Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Факультет физики и информационных технологий
Кафедра автоматизированных систем обработки информации

СОГЛАСОВАНО
Заведующий кафедрой
автоматизированных систем
обработки информации
А.В.Воруев
__ _____ 2023 г.

СОГЛАСОВАНО
Декан
факультета физики и
информационных технологий
А.Л.Самофалов
__ _____ 2023 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

для специальности
1-53 01 02 Автоматизированные системы обработки информации

составители: ассистент кафедры АСОИ Рафалова Е.В.
старший преподаватель кафедры АСОИ Пугачева Е.Е.
старший преподаватель кафедры АСОИ Леванцов В.Н.

Рассмотрено и утверждено
на заседании кафедры АСОИ
__17 октября___ 2023 г., протокол № _3_

Рассмотрено и утверждено
на заседании научно-методического
совета университета
__28 ноября___ 2023 г., протокол № _4_

Гомель 2023

1 ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по дисциплине «Шаблоны проектирования» представляет собой комплекс систематизированных учебных, методических и вспомогательных материалов, предназначенных для использования в образовательном процессе специальности 1-53 01 02 – Автоматизированные системы обработки информации.

ЭУМК разработан в соответствии со следующими нормативными документами:

1. Положением об учебно-методическом комплексе на уровне высшего образования, утвержденном постановлением Министерства образования Республики Беларусь от 08.11.2022 № 427.

2. Учебная программа составлена на основе образовательного стандарта ОСВО 1-53 01 02-2013 г. и учебного плана ГГУ имени Ф.Скорины регистрационный № I 53-01-13, дата утверждения 25.08.2013.

3. Учебной программой по учебной дисциплине «Шаблоны проектирования» для специальности 1-53 01 02 Автоматизированные системы обработки информации, утвержденной 07.06.2014, регистрационный номер УД-37-2017-17/уч.

Цель создания ЭУМК – обеспечить приобретение теоретических знаний и практических навыков при подготовке специалистов в области практического применения технологий проектирования программного обеспечения, используя шаблоны проектирования.

ЭУМК направлен на всестороннюю подготовку студентов теоретическим основам и практическим навыками разработки, введения в эксплуатацию и использования медиапродуктов. Отдельное внимание уделяется применению паттернов проектирования в реализации программного обеспечения. Организация изучения дисциплины на основе ЭУМК предполагает продуктивную образовательную деятельность, позволяющую сформировать социально-личностные и профессиональные компетенции будущих специалистов.

ЭУМК способствует успешному осуществлению учебной деятельности, дает возможность планировать и осуществлять самостоятельную управляемую работу студентов, обеспечивает рациональное распределение учебного времени по темам учебной дисциплины и совершенствование методики проведения занятий.

ЭУМК состоит из теоретического, практического и вспомогательного разделов. Теоретический раздел содержит тексты лекций. Практический раздел содержит методические рекомендации к лабораторным работам, тестовые задания и вопросы для самоконтроля. Вспомогательный раздел содержит учебную программу и список литературы.

Теоретический раздел содержит лекционный материал по всем темам учебной программы, включая и темы, вынесенные на самостоятельное изучение.

Практический раздел включает в себя темы лабораторных занятий и задания с краткими методическими указаниями по выполнению лабораторных работ. В разделе так же приводятся некоторый набор тестовых заданий и к каждой теми указаны вопросы для самоконтроля.

Вспомогательный раздел содержит необходимые элементы учебно-программной документации по дисциплине с указанием рекомендуемой литературы (основной, дополнительной, вспомогательной).

Все разделы ЭУМК в полной мере соответствуют содержанию учебной программы и объему учебного плана.

Дисциплина компонента учреждения образования «Шаблоны проектирования» изучается студентами 3 курса дневной формы обучения специальности 1-53 01 02 - «Автоматизированные системы обработки информации», студентами 3,4 курсов заочной интегрированной формы обучения на основе среднего специального образования и студентами 2,3 курсов заочной сокращенной формы обучения.

Дневная форма обучения: всего часов по плану – 156; аудиторное количество часов – 64, из них: лекции – 30, лабораторные занятия – 34.

Форма отчётности – экзамен в 5 семестре.

Заочная форма обучения: всего часов по плану – 158, аудиторное количество часов – 16, из них: лекции – 10, лабораторных работ – 6.

Форма отчетности – экзамен в 7 семестре.

Заочная форма обучения (интегрированная форма обучения на основе среднего специального образования): всего часов по плану – 158, аудиторное количество часов – 16, из них: лекции – 10, лабораторных работ – 6.

Форма отчетности – экзамен в 5 семестре.

2 ТЕКСТЫ ЛЕКЦИЙ

1. ООП. Парадигмы ООП. Классы и объекты

ООП — методология программирования, основанная на функционировании программного продукта как результата взаимодействия совокупности объектов, каждый из которых является экземпляром конкретного класса.

Объект — именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение.

Класс — модель информационной сущности, представляющая универсальный тип данных, состоящая из набора полей данных и методов их обработки. В применении к объектно-ориентированным языкам программирования понятия объекта и класса конкретизируются. Во всех языках классы и объекты обладают рядом общих свойств, таких как инкапсуляция (объединение открытых данных и закрытых методов), наследование (заимствование функциональности базовых классов производными), полиморфизм (возможность использования объектов с одинаковым интерфейсом при наследовании).

Объектно-ориентированный язык Java был разработан в компании Sun Microsystems в 1995 году для программирования небольших устройств и введения динамики на сайтах в виде апплетов. Немного позже язык Java нашел широкое применение в интернет-приложениях, добавив на клиентские веб-страницы динамический интерфейс, улучшив вычислительные возможности.

Системная библиотека классов языка Java содержит классы и пакеты, реализующие и расширяющие базовые возможности языка, а также сетевые средства, взаимодействие с базами данных, многопоточность и многое другое. Методы классов, включенные в эти библиотеки, вызываются JVM (Java Virtual Machine) во время интерпретации программы.

В Java все объекты программы расположены в динамической памяти — куче данных (heap) — и доступны по объектным ссылкам, которые хранятся в стеке (stack). Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов и сделало ее менее эффективной по сравнению с программами на C++. В свою очередь, в Java предложен усовершенствованный механизм работы с коллекциями, реализующими основные динамические структуры данных. Необходимо отметить, что объектная ссылка языка Java содержат информацию о классе объекта, на который она ссылается, так что объектная ссылка — это не только ссылка на объект, размещенный в динамической памяти, но и дескриптор (описание) объекта. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки. В Java изменена концепция организации динамического распределения памяти: отсутствуют способы программного освобождения динамически выделенной памяти с помощью деструктора, понятие которого исключено из Java. Вместо этого реализована система автоматического освобождения памяти «сборщик мусора», выделенной с помощью оператора new. Программист может только рекомендовать системе освободить выделенную динамическую память.

JDK (Java Development Kit) — полный набор для разработки и запуска приложений, состоящий из компилятора, утилит, исполнительной системы JRE, библиотек, документации.

JRE (Java Runtime Environment) — минимальный набор для исполнения приложений, включающий JVM, но без средств разработки.

JVM (Java Virtual Machine) — базовая часть исполняющей системы Java, которая интерпретирует байт-код Java, скомпилированный из исходного текста Java-программы для конкретной операционной системы. Однако при одновременном использовании нескольких различных версий компилятора и различных библиотек применение переменных среды окружения начинает мешать эффективной работе, так как при выполнении приложения поиск класса осуществляется независимо от версии. Когда виртуальная машина обнаруживает класс с подходящим именем, она его и подгружает.

Для компиляции и запуска приложений можно использовать два способа:

1) Командная строка;

2) IDE (IntelliJ IDEA, Eclipse, NetBeans etc.).

IDE позволяют создавать, компилировать и запускать приложения в значительно более удобной форме, чем с помощью командной строки.

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для Java, разработанная компанией JetBrains. Среда IDEA используется для разработки приложений, поэтому необходимо установить Java Development Kit (JDK). После установки IntelliJ IDEA программа предложит настроить среду разработки: выбрать тему и установить дополнительные плагины.

Классы в языке Java объединяют поля класса, методы, конструкторы, логические блоки и внутренние классы. Основные отличия от классов C++: все функции определяются внутри классов и называются методами; невозможно создать метод, не являющийся методом класса, или объявить метод вне класса; спецификаторы доступа public, private, protected воздействуют только на те объявления полей, методов и классов, перед которыми они стоят, а не на участок от одного до другого спецификатора, как в C++; элементы по умолчанию не устанавливаются в private, а доступны для классов из данного пакета.

Спецификатор доступа к классу может быть public (класс доступен в данном пакете и вне пакета). По умолчанию, если спецификатор класса public не задан, он устанавливается в дружественный (friendly или private-package). Такой класс доступен только в текущем пакете. Кроме этого, спецификатор может быть final (класс не может иметь подклассов) и abstract (класс может содержать абстрактные нереализованные методы, объект такого класса создать нельзя).

Класс наследует все свойства и методы суперкласса (базового класса), указанного после ключевого слова extends, и может включать множество интерфейсов, перечисленных через запятую после ключевого слова implements. Интерфейсы относительно похожи на абстрактные классы, содержащие только статические константы и не имеющие конструкторов, но имеют целый ряд серьезных архитектурных различий.

Все классы любого приложения условно разделяются на две группы: классы — носители информации, и классы, работающие с этой информацией.

Объектные ссылки Java работает не с объектами, а со ссылками на объекты, размещаемыми в динамической памяти с помощью оператора new. Это объясняет то, что операции сравнения ссылок на объекты не имеют смысла, так как при этом сравниваются адреса. Для сравнения объектов на эквивалентность по значению необходимо использовать специальные методы, например, equals(Object ob).

Этот метод наследуется в каждый класс из суперкласса Object, который лежит в корне дерева иерархии всех классов и должен переопределяться в подклассе для определения эквивалентности содержимого двух объектов этого класса.

