в строке может стоять один из перечисленных символов (`[abc]`, `[12890]`). Возможно указание диапазонов символов: например, `[А-Яа-я]`.

Если требуется указать символы, которые не входят в указанный набор, то используют символ ^ внутри квадратных скобок, например [^0-9] означает любой символ, кроме цифр.

Метасимволы

Символ	Эквивалент	Соответствие
`\d`	`[0-9]`	Цифровой символ
`\D`	`[^0-9]`	Нецифровой символ
`\s`	`[ \f\n\r\t\v]`	Пробельный символ
`\S`	`[^ \f\n\r\t\v]`	Непробельный символ
`\w`	`[[:word:]]`	Буквенный или цифровой символ или знак подчёркивания
`\W`	`[^[:word:]]`	Любой символ, кроме буквенного или цифрового символа или знака подчёркивания

Позиция внутри строки

Представление	Позиция	Пример
`^`	Начало текста	`^a`
`$`	Конец текста	`a$`

Обозначение группы

Круглые скобки используются для определения области действия и приоритета операций. Шаблон внутри группы обрабатывается как единое целое. Пример: `the ([0-9]th) of (\S)`, где `([0-9]th)` и `(\S)` – группы №1 и №2. Группа №0 – все регулярное выражение.

Перечисление

Вертикальная черта разделяет допустимые варианты. Перебор вариантов выполняется слева направо, как они указаны. Пример: `red | green | blue` соответствует либо `red`, либо `green`, либо `blue`. Перечисления можно заключать в группы: `(red | green | blue)`

Квантификация

Квантификатор после символа, символьного класса или группы определяет, сколько раз предшествующее выражение может встречаться.

Представление	Число повторений	Пример
`?`	Ноль или одно	`colou?r`
`*`	Ноль или более	`colou*r`
`+`	Одно или более	`colou+r`
`{n}`	Ровно n раз	`colou{n}r`
`{m,n}`	От m до n включительно	`colou{m,n}r`
`{m,}`	Не менее m	`colou{m,}r`
`{,n}`	Не более n	`colou{,n}r`

Примеры сложных выражений

Дробное число

`(\+|-)?(0|[1-9][0-9]*)(\.[0-9]*)?(e(\+|-)?[0-9]+)?`

`(\+|-)?` – знак (‘+’ или ‘-‘)

`(\+|-)?(0|[1-9][0-9]*)` – целое число (‘0’ или любое другое)

`(\.[0-9]*)?` – дробная часть (‘.<цифры от 0 до 9 любое число раз>’)

`(e(\+|-)?[0-9]+)?` – экспонента (‘e<число>, e+<число>, e-<число>)

`0, 123, 123.123, 0.123, 1e10, 123.123e10, 123.123e+010, 123.123e-010, 123e10`

Идентификатор в языках программирования

`([a-zA-Z]|_)+[_0-9a-zA-Z]*`

`([a-zA-Z]|_)+` – идентификатор не может начинаться с цифры

`[_0-9a-zA-Z]*` – в идентификаторах можно использовать символы латинского алфавита, цифры и знак подчеркивания

`_hello_123_abc`

`hello`

`hello12_2_fd`

Поддержка в Java

Класс java.util.regex.Pattern – построение регулярного выражения (образца).

Методы:

- compile(String regex) – переводит регулярное выражение во внутреннее представление.
- matcher(CharSequence input) – создает объект Matcher, выполняющий сопоставление входной строки с образцом.
- matches(String regex, CharSequence input) – сопоставляет регулярное выражение regex со строкой input. Если строка подходит – True, иначе False.

- split(CharSequence input) – разбивает строку input на подстроки, используя в качестве разделителя регулярное выражение.

Примеры:

```
Regex.matches(“[0-9]+”, “234”); //True

Regex.matches(“[0-9]+”, “23p3”); //False

Regex.compile(“[ABC]”).split(“helloAhowBareCyou”);
// {“hello”, “how”, “are”, “you”}
```

Класс java.util.Matcher – выполнение операций по сопоставлению строки с регулярным выражением.

Методы:

- group() – возвращает подстроку, которая соответствует данному регулярному выражению. Аналогично group(0).
- group(int group) – возвращает подстроку, которая соответствует группе group в данном регулярном выражении.
- replaceAll(String replacement) – заменяет все подстроки, соответствующие регулярному выражению, на строку replacement.
- replaceFirst(String replacement) – аналогично replaceAll, но только первую подстроку.

Примеры:

```
Regex.compile(“[ABC]”)
      .matcher(“helloAhowBareCyou”)
      .replaceAll(“_”); // “hello_how_are_you”

Regex.compile(“[ABC]”)
      .matcher(“helloAhowBareCyou”)
      .replaceFirst(“_”); // “hello_howBareCyou”

Regex.compile(“[0-9]+\.([0-9]+)”)
      .matcher(“123.456”)
      .group(1); // “456”
```

Поддержка в Python

re – модуль стандартной библиотеки Python для работы с регулярными выражениями

Функции:

- re.compile(pattern) – переводит регулярное выражение во внутреннее представление, возвращает объект типа RegEx, к которому также применимы функции модуля. Это лучше, чем каждый раз компилировать регулярное выражение из строки.
- re.search(pattern, string) – ищет первую подстроку в string, соответствующую регулярному выражению pattern.
- re.match(pattern, string) – сопоставляет регулярное выражение regex с префиксом строки string. Если строка не подходит, возвращает None.
- re.split(pattern, string) – разбивает строку input на подстроки, используя в качестве разделителя регулярное выражение.

- re.sub(pattern, repl, string) – заменяет все подстроки, соответствующие регулярному выражению, на строку repl.

Примеры:

```
re.search(r'[ABC]', 'helloAhowBareCyou')
#<_sre.SRE_Match; span=(5, 6), match='A'>

re.match(r'[ABC]', 'helloAhowBareCyou') # None

re.split(r'[ABC]', 'helloAhowBareCyou') # ['hello', 'how', 'are', 'you']

re.sub(r'[ABC]', '_', 'helloAhowBareCyou') # 'hello_how_are_you'
```

Реализации на нижнем уровне

- **NFA** (Недетерминированные Конечные Автоматы) используют «жадный» алгоритм отката, проверяя все возможные расширения регулярного выражения в определённом порядке и выбирая первое подходящее значение. NFA может обрабатывать подвыражения и обратные ссылки. Но из-за алгоритма отката традиционный NFA может проверять одно и то же место несколько раз, что отрицательно сказывается на скорости работы. Поскольку традиционный NFA принимает первое найденное соответствие, он может и не найти самое длинное из вхождений (этого требует стандарт POSIX, и существуют модификации NFA, выполняющие это требование — GNU sed). Именно такой механизм регулярных выражений используется, например, в Perl, Tcl и .NET.
- **DFA** (Детерминированные Конечные Автоматы) работают линейно по времени, поскольку не используют откаты и никогда не проверяют какую-либо часть текста дважды. Они могут гарантированно найти самую длинную строку из возможных. DFA содержит только конечное состояние, следовательно, не обрабатывает обратных ссылок, а также не поддерживает конструкций с явным расширением, то есть, не способен обработать и подвыражения. DFA используется, например, в lex и egrep.

Хеш-функции

Основные понятия

Хеш-функция – функция преобразования массива входных данных произвольной длины (X) в битовую строку фиксированной длины Y (называются «**хеш**»), выполняемая определенным алгоритмом:

$$f: X \rightarrow Y$$

Как правило, кол-во различных значений входных данных значительно больше кол-ва значений выходных данных:

$$|Y| \ll |X|$$

Случай, при котором хеш-функция преобразует несколько разных сообщений в одинаковые сводки называется «**коллизией**». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Виды хеш-функций

«Хорошая» хеш-функция должна удовлетворять двум свойствам:

- быстрое вычисление;
- минимальное количество «коллизий».

Хеш-функции, основанные на делении

1. «Хеш» как остаток от деления на число всех возможных «хешей»

Хеш-функция может вычислять «хеш» как остаток от деления входных данных на M:

$$h(k) = k \; mod \; M$$

где M — количество всех возможных «хешей» (выходных данных), k – ключ.

На практике обычно выбирают простое M; в большинстве случаев этот выбор вполне удовлетворителен или используют степень двойки.

2. «Хеш» как набор коэффициентов получаемого полинома

Хеш-функция может выполнять деление входных данных на полином по модулю два. В данном методе M должна являться степенью двойки, а бинарные ключи ($K = k_{n-1}k_{n-2} \dots k_0$) представляются в виде полиномов, в качестве «хеш-кода» «берутся» значения коэффициентов полинома, полученного как остаток от деления входных данных K на заранее выбранный полином P степени m:

$$K(x) \; mod \; P(x) = h_{m-1}x^{m-1} + \cdots + h_1 x + h_0$$

$$h(x) = h_{m-1} \dots h_1 h_0$$

При правильном выборе $P(x)$ гарантируется отсутствие коллизий между почти одинаковыми ключами.

Хеш-функции, основанные на умножении

Обозначим символом ω количество чисел, представимых машинным словом. Например, для 32-разрядных компьютеров, совместимых с IBM PC, $\omega = 2^{32}$.

Выберем некую константу A так, чтобы A была взаимно простой с ω. Тогда хеш-функция, использующая умножение, может иметь следующий вид:

$$h(K) = \left[M \left[\frac{A}{\omega} * K\right]\right]$$

В этом случае на компьютере с двоичной системой счисления M является степенью двойки, и $h(K)$ будет состоять из старших битов правой половины произведения $A * K$.

Среди преимуществ хеш-функций, основанных на делении и умножении, стоит отметить выгодное использование неслучайности реальных ключей. Например, если ключи представляют собой арифметическую прогрессию (например, последовательность имён «Имя 1», «Имя 2», «Имя 3»), хеш-функция, использующая умножение, отобразит арифметическую прогрессию в приближенно арифметическую прогрессию различных хеш-значений, что уменьшит количество коллизий по сравнению со случайной ситуацией.

Применение

- при построении ассоциативных массивов;
- при поиске дубликатов в сериях наборов данных;
- при построении уникальных идентификаторов для наборов данных;
- при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
- при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
- при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

Использование хеш-функций в хеш-таблицах

Реализация

- `dict`, `set` – Python
- `std::unordered_map`, `std::unordered_set` – C++
- `HashMap`, `HashSet`, `Hashtable` - Java

Методы борьбы с коллизиями в хеш-таблицах

Существует два основных метода борьбы с коллизиями в хеш-таблицах:

- метод цепочек (метод прямого связывания, метод корзин);
- метод открытой адресации.

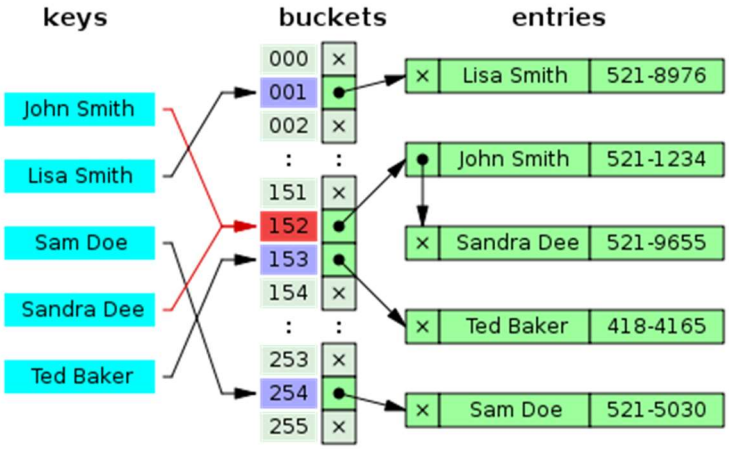
При использовании **метода цепочек** в хеш-таблице хранятся пары «связный список ключей» — «хеш». Для каждого ключа хеш-функцией вычисляется хеш; если хеш был получен ранее (для другого ключа), ключ добавляется в существующий список ключей, парный хеш; иначе, создаётся новая пара «список ключей» — «хеш-код», и ключ добавляется в созданный список.

В общем случае, если имеется N ключей и M списков, средний размер хеш-таблицы составит $\alpha = \frac{N}{M}$. Число α называет **коэффициентом заполнения** (load factor). В HashMap (Java) load factor равен 0,75. В этом случае при поиске по таблице по сравнению со случаем, в котором поиск выполняется последовательно, средний объём работ уменьшится примерно в M раз.

Если load factor становится слишком большим, то происходит увеличение кол-ва корзин. Обычно число корзин – простое число или степень двойки. В случае со степенью двойки кол-во корзин просто удваивается, следовательно, хеш становится на один бит больше. Предполагается, что в среднем у половины ключей этот бит будет равен единице, а у другой – ноль и размер каждой корзины уменьшится в два раза. Если число корзин простое число, то берется следующее простое как кол-во корзин.

При этом слишком маленький load_factor также неэффективен, т. к. это означает, что кол-во корзин слишком велико для данного числа элементов в таблице.

При удалении элементов из таблицы кол-во корзин не меняется.

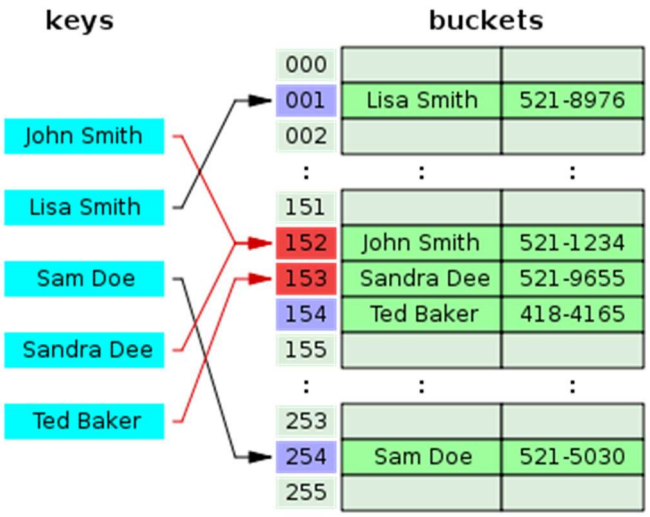


При использовании **метода открытой адресации** в хеш-таблице хранятся пары «ключ» — «хеш». Для каждого ключа хеш-функцией вычисляется хеш; пара «ключ» — «хеш» сохраняется в таблице. В этом случае при поиске по таблице по сравнению со случаем, в котором используются связные списки, ссылки не используются, выполняется последовательный перебор пар «ключ» — «хеш», перебор прекращается после обнаружения нужного ключа. Последовательность, в которой просматриваются ячейки таблицы, называется последовательностью проб.

Типы последовательностей проб:

- Линейное пробирование: ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом k между ячейками (обычно k = 1), то есть i-й элемент последовательности проб — это ячейка с номером $(hash(x) + ik) \ mod \ N$. Для того, чтобы все ячейки оказались просмотренными по одному разу, необходимо, чтобы k было взаимно-простым с размером хеш-таблицы.
- Квадратичное пробирование: интервал между ячейками с каждым шагом увеличивается на константу. Если размер хеш-таблицы равен степени двойки ($N = 2^p$), то одним из примеров последовательности, при которой каждый элемент будет просмотрен по одному разу, является:
  $$hash(x) \ mod \ N, (hash(x) + 1 * 1) \ mod \ N, (hash(x) + 2 * 2) \ mod \ N, (hash(x) + 3 * 3) \ mod \ N, \dots$$
- Двойное хеширование: интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-

таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до N — 1.



Красно-черные деревья

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или чёрный. Дополнительные требования:

1. Узел либо красный, либо чёрный.
2. Корень — чёрный. (Может и красный, не суть).
3. Все листья (NIL, пустышки) — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

Реализации

- `std::map`, `std::multimap`, `std::set`, `std::multiset` в C++
- `TreeMap`, `TreeSet` в Java

Почему они так популярны?

Популярность красно-чёрных деревьев связана с тем, что на них часто достигается подходящий баланс между степенью сбалансированности и сложностью поддержки сбалансированности.

Операции

Операции чтения для красно-чёрного дерева ничем не отличаются от иных для бинарного дерева поиска, потому что любое красно-чёрное дерево является особым случаем обычного бинарного дерева поиска. Однако непосредственный результат вставки или удаления может привести к нарушению свойств красно-чёрных деревьев. Восстановление свойств требует небольшого ($O(log\ n)$ или $O(1)$) числа операций смены цветов (которая на практике очень быстрая) и не более чем трех поворотов дерева (для вставки — не более двух). Хотя вставка и удаление сложны, их трудоемкость остается $O(log\ n)$.

Короче:

Поиск, вставка и удаление за $O(log\ n)$

Вставка

Вставка начинается с добавления красного узла с двумя пустыми листьями.

Что происходит дальше зависит от цвета близлежащих узлов. **Дядя** – брат родительского узла, как и в фамильном дереве. Заметим, что:

- Свойство 3 (Все листья чёрные) выполняется всегда.
- Свойство 4 (Оба потомка любого красного узла — чёрные) может нарушиться только при добавлении красного узла, при перекрашивании чёрного узла в красный или при повороте.
- Свойство 5 (Все пути от любого узла до листовых узлов содержат одинаковое число чёрных узлов) может нарушиться только при добавлении чёрного узла, перекрашивании красного узла в чёрный (или наоборот), или при повороте.