2. Основы синтаксиса Java. Java Code Convention

Любая программа манипулирует данными и объектами с помощью операторов. Каждый оператор производит результат из значений своих операндов или изменяет непосредственно значение операнда.

Java — язык объектно-ориентированного программирования, однако не все данные в языке есть объекты. Для повышения производительности в нем, кроме объектов, используются базовые типы данных, значения которых (литералы) размещаются в стековой памяти. Для каждого базового типа имеются также классы-оболочки, которые инкапсулируют данные базовых типов в объекты, располагаемые в динамической памяти (heap). Базовые типы обеспечивают более высокую производительность вычислений по сравнению с объектами классов-оболочек и другими объектами. Что является основной причиной применения в Java базовых типов, а не объектной модели полностью.

Значения базовых типов данных, записанных по правилам языка Java, называют литералами.

Определено восемь базовых типов данных, размер каждого из которыхостается неизменным независимо от платформы. Беззнаковых типов в Java не существует. Каждый тип данных определяет множество значений и их представление в памяти. Для каждого типа определен набор операций над его значениями.

В Java используются целочисленные литералы, например: 42 — целое (int) десятичное число, 042 — восьмеричное целое число, 0x42b — шестнадцатеричное целое число, 0b101010 — двоичное целое число. Целочисленные литералы по умолчанию относятся к типу int. Если необходимо определить длинный литерал типа long, в конце указывается символ L (например: 0xffffL). Если значение числа больше значения, помещающегося в int (2147483647), то Java автоматически полагает, что оно типа long.

В Java для удобства восприятия литералов стало возможно использовать знак «_» при объявлении больших чисел, то есть вместо int value = 4200000 можно записать int value = 4_200_000. Эта форма применима и для чисел с плавающей запятой. Однако некорректно: _7 или 7_.

Литералы с плавающей точкой записываются в виде 1.618 или в экспоненциальной форме 0.117E-5 и относятся к типу double. Таким образом, действительные числа относятся к типу double. При объявлении такого литерала можно использовать символы d и D, а именно 1.618d. Если необходимо определить литерал типа float, то в конце литерала необходимо добавить символ F или f. По стандарту IEEE 754 введены понятие бесконечности +Infinity и –Infinity, число большее или меньшее любого другого числа при делении на ноль в типах с плавающей запятой, а также значение NaN (Not a Number), которое может быть получено, например, при извлечении квадратного корня из отрицательного числа.

К булевским литералам относятся значения true и false. Литерал null — значение по умолчанию для объектной ссылки.

Символьные литералы определяются в апострофах ('a', '\n', '\042', '\ub76f'). Для размещения символов используется формат Unicode, в соответствии с которым для каждого символа отводится два байта. В формате Unicode первый байт содержит код управляющего символа или национального алфавита, а второй байт соответствует стандартному ASCII коду, как в C++. Любой символ можно представить в виде '\ucode', где code представляет двухбайтовый шестнадцатеричный код символа. Java поддерживает управляющие символы, не имеющие графического изображения; '\n' — новая строка, '\r' — переход к началу, '\f' — новая страница, '\t' — табуляция, '\b' — возврат на один символ, '\uxxxx' — шестнадцатеричный символ Unicode, '\ddd' — восьмеричный символ и др.

В настоящее время Java обеспечивает поддержку стандарта Unicode 10.0.0 возможностями классов Character, String.

Переменные могут быть либо членами класса, либо локальными переменными метода, либо параметрами метода. По стандартным соглашениям имена переменных не могут начинаться с цифры, в именах не могут использоваться символы арифметических и логических операторов, а также символ '#'. Применение символов '$' и '_' допустимо, в том числе и в первой позиции имени, но нежелательно. Каждая переменная должна быть объявлена с одним из указанных выше типов.

Переменная базового типа, объявленная как член класса, хранит нулевое значение, соответствующее своему типу. Если переменная объявлена как локальная переменная в методе, то перед использованием она обязательно должна быть проинициализирована, так как локальная переменная не инициализируется по умолчанию нулем. Область действия и время жизни такой переменной ограничена блоком {}, в котором она объявлена.

Строки, заключенные в двойные апострофы, считаются литералами и размещаются в пуле литералов, но в то же время такие строки представляют собой объекты класса String. При инициализации строки создается объект класса String. При работе со строками, кроме методов класса String, можно использовать единственный в языке перегруженный оператор «+» конкатенации (слияния) строк. Конкатенация строки с объектом любого другого типа добавляет к исходному объекту-строке строковое представление объекта другого типа.

Строковый литерал заключается в двойные кавычки, это не ASCII-строка, а объект из набора (массива) символов. Строковые литералы в большинстве своем должны объявляться как константы final static поля классов для экономии памяти.

В арифметических выражениях автоматически выполняются расширяющие преобразования типа byte -> short -> int -> long -> float-> double

Операторы Java практически совпадают с операторами в других современных языках программирования и имеют приоритет. Операторы работают с базовыми типами, для которых они определены, и объектами классов-оболочек над базовыми типами. Кроме этого, операторы «+» и «+=» производят также действия по конкатенации операндов типа String. Логические операторы «==», «!=» и оператор присваивания «=» применимы к операндам любого объектного и базового типов, а также литералам.

Применение оператора присваивания к объектным типам часто приводит к ошибке несовместимости типов во время выполнения приложения с генерацией исключения ClassCastException, поэтому такие операции необходимо контролировать. Деление на ноль в целочисленных типах вызывает исключительную ситуацию, переполнение не контролируется.

Кроме базовых типов данных, в языке Java используются соответствующие классы-оболочки (wrapper-классы) из пакета java.lang: Boolean, Character, Integer, Byte, Short, Long, Float, Double. Объекты этих классов хранят те же значения, что и соответствующие им базовые типы. Объект любого из этих классов представляет собой экземпляр класса в динамической памяти, в котором хранится его неизменяемое значение. Значения базовых типов хранятся в стеке и не являются объектами. Классы, соответствующие числовым базовым типам, находятся в библиотеке java.lang, являются наследниками абстрактного класса Number и реализуют интерфейс Comparable<T>. Этот интерфейс определяет возможность сравнения объектов одного типа между собой с помощью метода int compareTo(T ob).

Объекты классов-оболочек по умолчанию получают значение null. Как было сказано выше, базовые типы данных по умолчанию инициализируются нулями. Создаются экземпляры числовых, символьного и булевского, классов с помощью статических методов valueOf(String s), parseType(String s) и decode(String s) с параметрами типа String или статического метода-фабрики valueOf(type v), где type параметр базового типа.

Автораспаковка — процесс извлечения из объекта-оболочки значения базового типа. Вызовы методов intValue(), doubleValue() и им подобных для преобразования объектов в значения базовых типов становятся излишними.

Допускается участие объектов оболочек в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким процессом. Сравнение объектов классов оболочек между собой происходит по ссылкам, как и для других объектных типов. Для сравнения значений объектов следует использовать метод equals(). Метод equals() сравнивает не значения объектных ссылок, а значения объектов, на которые установлены эти ссылки.

3. Наследование в Java. Интерфейсы

Отношение между классами, при котором характеристики одного класса (суперкласса) передаются другому классу (подклассу) без необходимости их повторного определения, называется наследованием.

Подкласс наследует поля и методы суперкласса, используя ключевое слово extends. Класс может также реализовать любое число интерфейсов, используя ключевое слово implements. Подкласс имеет прямой доступ ко всем открытым переменным и методам родительского класса, как будто они находятся в подклассе. Исключение составляют члены класса, помеченные private (во всех случаях) и «по умолчанию» для подкласса в другом

пакете. В любом случае (даже если ключевое слово extends отсутствует) класс автоматически наследует свойства суперкласса всех классов — класса Object.

Множественное наследование классов запрещено, аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, задающих поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены суперкласса своими полями и\или методами и\или переопределяет методы суперкласса. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов (статическим полиморфизмом). Если же совпадают имена и параметры методов, то этот механизм называется динамическим полиморфизмом. То есть в подклассе можно объявить (переопределить) метод с тем же именем, списком параметров и возвращаемым значением, что и у метода суперкласса.

Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного по ссылке типа объекта называется полиморфизмом. Полиморфизм является основой для реализации механизма динамического или «позднего связывания».

При объявлении совпадающих по сигнатуре (имя, тип, область видимости) полей в суперклассе и подклассах их значения не переопределяются и никак не пересекаются, т.е. существуют в одном объекте независимо друг от друга. Такое решение является плохим примером кода, который не используется в практическом программировании. Не следует использовать вызов методов, которые можно переопределить, в конструкторе. Это действие может привести к некорректной работе конструктора при инициализации полей объекта и в целом некачественному созданию объекта. Для доступа к полям текущего объекта можно использовать указатель this, для доступа к полям суперкласса — указатель super. Если разработчик объявляет метод как final, следовательно, он считает, что его версия в этой ветви наследования метода окончательна и переопределению\совершенствованию не подлежит.

Применение final-методов также показательно при разработке конструктора класса. Процесс инициализации экземпляра должен быть строго определен и не подвергаться изменениям. Исключить подмену реализации метода, вызываемого в конструкторе, следует объявлением метода как final, т.е. при этом метод не может быть переопределен в подклассе. Подобное объявление гарантирует обращение именно к этой реализации.

Ключевое слово super применяется для обращения к конструктору суперкласса и для доступа к полю или методу суперкласса. Ссылка this используется, если в методе объявлены локальные переменные с тем же именем, что и переменные экземпляра класса. Локальная переменная имеет преимущество перед полем класса и закрывает к нему доступ. Чтобы получить доступ к полю класса, требуется воспользоваться явной ссылкой this перед именем поля, так как поле класса является частью объекта, а локальная переменная нет.

Инструкция this() должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией, иначе возникает возможность вызова нескольких конструкторов суперкласса или ветвления при обращении к конструктору суперкласса. Компилятор выполнять подобные действия запрещает.

Способность Java делать выбор метода, исходя из типа объекта во время выполнения, называется «поздним связыванием». При вызове метода его поиск происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут Object — суперкласс для всех классов.

Если два метода с одинаковыми именами и возвращаемыми значениями находятся в одном классе, то списки их параметров должны отличаться. То же относится к методам, наследуемым из суперкласса. Такие методы являются перегружаемыми (overloading). При обращении вызывается доступный метод, список параметров которого совпадает со списком параметров вызова.

Если объявление метода подкласса полностью, включая параметры, совпадает с объявлением метода суперкласса (порождающего класса), то метод подкласса переопределяет (overriding) метод суперкласса. Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения. В следующем примере рассматривается реализация полиморфизма на основе динамического связывания. Так как суперкласс содержит методы, переопределенные подклассами, то объект суперкласса будет вызывать методы различных подклассов в зависимости от того, на объект какого подкласса у него имеется ссылка.

Аннотация @Override позволяет выделить в коде переопределенный метод и сгенерирует ошибку компиляции в случае, если программист изменит имя метода, типы его параметров или их количество в описании сигнатуры полиморфного метода.

Следует помнить, что при вызове метода обращение super всегда производится к ближайшему суперклассу. Переадресовать вызов, минуя суперкласс, невозможно! Аналогично при вызове super() в конструкторе обращение происходит к соответствующему конструктору непосредственного суперкласса.

Основной вывод: выбор версии переопределенного метода производится на этапе выполнения кода.

На вершине иерархии классов находится класс Object, суперкласс для всех классов. Изучение класса Object и его методов необходимо, т.к. его свойствами обладают все классы Java. Ссылочная переменная типа Object может указывать на объект любого другого класса, на любой массив, так как массив реализован как класс-наследник Object.

В классе Object определен набор методов, который наследуется всеми классами:

• protected Object clone() — создает и возвращает копию вызывающего объекта;

• public boolean equals(Object ob) — предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;

• public Class<? extends Object> getClass() — возвращает экземпляр типа Class;

• protected void finalize() — (deprecated) автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;

• public int hashCode() — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);

• public String toString() — возвращает представление объекта в виде строки.

4. Обработка строк

Строка в языке Java — это основной носитель текстовой информации. Системная библиотека Java содержит классы String, StringBuilder и StringBuffer, поддерживающие хранение строк, их обработку и определенные в пакете java.lang, подключаемом к приложению автоматически. Эти классы объявлены как final, что означает невозможность создания собственных порожденных классов со свойствами строки. Для форматирования и обработки строк применяются также классы Formatter, Pattern, Matcher, StringJoiner и другие.

Каждая строка, создаваемая с помощью оператора new, литерала (заключенная в двойные апострофы) или метода класса, создающего строку, является экземпляром класса String. Особенностью объекта класса String является то, что его значение не может быть изменено после создания объекта при помощи любого метода класса. Изменение строки всегда приводит к созданию нового объекта в heap. Сама объектная ссылка при этом сохраняет прежнее значение и хранится в стеке. Произведенные изменения можно сохранить переинициализируя ссылку.

Класс String поддерживает несколько конструкторов, например: String(), String(String original), String(byte[] bytes), String(char[] value), String(char[] value, int offset, int count), String(StringBuffer buffer), String(StringBuilder builder) и др. Эти конструкторы используются для создания объектов класса String на основе их инициализации значениями из массива типа char, byte и др.

В Java 8 класс String был подвержен серьезному изменению внутренней структуры. Вместо массива символов char теперь строка хранится в массиве типа byte, а ее кодировка в отдельном поле. Изменен алгоритм хэширования, что, как говорит Oracle, даст лучшее распределение хэш-кодов, улучшит производительность основанных на хэшировании коллекций типа Set и Map.

Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект-литерал типа String, на который можно установить ссылку. Теперь нет необходимости использовать конструктор String(String original) при создании новой строки на основе части другой строки, если новая строка была получена выделением подстроки, например, методом substring(). Ранее «обрезанная» часть сохраняла полную строку, что влекло за собой утечки памяти, порой существенные.

Некоторые методы класса String:

−String concat(String s) или оператор «+» — слияние строк;

−boolean equals(Object ob) и equalsIgnoreCase(String s) — сравнение строк с учетом и без учета нижнего и верхнего регистра символов соответственно;

−int compareTo(String s) и compareToIgnoreCase(String s) — лексикографическое сравнение строк с учетом и без учета их регистра. Метод осуществляет вычитание кодов первых различных символов вызывающей и передаваемой строки в метод строк и возвращает целое значение. Метод возвращает значение 0 в случае, когда equals() возвращает значение true;

−boolean contentEquals(CharSequence ob) — сравнение строки и содержимого объекта типа StringBuffer, StringBuilder и пр.;

−boolean matches(String regex) — проверка строки на соответствие регулярному выражению;

−String substring(int n, int m) — извлечение из строки подстроки длины m-n, начиная с позиции n. Нумерация символов в строке начинается с нуля;

−String substring(int n) — извлечение из строки подстроки, начиная с позиции n;

−int length() — определение длины строки;

−int indexOf(char ch) — определение позиции символа в строке;

−static String valueOf(type v) — преобразование переменной базового типа к строке;

−String toUpperCase()/toLowerCase() — преобразование всех символов вызывающей строки в верхний/нижний регистр;

−String replace(char c1, char c2) — замена в строке всех вхождений первого символа вторым символом;

−String replaceAll(String regex, String replacement) — замена в строке всех подстрок, соответствующих регулярному выражению, новой строкой, см. также replaceFirst();

−String intern() — заносит строку в «пул» литералов и возвращает ее объектную ссылку;

−String strip() — удаление всех пробелов в начале и конце строки, более совершенный аналог метода trim(), см. также методы stripLeading() и stripTrailing();

−char charAt(int position) — возвращение символа из указанной позиции (нумерация с нуля);

−boolean isEmpty() — возвращает true, если длина строки равна 0;

−boolean isBlank() — возвращает true, если строка пуста или содержит только пробельные символы;

−static String join(CharSequence delimiter, CharSequence... elements) — объединение произвольного набора строк (коллекции строк) в одну строку с заданной строкой-разделителем;

−char[] getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) — извлечение символов строки в массив символов;

−static String format(String format, Object… args), format(Locale l, String format, Object… args) — создание форматированной строки, полученной с использованием формата, локализации и др.;

−String[] split(String regex), String[] split(String regex, int limit) — поиск вхождения в строку заданного регулярного выражения-шаблона в качестве разделителя и деление исходной строки в соответствии с этим разделителем на массив строк;

−IntStream codePoints() — извлечение символов строки в поток (stream) их кодов;

−IntStream chars() — преобразование строки в stream ее символов;

−Stream<String> lines() — извлечение строк, разделенных символом перехода на другую строку, в поток (stream) строк.

5. Исключения и ошибки

Исключительные ситуации (исключения) и ошибки возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Обычно считается, что исключения и ошибки — тождественные понятия. Примерами «популярных» ошибок являются: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на ноль. При возникновении исключения в приложении создается объект, описывающий это исключение. Затем текущий ход выполнения приложения останавливается, и включается механизм обработки исключений. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Если в классе используется метод, в котором может возникнуть проверяемая исключительная ситуация, но не предусмотрена ее обработка, то ошибка возникает еще на этапе компиляции. При создании такого метода программист обязан включить в код метода обработку исключений, которые могут генерироваться в этом методе, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод. Схема обработки исключения подобна схеме обработки событий.

Исключение не должно восприниматься как нечто вредное, от которого следует избавиться любой ценой. Исключение — это источник дополнительной информации о ходе выполнения приложения. Такая информация позволяет лучше адаптировать код к конкретным условиям его использования, а также на ранней стадии выявить ошибки или защититься от их возникновения в будущем. В противном случае «подавление» исключений приведет к тому, что о возникшей ошибке никто не узнает или узнает на стадии некорректно обработанной информации. Поиск места возникновения ошибки может быть затруднительным.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при ее появлении. Если подходящего класса не существует, то он может быть создан разработчиком. Все исключения являются наследниками суперкласса Throwable и его подклассов Error и Exception из пакета java.lang.

Исключительные ситуации типа Error возникают только во время выполнения программы. Такие исключения, связанные с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением. Собственные подклассы от Error создавать мало смысла по причине невозможности управления прерываниями.

Возможность возникновения проверяемого исключения может быть отслежена еще на этапе компиляции кода. Компилятор проверяет, может ли данный метод генерировать или обрабатывать исключение. Проверяемые исключения должны быть обработаны в методе,

который может их генерировать, или включены в throws-список метода для дальнейшей обработки в вызывающих методах.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы.

В отличие от проверяемых исключений, класс RuntimeException и порожденные от него классы относятся к непроверяемым исключениям. Компилятор не проверяет, может ли генерировать и/или обрабатывать метод эти исключения. Исключения типа RuntimeException генерируются при возникновении ошибок во время выполнения приложения.

Ниже приведен список часто встречаемых в практике программирования непроверяемых исключений, знание причин возникновения которых необходимо при создании качественного кода.

Почему возникла необходимость деления исключений на проверяемые

и непроверяемые? Представим, что следующие ситуации проверяются на этапе компиляции, а именно:

– деление в целочисленных типах вида a/b при b=0 генерирует исключение ArithmeticException;

– индексация массивов, строк, коллекций. Выход за пределы такого объекта приводит к исключению ArrayIndexOutOfBoundsException и аналогичных;

– вызов метода на ссылке вида obj.method(), если obj ссылается на null.

Если бы возможность появления перечисленных исключений проверялась на этапе компиляции, то любая попытка индексации массива или каждый вызов метода требовали бы или блока try-catch, или секции throws. Такой код был бы практически непригоден для понимания и поддержки, поэтому часть исключений была выделена в группу непроверяемых и ответственность за защиту приложения от последствий их возникновения возложена на программиста.

Если при возникновении исключения в текущем методе обработчик не будет обнаружен, то его поиск будет продолжен в методе, вызвавшем данный метод, и так далее вплоть до метода main() для консольных приложений или другого метода, запускающего соответствующее приложение. Если же и там исключение не будет перехвачено, то JVM выполнит аварийную остановку приложения с вызовом метода printStackTrace(), выдающего данные трассировки. Для проверяемого исключения возможность его генерации отслеживается.