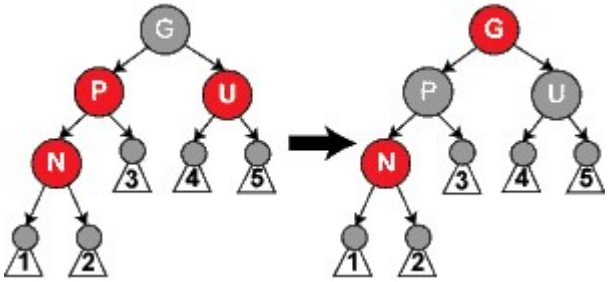
Обозначения:

- **N** – текущий узел
- **P –** родительский узел
- **G** – дедушка **N**
- **U** – дядя **N**

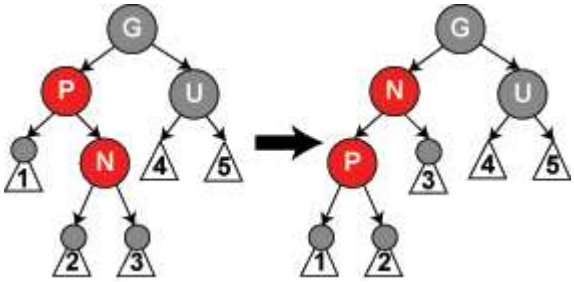
Случай 1: Текущий узел **N** в корне дерева. В этом случае, он перекрашивается в чёрный цвет. Все.

Случай 2: Предок **P** текущего узла чёрный – дерево остаётся корректным.

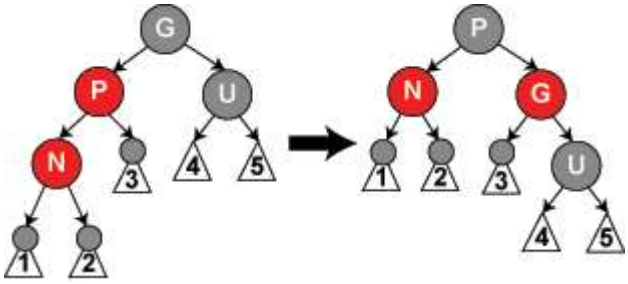
Случай 3: Если и родитель **P** и дядя **U** — красные, то они оба могут быть перекрашены в чёрный и дедушка **G** станет красным. Теперь у текущего красного узла **N** чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка **G** теперь может нарушить свойства 2 (Корень — чёрный) или 4 (Оба потомка каждого красного узла — чёрные). Чтобы это исправить, вся процедура рекурсивно выполняется на **G**.



Случай 4: Родитель **P** является красным, но дядя **U** — чёрный. Также, текущий узел **N** — правый потомок **P**, а **P** в свою очередь — левый потомок своего предка **G**. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла **N** и его предка **P**. Тогда, для бывшего родительского узла **P** в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено, но теперь задача сводится к Случаю 5.



Случай 5: Родитель **P** является красным, но дядя **U** — чёрный, текущий узел **N** — левый потомок **P** и **P** — левый потомок **G**. В этом случае выполняется поворот дерева на **G**. Цвета **P** и **G** меняются.



Для последних трех случаев есть еще три симметричных (буквально меняем слова «лево» на «право»).

Удаление

При удалении узла с двумя нелистовыми потомками в обычном двоичном дереве поиска мы ищем либо наибольший элемент в его левом поддереве, либо наименьший элемент в его правом поддереве и перемещаем его значение в удаляемый узел. Затем мы удаляем узел, из которого

копировали значение. Копирование значения из одного узла в другой не нарушает свойств красно-чёрного дерева, так как структура дерева и цвета узлов не изменяются.

Что потом делать, думаю сам Геннадий Андреевич не помнит.

тут Рысев

Абстрактные классы и интерфейсы Java. Особенности реализации. Решение проблемы множественного наследования.

Абстрактный класс - это класс, объявленный как `abstract`. Он может содержать или не содержать абстрактные методы. Абстрактные классы нельзя создать, но их можно наследовать.

Абстрактный метод - это метод, объявленный без реализации. Пример:

```
abstract void moveTo(double deltaX, double deltaY);
```
Если в классе содержится хотя бы один абстрактный метод, тогда сам класс должен быть объявлен как `abstract`. Пример:

```
public abstract class GraphicObject {
   abstract void draw();
}
```
Когда абстрактный класс наследуется, подкласс обычно обеспечивает реализацию для всех абстрактных методов в родительском классе. Однако, если подкласс реализует не все методы абстрактного класса, то подкласс должен так же быть объявлен как `abstract`.

Интерфейс - это ссылочный тип, который может содержать *только* константы, сигнатуры методов, методы по умолчанию, статические методы и вложенные типы (пример: другой интерфейс). Реализации методов могут быть только в методах по умолчанию и статических методах. Интерфейс нельзя создать - он может быть реализован классами или унаследован другими интерфейсами.

Все методы интерфейса по умолчанию являются публичными (public) и абстрактными (abstract), а поля - public static final.

Пример интерфейса:

```
interface Bicycle {
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```
Главная причина для внедрения методов по умолчанию в интерфейсы - предоставление средства, позволявшего расширять интерфейсы, не нарушая уже существующий код.

Статические методы из интерфейсов не наследуются ни реализующими их классами, ни подчинёнными интерфейсами.

Методы по умолчанию предоставляют одну из форм реализации множественного наследования.

В Java нет множественного наследования классов, но есть множественное наследование интерфейсов. При определении класса можно указать, что класс будет реализовать несколько интерфейсов. В Java класс может наследоваться (реализовывать) от многих интерфейсов, но только от одного абстрактного класса.

Static и final классы и члены классов.

Иногда желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Как правило, обращение к члену класса должно осуществляться только в сочетании с объектом его класса. Но можно создать член класса, чтобы пользоваться им отдельно, не ссылаясь на конкретный экземпляр. Такой член класса объявляется как static. Статическими могут быть объявлены как методы, так и переменные. Статические переменные являются глобальными: их копии не создаются, вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

Ограничения static методов:

1) они могут непосредственно вызывать только другие статические методы;
2) им непосредственно доступны только статические переменные;
3) они не могут делать ссылки типа this или super.

Пример:

```
class Example {
    static int a = 42;
    static void callme() {
        System.out.println("a = " + a);
    }
}
Вызов из другого класса: Example.callme();
```

Статическим классом может быть только вложенный класс. Поскольку такой класс является статическим, то должен обращаться к нестатическим членам своего внешнего класса посредством объекта.

Поле может быть объявлено как final (завершённое). Это позволяет предотвратить изменение содержимого переменной, сделав её, по существу, константой. Завершённое поле должно быть инициализировано во время его объявления. Пример: final int FILE_NEW = 1.

Методы, объявленные как final, переопределяться (при наследовании) не могут.

Для предотвращения наследования класса следует в объявлении класса указать ключевое слово final. Такое объявление класса неявно делает завершенными (final) и все его методы. Одновременное объявление класса как abstract и final недопустимо, поскольку абстрактный класс является незавершённым.

ВАЖНО: final для объекта делает константной только саму ссылку. Пример:

```
final List<Integer> list = new ArrayList<Integer>();

list = new ArrayList<Integer>(); // ошибка - переопределение ссылки.
```

Многопоточное программирование в Java.

Многопоточная система в Java построена на основе класса Thread, его методах и дополняющем его интерфейсе Runnable. Чтобы создать новый поток исполнения, следует расширить класс Thread или же реализовать интерфейс Runnable.

Основные методы класса Thread:

1) isAlive - определяет, выполняется ли поток;
2) join - ожидает завершения потока управления;
3) run - задаёт точку входа в поток исполнения;
4) sleep - приостанавливает выполнение потока на заданное время;
5) start - запускает потом исполнения, вызывая его метод run().

Когда программа на Java запускается на выполнение, сразу же начинает выполняться один поток. Он обычно называется главным потоком программы, потому что он запускается вместе с ней. От главного потока исполнения порождаются все дочерние потоки. Зачастую он должен быть последним

потом, завершающим выполнение программы, поскольку в нём производятся различные завершающие действия. Для управления главным потоком можно получить ссылку на него, вызвав метод currentThread().

Создание потока:

1) реализация интерфейса Runnable. Необходимо переопределять метод run() из интерфейса.

Пример:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

2) расширение класса Thread. Класс Thread сам реализует интерфейс Runnable, но его метод run() ничего не делает.

Пример:

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Синхронизация. Монитор - это объект, используемый в качестве взаимоисключающей блокировки. Только один поток исполнения может в одно и то же время владеть монитором, остальные потоки, пытающиеся войти в заблокированный монитор будут приостановлены (ожидают монитор). Для синхронизации методов применяют модификатор доступа synchronized.

Методы взаимодействия потоков исполнения:

- wait() - вызывающий поток исполнения уступает монитор и переходит в состояние ожидания до тех пор, пока какой-нибудь другой поток исполнения не выйдёт в тот монитор и не вызовет notify();
- notify() - возобновляет исполнение потока, из которого был вызван метод wait() для того же самого объекта;
- notifyAll() - возобновляет исполнение всех потоков, из которых был вызван метод wait() для того же самого объекта. Одному из этих потоков предоставляется доступ.

Приостановка и возобновление потоков исполнения. Для этих действий необходимо периодически в методе run() проверять флаговую переменную и использовать методы wait() и notify().

Изменяемые и неизменяемые классы в Java.

Изменяемый объект - можно изменить состояние и поля после того как объект создан. Например: класс StringBuilder, java.util.Date. Все методы, которые якобы изменяют состояние объекта String, на самом деле возвращают новые строки.

Неизменяемый класс – это класс, состояние которого не может быть изменено после создания. Здесь состоянием объекта по существу считаются значения, хранимые в экземпляре класса, будь то примитивные типы или ссылочные типы. Примеры: String, упакованные примитивные объекты (Integer, Long и др.).

Для того чтобы сделать класс неизменяемым, необходимо выполнить следующие условия:

1. Не предоставляйте сеттеры или методы, которые изменяют поля или объекты, ссылающиеся на поля. Сеттеры подразумевают изменение состояния объекта а это то, чего мы хотим тут избежать.
2. Сделайте все поля final и private. Поля, обозначенные private, будут недоступными снаружи класса, а обозначение их final гарантирует, что вы не измените их даже случайно.
3. Не разрешайте субклассам переопределять методы. Самый простой способ это сделать – объявить класс как final. Финализированные классы в Java не могут быть переопределены.

Преимущества неизменяемых классов:

- легко конструировать, тестировать и использовать;
- автоматически потокобезопасны и не имеют проблем синхронизации;
- не требуют конструктора копирования;
- позволяют выполнить «ленивую инициализацию» хэшкода и кэшировать возвращаемое значение;
- делают хорошие Map ключи и Set элементы (эти объекты не должны менять состояние, когда находятся в коллекции);
- делают свой класс постоянным, единожды создав его, а он не нуждается в повторной проверке.

Сборка мусора. Управление сборщиком мусора в Java. Алгоритмы работы сборщика мусора в Java.

В C++ все объекты должны быть уничтожены. При создании объекта с оператором **new** в C++ деструктор вызывается оператором **delete.** Если программист забудет вызвать **delete**, то деструктор никогда не будет вызван, и будет утечка памяти. Такие баги очень трудно обнаружить, поэтому в Java применяется механизм сборки мусора. В Java нет оператора **delete**, очистку объекта осуществляет сборщик мусора. Сборка мусора не гарантируется: если у JVM не заканчивается память, то JVM может и не вызывать сборку мусора.

Java-программисту не нужно следить за распределением памяти, так как сборщик мусора управляет памятью автоматически. Сборщик мусора запускается виртуальной машиной Java (JVM). Сборщик мусора — это низкоприоритетный процесс, который запускается периодически и освобождает память, использованную объектами, которые больше не нужны.

[Bruce Eckel - Thinking in Java] Алгоритмы. Если проследовать по ссылкам из стека или из статической области памяти, то можно найти все живые объекты: для каждой ссылки на объект нужно просмотреть в самом объекте все ссылки на другие объекты и т.д. JVM использует *адаптивную схему* сборки мусора и что она делает с живыми (на них есть ссылки) объектами зависит от текущего варианта схемы.

1. Один из вариантов - *stop-and-copy* (остановка и копирование). Программа сначала останавливается. Затем каждый живой объект копируется из одной кучи во вторую, оставляя в первой мусор. Так же при копировании во вторую кучу они компактно упаковываются. После перемещения объектов все ссылки на них должны быть изменены. У этого варианта есть недостатки: 1) нужно управлять памятью в двух кучах; 2) при работе программы мусора может не быть или быть немного, но несмотря на это, copy collector всё рано будет копировать всю память из одной кучу в другую, затрачивая ресурсы. Чтобы избежать такой ситуации некоторые JVM обнаруживают, что новый мусор не создаётся и переключаются на другую схему (это адаптивная часть). Другая схема называется *mark-and-sweep*.

2. *Mark-and-sweep* (пометка и очистка) так же находит все живые объекты, следуя ссылкам. Каждый раз находя живой объект, он помечается флагом. Только после завершения процесса пометки начинается процесс очистки. Во время очистки происходит уничтожение объектов. Хотя копирование объектов не происходит, сборщик может упаковать фрагментированную кучу (дефрагментировать), перемещая объекты. Эта схема работы сборщика мусора так же останавливает программу перед началом своей работы.

Подавляющее большинство объектов создаются на очень короткое время, они становятся ненужными практически сразу после их первого использования (итераторы, локальные переменные методов и т.д.). Далее идут объекты, создаваемые для выполнения более-менее долгих вычислений. И, наконец, объекты-старожилы, переживающие почти всех — это постоянные данные программы, загружаемые часто в самом начале и проживающие долгую жизнь до остановки приложения. Это навело разработчиков на мысль, что в первую очередь необходимо сосредотачиваться на очистке тех объектов, которые были созданы совсем недавно. Именно среди них чаще всего находится бóльшее число тех, кто уже отжил свое, и именно здесь можно получить максимум эффекта при минимуме трудозатрат. Тут и возникает идея разделения объектов на младшее поколение (young generation) и старшее поколение (old generation). В соответствии с этим разделением и процессы сборки мусора разделяются на малую сборку (minor GC), затрагивающую только младшее поколение, и полную сборку (full GC), которая может затрагивать оба поколения. Малые сборки выполняются достаточно часто и удаляют основную часть мертвых объектов. Полные сборки выполняются тогда, когда текущий объем выделенной программе памяти близок к исчерпанию и малой сборкой уже не обойтись.

В JVM память выделяется блоками. Сборщик мусора обычно может копировать объекты в мёртвые блоки. У каждого блока есть счётчик поколения. Обычно только новые блоки, созданные с момента последней сборки мусора, уплотняются (дефрагментируются). У всех остальных блоков увеличивается счётчик поколения. Это решает проблему с мало-живущими объектами. Периодически выполняется полная очистка - большие объекты не копируются (у них увеличивается счётчик поколения), а блоки, содержащие небольшие объекты копируются и дефрагментируются.

JVM следит за эффективностью сборки мусора. Если сборка мусора только затрачивает много времени на копирование (большинство объектов - долгоживущие), то схема переключается на *mark-and-sweep*. А если куча становится сильно фрагментированной - переключается на *stop-and-copy*.

Вы можете запросить запуск сборщика мусора, но вы не можете принудительно задавать это действие (JVM сама решает выполнить ли сборку мусора). Для запроса вы можете вызвать один из следующих методов:

- System.gc();
- Runtime.getRuntime().gc().

Рефлексия в Java. Использование метаданных.

Аннотации представляют собой некие метаданные, которые могут добавляться в исходный код программы и семантически не влияют на нее, но могут использоваться в процессе анализа кода, компиляции и даже во время выполнения.

Вот основные варианты использования аннотаций:

- предоставлять необходимую информацию для компилятора;
- предоставлять метаданные различным инструментам для генерации кода, конфигураций и т.д.;
- использоваться в коде во время выполнения программного кода (рефлексия).

Пример объявления аннотации:

```
// Простой тип аннотации.
@interface MyAnno {
    String str();
    int val();
}
```

Знак @ указывает компилятору, что объявлен тип аннотации. Все аннотации состоят только из объявлений методов. Но тела этих методов в них не определяются. Вместо этого они реализуются средствами Java. Более того, эти методы ведут себя аналогично полям.

Все аннотации автоматически расширяют интерфейс Annotation (он объявлен в пакете java.lang.annotation.). Как только аннотация будет объявлена, ею можно воспользоваться для аннотирования любых элементов прикладного кода. Аннотацию можно связать с любым объявлением. Например, аннотировать можно классы, методы, поля, параметры и константы перечислимого типа. Аннотированной может быть даже сама аннотация. Когда применяется аннотация, её членам присваиваются соответствующие значения. Пример аннотирования метода myMeth():

```
// Аннотирование метода
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() { //…
```

Правила удержания аннотаций. В Java определены 3 правила:

1) SOURCE - содержаться только в исходном файле и отбрасываются при компиляции;
2) CLASS - сохраняются в файле с расширением .class во время компиляции (кроме аннотаций объявления локальных переменных);
3) RUNTIME - тоже что и 2, и остаются доступными для виртуальной машины JVM во время выполнения.

Правило удержания аннотации задаётся с помощью одной из встроенных аннотаций Java: @Retention(правило_удержания). Если не указано правило удержания, то применяется правило CLASS.

Аннотации-маркеры - это специальный вид аннотаций, которые не содержат членов. Их единственное назначение - пометить объявление. Пример: @MyMarker public static void myMeth().

Для членов аннотации можно указать значения по умолчанию. Пример:

```
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}
```

Некоторые встроенные аннотации: @Documented, @Target, @Inherited, @Override, @Deprecated, @SupressWarnings.

Рефлексия - это языковое средство для получения сведений о классе во время выполнения программы. API для рефлексии входит в состав пакета java.lang.reflect.

Первый шаг с целью воспользоваться рефлексией - получение объекта типа Class вызвав метод getClass(). Этот объект представляет класс, аннотацию которого требуется получить. А Class относится к числу встроенных в Java классов и определён в пакете java.lang. Если требуются аннотации, связанные с определённым элементом, объявленным в классе, сначала следует получить объект, представляющий этот элемент. Например: getMethod(), getField(), getConstructor() - возвращают объекта типа Method, Field, Constructor. Если метод не удаётся найти, то генерируется исключение типа NoSuchMethodException. Из объекта типа Class, Method, Field или Constructor можно получить конкретные аннотации, связанные с этим объектом, вызвав метод getAnnotation(). Он возвращает null, если аннотация не найдена. Пример вывода аннотации (MyAnno) метода (myMeth):

```
try {
  Class c = myObject.getClass();
  Method m = c.getMethod("myMeth");
  MyAnno anno = m.getAnnotation(MyAnno.Class);
  System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
  System.out.println("Метод не найден");
}
```

Выведет строку: Пример аннотации 100

Для того чтобы получить сразу все аннотации, имеющие аннотацию @Retention(RetentionPolicy.RUNTIME) и связанные с искомым элементом, достаточно вызвать метод getAnnotations() для этого элемента. Он возвращает массив аннотаций. Может быть вызван для объектов типа Class, Method, Contructor и Field.

Сеарилизация в java. Структура сеарилизованного файла

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется **десериализацией**. Сериализация очень удобна, когда идет работа со сложными объектами.

Интерфейс Serializable

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс **Serializable**. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

Сериализация. Класс ObjectOutputStream

Для сериализации объектов в поток используется класс **ObjectOutputStream**. Он записывает данные в поток.

Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись:

Алгоритм:

- запись метаданных о классе ассоциированном с объектом
- рекурсивная запись описания суперклассов, до тех пор пока не будет достигнут java.lang.object
- после окончания записи метаданных начинается запись фактических данных ассоциированных с экземпляром, только в этот раз начинается запись с самого верхнего суперкласса
- рекурсивная запись данных ассоциированных с экземпляром начиная с самого низшего суперкласса

Есть два способа сделать это:

- Каждый параметр класса объявленный как transient, не сериализуются(по умолчанию все параметры класса сериализуются)
- Или каждый параметр класса, который мы хотим сериализовать, помечается тегом Externalizable(по умолчанию никакие параметры не сериализуются).
- Поле данных не будет сериализовано с помощью ObjectOutputStream, когда оно будет вызвано для объекта, если это поле данных, данного объекта помечено как transient. Например – private transient String password.

Поля, объявленные как `transient` или `static`, игнорируются в процессе сериализации/десериализации.

1 ObjectOutputStream(OutputStream out)

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

- **void close()**: закрывает поток
- **void flush()**: очищает буфер и сбрасывает его содержимое в выходной поток
- **void write(byte[] buf)**: записывает в поток массив байтов
- **void write(int val)**: записывает в поток один младший байт из val
- **void writeBoolean(boolean val)**: записывает в поток значение boolean
- **void writeByte(int val)**: записывает в поток один младший байт из val

- **void writeChar(int val)**: записывает в поток значение типа char, представленное целочисленным значением
- **void writeDouble(double val)**: записывает в поток значение типа double
- **void writeFloat(float val)**: записывает в поток значение типа float
- **void writeInt(int val)**: записывает целочисленное значение int
- **void writeLong(long val)**: записывает значение типа long
- **void writeShort(int val)**: записывает значение типа short
- **void writeUTF(String str)**: записывает в поток строку в кодировке UTF-8
- **void writeObject(Object obj)**: записывает в поток отдельный объект

Эти методы охватывают весь спектр данных, которые можно сериализовать.

Пример:

```
import java.io.*;
public class FilesApp {
   public static void main(String[] args) {

      try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.dat")))
      {
         Person p = new Person("Джон", 33, 178, true);
         oos.writeObject(p);
      }
      catch(Exception ex){

         System.out.println(ex.getMessage());
      }
   }
}
class Person implements Serializable{

   public String name;
   public int age;
   public double height;
   public boolean married;

   Person(String n, int a, double h, boolean m){

      name=n;
      age=a;
      height=h;
      married=m;
   }
}
```

Структура
Листинг 6.

```
class parent implements Serializable {
  int parentVersion = 10;
}
```

```
class contain implements Serializable{
  int containVersion = 11;
}
public class SerialTest extends parent implements Serializable {
  int version = 66;
  contain con = new contain();

  public int getVersion() {
    return version;
  }
  public static void main(String args[]) throws IOException {
    FileOutputStream fos = new FileOutputStream("temp.out");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    SerialTest st = new SerialTest();
    oos.writeObject(st);
    oos.flush();
    oos.close();
  }
}

* This source code was highlighted with Source Code Highlighter.
```

В примере сериализуется объект класса SerialTest, который наследуется от parent и содержит объект-контейнер классacontain. В листинге 7 показан сериализованный объект.

Листинг 7.

```
AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65
73 74 05 52 81 5A AC 66 02 F6 02 00 02 49 00 07
76 65 72 73 69 6F 6E 4C 00 03 63 6F 6E 74 00 09
4C 63 6F 6E 74 61 69 6E 3B 78 72 00 06 70 61 72
65 6E 74 0E DB D2 BD 85 EE 63 7A 02 00 01 49 00
0D 70 61 72 65 6E 74 56 65 72 73 69 6F 6E 78 70
00 00 00 0A 00 00 00 42 73 72 00 07 63 6F 6E 74
61 69 6E FC BB E6 0E FB CB 60 C7 02 00 01 49 00
0E 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E 78
70 00 00 00 0B
```

На рисунке 2 показан сценарий алгоритма сериализации.

Рисунок 2.

Давайте рассмотрим, что собой представляет каждый байт в сериализованном объекте. В начале идёт информация о протоколе сериализации:

- AC ED: STREAM_MAGIC. Говорит о том, что используется протокол сериазизации.
- 00 05: STREAM_VERSION. Версия сериализации.
- 0x73: TC_OBJECT. Обозначение нового объекта.

На первом шаге алгоритм сериализации записывает описание класса ассоциированного с объектом. В примере был сериализован объект класса SerialTest, следовательно алгоритм начал записывать описание класса SerailTest.

- 0x72: TC_CLASSDESC. Обозначение нового класса.
- 00 0A: Длина имени класса.
- 53 65 72 69 61 6c 54 65 73 74: SerialTest, имя класса.

- 05 52 81 5A AC 66 02 F6: SerialVersionUID, идентификатор класса.
- 0x02: Различные флаги. Этот специфический флаг говорит о том, что объект поддерживает сериализацию.
- 00 02: Число полей в классе.

Теперь алгоритм записывает поле int version = 66;.

- 0x49: Код типа поля. 49 это «I», которое закреплено за Int.
- 00 07: Длина имени поля.
- 76 65 72 73 69 6F 6E: version, имя поля.

Затем алгоритм записывает следующее поле, contain con = new contain();. Это объект следовательно будет записано каноническое JVM обозначение этого поля.

- 0x74: TC_STRING. Обозначает новую строку.
- 00 09: Длина строки.
- 4C 63 6F 6E 74 61 69 6E 3B: Lcontain;, Каноническое JVM обозначаение.
- 0x78: TC_ENDBLOCKDATA, Конец опционального блока данных для объекта.

Следующим шагом алгоритма является запись описания класса parent, который является непосредственным суперклассом для SerialTest.

- 0x72: TC_CLASSDESC. Обозначение нового класса.
- 00 06: Длина имени класса.
- 70 61 72 65 6E 74: parent, имя класса
- 0E DB D2 BD 85 EE 63 7A: SerialVersionUID, идентификатор класса.
- 0x02: Различные флаги. Этот флаг обозначает что класс поддерживает сериализацию.
- 00 01: Число полей в классе.

Теперь алгоритм записывает описание полей класса parent, класс имеет одно поле, int parentVersion = 100;.

- 0x49: Код типа поля. 49 обозначает «I», которое закреплено за Int.
- 00 0D: Длина имени поля.
- 70 61 72 65 6E 74 56 65 72 73 69 6F 6E: parentVersion, имя поля.
- 0x78: TC_ENDBLOCKDATA, конец опционального блока данных для объекта.
- 0x70: TC_NULL, обозначает то что больше нет суперклассов, потому что мы достигли верха иерархии классов.

До этого алгоритм сериализации записывал описание классов ассоциированных с объектом и всех его суперклассов. Теперь будут записаны фактические данные ассоциированные с объектом. Сначала записываются члены класса parent:

- 00 00 00 0A: 10, Значение parentVersion.

Дальше перемещаемся в SerialTest

- 00 00 00 42: 66, Значение version.

Следующие несколько байт очень интересны. Алгоритму необходимо записать информацию объекте класса contain.

Листинг 8.

```
contain con = new contain();

* This source code was highlighted with Source Code Highlighter.
```

Алгоритм сериализации ещё не записал описание класса contain. Настал момент это сделать.

- 0x73: TC_OBJECT, обозначает новый объект.
- 0x72: TC_CLASSDESC, обозначает новый класс.
- 00 07: Длина имени класса.
- 63 6F 6E 74 61 69 6E: contain, имя класса.
- FC BB E6 0E FB CB 60 C7: SerialVersionUID, идентификатор этого класса.
- 0x02: Различные флаги. Этот флаг обозначает что класс поддерживает сериализацию.
- 00 01: Число полей в классе.

Алгоритм должен записать описание единственного поля класса conatin, int containVersion = 11;.

- 0x49: Код типа поля. 49 обозначает «I», которое закреплено за Int.
- 00 0E: Длина имени поля.
- 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E: containVersion, имя поля.
- 0x78: TC_ENDBLOCKDATA, конец опционального блока данных для объекта.

Дальше алгоритм проверяет, имеет ли contain родительский класс. Если имеет, то алгоритм начинает запись этого класса; но в нашем случае суперкласса у contain нету, и алгоритм записывает TC_NULL.

- 0x70: TC_NULL

В конце алгоритм записывает фактические данные ассоциированные с объектом класса conatin.

- 00 00 00 0B: 11, значение containVersion.

реализация механизма исключений в java
Когда может понадобиться обработка исключений В Java исключения могут быть вызваны в результате неправильного ввода данных пользователем, отсутствия необходимого для работы программы ресурса или внезапного отключения сети. Для комфортного использования созданного разработчиком приложения, необходимо контролировать появление внештатных ситуаций. Потребитель не должен ждать завершения работы зависшей программы, терять данные в результате необработанных исключений или просто часто появляющихся сообщений о том, что что-то пошло не так.

Существует пять ключевых слов, используемых в исключениях: **try**, **catch**, **throw**, **throws**, **finally**. Порядок обработки исключений следующий.

Операторы программы, которые вы хотите отслеживать, помещаются в блок **try**. Если исключение произошло, то оно создаётся и передаётся дальше. Ваш код может перехватить исключение при помощи блока **catch** и обработать его. Системные исключения автоматически передаются самой системой. Чтобы передать исключение вручную, используется **throw**. Любое исключение, созданное и передаваемое внутри метода, должно быть указано в его интерфейсе ключевым словом **throws**. Любой код, который следует выполнить обязательно после завершения блока **try**, помещается в блок **finally**

Схематически код выглядит так:

```
try {
   // блок кода, где отслеживаются ошибки
}
catch (тип_исключения_1 exceptionObject) {
   // обрабатываем ошибку
}
catch (тип_исключения_2 exceptionObject) {
   // обрабатываем ошибку
}
finally {
   // код, который нужно выполнить после завершения блока try
}
```

Существует специальный класс для исключений **Trowable**. В него входят два класса **Exception** и **Error**.

Класс **Exception** используется для обработки исключений вашей программой. Вы можете наследоваться от него для создания собственных типов исключений. Для распространённых ошибок уже существует класс **RuntimeException**, который может обрабатывать деление на ноль или определять ошибочную индексацию массива.

Класс **Error** служит для обработки ошибок в самом языке **Java** и на практике вам не придётся иметь с ним дело.

Если вы хотите увидеть описание ошибки, то параметр **e** и поможет увидеть ёго.

```
catch (ArithmeticException e) {
        Toast.makeText(this, e + ": Нельзя котов делить на ноль!", Toast.LENGTH_LONG).show();
}
```

Оператор throw

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора **throw**. Код выглядит так:

```
throw экземпляр_Throwable
```

Вам нужно создать экземпляр класса **Throwable** или его наследников. Получить объект класса **Throwable** можно в операторе **catch** или стандартным способом через оператор **new**.

Мы могли бы написать такой код для кнопки:

```
Cat cat;

public void onClick(View view) {
   if(cat == null){
      throw new NullPointerException("Котик не инициализирован");
   }
}
```

Мы объявили объект класса **Cat**, но забыли его проинициализировать, например, в **onCreate()**. Теперь нажатие кнопки вызовет исключение, которое обработает система, а в логах мы можем прочитать сообщение об ошибке. Возможно, вы захотите использовать другое исключение, например, **throw new UnsupportedOperationException("Котик не инициализирован");**.

Оператор throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен задать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Для этого к объявлению метода добавляется конструкция **throws**, которая перечисляет типы исключений (кроме исключений Error и RuntimeException и их подклассов).

Общая форма объявления метода с оператором **throws**:

```
тип имя_метода(список_параметров) throws список_исключений {
   // код внутри метода
}

// Без изменений
public void createCat() throws NullPointerException {
		Toast.makeText(this, "Вы создали котёнка", Toast.LENGTH_LONG).show();
		throw new NullPointerException("Кота не существует");
}

// Щелчок кнопки
public void onClick(View v) {
		try {
			createCat();
		} catch (NullPointerException e) {
			// TODO: handle exception
			Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
		}
}
```

Мы поместили вызов метода в блок **try** и вызвали блок **catch** с нужным типом исключения. Теперь ошибки не будет.

Оператор finally

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм **finally**.

Ключевое слово **finally** создаёт блок кода, который будет выполнен после завершения блока **try/catch**, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет.
Оператор **finally** не обязателен, однако каждый оператор **try** требует наличия либо **catch**, либо **finally**.

Встроенные исключения Java

Существуют несколько готовых системных исключений. Большинство из них являются подклассами типа **RuntimeException** и их не нужно включать в список **throws**. Вот небольшой список непроверяемых исключений.

- ArithmeticException - арифметическая ошибка, например, деление на нуль
- ArrayIndexOutOfBoundsException - выход индекса за границу массива
- ArrayStoreException - присваивание элементу массива объекта несовместимого типа
- ClassCastException - неверное приведение
- EnumConstantNotPresentException - попытка использования неопределённого значения перечисления
- IllegalArgumentException - неверный аргумент при вызове метода
- IllegalMonitorStateException - неверная операция мониторинга
- IllegalStateException - некорректное состояние приложения
- IllegalThreadStateException - запрашиваемая операция несовместима с текущим потоком
- IndexOutofBoundsException - тип индекса вышел за допустимые пределы
- NegativeArraySizeException - создан массив отрицательного размера
- NullPointerException - неверное использование пустой ссылки
- NumberFormatException - неверное преобразование строки в числовой формат
- SecurityException - попытка нарушения безопасности
- StringIndexOutOfBounds - попытка использования индекса за пределами строки
- TypeNotPresentException - тип не найден
- UnsupportedOperationException - обнаружена неподдерживаемая операция

Список проверяемых системных исключений, которые можно включать в список **throws**.

- ClassNotFoundException - класс не найден
- CloneNotSupportedException - попытка клонировать объект, который не реализует интерфейс **Cloneable**
- IllegalAccessException - запрещен доступ к классу
- InstantiationException - попытка создать объект абстрактного класса или интерфейса
- InterruptedException - поток прерван другим потоком
- NoSuchFieldException - запрашиваемое поле не существует
- NoSuchMethodException - запрашиваемый метод не существует
- ReflectiveOperationException - исключение, связанное с рефлексией

Создание собственных классов исключений

Система не может предусмотреть все исключения, иногда вам придётся создать собственный тип исключения для вашего приложения. Вам нужно наследоваться от **Exception** (напомню, что этот класс наследуется

от **Trowable**) и переопределить нужные методы класса **Throwable**. Либо вы можете унаследоваться от уже существующего типа, который наиболее близок по логике с вашим исключением.

- final void addSuppressed(Throwable exception) - добавляет исключение в список подавляемых исключений (JDK 7)
- Throwable fillInStackTrace() - возвращает объект класса **Throwable**, содержащий полную трассировку стека.
- Throwable getCause() - возвращает исключение, лежащее под текущим исключение или null
- String getLocalizedMessage() - возвращает локализованное описание исключения
- String getMessage() - возвращает описание исключения
- StackTraceElement[] getStackTrace() - возвращает массив, содержащий трассировку стека и состояний из элементов класса **StackTraceElement**
- final Throwable[] getSuppressed() - получает подавленные исключения (JDK 7)
- Throwable initCause(Throwable exception) - ассоциирует исключение с вызывающим исключением. Возвращает ссылку на исключение.
- void printStackTrace() - отображает трассировку стека
- void printStackTrace(PrintStream stream) - посылает трассировку стека в заданный поток
- void printStackTrace(PrintWriter stream) - посылает трассировку стека в заданный поток
- void setStackTrace(StackTraceElement elements[]) - устанавливает трассировку стека для элементов (для специализированных приложений)
- String toString() - возвращает объект класса **String**, содержащий описание исключения.

Перехват произвольных исключений

Можно создать универсальный обработчик, перехватывающий любые типы исключения. Осуществляется это перехватом базового класса всех исключений **Exception**:

```
cacth(Exception e) {
   Log.w("Log", "Перехвачено исключение");
}
```

Подобная конструкция не упустит ни одного исключения, поэтому её следует размещать в самом конце списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

Основные правила обработки исключений

Используйте исключения для того, чтобы:

- обработать ошибку на текущем уровне (избегайте перехватывать исключения, если не знаете, как с ними поступить)

- исправить проблему и снова вызвать метод, возбудивший исключение
- предпринять все необходимые действия и продолжить выполнение без повторного вызова действия
- попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод
- сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень
- сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень
- завершить работу программы
- упростить программу (если используемая схема обработки исключений делает все только сложнее, значит, она никуда не годится)
- добавить вашей библиотеке и программе безопасности

реализация ввода-вывода в java

Java имеет в своём составе множество классов, связанных с вводом/выводом данных. Рассмотрим некоторые из них.

Класс File

В отличие от большинства классов ввода/вывода, класс **File** работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

Подробнее о классе java.io.File

Поток

При работе с данными ввода/вывода вам будет часто попадаться термин **Поток** (Stream). Поток - это абстрактное значение источника или приёмника данных, которые способны обрабатывать информацию. Вы в реальности не видите, как действительно идёт обработка данных в устройствах ввода/вывода, так как это сложно и вам это не нужно. Это как с телевизором - вы не знаете, как сигнал из кабеля превращается в картинку на экране, но вполне можете переключаться между каналами через пульт.

Есть два типа потоков: байтовые и символьные. В некоторых ситуациях символьные потоки более эффективны, чем байтовые.

За ввод и вывод отвечают разные классы Java. Классы, производные от базовых классов **InputStream** или **Reader**, имеют методы с именами **read()** для чтения отдельных байтов или массива

байтов (отвечают за ввод данных). Классы, производные от классов **OutputStream** или **Write**, имеют методы с именами **write()** для записи одиночных байтов или массива байтов (отвечают за вывод данных).

Подробнее о классе InputStream

Класс OutputStream

Класс **OutputStream** - это абстрактный класс, определяющий потоковый байтовый вывод.

В этой категории находятся классы, определяющие, куда направляются ваши данные: в массив байтов (но не напрямую в String; предполагается что вы сможете создать их из массива байтов), в файл или канал.

BufferedOutputStream

Буферизированный выходной поток

ByteArrayOutputStream

Создает буфер в памяти. Все данные, посылаемые в этот поток, размещаются в созданном буфере

DataOutputStream

Выходной поток, включающий методы для записи стандартных типов данных Java

FileOutputStream

Отправка данных в файл на диске. Реализация класса OutputStream

ObjectOutputStream

Выходной поток для объектов

PipedOutputStream

Реализует понятие выходного канала.

FilterOutputStream

Абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства.

Методы класса:

- int close() - закрывает выходной поток. Следующие попытки записи передадут исключение IOException
- void flush() - финализирует выходное состояние, очищая все буферы вывода
- abstract void write (int oneByte) - записывает единственный байт в выходной поток
- void write (byte[] buffer) - записывает полный массив байтов в выходной поток
- void write (byte[] buffer, int offset, int count) - записывает диапазон из *count* байт из массива, начиная с смещения *offset*

BufferedOutputStream

Класс **BufferedOutputStream** не сильно отличается от класса **OutputStream**, за исключением дополнительного метода **flush()**, используемого для обеспечения записи данных в буферизируемый поток. Буферы вывода нужно для повышения производительности.

ByteArrayOutputStream

Класс **ByteArrayOutputStream** использует байтовый массив в выходном потоке. Метод **close()** можно не вызывать.

DataOutputStream

Класс **DataOutputStream** позволяет писать элементарные данные в поток через интерфейс **DataOutput**, который определяет методы, преобразующие элементарные значения в форму последовательности байтов. Такие потоки облегчают сохранение в файле двоичных данных.

Класс **DataOutputStream** расширяет класс **FilterOutputStream**, который в свою очередь, расширяет класс **OutputStream**.

Методы интерфейса **DataOutput**:

- writeDouble(double value)
- writeBoolean(boolean value)
- writeInt(int value)

FileOutputStream

Класс **FileOutputStream** создаёт объект класса **OutputStream**, который можно использовать для записи байтов в файл. Создание нового объекта не зависит от того, существует ли заданный файл, так как он создаёт его перед открытием. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

Классы символьных потоков

Символьные потоки имеют два основных абстрактных класса **Reader** и **Writer**, управляющие потоками символов Unicode.

Reader

Методы класса **Reader**:

- abstract void close() - закрывает входной поток. Последующие попытки чтения передадут исключение IOException
- void mark(int readLimit) - помещает метку в текущую позицию во входном потоке
- boolean markSupported() - возвращает *true*, если поток поддерживает методы **mark()** и **reset()**
- int read() - возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает значение -1. Есть и другие перегруженные версии метода
- boolean ready() - возвращает значение *true*, если следующий запрос не будет ожидать.
- void reset() - сбрасывает указатель ввода в ранее установленную позицию метки
- logn skip(long charCount) - пропускает указанное число символов ввода, возвращая количество действительно пропущенных символов

BufferedReader
: Буферизированный входной символьный поток

CharArrayReader
: Входной поток, который читает из символьного массива

FileReader
: Входной поток, читающий файл

FilterReader
: Фильтрующий читатель

InputStreamReader
: Входной поток, транслирующий байты в символы

LineNumberReader
: Входной поток, подсчитывающий строки

PipedReader
: Входной канал

PushbackReader
: Входной поток, позволяющий возвращать символы обратно в поток

Reader
: Абстрактный класс, описывающий символьный ввод

StringReader
: Входной поток, читающий из строки

Класс BufferedReader

Класс **BufferedReader** увеличивает производительность за счёт буферизации ввода.

Класс CharArrayReader

Класс **CharArrayReader** использует символьный массив в качестве источника.

Класс FileReader

Класс **FileReader**, производный от класса **Reader**, можно использовать для чтения содержимого файла. В конструкторе класса нужно указать либо путь к файлу, либо объект типа **File**.

Writer

Класс **Writer** - абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение **IOException**.

Методы класса:

- Writer append(char c) - добавляет символ в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- Writer append(CharSequence csq) - добавляет символы в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- Writer append(CharSequence csq, int start, int end) - добавляет диапазон символов в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- abstract void close() - закрывает вызывающий поток
- abstract void flush() - финализирует выходное состояние так, что все буферы очищаются
- void write(int oneChar) - записывает единственный символ в вызывающий выходной поток. Есть и другие перегруженные версии метода

BufferedWriter
: Буферизированный выходной символьный поток

CharArrayWriter

: Выходной поток, который пишет в символьный массив

FileWriter

: Выходной поток, пишущий в файл

FilterWriter

Фильтрующий писатель

OutputStreamWriter

Выходной поток, транслирующий байты в символы

PipedWriter

Выходной канал

PrintWriter

Выходной поток, включающий методы print() и println()

StringWriter

Выходной поток, пишущий в строку

Writer

Абстрактный класс, описывающий символьный вывод

Класс BufferedWriter

Класс **BufferedWriter** - это класс, производный от класса **Writer**,который буферизует вывод. С его помощью можно повысить производительность за счёт снижения количества операций физической записи в выходное устройство.

Класс CharArrayWriter

Класс **CharArrayWriter** использует массив для выходного потока.

Класс FileWriter

Класс **FileWriter** создаёт объект класса, производного от класса **Writer**, который вы можете применять для записи файла. Есть конструкторы, которые позволяют добавить вывод в конец файла. Создание объекта не зависит от наличия файла, он будет создан в случае необходимости. Если файл существует и он доступен только для чтения, то передаётся исключение **IOException**.

Чтение и запись файлов

Существует множество классов и методов для чтения и записи файлов. Наиболее распространённые из них - классы **FileInputStream** и **FileOutputStream**, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, нужно создать объект одного из этих файлов, указав имя файла в качестве аргумента конструктора.

```
FileInputStream(String filename) throws FileNotFoundException
FileOutputStream(String filename) throws FileNotFoundException
```

В **filename** нужно указать имя файла, который вы хотите открыть. Если при создании входного потока файл не существует, передаётся исключение **FileNotFoundException**. Аналогично для выходных потоков, если файл не может быть открыт или создан, также передаётся исключение. Сам класс исключения происходит от класса **IOException**. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

После завершения работы с файлом, его необходимо закрыть с помощью метода **close()** для освобождения системных ресурсов. Незакрытый файл приводит к утечке памяти.

В JDK 7 метод **close()** определяется интерфейсом **AutoCloseable** и можно явно не закрывать файл, а использовать новый оператор **try-c-ресурсами**, что для Android пока не слишком актуально.

Чтобы читать файл, нужно вызвать метод **read()**. Когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. Когда будет достигнут конец файла, то метод вернёт значение -1. Примеры использования методов есть в различных статьях на сайте.

Иногда используют вариант, когда метод **close()** помещается в блок **finally**. При таком подходе все методы, которые получают доступ к файлу, содержатся в пределах блока **try**, а блок **finally** используется для закрытия файла. Таким образом, независимо от того, как закончится блок **try**, файл будет закрыт.

Так как исключение **FileNotFoundException** является подклассом **IOException**, то не обязательно обрабатывать два исключения отдельно, а оставить только **IOException**, если вам не нужно отдельно обрабатывать разные причины неудачного открытия файла. Например, если пользователь вводит вручную имя файла, то более конкретное исключение будет к месту.

Для записи в файл используется метод **write()**.

```
void write(int value) throws IOException
```

Метод пишет в файл байт, переданный параметром **value**. Хотя параметр объявлена как целочисленный, в файл записываются только младшие восемь бит. При ошибке записи передаётся исключение.

В JDK 7 есть способ автоматического управления ресурсами:

```
try (спецификация_ресурса) {
    // использование ресурса
}
```

Когда в Android будет полноценная поддержка JDK 7, то дополним материал.

Буферизированное чтение из файла - BufferedReader

Чтобы открыть файл для посимвольного чтения, используется класс **FileInputReader**; имя файла задаётся в виде строки (String) или объекта **File**. Ускорить процесс чтения помогает буферизация ввода, для этого полученная ссылка передаётся в конструктор класса **BufferedReader**. Так как в интерфейсе класса имеется метод **readLine()**, все необходимое для чтения имеется в вашем распоряжении. При достижении конца файла метод **readLine()** возвращает ссылку **null**.

Вывод в файл - FileWriter

Объект **FileWriter** записывает данные в файл. При вводе/выводе практически всегда применяется буферизация, поэтому используется **BufferedWriter**.

Когда данные входного потока исчерпываются, метод **readLine()** возвращает **null**. Для потока явно вызывается метод **close()**; если не вызвать его для всех выходных файловых потоков, в буферах могут остаться данные, и файл получится неполным.

Сохранение и восстановление данных - PrintWriter

PrintWriter форматирует данные так, чтобы их мог прочитать человек. Однако для вывода информации, предназначенной для другого потока, следует использовать классы **DataOutputStream** для записи данных и **DataInputStream** для чтения данных.

Единственным надежным способом записать в поток **DataOutputStream** строку так, чтобы ее можно было потом правильно считать потоком **DataInputStream**, является кодирование UTF-8, реализуемое методами **readUTF()** и **writeUTF()**. Эти методы позволяют смешивать строки и другие типы данных, записываемые потоком **DataOutputStream**, так как вы знаете, что строки будут правильно сохранены в Юникоде и их будет просто воспроизвести потоком **DataInputStream**.

Метод **writeDouble()** записывает число **double** в поток, а соответствующий ему метод **readDouble()** затем восстанавливает его (для других типов также существуют подобные методы).

RandomAccessFile - Чтение/запись файлов с произвольным доступом

Работа с классом **RandomAccessFile** напоминает использование совмещенных в одном классе потоков **DataInputStream** и **DataOutputStream** (они реализуют те же интерфейсы **DataInput** и **DataOutput**).

Кроме того, метод **seek()** позволяет переместиться к определенной позиции и изменить хранящееся там значение.

При использовании **RandomAccessFile** необходимо знать структуру файла. Класс **RandomAccessFile** содержит методы для чтения и записи примитивов и строк UTF-8.

RandomAccessFile может открываться в режиме чтения ("r") или чтения/записи ("rw"). Также есть режим "rws", когда файл открывается для операций чтения-записи и каждое изменение данных файла немедленно записывается на физическое устройство.

Исключения ввода/вывода

В большинстве случаев у классов ввода/вывода используется исключение **IOException**. Второе исключение **FileNotFoundException** передаётся в тех случаях, когад файл не может быть открыт. Данное исключение происходит от **IOException**, поэтому оба исключения можно обрабатывать в одном блоке **catch**, если у вас нет нужды обрабатывать их по отдельности.

механизмы выделения и уничтожение памяти jvm
наверное, все, работающие с Java, знают об управлении памяти на уровне, что для ее распределения используется сборщик мусора. Не все, к сожалению, знают, как именно этот сборщик (-и) работает, и как именно организована память внутри процесса Java.

Из-за этого иногда делается неверный вывод, что memory leaks в Java не бывает, и слишком задумываться о памяти не надо.

Итак, память процесса различается на heap (куча) и non-heap (стек) память, и состоит из 5 областей (memory pools, memory spaces):

• Eden Space (heap) – в этой области выделятся память под все создаваемые из программы объекты. Большая часть объектов живет недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора это области памяти, не перемещаются в другие области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), GC выполняет быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора она занимает мало времени, и затрагивает только эту область памяти — очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.

• Survivor Space (heap) – сюда перемещаются объекты из предыдущей, после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.

• Tenured (Old) Generation (heap) — Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и проч.). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.

• Permanent Generation (non-heap) – Здесь хранится метаинформация, используемая JVM (используемые классы, методы и т.п.). В частноси

• Code Cache (non-heap) — эта область используется JVM, когда включена JIT-компиляция, в ней кешируется скомпилированный платформенно — зависимый код.

Сборщик мусора
Управление памятью - это процесс размещения новых объектов и удаление неиспользуемых объектов, чтобы освободить место для этих новых ассигнований объектов. Традиционным для языков программирование способом управления памятью является ручной. Его сущность является в следующем:

- Для создания объекта в динамической памяти программист явно вызывает команду выделения памяти. Эта команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней.
- До тех пор, пока созданный объект нужен для работы программы, программа обращается к нему через ранее сохранённый указатель.
- Когда надобность в объекте проходит, программист явно вызывает команду освобождения памяти, передавая ей указатель на удаляемый объект.
- Ручное управление памятью допускает потенциально возможные две проблемы: висячие ссылки и утечки памяти.

Висячая ссылка — это оставшаяся в использовании ссылка на объект, который уже удалён. После удаления объекта все сохранившиеся в программе ссылки на него становятся «висячими». Память, занимаемая ранее объектом, может быть передана операционной системе и стать недоступной, или быть использована для размещения нового объекта в той же программе.

Утечка памяти - процесс неконтролируемого уменьшения объёма свободной оперативной или виртуальной памяти компьютера, связанный с ошибками в работающих программах, вовремя не освобождающих ненужные уже участки памяти.

Все проблемы ручного способа управления памяти в Java решает автоматический сборщик мусора. Но пред ознакомлением со сборщиком мусора, нужно разъяснить понятие кучи(heap).

Куча

В Java все объекты находятся в области памяти под названием куча. Куча создается, когда JVM запускается и может увеличиваться или уменьшаться в размерах во время выполнения приложения. Когда куча становится полной, происходит механизм сборки мусора. Все объекты, которые никогда больше не будут использоватся, очищаются. тем самым освобождая место для новых объектов.

Также нужно обратить внимание, что JVM использует больше памяти, чем занимает куча. Например, для методов Java и стеков потоков выделяется память отдельно от кучи.

Размер кучи зависит от используемой платформы, но, как правило, это где-то между 2 и 128 Кб.

Garbage Collection

Механизм сборки мусора - это процесс освобождения места в куче, для возможности добавления новых объектов.

Объекты создаются посредством оператора new, тем самым присваивая объекту ссылку. Закончив работу с объектом, вы просто перестаете на него ссылаться — достаточно присвоить переменной ссылку на другой объект или значение null либо прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом. Объекты, ссылки на которые отсутствуют, принято называть мусором (garbage), который будет удален.

Виртуальная машина Java, применяя механизм сборки мусора, гарантирует, что любой объект, обладающий ссылками, остается в памяти — все объекты, которые недостижимы из выполняемого кода ввиду отсутствия ссылок на них, удаляются с высвобождением отведенной для них памяти. Точнее говоря, объект не попадает в сферу действия процесса сборки мусора, если он достижим посредством цепочки ссылок, начиная с корневой (root) ссылки, т.е. ссылки, непосредственно существующей в выполняемом коде.

Память освобождается сборщиком мусора по его собственному "усмотрению" и обычно только в тех случаях, когда для дальнейшей работы программы необходим фрагмент свободной памяти большего размера, нежели тот, который имеется в распоряжении виртуальной машины в данный момент, либо если сборщик мусора "предвидит" потенциальную нехватку памяти в ближайшем будущем. Программа может (часто именно так и происходит) завершить работу, не исчерпав ресурсов свободной памяти или даже не приблизившись к этой черте, и поэтому ей так и не потребуются "услуги" сборщика мусора. Объект считается "более недостижимым", если ни одна из переменных в коде, выполняемом в данный момент, не содержит ссылок на него либо цепочка ссылок, которая могла бы связать объект с некоторой переменной программы, обрывается.

Мусор собирается системой без вашего вмешательства, но это не значит, что процесс не требует внимания вовсе. Необходимость создания и удаления большого количества объектов существенным образом сказывается на производительности приложений, и если быстродействие программы является важным фактором, следует тщательно обдумывать решения, связанные с созданием объектов, — это, в свою очередь, уменьшит и объем мусора, подлежащего утилизации.

Создание GUI в Java. Swing
Общие сведения об интерфейсах

Abstract Window Toolkit

AWT была первой попыткой Sun создать графический интерфейс для Java. Они пошли легким путем и просто сделали прослойку на Java, которая вызывает методы из библиотек, написанных на C. Библиотечные методы создают и используют графические компоненты операционной среды. С одной стороны, это хорошо, так как

программа на Java похожа на остальные программы в рамках данной ОС. Но с другой стороны, нет никакой гарантии, что различия в размерах компонентов и шрифтах не испортят внешний вид программы при запуске ее на другой платформе. Кроме того, чтобы обеспечить мультиплатформенность, пришлось унифицировать интерфейсы вызовов компонентов, из-за чего их функциональность получилась немного урезанной. Да и набор компонентов получился довольно небольшой. К примеру, в AWT нет таблиц, а в кнопках не поддерживается отображение иконок.

Использованные ресурсы AWT старается освобождать автоматически. Это немного усложняет архитектуру и влияет на производительность. Освоить AWT довольно просто, но написать что-то сложное будет несколько затруднительно. Сейчас ее используют разве что для апплетов.

Достоинства:

- часть JDK;
- скорость работы;
- графические компоненты похожи на стандартные.

Недостатки:

- использование нативных компонентов налагает ограничения на использование их свойств. Некоторые компоненты могут вообще не работать на «неродных» платформах;
- некоторые свойства, такие как иконки и всплывающие подсказки, в AWT вообще отсутствуют;
- стандартных компонентов AWT очень немного, программисту приходится реализовывать много кастомных;
- программа выглядит по-разному на разных платформах (может быть кривоватой).

заключение:

В настоящее время AWT используется крайне редко — в основном в старых проектах и апплетах. Oracle припрятал обучалки и всячески поощряет переход на Swing. Оно и понятно, прямой доступ к компонентам оси может стать серьезной дырой в безопасности.

Swing

Вслед за AWT Sun разработала набор графических компонентов под названием Swing. Компоненты Swing полностью написаны на Java. Для отрисовки используется 2D, что принесло с собой сразу несколько преимуществ. Набор стандартных компонентов значительно превосходит AWT по разнообразию и функциональности. Стало легко создавать новые компоненты, наследуясь от существующих и рисуя все, что душе угодно. Стала возможной поддержка различных стилей и скинов. Вместе с тем скорость работы первых версий Swing оставляла желать лучшего. Некорректно написанная программа и вовсе могла повесить винду намертво.

Тем не менее благодаря простоте использования, богатой документации и гибкости компонентов Swing стал, пожалуй, самым популярным графическим фреймворком в Java. На его базе появилось много расширений, таких как SwingX, JGoodies, которые значительно упрощают создание сложных пользовательских интерфейсов. Практически все популярные среды программирования Java включают графические редакторы для Swing-форм. Поэтому разобраться и начать использовать Swing не составит особого труда.

Достоинства:

- часть JDK, не нужно ставить дополнительных библиотек;
- по Swing гораздо больше книжек и ответов на форумах. Все проблемы, особенно у начинающих, гуглу досконально известны;
- встроенный редактор форм почти во всех средах разработки;
- на базе свинга есть много расширений типа SwingX;
- поддержка различных стилей (Look and feel).

Недостатки:

- окно с множеством компонентов начинает подтормаживать;
- работа с менеджерами компоновки может стать настоящим кошмаром в сложных интерфейсах.

Заключение:

Swing жил, Swing жив, Swing будет жить. Хотя Oracle и старается продвигать JavaFX, на сегодняшний день Swing остается самым популярным фреймворком для создания пользовательских интерфейсов на Java.

Standard Widget Toolkit

SWT был разработан в компании IBM в те времена, когда Swing еще был медленным, и сделано это было в основном для продвижения среды программирования Eclipse. SWT, как и AWT, использует компоненты операционной системы, но для каждой платформы у него созданы свои интерфейсы взаимодействия. Так что для каждой новой системы тебе придется поставлять отдельную JAR-библиотеку с подходящей версией SWT. Это позволило более полно использовать существующие функции компонентов на каждой оси. Недостающие функции и компоненты были реализованы с помощью 2D, как в Swing. У SWT есть много приверженцев, но, положа руку на сердце, нельзя не согласиться, что получилось не так все просто, как хотелось бы. Новичку придется затратить на изучение SWT намного больше времени, чем на знакомство с тем же Swing. Кроме того, SWT возлагает задачу освобождения ресурсов на программиста, в связи с чем ему нужно быть особенно внимательным при написании кода, чтобы случайное исключение не привело к утечкам памяти.

Достоинства:

- использует компоненты операционной системы — скорость выше;
- Eclipse предоставляет визуальный редактор форм;
- обширная документация и множество примеров;
- возможно использование AWT- и Swing-компонентов.

Недостатки:

- для каждой платформы необходимо поставлять отдельную библиотеку;
- нужно все время следить за использованием ресурсов и вовремя их освобождать;
- сложная архитектура, навевающая суицидальные мысли после тщетных попыток реализовать кастомный интерфейс.

Заключение:

Видно, что в IBM старались. Но получилось уж очень на любителя…

JavaFX

JavaFX можно без преувеличения назвать прорывом. Для отрисовки используется графический конвейер, что значительно ускоряет работу приложения. Набор встроенных компонентов обширен, есть даже отдельные компоненты для отрисовки графиков. Реализована поддержка мультимедийного контента, множества эффектов

отображения, анимации и даже мультитач. Внешний вид всех компонентов можно легко изменить с помощью CSS-стилей. И самое прекрасное — в JavaFX входит набор утилит, которые позволяют сделать родной инсталлятор для самых популярных платформ: exe или msi для Windows, deb или rpm для Linux, dmg для Mac. На сайте Oracle можно найти подробную документацию и огромное количество готовых примеров. Это превращает программирование с JavaFX в легкое и приятное занятие.

Достоинства:

- быстрая работа за счет графического конвейера;
- множество различных компонентов;
- поддержка стилей;
- утилиты для создания установщика программы;
- приложение можно запускать как десктопное и в браузере как часть страницы.

Недостатки:

- фреймворк еще разрабатывается, поэтому случаются и падения и некоторые глюки;
- JavaFX пока не получил широкого распространения.

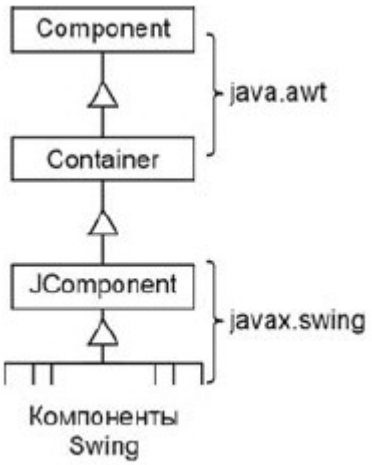
Заключение:

Хорошая работа, Oracle. Фреймворк оставляет только позитивные впечатления. Разобраться несложно, методы и интерфейсы выглядят логичными. Хочется пользоваться снова и снова!

Основные концепции SWING

Вслед за **AWT** Sun разработала графическую библиотеку компонентов **Swing**, полностью написанную на Java. Для отрисовки используется 2D, что принесло с собой сразу несколько преимуществ. Набор стандартных компонентов значительно превосходит AWT по разнообразию и функциональности. Swing позволяет легко создавать новые компоненты, наследуясь от существующих, и поддерживает различные стили и скины.

Создатели новой библиотеки пользовательского интерфейса **Swing** не стали «изобретать велосипед» и в качестве основы для своей библиотеки выбрали AWT. Конечно, речь не шла об использовании конкретных тяжеловесных компонентов AWT (представленных классами Button, Label и им подобными). Нужную степень гибкости и управляемости обеспечивали только легковесные компоненты. На диаграмме наследования представлена связь между AWT и Swing.



Важнейшим отличием **Swing** от AWT является то, что компоненты Swing вообще не связаны с операционной системой и поэтому гораздо более стабильны и быстры. Такие компоненты в Java называются легковесными (lightweight), и понимание основных принципов их работы во многом объяснит работу Swing.

Swing контейнеры высшего уровня

Для создания графического интерфейса приложения необходимо использовать специальные компоненты библиотеки Swing, называемые контейнерами высшего уровня (top level containers). Они представляют собой окна операционной системы, в которых размещаются компоненты пользовательского интерфейса. К контейнерам высшего уровня относятся окна JFrame и JWindow, диалоговое окно JDialog, а также апплет JApplet (который не является окном, но тоже предназначен для вывода интерфейса в браузере, запускающем этот апплет). Контейнеры высшего уровня Swing представляют собой тяжеловесные компоненты и являются исключением из общего правила. Все остальные компоненты Swing являются легковесными.

Простой **Swing** пример создания оконного интерфейса **JFrame**.

```
import java.awt.Dimension;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class JFrameTest
{
   public static void createGUI()
   {
      JFrame frame = new JFrame("Test frame");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JLabel label = new JLabel("Test label");
      frame.getContentPane().add(label);

      frame.setPreferredSize(new Dimension(200, 100));

      frame.pack();
      frame.setVisible(true);
   }

  public static void main(String[] args)
  {
     JFrame.setDefaultLookAndFeelDecorated(true);
     javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
           createGUI();
        }
     });
  }
}
```

Конструктор **JFrame()** без параметров создает пустое окно. Конструктор **JFrame(String title)** создает пустое окно с заголовком title. Чтобы создать простейшую программу с пустым окном необходимо использовать следующие методы :

- setSize(int width, int height) - определение размеров окна;
- setDefaultCloseOperation(int operation) - определение действия при завершении программы;
- setVisible(boolean visible) - сделать окно видимым.

Если не определить размеры окна, то оно будет иметь нулевую высоту независимо от того, что в нем находится. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.

Метод setDefaultCloseOperation определяет действие, которое необходимо выполнить при "выходе из программы". Для этого следует в качестве параметра operation передать константу EXIT_ON_CLOSE, описанную в классе JFrame.

По умолчанию окно создается невидимым. Чтобы отобразить окно на экране вызывается метод setVisible с параметром true. Если вызвать его с параметром false, окно станет невидимым.

Графический интерфейс **java swing** примера создания окна **JFrame** представлен на следующем рисунке.

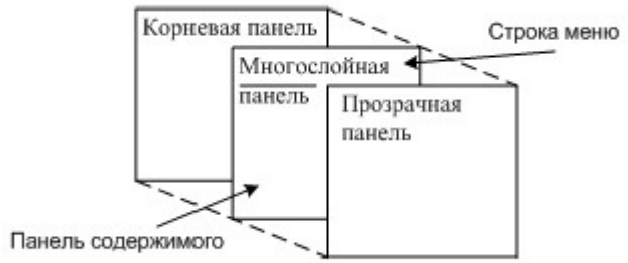
Для подключения библиотеки **Swing** в приложении необходимо импортировать библиотеку **javax.swing**.

Корневая панель JRootPane

Каждый раз, как только создается контейнер высшего уровня, будь то обычное окно, диалоговое окно или апплет, в конструкторе этого контейнера создается **корневая панель JRootPane**. Контейнеры высшего уровня Swing следят за тем, чтобы другие компоненты не смогли "пробраться" за пределы JRootPane.

Корневая палель **JRootPane** добавляет в контейнеры свойство "глубины", обеспечивая возможность не только размещать компоненты один над другим, но и при необходимости менять их местами, увеличивать или уменьшать глубину расположения компонентов. Такая возможность необходима при создании многодокументного приложения **Swing**, у которого окна представляют легковесные компоненты, располагающиеся друг над другом, а также выпадающими (контекстными) меню и всплывающими подсказками.

На следующем рисунке наглядно представлена структура корневой панели **JRootPane**.



Корневая панель **JRootPane** представляет собой контейнер, унаследованный от базового класса Swing JComponent. В этом контейнере за расположение компонентов отвечает специальный менеджер расположения, реализованный во внутреннем классе RootPaneLayout. Этот менеджер расположения отвечает за то, чтобы все составные части корневой панели размещались так, как им следует: многослойная панель занимает все пространство окна; в ее слое FRAME_CONTENT_LAYER располагаются строка меню и панель содержимого, а над всем этим располагется прозрачная панель.

Все составляющие корневой панели **JRootPane** можно получить или изменить. Для этого у нее есть набор методов get/set. Программным способом **JRootPane** можно получить с использованием метода getRootPane().

Кроме контейнеров высшего уровня корневая панель применяется во внутренних окнах JInternalFrame, создаваемых в многодокументных приложениях и располагающихся на "рабочем столе" JDesktopPane. Это позволяет забыть про то, что данные окна представляют собой обычные легковесные компоненты, и работать с ними как с настоящими контейнерами высшего уровня.

Многослойная панель JLayeredPane

В основании корневой панели (контейнера) лежит так называемая многослойная панель **JLayeredPane**, занимающая все доступное пространство контейнера. Именно в этой панели располагаются все остальные части корневой панели, в том числе и все компоненты пользовательского интерфейса.

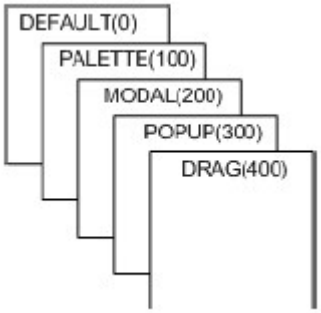
JLayeredPane используется для добавления в контейнер свойства глубины (depth). То есть, многослойная панель позволяет организовать в контейнере третье измерение, вдоль которого располагаются слои (layers)

компонента. В обычном контейнере расположение компонента определяется прямоугольником, который показывает, какую часть контейнера занимает компонент. При добавлении компонента в многослойную панель необходимо указать не только прямоугольник, занимаемый компонентом, но и слой, в котором он будет располагаться. Слой в многослойной панели определяется целым числом. Чем больше определяющее слой число, тем выше слой находится.

Первый добавленный в контейнер компонент оказывается выше компонентов, добавленных позже. Чаще всего разработчик не имеет дело с позициями компонентов. При добавлении компонентов их положение меняются автоматически. Тем не менее многослойная панель позволяет менять позиции компонентов динамически, уже после их добавления в контейнер.

Возможности многослойной панели широко используются некоторыми компонентами **Swing**. Особенно они важны для многодокументных приложений, всплывающих подсказок и меню.
Многодокументные **Swing** приложения задействуют специальный контейнер **JDesktopPane** («рабочий стол»), унаследованный от **JLayeredPane**, в котором располагаются внутренние окна Swing. Самые важные функции многодокументного приложения — расположение «активного» окна над другими, сворачивание окон, их перетаскивание — обеспечиваются механизмами многослойной панели. Основное преимущество от использования многослойной панели для всплывающих подсказок и меню — это ускорение их работы. Вместо создания для каждой подсказки или меню нового тяжеловесного окна, располагающегося над компонентом, в котором возник запрос на вывод подсказки или меню, **Swing** создает быстрый легковесный компонент. Этот компонент размещается в достаточно высоком слое многослойной панели выше в стопке всех остальных компонентов и используется для вывода подсказки или меню.

Многослойная панель позволяет организовать неограниченное количество слоев.
Структура **JLayeredPane** включает несколько стандартных слоев, которые и используются всеми компонентами Swing, что позволяет обеспечить правильную работу всех механизмов многослойной панели. Стандартные слои JLayeredPane представлены на следующем рисунке.



Default

Слой Default используется для размещения всех обычных компонентов, которые добавляются в контейнер. В этом слое располагаются внутренние окна многодокументных приложений.

Palette

Слой Palette предназначен для размещения окон с набором инструментов, которые обычно перекрывают остальные элементы интерфейса. Создавать такие окна позволяет панель JDesktopPane, которая размещает их в этом слое.

Modal

Слой Modal планировался для размещения легковесных модальных диалоговых окон. Однако такие диалоговые окна пока не реализованы, так что этот слой в Swing в настоящее время не используется.

Popup

Наиболее часто используемый слой, служащий для размещения всплывающих меню и подсказок.

Drag

Самый верхний слой. Предназначен для операций перетаскивания (drag and drop), которые должны быть хорошо видны в интерфейсе программы.

Панель содержимого ContentPane

Панель содержимого ContentPane - это следующая часть корневой панели, которая используется для размещения компонентов пользовательского интерфейса программы. **ContentPane** занимает большую часть пространства многослойной панели (за исключением места, занимаемого строкой меню). Чтобы панель содержимого не закрывала добавляемые впоследствии в окно компоненты, многослойная панель размещает ее в специальном очень низком слое с названием FRAME_CONTENT_LAYER, с номером -30000.

Обратиться к панели содержимого можно методом **getContentPane()** класса JFrame. С помощью метода add(Component component) можно добавить на нее любой элемент управления. Заменить **ContentPane** любой другой панелью типа JPanel можно методом **setContentPane()**

Пример добавления кнопки в панель содержимого :