Передача обработки вызывающему методу осуществляется с помощью оператора throws. В конце концов исключение может быть передано в метод main(), где и находится крайняя точка обработки. Добавлять оператор throws методу main() представляется дурным тоном программирования, как безответственное действие программиста, не обращающего никакого внимания на альтернативное выполнение программы.

На практике используется один из двух способов обработки исключений:

– перехват и обработка исключения в блоке try-catch метода;

– объявление исключения в секции throws метода и передача вызывающему методу (в первую очередь для проверяемых исключений).

Первый подход можно рассмотреть на следующем примере. При преобразовании содержимого строки к числу в определенных ситуациях может возникать проверяемое исключение типа ParseException. Например:

```
public double parseFromFrance(String numberStr) {
NumberFormat format = NumberFormat.getInstance(Locale.FRANCE);
double numFrance = 0;
try {
numFrance = format.parse(numberStr).doubleValue();
} catch (ParseException e) { // checked exception
// 1. throwing a standard exception,: IllegalArgumentException() — not very good
// 2. throwing a custom exception, where ParseException as a parameter
```

```
// 3. setting the default value - if possible
// 4. logging if an exception is unlikely
}
return numFrance;}
```

Исключительная ситуация возникнет в случае, если переданная строка содержит нечисловые символы или не является числом. Генерируется объект исключения, и управление передается соответствующему блоку catch, в котором он обрабатывается, иначе блок catch пропускается. Блок try похож на обычный логический блок. Блок catch(){} похож на метод, принимающий в качестве единственного параметра ссылку на объект-исключение и обрабатывающий этот объект.

Второй подход демонстрируется на этом же примере. Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод должен объявить о таком поведении с помощью ключевого слова throws, чтобы вызывающий метод мог защитить себя от этих исключений. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача соответствующему методу.

При этом объявляемый метод может содержать блоки try-catch, а может и не содержать их. Например, метод parseFrance() можно объявить:

```
public double parseFrance(String numberStr) throws ParseException {
NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
double numFrance = formatFrance.parse(numberStr).doubleValue();}
```

В практическом программировании такой подход допустим для private-методов.

Ключевое слово throws после имени метода позволяет разобраться с исключениями методов «чужих» классов, код которых отсутствует. Обрабатывать исключение при этом должен будет метод, вызывающий parseFrance():

```
public void doAction() {
// code here
try {
parseFrance(numberStr);
} catch (ParseException e) {
// code}}
```

Если в блоке try может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков catch, если только блок catch не обрабатывает все типы исключений.

```
public void doAction() {
try { int a = (int) (Math.random() * 2);
System.out.println("a = " + a);
int c[] = { 1 / a }; // place of occurrence of exception #1
c[a] = 71; // place of occurrence of exception #2
} catch (ArithmeticException e) {
System.err.println("divide by zero " + e);
} catch (ArrayIndexOutOfBoundsException e) {
System.err.println("out of bound: " + e);
} // end try-catch block
System.out.println("after try-catch"); }
```

Исключение ArithmeticException при делении на 0 возникнет при инициализации элемента массива c[0] действием 1/a при a=0. В случае a=1 генерируется исключение «превышение границ массива» при попытке присвоить значение второму элементу массива c[], содержащего только один элемент. Однако пример, приведенный выше, носит чисто демонстративный характер обойтись простой проверкой аргументов на допустимые значения перед выполнением операций. К тому же генерация и обработка исключения — операция

значительно более ресурсоемкая, чем вызов оператора if для проверки аргумента. Исключения должны применяться только для обработки исключительных ситуаций, и если существует возможность обойтись без них, то следует так и поступить.

Подклассы исключений в блоках catch должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения.

Например:

```
try { //...
} catch(IllegalArgumentException e) { //...
} catch(PatternSyntaxException e) { // unreachable code }
```

где класс PatternSyntaxException представляет собой подкласс класса IllegalArgumentException. Корректно будет просто поменять местами блоки

catch:

```
try {
} catch(PatternSyntaxException e) { //..
} catch(IllegalArgumentException e) { //..}
```

На практике иногда возникают ситуации, когда инструкций catch несколько, и обработка производится идентичная, например, вывод сообщения об исключении в журнал.

```
try {
// some code
} catch(NumberFormatException e) {
e.printStackTrace(); // or log
} catch(ClassNotFoundException e) {
e.printStackTrace(); // or log
} catch(InstantiationException e) {
e.printStackTrace(); // or log }
```

В версии Java 7 появилась возможность объединить все идентичные инструкции в одну, используя для разделения оператор «|».

```
try {
// some code
} catch(NumberFormatException | ClassNotFoundException | InstantiationException e){
e.printStackTrace(); }
```

В catch не могут находиться исключения из одной иерархической цепочки. Такая запись позволяет избавиться от дублирования кода.

Введено понятие более точной переброски исключений (more precise rethrow). Это решение применимо в случае, если обработка возникающих исключений не предусматривается в методе и должна быть передана вызывающему данный метод методу.

До введения этого понятия код выглядел так:

```
public double parseFromFileBefore(String filename)
throws FileNotFoundException, ParseException, IOException {
NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
double numFrance = 0;
BufferedReader bufferedReader = null;
try {
FileReader reader = new FileReader(filename);
bufferedReader = new BufferedReader(reader);
String number = bufferedReader.readLine();
numFrance = formatFrance.parse(number).doubleValue();
} catch (FileNotFoundException e) {
throw e;
} catch (IOException e) {
throw e;
} catch (ParseException e) {
```

```
throw e;
} finally {
if (bufferedReader != null) {
bufferedReader.close(); }}
return numFrance; }
```

More precise rethrow разрешает записать в единственную инструкцию catch более общее исключение, чем может быть генерировано в инструкции try, с последующей генерацией перехваченного исключения для его передачи в вызывающий метод.

```
public double parseFile(String filename)
throws FileNotFoundException, ParseException, IOException {
NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
double numFrance = 0;
BufferedReader bufferedReader = null;
try {
FileReader reader = new FileReader(filename);
bufferedReader = new BufferedReader(reader);
String number = bufferedReader.readLine();
numFrance = formatFrance.parse(number).doubleValue();
} catch (final Exception e) { // final — optional
throw e; // more precise rethrow
} finally {
if (bufferedReader != null) {
bufferedReader.close(); }}
return numFrance;}
```

Наличие секции throws контролируется компилятором на предмет точного указания списка проверяемых исключений, которые могут быть генерированы в блоке try-catch. При возможности возникновения непроверяемых исключений последние в секции throws обычно не указываются. Ключевое слово final не позволяет подменить экземпляр исключения для передачи за пределы метода. Однако данную конструкцию можно использовать и без final.

Операторы try можно вкладывать друг в друга. Если у оператора try низкого уровня нет раздела catch, соответствующего возникшему исключению, поиск будет развернут на одну ступень выше и будут проверены разделы catch внешнего оператора try.

```
try { // outer block
int a = (int) (Math.random() * 2) - 1;
System.out.println("a = " + a);
try { // inner block
int b = 1 / a;
StringBuilder builder = new StringBuilder(a);
} catch (NegativeArraySizeException e) {
System.err.println("invalid buffer size: " + e);}
} catch (ArithmeticException e) {
System.err.println("divide by zero: " + e);}
```

В результате запуска приложения при a=0 будет сгенерировано исключение ArithmeticException, а подходящий для его обработки блок try-catch является внешним по отношению к месту генерации исключения. Этот блок и будет задействован для обработки возникшей исключительной ситуации. Вкладывание блоков try-catch друг в друга загромождает код, поэтому такими конструкциями следует пользоваться с осторожностью.

При разработке кода возникают ситуации, когда в приложении необходимо инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Для генерации исключительной ситуации и создания экземпляра исключения используется

оператор throw. В качестве исключения должен быть использован объект подкласса класса Throwable, а также ссылки на них.

Общая форма записи инструкции throw, генерирующей исключение:

throw subclassThrowable;

Объект-исключение может уже существовать или создаваться с помощью оператора new:

throw new IllegalArgumentException();

При достижении оператора throw, выполнение кода прекращается. Ближайший блок try проверяется на наличие соответствующего обработчика catch. Если он существует, управление передается ему, иначе проверяется следующий из вложенных операторов try. Инициализация объекта-исключения без оператора throw никакой исключительной ситуации не вызовет. В ситуации, когда получение методом достоверной информации критично для выполнения им своей функциональности, у программиста может возникнуть необходимость в генерации исключения, так как метод не может выполнить ожидаемых от него действий, основываясь на некорректных или ошибочных данных. Ниже приведен пример, в котором оператор throw генерирует исключение, обрабатываемое виртуальной машиной при выбросе из метода

main().

```
public class ResourceAction {
public static void load(Resource resource) {
if (resource == null || !resource.exists() || !resource.isCreate()) {
throw new IllegalArgumentException();
// better custom exception, eg., throw new ResourceException();
}
// more code }}
public class ActionMain {
public static void main(String[] args) {
Resource resource = new Resource(); //or Resource resource = null;//!!
ResourceAction.load(resource); }}
public class Resource {
// fields
public boolean isCreate() {
// more code}
public boolean exists() {
// more code}
public void execute() {
// more code
}
public void close() {
// more code }}
```

Вызываемый метод load() может при отсутствии требуемого ресурса или при аргументе null генерировать исключение, перехватываемое обработчиком.

В результате экземпляр непроверяемого исключения IllegalArgumentException как подкласса класса RuntimeException передается обработчику исключений в методе main().

6. Работа с файлами. Сериализация

Для работы с физическими файлами и каталогами (директориями), расположенными на внешних носителях, в приложениях Java используются классы из пакетов java.io и java.nio.