```
JButton newButton = new JButton();
getContentPane().add(newButton);
```

Строка меню JMenuBar

Одной из важных особенностей использования корневой панели JRootPane в Swing, является необходимость размещения в окне строки меню **JMenuBar**. Серьезное приложение нельзя построить без какого-либо меню для получения доступа к функциям программы. Библиотека Swing предоставляет прекрасные возможности для создания удобных меню JMenuBar, которые также являются легковесными компонентами.

Строка меню **JMenuBar** размещается в многослойной панели в специальном слое FRAME_CONTENT_LAYER и занимает небольшое пространство в верхней части окна. По размерам в длину строка меню равна размеру окна. Ширина строки меню зависит от содержащихся в ней компонентов.

Корневая панель следит, чтобы панель содержимого и строка меню **JMenuBar** не перекрывались. Если строка меню не требуется, то корневая панель использует все пространство для размещения панели содержимого.

Коллекции в Java

Коллекции в Java

Java Коллекции являются одним из столпов Java Core. Они используются почти в каждом приложении, поэтому мы просто обязаны уметь использовать Java Collections Framework эффективно.

Что такое Коллекции?

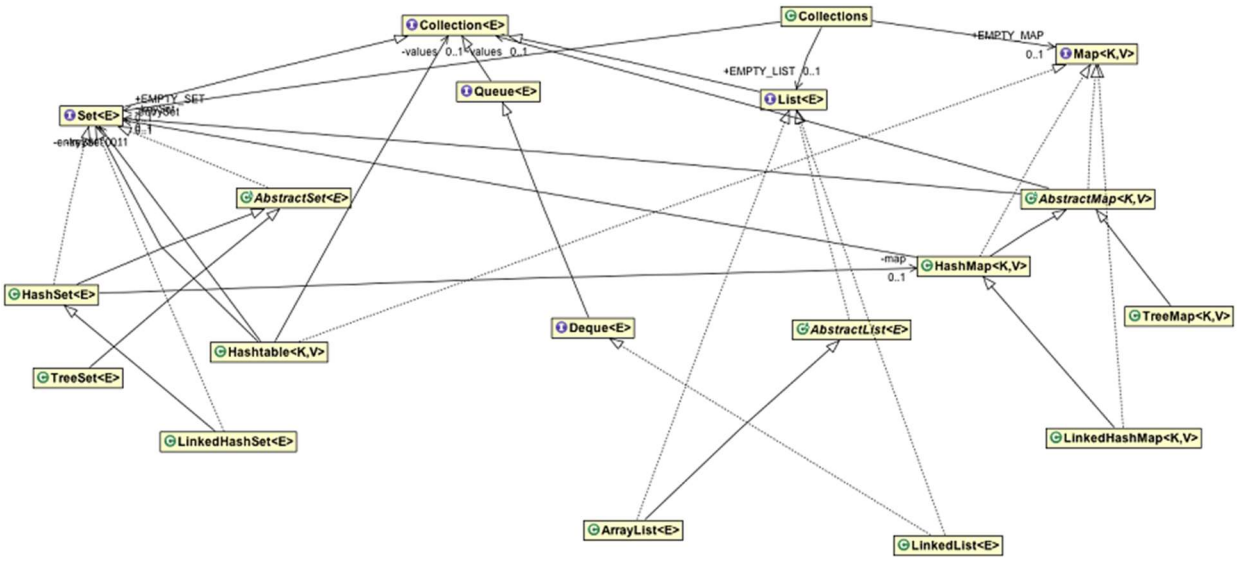
Коллекции — это контейнеры, группы элементов, которые представляют собой единое целое.

Например: банка конфет, список имен и т.д. Коллекции используются почти в каждом языке программирования и Java не является исключением. Как только коллекции появились в Java, то насчитывали всего несколько классов: Vector, Stack, Hashtable, Array. Но уже в Java 1.2 появился полноценный Java Collections Framework, с которым мы и будем сегодня знакомиться.

Коллекции в Java состоят нескольких частей

- **Интерфейсы:** В коллекциях интерфейсы обеспечивают абстрактный тип данных для представления коллекции java.util.Collection — корневого интерфейса фреймворка. Он находится на вершине иерархии Коллекций. Он содержит наиболее важные методы: size(), iterator(), add(), remove(), clear(). Каждая коллекция должна реализовывать эти методы. Также есть другие важные интерфейсы java.util.List, java.util.Set,java.util.Queue и java.util.Map.Map является единственным интерфейсом, который не наследует интерфейс Collection, но является неотъемлемой частью коллекций. Все интерфейсы фреймворка находятся в пакете java.util.

- **Реализация**: Java предоставляет готовые классы с реализацией вышеупомянутых коллекций. Мы можем использовать их для создания новых типов коллекций в нашей программе. С помощью классов ArrayList, LinkedList, HashMap, TreeMap, HashSet, TreeSet можно решить огромное количество задач, но если нам нужна специальная реализация той или иной коллекции, мы можем наследовать её и работать со своей реализацией. В Java 1.5 придумали потокобезопасные коллекции, которые позволили изменять содержимое коллекции время итерации по элементам. Наиболее популярными являются: CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet. Эти классы находятся в пакете java.util.concurrent. Все классы коллекций находятся в пакетах java.util и java.util.concurrent.
- **Алгоритмы: алгоритмы — это полезны методы, которые решают тривиальные задачи, например: поиск, сортировка и перетасовка элементов коллекции.**
- Ниже на диаграмме классов показана иерархия Java Collections Framework. Для простоты я включил только часто используемые интерфейсы и классы.



Преимущества Java Collections Framework

В Java Collections Framework есть следующие преимущества:

- **Требует меньше усилий.** Фреймворк располагает множеством распространенных типов коллекций и полезных методов для манипуляции данными. Таким образом, мы можем сосредоточиться на бизнес-логике, а не разработке наших API.
- **Отличное качество** — использование хорошо проверенных коллекций увеличивает качество нашей программы.
- **Повторное использование и совместимость**

Интерфейсы коллекций

Интерфейсы являются основой Java Collections Framework. Обратите внимание, что все интерфейсы являются Generic, например public interface Collection<E>. Использование <E> — это указание типа объекта, который коллекция может содержать. Это помогает сократить ошибки времени выполнения с помощью проверки типов объектов во время компиляции.

Следует отметить, что платформа Java не предоставляет отдельные интерфейсы для каждого типа коллекций. Если какая-то операция не поддерживается, то реализация коллекции бросает UnsupportedOperationException.

Кратко по каждой коллекции

Интерфейс итератора (Iterator)

Итератор предоставляет методы для перебора элементов любой коллекции. Мы можем получить экземпляр итератора из коллекции с помощью метода iterator. Итераторы позволяют удалить элементы из базовой коллекции во время выполнения итерации.

Интерфейс множества (Set)

Набор представляет собой коллекцию, которая не может содержать повторяющиеся элементы. Этот интерфейс представляет математическую абстракцию для представления множеств в виде колоды карт.

Платформа Java содержит три реализации Set : HashSet, TreeSet и LinkedHashSet.Интерфейс Set не позволяет осуществлять произвольный доступ к элементу в коллекции. Мы можем использовать итератор или цикл по каждому элементу для перебора элементов.

Интерфейс Список (List)

Список представляет собой упорядоченный набор элементов и может содержать повторяющиеся элементы. Вы можете получить доступ к любому элементу по индексу. Список представляет собой динамический массив. Список является одним из наиболее используемых типов коллекций. ArrayList и LinkedList классы являются реализацией интерфейса List.

Вот небольшой пример использования:

```
1 List strList = new ArrayList<>();

2 //добавить в конец

3 strList.add(0, "0");

4 //добавить элемент в определенное место

5 strList.add(1, "1");

6 //заменить элемент

7 strList.set(1, "2");

8 //удалить элемент

9 strList.remove("1");
```

Интерфейс Очередь (Queue)

Очередь — коллекция, которая используется для хранения нескольких элементов.

В очереди обычно, но не обязательно, элементы располагаются по принципу FIFO (**f**irst-**i**n, **f**irst-**o**ut = первый вошел, первый вышел). В очереди FIFO, все новые элементы вставляются в конец очереди.

Интерфейс Dequeue

Коллекция Dequeue поддерживает вставку элемента и удаление элемента как в начало, так и в конец коллекции. Название Deque это сокращение от «двухконцевой очереди» и, как правило, произносится как «deck». Большинство реализаций DEQUE не устанавливают ограничения на количество элементов.

Этот интерфейс определяет методы для доступа к элементам на концах дека. Методы предоставляются для вставки, удаления, извлечения элемента.

Интерфейс Map

Map является объектом, который содержит ключи и значения. Map не может содержать дубли ключей: Каждый ключ может иметь только одно значение.

Платформа Java содержит три реализации Map: HashMap, TreeMap и LinkedHashMap.

Интерфейс ListIterator

ListIterator (итератор для списков) позволяет программисту проходить список в любом направлении, изменять список во итерации, и получать текущую позицию итератора в списке.

Интерфейс SortedSet

SortedSet представляет собой множество, в котором элементы хранятся в порядке возрастания.

Интерфейс SortedMap

SortedMap содержит элементы в порядке возрастания ключей. Эта Map является аналогом SortedSet. SortedMap используются для естественно упорядоченных пар ключ/значение, например, словарей и телефонных справочников.

Классы реализующие Java коллекции

Java Collections framework предоставляет большое количество классов с реализацией интерфейсов коллекций. Наиболее используемыми и распространенными реализациями являются ArrayList, HashMap и HashSet. Обычно классы, реализующие коллекции, не являются потокобезопасными.

Далее в этой статье мы разберем наиболее используемые классы в Java Collections framework.

HashSet Класс

Это базовая реализация интерфейса Set, которая базируется на HashMap.

Этот класс предлагает одинаковое время выполнения базовых операций (add, remove,contains и size). Мы можем установить начальную емкость и коэффициент нагрузки для этой коллекции.

Класс TreeSet

NavigableSet создан на основе TreeMap. Элементы могут быть упорядочены в порядке их добавления или с помощью компаратора.

Эта реализация обеспечивает log(n) время выполнения для основных операций (add, remove и contains).

Класс ArrayList

ArrayList — реализация интерфейса List в виде массива переменной длины. Реализует все операции со списком. Плюс к этому ArrayList обеспечивает методы для манипулирования размером массива, который используется для хранения списка. (Этот класс примерно соответствует вектору, но не является synchronized).

LinkedList класс

LinkedList — реализация интерфейсов List и Deque в виде двусвязного списка. Осуществляет все дополнительные операции со списком.

Класс HashMap

HashMap представляет собой реализацию интерфейса Map. Эта реализация обеспечивает все дополнительные операции Map и позволяет нулевые значения и ключи со значением null.
Класс HashMap примерно эквивалентно Hashtable, кроме того, что он не синхронизирован и позволяет null.
Этот класс не дает никаких гарантий упорядоченного размещения элементов.

Класс TreeMap

TreeMap представляет собой красно-черное дерево, основанное на NavigableMap. Map сортируются с помощью компаратора.

Эта реализация обеспечивает log(n) время выполнения для операций containsKey, get, put и remove .

Класс PriorityQueue

В очереди элементы додаются в порядке FIFO, но иногда мы хотим добавлять элементы на основании их приоритета. В этом случае мы можем использовать PriorityQueue, обеспечив при этом реализацию компаратора для элементов PriorityQueue. Следует отметить, что PriorityQueue не позволяют хранить null

Класс Collections

Этот класс состоит исключительно из статических методов, которые работают или возвращают коллекции. Он содержит полиморфные алгоритмы, которые используются при работе с коллекциями.

Этот класс содержит методы основных алгоритмов Collection framework, а именно методы бинарного поиска, сортировка, перемешивание, а также метод, возвращающий обратный порядок элементов и многие другие.

Синхронизированные оболочки

Синхронизированные оболочки добавляют автоматическую синхронизацию (потоко-безопасность) к определенной коллекции. Каждый из шести основных интерфейсов коллекций (Collection, Set, List, Map, SortedSet, и SortedMap) располагает статическим фабричным методом синхронизации.

```
1 public static  Collection synchronizedCollection(Collection c);

2 public static  Set synchronizedSet(Set s);

3 public static  List synchronizedList(List list);

4 public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

5 public static  SortedSet synchronizedSortedSet(SortedSet s);

6 public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Каждый из этих методов возвращает синхронизированную (потокобезопасную) коллекцию.