В Java 7 были добавлены класс java.nio.file.Files и интерфейс java.nio.file.Path, дублирующие и существенно расширяющие возможности класса java.io.File, возможности которого будут рассмотрены ниже.

Интерфейс Path представляет более совершенный аналог класса File, а класс Files, по сути, утилитный класс, содержащий только статические методы для доступа к файлам, директориям, их свойствам и их содержимому.

Получить объект Path можно как из объекта File:

```
File file = new File("data/info.txt");
Path path = file.toPath();
```

так и прямой инициализацией:

```
Path path1 = Paths.get("data/info.txt");
```

или

```
Path path2 = FileSystems.getDefault().getPath("data/info.txt");
```

Доступ и управление файловой системой, доступной для текущей версии JVM, осуществляют классы java.nio.file.FileSystem и java.nio.file.FileSystems.

Класс FileSystems определяет набор статических методов для получения и создания файловых систем. Вызов метода FileSystems.getDefault() предоставляет доступ к текущей файловой системе.

При работе с файлами и директориями\каталогами лучше воспользоваться возможностями пакета java.nio.file, так как его классы позволяют учитывать очень широкий набор свойств объектов.

Класс java.io.File обладает достаточно ограниченной функциональностью.

Класс File служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как право доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Объект класса File создается одним из способов:

```
File file = new File("\\com\\file.txt");
File dir = new File("c:/jdk/src/java/io");
File file1 = new File(dir, "File.java");
File file2 = new File("c:\\com", "file.txt");
```

В первом случае создается объект, соответствующий файлу, во втором — подкаталогу. Третий и четвертый случаи практически идентичны. Для создания объекта указывается каталог и имя файла.

При создании объекта класса File любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Существует разница между разделителями, употребляющимися при записи пути к файлу: для системы Unix — «/», а для Windows — «\\». Для случаев, специальные поля в классе File:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File file = new File(File.separator + "com" + File.separator + "data.txt");
```

Также предусмотрен еще один тип разделителей для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

К примеру, для ОС Unix значение pathSeparator принимает значение «:», а для ОС MS-DOS — «;».

В отличие от Java 1.1 в языке Java 1.2 для ввода используется не байтовый, а символьный поток. В этой ситуации для ввода используется подкласс BufferedReader абстрактного класса Reader и методы read() и readLine() для чтения символа и строки соответственно. Этот поток для организации чтения из файла лучше всего инициализировать объектом класса FileReader в виде:

```
new BufferedReader(new FileReader(new File("data\\res.txt")));
```

Кроме данных базовых типов, в поток можно отправлять объекты классов целиком в байтовом представлении для передачи клиентскому приложению, а также для хранения в файле или базе данных.

Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией. Существует два способа сделать объект сериализуемым.

Для того, чтобы объекты класса могли быть подвергнуты процессу сериализации, этот класс должен имплементировать интерфейс java.io.Serializable. Все подклассы такого класса также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора static или transient. Спецификаторы transient и static означают, что поля, помеченные ими, не могут быть предметом сериализации, но существует различие в десериализации. Так, поле со спецификатором transient после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением null), а поле со спецификатором static получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта. На поля, помеченные как final, ключевое слово transient не действует.

Интерфейс Serializable не имеет методов, которые необходимо реализовать, поэтому его использование ограничивается упоминанием при объявлении класса. Все действия в дальнейшем производятся по умолчанию. Для записи объектов в поток необходимо использовать класс ObjectOutputStream. После этого достаточно вызвать метод writeObject(Object ob) этого класса для сериализации объекта ob и пересылки его в выходной поток данных. Для чтения используются, соответственно, класс ObjectInputStream и его метод readObject(), лученный объект к нужному типу. Необходимо знать, что при использовании Serializable десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор сериализуемого класса при этом не вызывается, но вызываются все конструкторы суперклассов в заданной последовательности до класса, имплементирующего Serializable.

При сериализации объекта класса, реализующего интерфейс Serializable, учитывается порядок объявления полей в классе. Поэтому при изменении порядка, имен и типов полей или добавлении новых полей в класс структура информации, содержащейся в сериализованном объекте, будет серьезно отличаться от новой структуры класса. Поэтому десериализация может пройти некорректно. Этим обусловлена необходимость добавления программистом в каждый класс, реализующий интерфейс Serializable, поля private static final long serialVersionUID на стадии разработки класса. Это поле содержит уникальный идентификатор версии класса. Оно задается программистом или вычисляется по содержимому класса — полям, их порядку объявления, методам, их порядку объявления. Для этого применяются специальные программы-генераторы UID.

Это поле записывается в поток при сериализации класса. Это тот случай, когда static-поле сериализуется. При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение java.io.InvalidClassException. Соответственно, при любом изменении в первую очередь полей класса значение поля serialVersionUID должно быть изменено программистом или генератором.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации и десериализации ничего не угрожает.

Вместо реализации интерфейса Serializable можно реализовать Externalizable, который содержит два метода: writeExternal(ObjectOutput out) и readExternal(ObjectInput in).

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра

должен сам класс. Для этого в нем должны быть переопределены методы writeExternal() и readExternal() интерфейса Externalizable.

Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении Externalizable-объекта экземпляр создается путем вызова конструктора без аргументов, после чего вызывается метод readExternal(), поэтому необходимо проследить, чтобы в классе был конструктор по умолчанию. Для сохранения состояния вызываются методы ObjectOutput, с помощью которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе readExternal() эти значения должны быть считаны в том же порядке.

Для чтения и записи в поток значений отдельных полей объекта используются методы внутренних классов: ObjectInputStream.GetField и ObjectOutputStream.PutField.

7. Коллекции в Java

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных структур данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: заменить, просмотреть элементы, подсчитать их количество и др.

Для работы с коллекциями разработчиками был создан Collection Framework.

Применение коллекций обусловливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч или миллионов, массивы не обеспечивают ни должной скорости, ни экономии ресурсов.

Примером коллекции является стек (структура LIFO — Last In First Out), в котором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO — First In First Out) используется другое правило удаления: всегда удаляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщенные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

Коллекции в языке Java объединены в библиотеке классов java.util и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с простейшими структурами данных: Vector, Stack, Hashtable, BitSet, а также интерфейс Enumeration для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют собой общую технологию хранения и доступа к объектам.

Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому, если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, возможно использование коллекции из Java 1.

В Java 5 в новом пакете java.util.concurrent появились ограниченно потокобезопасные коллекции, гарантирующие более высокую производительность в многопоточной среде для конкурирующих потоков.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие, не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты

на исправление и верификацию кода. Поэтому, начиная с версии Java SE 5, коллекции стали типизированными или generic.

Более удобным стал механизм работы с коллекциями, а именно:

• предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;

• отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип Object) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как Object — суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых.

Коллекции — это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

Map<K, V> — карта отображения вида «ключ-значение»;

Collection<E> — основной интерфейс коллекций, вершина иерархии коллекций List, Set. Также наследует интерфейс Iterable<E>;

List<E> — специализирует коллекции для обработки упорядоченного набора элементов;

Set<E> — множество, содержащее уникальные элементы;

Queue<E> — очередь, где элементы добавляются в один конец списка, а извлекаются из другого конца.

Все классы коллекций реализуют интерфейсы Serializable, Cloneable (кроме WeakHashMap).

В интерфейсе Collection<E> определены методы, которые работают на всех коллекциях:

boolean add(E obj) — добавляет obj к вызывающей коллекции и возвращает true, если объект добавлен, и false, если obj уже элемент коллекции;

boolean remove(Object obj) — удаляет obj из коллекции;

boolean addAll(Collection<? extends E> c) — добавляет все элементы коллекции к вызывающей коллекции;

void clear() — удаляет все элементы из коллекции;

boolean contains(Object obj) — возвращает true, если вызывающая коллекция содержит элемент obj;

boolean equals(Object obj) — возвращает true, если коллекции эквивалентны;

boolean isEmpty() — возвращает true, если коллекция пуста;

int size() — возвращает количество элементов в коллекции;

Object[] toArray() — копирует элементы коллекции в массив объектов;

<T> T[] toArray(T a[]) — копирует элементы коллекции в массив объектов определенного типа.

Появление Stream API обусловило возникновение методов для создания потоков объектов и работы с функциональными интерфейсами:

default Stream<E> stream() — преобразует коллекцию в stream объектов;

default Stream<E> parallelStream() — преобразует коллекцию в параллельный stream объектов. Повышает производительность при работе с очень большими коллекциями на многоядерных процессорах;

default boolean removeIf(Predicate<? super E> filter) — удаляет все элементы коллекции в зависимости от условия.

Методы void forEach(Consumer<? super T> action), Iterator<T> iterator(), Spliterator<T> spliterator() унаследованы от интерфейса Iterable<T>.

При работе с элементами коллекции применяются интерфейсы: Iterator<E>, ListIterator<E>, Map.Entry<K, V> — для перебора коллекции и доступа к объектам коллекции.

Интерфейс Iterator<E> строит объект, обеспечивающий доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом iterator(). Такой объект позволяет осуществлять навигацию по содержимому коллекции последовательно, элемент за элементом. Позиции итератора условно располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса Iterator<E>, представляющего собой одну из реализаций дизайн-паттерна с одноименным названием:

boolean hasNext() — проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает false. Итератор при этом остается неизменным;

E next() — возвращает ссылку на объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ метод next() генерирует исключение NoSuchElementException;

void remove() — удаляет объект, возвращенный последним вызовом метода next(). Если метод next() до вызова remove() не вызывался, то будет сгенерировано исключение IllegalStateException;

void forEachRemaining(Consumer<? super E> action) — выполняет действие над каждым оставшимся необработанным элементом коллекции.

Интерфейс Map.Entry предназначен для извлечения ключей и значений карты с помощью методов K getKey() и V getValue() соответственно. Вызов метода V setValue(V value) заменяет значение, ассоциированное с текущим ключом.

Иерархия наследования следующая:

java.util.AbstractCollection<E>

java.util.AbstractList<E>

java.util.ArrayList<E>

В классе объявлены конструкторы:

ArrayList()

ArrayList(Collection <? extends E> c)

ArrayList(int capacity)

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов или дефолтными методами интерфейсов Collection, List, Iterable. Методы интерфейса List<E> позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

E get(int index) — возвращает элемент в виде объекта из позиции index, представляет собой одно из главных достоинств класса из-за скорости выполнения;

void add(int index, E element) — вставляет element в позицию, указанную в index;

E remove(int index) — удаляет объект из позиции index;

E set(int index, E element) — заменяет объект в позиции index, возвращает при этом удаляемый элемент;

boolean addAll(int index, Collection<? extends E> c) — вставляет в вызывающий список все элементы коллекции c, начиная с позиции index;

int indexOf(Object ob) — возвращает индекс указанного объекта;

default void sort(Comparator<? super E> c) — сортирует список на основе компаратора;

List<E> subList(int fromIndex, int toIndex) — извлекает часть коллекции в указанных границах;

static <E> List<E>copyOf(Collection <? extentds E> coll) — создает немодифицируемый список на основе передаваемой коллекции.

Удаление и добавление элементов для такой коллекции представляет ресурсоемкую задачу, поэтому объект ArrayList<E> лучше всего подходит для хранения списков с малым числом подобных действий. С другой стороны, навигация по списку осуществляется очень быстро, поэтому операции поиска производятся за более короткое время.

Коллекция LinkedList<E> реализует возможности связанного списка.

java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.AbstractSequentialList<E>
java.util.LinkedList<E>

Реализует, кроме интерфейсов, указанных при описании ArrayList, также интерфейсы Queue<E> и Deque<E>.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто называется двунаправленным списком.

Самый быстрый метод класса add(E element). Главным же достоинством класса является скорость работы метода remove() на Iterator, после получения его из LinkedList. Также очень быстро работает метод add(E element) на ListIterator.

Операция удаления из начала и конца списка выполняется достаточно быстро, в отличие от операций поиска и извлечения.

При тестировании на списке из десяти тысяч элементов LinkedList быстрее, чем ArrayList, при добавлении в середину списка методом add() в 2 раза, а в начало или конец примерно в 40 раз. Вставки и удаления элементов из LinkedList происходят за постоянное время, в том числе и с использованием итераторов, в тоже время вставка\удаление элемента в ArrayList приводит к сдвигу всех элементов после позиции добавления\удаления, а в случае, если базовый массив хранения переполняется, то еще и сам массив увеличивается в полтора раза с копированием старого массива в новый. Список ArrayList, в свою очередь, быстрее при вызове метода get(index) примерно в 50 раз. Происходит это вследствие того, что определение позиции в списке производится за конкретный интервал времени, не зависящий от размера списка, при поиске же индекса в LinkedList время поиска пропорционально размеру списка.

Список LinkedList занимает больший объем памяти за счет необходимости хранения ссылок на соседние объекты, что следует учитывать при создании списков больших размеров. Список LinkedList занимает от 3,5 до 5 раз больше памяти нежели аналогичный список ArrayList.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т.д. Двунаправленный список, кроме обычного, имеет особый «нисходящий» итератор, позволяющий двигаться от конца списка к началу, и извлекается методом descendingIterator().

Для манипуляций с первым и последним элементами списка в LinkedList<E> реализованы методы:

void addFirst(E ob), void addLast(E ob) — добавляющие элементы в начало и конец списка;

E getFirst(), E getLast() — извлекающие элементы;

E removeFirst(), E removeLast() — удаляющие и извлекающие элементы;

E removeLastOccurrence(E elem), E removeFirstOccurrence(E elem) — удаляющие и извлекающие элемент, первый или последний раз встречаемый в списке.

Класс LinkedList<E> реализует интерфейс Queue<E>, что позволяет списку придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (first in, first out). Элементы добавляются в конец и вынимаются из начала очереди. Но существует возможность не только добавлять и удалять элементы, также можно просмотреть, что находится в очереди. К тому же методы интерфейса Queue<E> по манипуляции первым и последним элементами такого списка E element(), boolean offer(E o), E peek(), E poll(), E remove() работают немного быстрее, чем соответствующие методы класса LinkedList<E>.

Методы интерфейса Queue<E>:

boolean add(E o) — вставляет элемент в очередь, но если же очередь полностью заполнена, то генерирует исключение IllegalStateException;

boolean offer(E o) — вставляет элемент в очередь, если возможно;

E element() — возвращает, но не удаляет головной элемент очереди;

E peek() — возвращает, но не удаляет головной элемент очереди, возвращает null, если очередь пуста;

E poll() — возвращает и удаляет головной элемент очереди, возвращает null, если очередь пуста;

E remove() — возвращает и удаляет головной элемент очереди.

Методы element() и remove() отличаются от методов peek() и poll() тем, что генерируют исключение NoSuchElementException, если очередь пуста.

Queue<Order> queue = new LinkedList<>();

Создается очередь простым присваиванием списка LinkedList ссылке типа Queue.

Интерфейс Deque определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди.

Реализацию этого интерфейса можно использовать для моделирования стека. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих методов существует в двух формах. Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-либо из значений (null или false в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций Deque, имеющих ограничение по размеру. В большинстве реализаций операции добавления заканчиваются успешно. Методы addFirst(), addLast() вставляют элементы в начало и в конец очереди соответственно. Метод add() унаследован от интерфейса Queue и абсолютно аналогичен методу addLast() интерфейса Deque. Объявить двуконечную очередь на основе связанного списка можно, например, следующим образом:

Deque<String> deque = new LinkedList<>();

Интерфейс Set<E> объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс SortedSet<E> наследует Set<E> и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса. Интерфейс NavigableSet<E> существенно облегчает поиск элементов, например, расположенных рядом с заданным.

Класс HashSet<E> наследуется от абстрактного суперкласса AbstractSet<E> и реализует интерфейс Set<E>, используя хэш-таблицу для хранения коллекции.

Ключ хэш-код используется в качестве индекса хэш-таблицы для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента. Скорость указанных процессов становится заметной для коллекций с большим количеством элементов. Множество HashSet не является сортированным. В таком множестве могут храниться элементы с одинаковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравнении. Для грамотной организации HashSet стоит следить, чтобы реализации методов equals() и hashCode() соответствовали правилам.

Класс TreeSet<E> для хранения объектов использует бинарное (красно-черное) дерево. С этим алгоритмом желательно ознакомиться самостоятельно.

Иерархия наследования TreeSet:

java.util.AbstractCollection<E>

java.util.AbstractSet<E>

java.util.TreeSet<E>

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что класс реализует интерфейс SortedSet, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс Comparable. Обработка операций удаления и вставки объектов происходит несколько медленнее, чем в хэш-множествах, где при любом числе элементов время этих операций постоянно.

Конструкторы класса:

TreeSet()

TreeSet(Collection <? extends E> c)

TreeSet(Comparator <? super E> c)

TreeSet(SortedSet <E> s)

Класс TreeSet<E> содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов E first() и E last(). Методы subSet(E from, E to), tailSet(E from) и headSet(E to) предназначены для извлечения определенной части множества. Метод Comparator <? super E> comparator() возвращает объект Comparator, используемый для сортировки объектов множества или null, если выполняется обычная сортировка.

Карта отображений — это объект, который хранит пару «ключ–значение». Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по уникальному ключу. Уникальность объектов-ключей должна обеспечиваться переопределением методов hashCode() и equals() или реализацией интерфейсов Comparable, Comparator пользовательским классом. Классы карт отображений:

AbstractMap<K, V> — реализует интерфейс Map<K, V>, является супер-

классом для всех перечисленных карт отображений;

HashMap<K, V> — использует хэш-таблицу для работы с ключами;

TreeMap<K, V> — использует дерево, где ключи расположены в виде дерева поиска в определенном порядке;

WeakHashMap<K, V> — позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения;

LinkedHashMap<K, V> — образует дважды связанный список ключей.

Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

Для класса IdentityHashMap<K, V> хэш-коды объектов-ключей вычисляются методом System.identityHashCode() по адресу объекта в памяти, в отличие от обычного значения hashCode(), вычисляемого сугубо по содержимому самого объекта.

Интерфейсы карт:

Map<K, V> — отображает уникальные ключи и значения;

Map.Entry<K, V> — описывает пару «ключ–значение»;

SortedMap<K, V> — содержит отсортированные ключи и значения;

NavigableMap<K, V> — добавляет новые возможности навигации и поиска по ключу.

Интерфейс Map<K, V> содержит следующие методы:

V get(Object obj) — возвращает значение, связанное с ключом obj. Если элемент с указанным ключом отсутствует в карте, то возвращается значение null;

V put(K key, V value) — помещает ключ key и значение value в вызывающую карту. При добавлении в карту элемента с существующим ключом, произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент;

default V putIfAbsent(K key, V value) — помещает ключ key и значение value в вызывающую карту. При добавлении в карту элемента с существующим ключом, замена не произойдет;

default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) — помещает ключ key и вычисляет значение value при добавлении в вызывающую карту;

default V computeIfAbsent(K key, Function<? super K, ? super V> mappingFunction) — помещает ключ key и значение value в вызывающую карту, если пары с таким ключом не существует, если ключ существует, то замена не производится;

default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) — заменяет значение value в вызывающей карте, если ключ с таким значением существует, если же пары с таким ключом не существует, то вставка пары не производится;

void putAll(Map <? extends K, ? extends V> m) — помещает карту m в вызывающую карту;

V remove(Object key) — удаляет пару «ключ–значение» по ключу key;

void clear() — удаляет все пары из вызываемой карты;

boolean containsKey(Object key) — возвращает true, если вызывающая карта содержит key как ключ;

boolean containsValue(Object value) — возвращает true, если вызывающая карта содержит value как значение;

Set<K> keySet() — возвращает множество ключей;

Set<Map.Entry<K, V>> entrySet() — возвращает множество, содержащее значения карты в виде пар «ключ–значение»;

Collection<V> values() — возвращает коллекцию, содержащую значения карты;

static <K, V> Map<K, V> copyOf(Map <? extends K,? extends V> map) — копирует исходную карту в немодифицируемую новую карту;

static <K, V> Map<K, V> of(parameters) — перегруженный метод для создания неизменяемых карт на основе переданных в метод параметров;

default void forEach(BiConsumer<? super K, ? super V> action) — выполняет действие над каждым элементом Map.

В коллекциях, возвращаемых тремя последними методами, можно только удалять элементы, добавлять нельзя. Данное ограничение обуславливается параметризацией возвращаемого методами значения.

Интерфейс Map.Entry<K, V> представляет пару «ключ–значение» и содержит следующие методы:

K getKey() — возвращает ключ текущего входа;

V getValue() — возвращает значение текущего входа;

V setValue(V obj) — устанавливает значение объекта obj в текущем входе.

Класс java.util.Collections содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

<T> void copy(List<? super T> dest, List<? extends T> src) — копирует все элементы из одного списка в другой;

boolean disjoint(Collection<?> c1, Collection<?> c2) — возвращает true, если коллекции не содержат одинаковых элементов;

<T> List <T> emptyList(), <K, V> Map <K, V> emptyMap(), <T> Set <T> emptySet() — возвращают пустой список, карту отображения и множество соответственно;

<T> void fill(List<? super T> list, T obj) — заполняет список заданным элементом;

int frequency(Collection<?> c, Object o) — возвращает количество вхождений в коллекцию заданного элемента;

<T> boolean replaceAll(List<T> list, T oldVal, T newVal) — заменяет все заданные элементы новыми;

void reverse(List<?> list) — «переворачивает» список;

void rotate(List<?> list, int distance) — сдвигает список циклически на заданное число элементов;

void shuffle(List<?> list) — перетасовывает элементы списка;

singleton(T o), singletonList(T o), singletonMap(K key, V value) — создают множество, список и карту отображения, позволяющие добавлять только один элемент;

<T> void sort(List<T> list, Comparator<? super T> c) — сортировка списка естественным порядком и с использованием Comparable или Comparator соответственно;

void swap(List<?> list, int i, int j) — меняет местами элементы списка, стоящие на заданных позициях;

<T> List<T> unmodifiableList(List<? extends T> list) — возвращает ссылку на список с запрещением его модификации. Аналогичные методы есть для всех коллекций.

8. Работа с XML документами

JAXP (Java API for XML Processing). XML-документ как набор байт в памяти, запись в базе или текстовый файл представляет собой данные, которые еще предстоит обработать. То есть из набора строк необходимо получить данные, пригодные для использования в проекте. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки — XML-анализаторы, или парсеры.

Парсер — это библиотека, которая читает XML-документ, а затем предоставляет набор методов для обработки информации из этого документа.

Древовидная и псевдособытийная модели: существуют три стандартных подхода (API) к обработке XML-документов:

——DOM (Document Object Model) — платформенно-независимый программный интерфейс, позволяющий программам и скриптам управлять содержимым документов HTML и XML, а также изменять их структуру и оформление. Модель DOM не накладывает ограничений на структуру документа. Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел которого содержит элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями родитель-потомок.

——SAX (Simple API for XML) базируется на модели последовательной одноразовой обработки и не создает внутренних деревьев. При прохождении по XML вызывает соответствующие методы у классов, реализующих интерфейсы, предоставляемые SAX-парсером.

——StAX (Streaming API for XML) не создает дерево объектов в памяти, но, в отличие от SAX-парсера, за переход от одной вершины XML к другой отвечает приложение, которое запускает разбор документа.

Анализаторы, которые строят древовидную модель, — это DOM- анализаторы. Анализаторы, которые генерируют квазисобытия, — это SAX-анализаторы.

Анализаторы, которые ждут команды от приложения для перехода к следующему элементу XML, — StAX-анализаторы.

В первом случае анализатор строит в памяти объект, представляющий собой дерево из элементов, соответствующее XML-документу. Далее вся работа ведется именно с этим объектом-деревом, который может быть очень большим, но предоставляет доступ к своим элементам в любой момент времени.

Во втором случае анализатор работает следующим образом: при чтении/анализе документа, анализатор вызывает методы, связанные с различными участками XML-файла, а программа, использующая анализатор, решает, как реагировать на тот или иной элемент XML-документа. Так, анализатор будет генерировать событие о том, что он встретил начало документа либо его конец, начало элемента либо его конец, символьную информацию внутри элемента и т.д.

Анализатор StAX работает подобно итератору, который указывает на наличие элемента с помощью метода hasNext() и для перехода к следующей вершине использует метод next().

Когда следует использовать DOM, а когда — SAX, StAX-анализаторы?

DOM-анализаторы следует использовать тогда, когда нужно знать структуру документа и может понадобиться изменять эту структуру либо использовать информацию из XML-документа в любой момент времени работы приложения.

SAX/StAX-анализаторы используются тогда, когда нужно извлечь информацию о нескольких элементах из XML-файла, либо когда информация из документа нужна только один раз.

Валидация. XML-документ может соответствовать двум видам корректности: синтаксической (well-formed) — документ сформирован в соответствии с синтаксическими

правилами построения — и действительной (valid) — документ синтаксически корректен и соответствует требованиям, заявленным в XSD.

Соответственно, есть невалидирующие и валидирующие анализаторы. И те, и другие проверяют XML-документ на соответствие синтаксическим правилам. Но только валидирующие анализаторы знают, как проверить XML-документ на соответствие структуре, описанной в XSD.

Никакой связи между видом анализатора и видом XML-документа не существует. Валидирующий анализатор может разобрать XML-документ, для которого нет XSD, и, наоборот, невалидирующий анализатор может разобрать XML-документ, для которого есть XSD. При этом он просто не будет учитывать описание структуры документа.

При проверке XML-документа разумно все ошибки фиксировать, в частности, с помощью логгера Log4J2, минимальные библиотеки которого log4japi-2.[version].jar и log4j-core-2.[version].jar следует подключить к проекту.

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует квазисобытие при нахождении элемента, атрибута или текста. На первый взгляд, такой подход менее естествен для приложения, использующего анализатор, так как он не строит дерево, а приложение само должно догадаться, какое дерево элементов описывается в XML-документе.

Однако нужно учитывать, для каких целей используются данные из XML- файла. Очевидно, что нет смысла строить дерево объектов, содержащее десятки тысяч элементов в памяти, если все, что необходимо, — просто посчитать точное количество элементов в файле.

SAX-анализаторы. SAX2 API определяет ряд интерфейсов, используемых при разборе документа. Чаще других используется org.xml.sax.ContentHandler и некоторые объявленные в нем методы:

void startDocument() — вызывается на старте обработки документа;

void endDocument() — вызывается при завершении разбора документа;

void startElement(String uri, String localName, String qName, Attributes attrs) — будет вызван, когда анализатор полностью обработает содержимое открывающего тега, включая его имя и все содержащиеся атрибуты;

void endElement(String uri, String localName, String qName) — сигнализирует о завершении элемента;

void characters(char[] ch, int start, int length) — вызывается в том случае, если анализатор встретил символьную информацию внутри элемента (тело тега). Если этой информации достаточно много, то метод может быть вызван более одного раза.

В пакете org.xml.sax в SAX2 API содержатся также интерфейсы, необходимые для обработки интересующего события.

Для того чтобы создать простейшее приложение, обрабатывающее XML-документ, достаточно сделать следующее:

1. Создать класс или классы, которые реализуют один или несколько интерфейсов ContentHandler, ErrorHandler или EntityResolver, или наследует их класс-адаптер org.xml.sax.helpers.DefaultHandler, и реализовать методы, отвечающие за обработку интересующих частей документа или ошибок. Класс DefaultHandler имплементирует все вышеуказанные интерфейсы и реализует все их абстрактные методы. Его реализации методов пустые, что позволяет разработчику переопределять только необходимые для выполнения задачи методы.

2. Используя SAX2 API, поддерживаемое всеми SAX-парсерами, создать org.xml.sax.XMLReader, например:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
XMLReader xmlReader = parser.getXMLReader();
```

3. Передать в объект XMLReader объект одного или нескольких классов, созданного на первом шаге с помощью методов: setContentHandler(Content Handler handler), setErrorHandler(ErrorHandler handler), setEntityResolver(EntityResolver resolver) и прочих.

4. Вызвать метод parse(String filename) или parse(InputSource input) класса XMLReader, которому в качестве параметра передать путь к анализируемому документу либо InputSource.

Древовидная модель. Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода, однако при совместном использовании библиотек с аналогичными классами необходимо следить за совместимостью и корректностью взаимодействия.

DOM строит дерево, которое представляет содержимое XML-документа и определяет набор классов, представляющих каждый элемент в XML-документе: элементы, атрибуты, сущности, текст и т.д.

В пакете org.w3c.dom можно найти интерфейсы, представляющие указанные объекты. Разработчики приложений, которые хотят использовать DOM-анализатор, имеют готовый набор методов для манипуляции деревом объектов и не зависят от конкретной реализации используемого анализатора. Существуют общепризнанные DOM-анализаторы: Xerces, JDOM и JAXP, входящий в JDK.

DOM. В стандартную конфигурацию Java входит набор пакетов для работы с XML. Но стандартная библиотека не всегда является самой простой в применении, поэтому часто в основе многих проектов, использующих XML, лежат библиотеки сторонних производителей. Эти библиотеки представлены проектами Xerces, Xalan, JDOM. Особенностью всех их является использование части стандартных возможностей XML-библиотек Java с добавлением собственных классов и методов, упрощающих и облегчающих обработку документов XML.

Ниже приведены интерфейсы и методы, необходимые для разбора XML- документа. org.w3c.dom.Node. Основным объектом DOM является Node — некоторый базовый элемент дерева. Большинство DOM-объектов унаследовано именно от Node. Для представления элементов, атрибутов, сущностей разработаны свои реализации Node. Некоторые из них — Element, Attr, Text, для работы с конкретными объектами дерева.