Неизменяемые оболочки

Неизменяемые оболочки/обертки не позволяют изменять коллекцию, перехватывая все операции, которые изменяют коллекции и бросают UnsupportedOperationException в том случае, если кто-то захочет это сделать. Вот эти методы:

```
1 public static  Collection unmodifiableCollection(Collection<? extends T> c);

2 public static  Set unmodifiableSet(Set<? extends T> s);

3 public static  List unmodifiableList(List<? extends T> list);

4 public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);

5 public static  SortedSet unmodifiableSortedSet(SortedSet<? extends T> s);

6 public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

Какую же коллекцию выбрать?

Этим вопросом часто задаются программисты выбирая между множеством коллекций, представленных в Java Collections Framework. Ниже представлена таблица, в которой собраны все наиболее значимые характеристики коллекций:

Коллекция	Упорядочивание	Random Access	Ключ-значение	Дубликат Элементы	Нулевой элемент	Потокобезопасность

ArrayList	Да	Да	Нет	Да	Да	Нет
LinkedList	Да	Нет	Нет	Да	Да	Нет
HashSet	Нет	Нет	Нет	Нет	Да	Нет
TreeSet	Да	Нет	Нет	Нет	Нет	Нет
HashMap	Нет	Да	Да	Нет	Да	Нет
TreeMap	Да	Да	Да	Нет	Нет	Нет
Vector	Да	Да	Нет	Да	Да	Да
Hashtable	Нет	Да	Да	Нет	Нет	Да
Properties	Нет	Да	Да	Нет	Нет	Да
Stack	Да	Нет	Нет	Да	Да	Да
CopyOnWriteArrayList	Да	Да	Нет	Да	Да	Да
ConcurrentHashMap	Нет	Да	Да	Нет	Нет	Да
CopyOnWriteArraySet	Нет	Нет	Нет	Нет	Да	Да

тут еще будут ответы

- Общая характеристика и особенности языка Python. Понятие динамической типизации. Преимущества и недостатки.

Python — высокоуровневый интерпретируемый язык программирования общего назначения. Для запуска написанных программ требуется наличие интерпретатора CPython.

Разработка языка Python была начата в конце 1980-х годов Гвидо ван Россумом. Новый язык позаимствовал некоторые наработки для языка ABC, который был ориентирован на обучение программированию. В феврале 1991 года Гвидо опубликовал исходный текст.

Python поддерживает несколько парадигм программирования, в том числе, объектно-ориентированное, функциональное, императивное. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных.

Динамическая типизация — приём, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

Преимущества	Недостатки
Упрощается написание несложных программ, например, скриптов. Облегчается работа прикладного программиста с СУБД, которые принципиально возвращают информацию в «динамически типизированном» виде. Поэтому динамические языки ценны, например, для программирования веб-служб. Иногда требуется работать с данными переменного типа. Например, функция поиска подстроки возвращает позицию найденного символа (число) или маркер «не найдено».	Статическая типизация позволяет уже при компиляции заметить простые ошибки «по недосмотру». Развитая статическая система типов значительную роль в самодокументировании программы; динамическая типизация по определению не проявляет этого свойства, что затрудняет разработку структурно сложных программ. Снижение производительности из-за трат процессорного времени на динамическую проверку типа, и излишние расходы памяти на переменные, которые могут хранить «что угодно».

- Структура Python – программы. Компиляция в байт-код. Пакетная организация, модули, механизмы импорта.

Про структуру, наверное, просто можно рассказать, чё и как выглядит: Есть пакеты, есть модули (об этом позже), есть функции и т.д. Вот код первой моей лабы:

```
import del_leks as dl
import sys

info = ""

def loadFile(inf):
   global info
   with open(inf, "r") as file:
      info = file.read()

def delete_leksems(leksf):
   global info
   with open(leksf, "r") as file:
      for lek in file:
         lek = lek.replace("\n", "")
```

```
        info = info.replace(lek, "")


def saveFile(outf):
   with open(outf, "w") as file:
      file.write(info)

def main():

   if len (sys.argv) < 3:
      print("Неверное количество параметров")
   else:

      print("\tУдаление лексем из файла...")

      dl.loadFile(sys.argv[1])
      dl.delete_leksems(sys.argv[2])
      dl.saveFile("out.txt")

      print("\tОперация выполнена.");


if __name__ == "__main__":
   main()
```

Python сначала компилирует исходный текст сценария в байт-код для виртуальной машины. Компиляция - это просто этап перевода, а байт-код это низкоуровневое платформонезависимое представление исходного текста программы. Python транслирует каждую инструкцию в исходном коде сценария в группы инструкций байт-кода для повышения скорости выполнения программы, так как байт-код выполняется намного быстрее. После компиляции в байт-код, создается файл с расширением ".pyc" по соседству с исходным текстом сценария.

В следующий раз, когда вы запустите свою программу интерпретатор минует этап компиляции и отдаст на выполнение откомпилированный файл с расширением ".pyc". Однако, если вы изменили исходные тексты вашей программы, то снова произойдет этап компиляции в байт-код, так как Python автоматически следит за датой изменения файла с исходным кодом.

Под модулем в Python понимается файл с расширением .py. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке Python, но и на других языках (например C).

Пакет в Python – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл __init__.py. Пакеты используются для формирования пространства имен, что позволяет работать с модулями через указание уровня вложенности (через точку).

Для импортирования пакетов используется тот же синтаксис, что и для работы с модулями.

Файл __init__.py может быть пустым или может содержать переменную __all__, хранящую список модулей, который импортируется при загрузке через конструкцию (__all__ = ["simper", "compper", "annuity"])

Как всё это импортировать

Импорт	Пример
import имя_модуля1, имя_модуля2	import math, datetime
import имя_модуля as новое_имя	import math as m
from имя_модуля import имя_объекта1, имя_объекта2	from math import cos, sin, pi
from имя_модуля import имя_объекта as псевдоним_объекта	from math import factorial as f
from имя_модуля import *	from math import *
Импорт пакета. Пускай пакет будут звать fincalc. В этом пакете есть модуль simper. Тогда импорт - import fincalc.simper	

- Объектное – ориентированное программирование в Python.

Пример объявления класса:

```
Специальные методы вызываются при создании экземпляра класса (конструктор), при инициализировании экземпляра класса (инициализатор) и при удалении класса (деструктор)
class Line (object):
    def __new__(cls):
        return super(Line, cls).__new__(cls)

    def __init__(self, p1, p2):
        self.line = (p1, p2)

    def __del__(self):
        print "Удаляется линия %s - %s" % self.line
```

Пример свойства в классе:

```
class Mine(object):

def __init__(self):
self._x = None

def get_x(self):
return self._x
```

```
def set_x(self, value):
self._x = value

def del_x(self):
self._x = 'No more'

x = property(get_x, set_x, del_x, 'Это свойство x.')

type(Mine.x) # property
mine = Mine()
mine.x # None
mine.x = 3
mine.x # 3
del mine.x
mine.x # No more
```

Инкапсуляция является одним из ключевых понятий ООП. Все значения в Python являются объектами, инкапсулирующими код (методы) и данные и предоставляющими пользователям общедоступный интерфейс.

Сокрытие информации о внутреннем устройстве объекта выполняется в Python на уровне соглашения между программистами о том, какие атрибуты относятся к общедоступному интерфейсу класса, а какие — к его внутренней реализации. Одиночное подчеркивание в начале имени атрибута говорит о том, что атрибут не предназначен для использования вне методов класса (или вне функций и классов модуля), однако, атрибут все-таки доступен по этому имени. Два подчеркивания в начале имени дают несколько большую защиту: атрибут перестает быть доступен по этому имени.

Полиморфизм в компилируемых языках программирования достигается за счёт создания виртуальных методов, которые в отличие от невиртуальных можно перегрузить в потомке. В Python все методы являются виртуальными, что является естественным следствием разрешения доступа на этапе исполнения.

```
>>> class Parent(object):
        def isParOrPChild(self) : return True
        def who(self) : return 'parent'
>>> class Child(Parent):
        def who(self): return 'child'
>>> x = Parent()
>>> x.who(), x.isParOrPChild()
('parent', True)
>>> x = Child()
>>> x.who(), x.isParOrPChild()
('child', True)
```

Python поддерживает как одиночное наследование, так и множественное, позволяющее классу быть производным от любого количества базовых классов.

```
>>> class Par1(object):                    # наследуем один базовый класс - object
        def name1(self): return 'Par1'
>>> class Par2(object):
        def name2(self): return 'Par2'
>>> class Child(Par1, Par2):               # создадим класс, наследующий Par1, Par2 (и, опосредованно, object)
        pass
>>> x = Child()
>>> x.name1(), x.name2()                   # экземпляру Child доступны методы из Par1 и Par2
'Par1','Par2'
```

Не хочу писать про метаклассы. Есть статья на хабре https://habrahabr.ru/post/145835/ , прочитайте там)

Следующий пример показывает, как работает сериализация и десериализация:

```
# сериализация
>>> import pickle
>>> p = set([1, 2, 3, 5, 8])
>>> pickle.dumps(p)
'c__builtin__\nset\np0\n((lp1\nI8\naI1\naI2\naI3\naI5\natp2\nRp3\n.'

# де-сериализация
>>> import pickle
>>> p =
pickle.loads('c__builtin__\nset\np0\n((lp1\nI8\naI1\naI2\naI3\naI5\natp2\nRp3\n.')
>>> print p
set([8, 1, 2, 3, 5])
```

- Элементы функционального программирования в Python.

Функциональное программирование является одной из парадигм, поддерживаемых языком программирования Python. Основными предпосылками для полноценного функционального программирования в Python являются: функции высших порядков, развитые средства обработки списков, рекурсия, возможность организации ленивых вычислений. Элементы функционального программирования в Python могут быть полезны любому программисту, так как позволяют гармонично сочетать выразительную мощность этого подхода с другими подходами.

Функция в Python может быть определена с помощью оператора def или лямбда-выражением. Следующие операторы эквивалентны:

```
def func(x, y):
    return x**2 + y**2

func = lambda x, y: x**2 + y**2
```

Списковое включение — наиболее выразительное из функциональных средств Python. Например, для вычисления списка квадратов положительных целых чисел, меньших 10, можно использовать выражение:

```
l = [x**2 for x in range(1,10)]
```

Функции высших порядков:

Кто	Что делает	Пример

`map()`	позволяет обрабатывать одну или несколько последовательностей с помощью заданной функции	`>>> list1 = [7, 2, 3, 10, 12]`
		`>>> list2 = [-1, 1, -5, 4, 6]`
		`>>> map(lambda x, y: x*y, list1, list2)`
		`[-7, 2, -15, 40, 72]`

`filter()`	позволяет фильтровать значения последовательности. В результирующем списке только те значения, для которых значение функции для элемента истинно	`>>> numbers = [10, 4, 2, -1, 6]`
		`>>> filter(lambda x: x < 5, numbers)`
		`[4, 2, -1]`

`reduce()`	Для организации цепочечных вычислений (последний параметр функции – начальное значение результата)	`>>>from functools import reduce`
		`>>>numbers = [2, 3, 4, 5, 6]`
		`>>> reduce(lambda res, x: res*x, numbers, 1)`
		`720`

Хрен знает. Наверное всё. Может ещё итераторы надо, но пофиг. Там вроде всё просто (и тоже есть статья на хабре https://habrahabr.ru/post/50026/).

- Работа со сложными структурами данных в Python (кортежи, списки, словари).

Список (list) представляет тип данных, который хранит набор или последовательность элементов. Для создания списка в квадратных скобках ([]) через запятую перечисляются все его элементы. Во многих языках программирования есть аналогичная структура данных, которая называется массив. Например, определим список чисел:

```
numbers = [1, 2, 3, 4, 5]
numbers1 = []
numbers2 = list()
numbers3 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
numbers4 = list(numbers)
```

Перебор элементов

```
companies = ["Microsoft", "Google", "Oracle", "Apple"]
for item in companies:
    print(item)
```

Не буду расписывать функции, ибо нафиг. Вот примеры:

```
users = ["Tom", "Bob"]

# добавляем в конец списка
users.append("Alice")  # ["Tom", "Bob", "Alice"]
# добавляем на вторую позицию
users.insert(1, "Bill")          # ["Tom", "Bill", "Bob", "Alice"]

# получаем индекс элемента
i = users.index("Tom")
# удаляем по этому индексу
removed_item = users.pop(i)             # ["Bill", "Bob", "Alice"]

last_user = users[-1]
# удаляем последний элемент
users.remove(last_user)           # ["Bill", "Bob"]

print(users)

# удаляем все элементы
users.clear()
```

```
#Проверка, есть ли элемент в списке
companies = ["Microsoft", "Google", "Oracle", "Apple"]
item = "Oracle"  # элемент для удаления
if item in companies:
    companies.remove(item)

print(companies)
```

```
#Копирование списков
users = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]

slice_users1 = users[:3]   # с 0 по 3
print(slice_users1)   # ["Tom", "Bob", "Alice"]
```

```
slice_users2 = users[1:3]   # с 1 по 3
print(slice_users2)   # ["Bob", "Alice"]

slice_users3 = users[1:6:2]   # с 1 по 6 с шагом 2
print(slice_users3)   # ["Bob", "Sam", "Bill"]
```

Кортеж (tuple) представляет последовательность элементов, которая во многом похожа на список за тем исключением, что кортеж является неизменяемым (immutable) типом. Поэтому мы не можем добавлять или удалять элементы в кортеже, изменять его.

Здесь всё то же самое, что и в списках, поэтому просто примеры кода:

```
user = ("Tom", 23)
print(user)


users_list = ["Tom", "Bob", "Kate"]
users_tuple = tuple(users_list)
print(users_tuple)      # ("Tom", "Bob", "Kate")


user = ("Tom", 22, False)
name, age, isMarried = user
print(name)             # Tom
print(age)              # 22
print(isMarried)        # False


def get_user():
    name = "Tom"
    age = 22
    is_married = False
    return name, age, is_married
user = get_user()
print(user[0])              # Tom
print(user[1])              # 22
print(user[2])              # False
```

Словарь хранит коллекцию элементов. Каждый элемент в словаре имеет уникальный ключ, с которым ассоциировано некоторое значение.

```
#Обьявление словарей
users = {1: "Tom", 2: "Bob", 3: "Bill"}
elements = {"Au": "Золото", "Fe": "Железо", "H": "Водород", "O": "Кислород"}


# Список - > Словарь
```

```
users_list = [
    ["+111123455", "Tom"],
    ["+384767557", "Bob"],
    ["+958758767", "Alice"]
]
users_dict = dict(users_list)
print(users_dict)  # {"+111123455": "Tom", "+384767557": "Bob",
"+958758767": "Alice"}


#изменение элесента
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
# получаем элемент с ключом "+11111111"
print(users["+11111111"])      # Tom
# установка значения элемента с ключом "+33333333"
users["+33333333"] = "Bob Smith"
print(users["+33333333"])      # Bob Smith


# удаление элемента
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
del users["+55555555"]
print(users)



# конкатонация словарей
users = {"+1111111": "Tom","+3333333": "Bob","+5555555": "Alice"}
users2 = {"+2222222": "Sam","+6666666": "Kate"}
users.update(users2)
print(users)    # {"+1111111": "Tom", "+3333333": "Bob", "+5555555":
"Alice", "+2222222": "Sam", "+6666666": "Kate"}
print(users2)   # {"+2222222": "Sam", "+6666666": "Kate"}
```

- Создание GUI в Python. Библиотеки wxPython и Tkinter.

Библа	Сыль
wxPython	https://habrahabr.ru/post/137369/
Tkinter	https://habrahabr.ru/post/133337/

Интеграция Python с другими языками.

C API

Используется для написания модулей расширения на C/C++. Методы из модулей расширения далее могут вызываться из программы на Python.

Использование языка C/C++ в модулях Python может иметь смысл, если аналогичная функция, написанная на Python, работает медленнее, например операции с массивами из модуля Numeric.

Документация по модулям расширения: “Python/C API Reference Manual”.

Все необходимые расширения описываются в заголовочном файле Python.h. Для связи с интерпретатором с кодом на C используют функции, определённые в интерпретаторе Python, причём всем функциям Python желательно иметь префикс _Py ил Py во избежание конфликтов.

Примеры возможностей

1. Высокоуровневый (The Very High Level Layer) интерфейс интерпретатора
    a. int Py_Main(int argc, wchar_t **argv) – главная программа. Возвращает 0, если всё хорошо, 1, если исключение, 2, если список параметров не являетсякорректной командной строкой Python.
    b. PyObject* PyRun_StringFlags(const char *str, int start, PyObject *globals, PyObject *locals, PyCompilerClass *flags) – выполняет код из строки str, контекст определён globals, locals, flags.
2. Функции для работы со встроенными интерпретатором и потоками
    a. Py_Initialize(), Py_Finalize()
3. Управление процессом и сервисы OS
    a. Py_FatalError(), Py_Exit()
4. Импорт модулей
    a. PyImport_Import()
5. Поддержка встроенных типов данных
    a. PyBool_Check()
6. Управление кучей интерпретатора
    a. PyMem_Realloc(), PyMem_Free(), PyMem_New()

Пример встраивания

```
#include "Python.h"

main(int argc, char **argv)
{
    /* Передает argv[0] интерпретатору Python */
    Py_SetProgramName(argv[0]);

    /* Инициализация интерпретатора */
    Py_Initialize();

    /* ... */

    /* Выполнение операторов Python (как бы модуль __main__) */
    PyRun_SimpleString("import time\n");
    PyRun_SimpleString("print time.localtime(time.time())\n");

    /* ... */

    /* Завершение работы интерпретатора */
    Py_Finalize();
}
```

Требуется компиляция с библиотекой libpython, libthread и т.д.

SWIG

Simplified Wrapper an Interface Generator – средство для упрощения использования библиотек, написанных на C, C++, Python.

Поддерживает многие возможности C++, например классы, указатели, наследование, препроцессинг и шаблоны. Однако, с другой стороны, требуется ручное управление памятью.

Пример файла расширения

```
%module freq

%typemap(out) int * {
  int i;
  $result = PyTuple_New(256);
  for(i=0; i<256; i++)
    PyTuple_SetItem($result, i, PyLong_FromLong($1[i]));
  free($1);
}

extern int * frequency(char s[]);
```

Создание модуля расширения

```
swig -python freq.i
              gcc -c -fpic freq_wrap.c freq.c  -DHAVE_CONFIG_H
                 -I/usr/local/include/python2.3 -I/usr/local/lib/python2.3/config
              gcc -shared freq.o freq_wrap.o -o _freq.so
```

Java

Jython – реализация Python на Java–платформе.

Достоинства:

+ Возможность динамической (JIT) и статической компиляции.
+ Поддержка объектной модели Java, наследование от абстрактных классов Java
+ Большая выразительная мощность Python, что приводит к уменьшению объёма кода

Недостатки

– В Java нет поддержки множественного наследования

Пример кода

```
from java.lang import System
from java.awt import *
# А это модуль из Jython
import random

# Класс для рисования линий на рисунке
class Lines(Canvas):
  # Реализация метода paint()
  def paint(self, g):
    X, Y = self.getSize().width, self.getSize().height
    label.setText("%s x %s" % (X, Y))
    for i in range(100):
      x1, y1 = random.randint(1, X), random.randint(1, Y)
      x2, y2 = random.randint(1, X), random.randint(1, Y)
      g.drawLine(x1, y1, x2, y2)

# Метки, кнопки и т.п.
panel = Panel(layout=BorderLayout())
label = Label("Size", Label.RIGHT)
panel.add(label, "North")
button = Button("QUIT", actionPerformed=lambda e: System.exit(0))
panel.add(button, "South")
lines = Lines()
panel.add(lines, 'Center')

# Запуск панели в окне
import pawt
pawt.test(panel, size=(240, 240))
```

Компиляция и сборка в Jar

```
jythonc -d -c -j lns.jar lines.py
```

OCaml

Язык программирования OCaml - это язык функционального программирования (семейства ML, что означает Meta Language), созданный в институте INRIA, Франция. Важной особенностью OCaml является то, что его компилятор порождает исполняемый код, по быстродействию сравнимый с C, родной для платформ, на которых OCaml реализован. В то же время, будучи функциональным по своей природе, он приближается к Python по степени выразительности. Именно поэтому для OCaml была создана библиотека Pycaml, фактически реализующая аналог C API для OCaml. Таким образом, в программах на OCaml могут использоваться модули языка Python, в них даже может быть встроен интерпретатор Python. Для Python имеется большое множество адаптированных C-библиотек, это дает возможность пользователям OCaml применять в разработке комбинированное преимущество Python и OCaml. Минусом является только необходимость знать функции Python/C API, имена которого использованы для связи OCaml и Python.

Следующий пример (из Pycaml) показывает программу для OCaml, которая определяет модуль для Python на OCaml и вызывает встроенный интерпретатор Python:

```
let foo_bar_print = pywrap_closure
                    (fun x -> pytuple_fromarray (pytuple_toarray x)) ;;
                  let sd = pyimport_getmoduledict () ;;
                  let mx = pymodule_new "CamlModule" ;;
                  let cd = pydict_new () ;;
                  let cx = pyclass_new (pynull (), cd, pystring_fromstring "CamlClass") ;;
                  let cmx = pymethod_new (foo_bar_print,(pynull ()),cx) ;;
                  let _ = pydict_setitemstring (cd, "CamlMethod", cmx) ;;
                  let _ = pydict_setitemstring (pymodule_getdict mx, "CamlClass", cx) ;;
                  let _ = pydict_setitemstring (sd, "CamlModule", mx) ;;
                  let _ = pyrun_simplestring
                    ("from CamlModule import CamlClass\n" ^
                     "x = CamlClass()\n" ^
                     "for i in range(100000):\n" ^
                     "  x.CamlMethod(1,2,3,4)\n" ^
                     "print 'Done'\n")
```

Pyrex

Для написания модулей расширения можно использовать специальный язык - Pyrex - который совмещает синтаксис Python и типы данных C. Компилятор Pyrex написан на Python и превращает исходный файл (например, primes.pyx) в файл на C - готовый для компиляции модуль расширения. Язык Pyrex заботится об управлении памятью, удаляя после себя ставшие ненужными объекты. Пример файла из документации к Pyrex (для вычисления простых чисел):

```
def primes(int kmax):
                  cdef int n, k, i
                  cdef int p[1000]
                  result = []
                  if kmax > 1000:
                    kmax = 1000
                  k = 0
                  n = 2
                  while k < kmax:
                    i = 0
                    while i < k and n % p[i] <> 0:
                      i = i + 1
                    if i == k:
                      p[k] = n
                      k = k + 1
                      result.append(n)
                    n = n + 1
                  return result
```

Компилция и компоновка

```
pyrexc primes.pyx
              gcc primes.c -c -fPIC -I /usr/local/include/python2.3
              gcc -shared primes.o -o primes.so
```

Источники
http://ideafix.name/wp-content/uploads/2012/09/Python-13.pdf

Многопоточность в Python

Питоновская реализация многопоточности ограниченная. Интерпретатор питона использует внутренний глобальный блокировщик (GIL), который позволяет выполняться только одному потоку. Это сводит на нет преимущества многоядерной архитектуры процессоров. Для многопоточных приложений, которые работают в основном на дисковые операции чтения/записи, это не имеет особого значения, а для приложений, которые делят процессорное время между потоками, это является серьезным ограничением.

Threading

Для создания потоков применяют стандартный модуль threading (есть также и модуль thread, но в нём определено меньше методов и есть конфликты атрибутов). Есть два варианта создания потоков:

– вызов функции: threading.Thread()

– вызов класса: threading.Thread

Пример использования первого варианта

```
import threading
import time
def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)
t = threading.Thread(target=clock, args=(15,))
t.daemon = True
t.start()
```

Пример вызова класса

```
import threading
import time
class ClockThread(threading.Thread):
    def __init__(self,interval):
        threading.Thread.__init__(self)
        self.daemon = True
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)
t = ClockThread(15)
t.start()
```

– start() – запуск потока;

– run() –этот метод представляет действия, которые должны быть выполнены в потоке;

– join([timeout]) – поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван.

Параметр timeout (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод join был вызван. Вызывать join()некоторого потока можно много раз. Поток не может вызвать метод join() самого себя. Также нельзя ожидать завершения еще не запущенного потока.

– getName() – возвращает имя потока.

– setName(name) – присваивает потоку имя name.

– isAlive() – возвращает истину, если поток работает (метод run() уже вызван).

– isDaemon() – возвращает истину, если поток имеет признак демона (работает в фоновом режиме).

– setDaemon(daemonic) – устанавливает признак daemonic того, что поток является демоном.

– activeCount() – возвращает количество активных в настоящий момент экземпляров класса Thread. Фактически это len(threading.enumerate()).

– currentThread() – возвращает текущий объект-поток, т.е. соответствующий потоку управления, который вызвал эту функцию.

– enumerate() – возвращает список активных потоков.

queue

Модуль queue предоставляет механизм связи между потоками, с по- мощью которого отдельные потоки могут совместно использовать данные. В данном случае создается очередь, через которую организуется связь потоков.

Атрибуты модуля:

Атрибут	Описание
Классы модуля Queue/queue	
`Queue(maxsize=0)`	Создает очередь с последовательной организацией, имеющую указанный размер *maxsize*, которая не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина очереди становится неограниченной
`LifoQueue(maxsize=0)`	Создает стек, имеющий указанный размер *maxsize*, который не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина стека становится неограниченной
`PriorityQueue(maxsize=0)`	Создает очередь по приоритету, имеющую указанный размер *maxsize*, которая не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина очереди становится неограниченной
Исключения модуля **Queue/queue**	
`Empty`	Активизируется при вызове метода `get*()` применительно к пустой очереди
`Full`	Активизируется при вызове метода `put*()` применительно к заполненной очереди
Методы объекта **Queue/queue**	
`qsize()`	Возвращает размер очереди (это — приблизительное значение, поскольку при выполнении этого метода может происходить обновление очереди другими потоками)
`empty()`	Возвращает `True`, если очередь пуста; в противном случае возвращает `False`
`full()`	Возвращает `True`, если очередь заполнена; в противном случае возвращает `False`
`put(item, block=True, timeout=None)`	Помещает элемент *item* в очередь; если значение block равно True (по умолчанию) и значение *timeout* равно `None`, устанавливает блокировку до тех пор, пока в очереди не появится свободное место. Если значение *timeout* является положительным, блокирует очередь самое больше на `timeout` секунд, а если значение *block* равно False, активизирует исключение `Empty`
`put_nowait(item)`	То же, что и `put(item, False)`
`get(block=True, timeout=None)`	Получает элемент из очереди, если задано значение `block` (отличное от 0); устанавливает блокировку до того времени, пока элемент не станет доступным
`get_nowait()`	То же, что и `get(False)`
`task_done()`	Используется для указания на то, что работа по постановке элемента в очередь завершена, в сочетании с описанным ниже методом `join()`
`join()`	Устанавливает блокировку до того времени, пока не будут обработаны все элементы в очереди; сигнал об этом вырабатывается путем вызова описанного выше метода `task_done()`

Пример:

```
#!/usr/bin/env python

from random import randint
from time import sleep
from Queue import Queue
from myThread import MyThread

def writeQ(queue):
    print 'producing object for Q...',
     queue.put('xxx', 1)
     print "size now", queue.qsize()

 def readQ(queue):
     val = queue.get(1)
     print 'consumed object from Q... size now', \
             queue.qsize()

 def writer(queue, loops):
     for i in range(loops):
         writeQ(queue)
         sleep(randint(1, 3))

 def reader(queue, loops):
     for i in range(loops):
         readQ(queue)
         sleep(randint(2, 5))

 funcs = [writer, reader]
 nfuncs = range(len(funcs))

def main():
    nloops = randint(2, 5)
    q = Queue(32)

    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (q, nloops),
            funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
         threads[i].join()

    print 'all DONE'

if __name__ == '__main__':
   main()
```

Альтернативы

subprocess

В первую очередь вместо модуля threading можно применить модуль subprocess, когда возникает необходимость запуска новых процессов, либо для выполнения кода, либо для обеспечения обмена данными с другими процессами через стандартные файлы ввода-вывода (stdin, stdout, stderr). Этот модуль был введен в версии Python 2.4.

multiprocessing

Этот модуль, впервые введенный в Python 2.6, позволяет запускать процессы для нескольких ядер или процессоров, но с интерфейсом, весьма напоминающим интерфейс модуля threading. Он также поддерживает различные механизмы передачи данных между процессами, применяемыми для выполнения совместной работы.

concurrent.futures

Это новая высокоуровневая библиотека, которая работает только на уровне заданий. Это означает, что при использовании модуля concurrent.futures исключается необходимость заботиться о синхронизации либо управлять потоками или процессами. Достаточно лишь указать поток или пул процесса с определенным количеством рабочих потоков, передать задания на выполнение и собрать полученные результаты. Этот модуль впервые появился в версии Python 3.2, но перенесен также в версию Python 2.6 и последующие версии. Модуль можно получить по адресу http://code.google.com/p/pythonfutures.

Дополнительные модули

Модуль	Описание
`thread`[a]	Основной, находящийся на низком уровне модуль поддержки потоков
`threading`	Объекты для многопоточной организации работы и синхронизации высокого уровня
`multiprocessing`[b]	Запуск/использование подпроцессов с помощью интерфейса threading
`subprocess`[c]	Полный отказ от потоков и выполнение вместо них процессов
`Queue`	Синхронизированная очередь с последовательной организацией для нескольких потоков
`mutex`[d]	Объекты мьютексов
`concurrent.futures`[e]	Библиотека высокого уровня для асинхронного выполнения
`SocketServer`	Создание/управление многопоточными серверами TCP или UDP

[a] Переименован в `_thread` в Python 3.0.
[b] Новое в версии Python 2,6.
[c] Новое в версии Python 2.4.
[d] Обозначен как устаревший в Python 2.6 и удален в версии 3.0.
[e] Впервые введенный в Python 3.2, но предоставляемый вне стандартной библиотеки для версии 2.6 и последующих версий.

Источники:

http://www.williamspublishing.com/PDF/978-5-8459-1793-5/part.pdf

Сетевое взаимодействие в Python. Механизм сокетов. Работка с объектами на удалённых серверах. Протокол XML–RPC.

Обмен сообщениями по сети осуществляется через стек протоколов TCP/IP.

Для идентификации сетевого интерфейса используют IP-адрес. В рамках одного сетевого интерфейса может быть несколько портов. Для установки соединения приложение клиента должно выбрать свободный порт и установить соединение с серверным приложением, прослушивающим порт. Пара IP/Порт характеризует сокет – программный интерфейс для обмена данными между процессами. Каждое сетевое приложение должно иметь серверный и клиентский сокеты.

В Python сокеты представлены классом socket

Методы класса socket:

Общие	
Socket	Создать новый сокет и вернуть файловый дескриптор
Send	Отправить данные по сети
Receive	Получить данные из сети
Close	Закрыть соединение
Серверные	
Bind	Связать сокет с IP-адресом и портом
Listen	Объявить о желании принимать соединения. Слушает порт и ждет когда будет установлено соединение
Accept	Принять запрос на установку соединения
Клиентские	
Connect	Установить соединение

Пример клиента:

```
import socket HOST = "" # удаленный компьютер (localhost)
PORT = 33333 # порт на удаленном компьютере
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))
sock.send("ПАЛИНДРОМ")
result = sock.recv(1024)
sock.close()
print "Получено:", result
```

socket.AF_INET – domain: IPv4

socket.SOCK_STREAM – type: надёжная потокоориентированная служба (сервис) или потоковый сокет

Сервер:

```
import socket, string
def do_something(x):
    lst = map(None, x);
    lst.reverse();
    return string.join(lst, "")
HOST = "" # localhost
PORT = 33333
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM) srv.bind((HOST, PORT))
while 1:
 print "Слушаю порт 33333"
 srv.listen(1)
 sock, addr = srv.accept()
 while 1:
   pal = sock.recv(1024)
   if not pal:
      break
```

```
print "Получено от %s:%s:" % addr, pal
lap = do_something(pal)
print "Отправлено %s:%s:" % addr, lap
sock.send(lap)
sock.close()
```

Также есть возможность работать с DNS:

```
>>> socket.gethostbyname('www.onego.ru')
('www.onego.ru', [], ['195.161.136.4'])
```

XML-RPC

XML-RPC – это простой и удобный способ для отправки сообщений через Internet. В Python реализовано в библиотеке xmlrpc.client Пример кода:

```
import xmlrpclib

XMLRPC_SERVER_URL = "http://www.python.org/cgi-bin/moinmoin/?action=xmlrpc"

pythoninfo = xmlrpclib.ServerProxy( XMLRPC_SERVER_URL )
allpages = pythoninfo.getAllPages() # this is the XML-RPC call

print ", ".join( allpages )
```

Пример составления строки вызова

```
import xmlrpclib

func_name = "foo"
arg_1 = "robot"
arg_2 = { "some":1, "dict":2 }
arg_3 = [1,2,3,4,5]

call_string = xmlrpclib.dumps( (arg_1,arg_2,arg_3,), func_name )
```

Строка call_string

```
<?xml version='1.0'?>
<methodCall>
<methodName>foo</methodName>
<params>
<param>
<value><string>robot</string></value>
</param>
<param>
<value><struct>
<member>
<name>dict</name>
<value><int>2</int></value>
</member>
<member>
<name>some</name>
<value><int>1</int></value>
</member>
</struct></value>
</param>
<param>
<value><array><data>
<value><int>1</int></value>
<value><int>2</int></value>
<value><int>3</int></value>
<value><int>4</int></value>
<value><int>5</int></value>
</data></array></value>
</param>
</params>
</methodCall>
```

Получение данных из строки

```
call_data = xmlrpclib.loads( call_string )
```

Результат:

```
(('robot', {'some': 1, 'dict': 2}, [1, 2, 3, 4, 5]), u'foo')
```

Пример применения xmlrpc.client

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(("localhost", 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x,y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

Клиент:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

Источники

https://docs.python.org/3/library/xmlrpc.server.html#module-xmlrpc.server

https://docs.python.org/3/library/xmlrpc.client.html#module-xmlrpc.client

Модуль OS

Модуль os предоставляет множество функций для работы с операционной системой, причём их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми.

Наиболее часто используемые:

– **os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

– **os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

– **os.getlogin**() - имя пользователя, вошедшего в терминал (Unix).

– **os.getpid**() - текущий id процесса.

– **os.uname**() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

– **os.access**(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True) - проверка доступа к объекту у текущего пользователя. Флаги: **os.F_OK** - объект существует, **os.R_OK** - доступен на чтение, **os.W_OK** - доступен на запись, **os.X_OK** - доступен на исполнение.

– **os.chdir**(path) - смена текущей директории.

– **os.chmod**(path, mode, *, dir_fd=None, follow_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

– **os.chown**(path, uid, gid, *, dir_fd=None, follow_symlinks=True) - меняет id владельца и группы (Unix).

– **os.getcwd**() - текущая рабочая директория.

– **os.link**(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True) - создаёт жёсткую ссылку.

– **os.listdir**(path=".") - список файлов и директорий в папке.

– **os.mkdir**(path, mode=0o777, *, dir_fd=None) - создаёт директорию. OSError, если директория существует.

– **os.makedirs**(path, mode=0o777, exist_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

– **os.remove**(path, *, dir_fd=None) - удаляет путь к файлу.

– **os.rename**(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает файл или директорию из src в dst.

– **os.renames**(old, new) - переименовывает old в new, создавая промежуточные директории.

– **os.replace**(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает из src в dst с принудительной заменой.

– **os.rmdir**(path, *, dir_fd=None) - удаляет пустую директорию.

– **os.removedirs(path)** - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

– **os.symlink**(source, link_name, target_is_directory=False, *, dir_fd=None) - создаёт символическую ссылку на объект.

– **os.sync**() - записывает все данные на диск (Unix).

– **os.truncate**(path, length) - обрезает файл до длины length.

– **os.utime**(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

– **os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

– **os.system**(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

– **os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

– **os.path** - модуль, реализующий некоторые полезные функции на работы с путями.

Источники

https://pythonworld.ru/moduli/modul-os.html

тут еще 2 вопросов Егора