Интерфейс Node определяет ряд общих для всех реализаций методов для работы с деревом:

short getNodeType() — возвращает тип объекта: элемент, атрибут, текст, CDATA и т.д.;

String getNodeValue() — возвращает значение Node;

Node getParentNode() — возвращает объект, являющийся родителем текущего узла Node;

NodeList getChildNodes() — возвращает список объектов, являющихся дочерними элементами;

NamedNodeMap getAttributes() — возвращает список атрибутов данного элемента.

Для получения информации о документе и изменения его структуры. Этот интерфейс представляет собой корневой элемент XML-документа и содержит методы доступа ко всему содержимому документа.

Element getDocumentElement() — возвращает корневой элемент документа.

org.w3c.dom.Element. Интерфейс предназначен для работы с элементом XML-документа и его содержимым. Некоторые методы:

String getTagName(String name) — возвращает имя элемента;

boolean hasAttribute() — проверяет наличие атрибутов;

String getAttribute(String name) — возвращает значение атрибута по его имени;

Attr getAttributeNode(String name) — возвращает атрибут по его имени;

NodeList getElementsByTagName(String name) — возвращает список дочерних элементов с определенным именем.

org.w3c.dom.Attr. Интерфейс служит для работы с атрибутами элемента XML-документа.

Некоторые методы интерфейса Attr:

String getName() — возвращает имя атрибута;

Element getOwnerElement() — возвращает элемент, который содержит этот атрибут;

String getValue() — возвращает значение атрибута;

boolean isId() — проверяет атрибут на тип ID.

Ниже приведен стандартный способ разбора документа students.xml с использованием DOM-анализатора и инициализация на его основе множества объектов.

Создание XML-документа. Документы можно не только читать, но также модифицировать и создавать совершенно новые. Для этого необходимо создать объекты классов Document, Element, добавить к последнему атрибуты и текстовое содержимое, после чего присоединить их к объекту, который в дереве XML-документа будет находиться выше.

StAX. StAX (Streaming API for XML), который еще называют pull-парсером, включен в JDK, начиная с версии Java 6. Он похож на SAX отсутствием объектной модели в памяти и последовательным продвижением по XML, но в StAX не требуется реализация интерфейсов, и приложение само указывает StAX парсеру перейти к следующему элементу XML. Кроме того, в отличие от SAX, данный парсер предлагает API для создания XML-документа.

Основными классами StAX являются XMLStreamReader и XMLStreamWriter, которые, соответственно, используются для чтения и создания XML-документа и расположены в пакете javax.xml.stream. Для чтения XML требуется получить ссылку на экземпляр XMLStreamReader:

StringReader input = new StringReader(filename);

или

InputStream input = new FileInputStream(new File(filename));

далее

XMLInputFactory inputFactory = XMLInputFactory.newInstance();

XMLStreamReader reader = inputFactory.createXMLStreamReader(input);

после чего по экземпляру XMLStreamReader можно организовать навигацию аналогично интерфейсу Iterator, используя методы hasNext() и next():

boolean hasNext() — показывает, есть ли еще элементы;

int next() — переходит к следующей вершине-константе XML, извлекая тип текущей.

При попытке вызова на константе несоответствующего ей метода генерируется исключительная ситуация IllegalStateException.

Чаще всего данные извлекаются с применением методов:

String getLocalName() — возвращает название тега (элемента) для текущей константы;

String getAttributeValue(String namespaceURI, String localName) — возвращает значение атрибута по имени;

String getAttributeValue(int index) — возвращает значение атрибута по номеру позиции;

String getText() — возвращает текст для констант CHARACTERS, CDATA, COMMENT и др.

Возможные типы вершин и методы, применимые к ним (см. таблицу ниже).

Типы вершин-констант Методы

Для всех типов констант getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(), getLocation(), hasText(), hasName()

START_ELEMENT next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()

ATTRIBUTE next(), nextTag() getAttributeXXX(), isAttributeSpecified(),

NAMESPACE next(), nextTag() getNamespaceXXX()

END_ELEMENT next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()

CHARACTERS, CDATA, COMMENT, SPACE next(), getTextXXX(), nextTag()
START_DOCUMENT next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT close()
PROCESSING_INSTRUCTION next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE next(), getLocalName(), getText(), nextTag()

StAX API более высокого уровня, базирующийся на итераторах, позволяет приложению обрабатывать XML как серию объектов событий, каждый из которых содержит определенную часть структуры XML-документа. Приложению требуется определить тип анализируемого события и использовать его методы для получения информации, связанной с ним.

Основными интерфейсами этого парсера StAX являются XMLEventReader и XMLEventWriter, которые, соответственно, используются для чтения и создания XML-документа и расположены в пакете javax.xml.stream. Для чтения XML требуется создать экземпляр XMLEventReader:

InputStream input = new FileInputStream(new File(filename)); далее
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLEventReader reader = inputFactory.createXMLEventReader(input);

после чего по экземпляру XMLEventReader можно организовать навигацию аналогично интерфейсу Iterator, используя методы hasNext() и nextEvent():

boolean hasNext() — показывает, есть ли еще элементы;
XMLEvent nextEvent() — переходит к следующему событию XML, извлекая текущее.

У интерфейса XMLEvent используются методы isStartElement() и isEndElement() для определения является ли тег начальным или конечным.

Метод asCharacters() извлекает информацию из тела элемента.

Методы getName() и getAttributeByName(QName qName) интерфейса StartElement позволяют получить имя элемента и значение его атрибутов.

Схема XSD представляет собой руководство по созданию и валидации XML-документа. XSD-схема является XML-документом, и поэтому она отличается гибкостью при использовании в приложениях, при задании правил документа, а также для дальнейшего расширения новой функциональностью. Схема содержит 44 базовых типа и имеет поддержку пространств имен (namespace). С помощью схемы XSD можно также проверить документ на валидность.

Схема XSD первой строкой содержит XML-декларацию. Любая схема своим корневым элементом должна содержать элемент schema.

В схеме нужно описать все элементы: их тип, количество повторений, дочерние элементы. Сам элемент создается элементом element, который может включать следующие атрибуты: name — определяет имя элемента; type — указывает тип элемента; ref — ссылается на определение элемента, находящегося в другом месте; minOccurs и maxOccurs — количество повторений этого элемента (по умолчанию принимает значение 1), чтобы указать, что количество элементов не ограничено, в атрибуте maxOccurs необходимо задать unbounded.

Если стандартных типов не хватает для полноты описания элемента, то можно создать свой собственный тип элемента. Типы элементов делятся на простые и сложные. Различия заключаются в том, что сложные типы могут содержать другие элементы, а простые — нет.

9. Работа с JSON документами

SON (JavaScript Object Notation) - текстовый формат обмена данными, легко читаемый людьми и основанный на синтаксисе Javascript. JSON, как правило, используется с Javascript, если быть более точным - при обмене данными между Javascript и сервером. Стоит отметить, что JSON обладает существенным преимуществом перед XML - он менее избыточен.

Чтобы использовать JSON в Java, мы должны использовать библиотеку json.simple для кодирования и декодирования. Для выполнения программы JSON и установки пути к классу необходимо установить jar (Java-архив) json.simple. Структуры данных, используемые в JSON:

JSON-объекты;

JSON-массивы.

Чтобы прочитать файл JSON в Java, необходимо использовать метод FileReader(). используем библиотеку json.simple.

```
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.*;
public class JSON{
        public static void main(Strings args[]){
                // file name is File.json
                Object o = new JSONParser().parse(new FileReader(File.json));
                JSONObject j = (JSONObject) o;
                String Name = (String) j.get("Name");
                String College = (String ) j.get("College");
                System.out.println("Name :" + Name);
                System.out.println("College :" +College);
}}
```

10. Работа с многопоточными приложениями

К большинству современных распределенных приложений (Rich Client) и веб-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов.

Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существует три способа создания и запуска потока: на основе расширения класса Thread, реализации интерфейсов Runnable или Callable.

При реализации интерфейса Runnable необходимо определить его единственный абстрактный метод run(). Запуск двух потоков для объектов классов WalkThread непосредственно и TalkThread через инициализацию экземпляра Thread приводит к выводу строк: Walk n и Talk -->n. Порядок вывода, как правило, различен при нескольких запусках приложения.

В конце работы каждого потока происходит вызов Thread.currentThread(). getName(), обеспечивающий вывод на консоль имени потока, в котором произошел вызов. В данном случае это будут строки Thread-1 и Thread-2. Имена даются потокам по умолчанию либо с помощью метода setName(String name) или конструктора потока. Статический метод currentThread() дает доступ к потоку, в котором он вызван.

Интерфейс Runnable не имеет метода start(), а только единственный метод run(). Поэтому для запуска такого потока, как TalkThread, следует создать экземпляр класса Thread с передачей экземпляра TalkThread его конструктору. Однако при прямом вызове метода run() поток не запустится, выполнится только тело самого метода.

При выполнении программы объект класса Thread может быть в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный». При создании потока он получает состояние «новый» (NEW) и не выполняется. Для перевода

потока из состояния «новый» в состояние «работоспособный» (RUNNABLE) следует выполнить метод start(), который вызывает метод run() — основной метод потока.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного класса-перечисления Thread.State:

NEW — поток создан, но еще не запущен;

RUNNABLE — поток выполняется;

BLOCKED — поток блокирован;

WAITING — поток ждет окончания работы другого потока;

TIMED_WAITING — поток некоторое время ждет окончания другого потока или просто в ожидании истечения времени;

TERMINATED — поток завершен.

Получить текущее значение состояния потока можно вызовом метода getState(). Поток переходит в состояние «неработоспособный» в режиме ожидания (WAITING) вызовом методов join(), wait() или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (TIMED_WAITING) с помощью методов sleep(long millis), join(long timeout) и wait(long timeout). Вернуть потоку работоспособность после вызова метода suspend() можно методом resume() (deprecated-метод), а после вызова метода wait() — методами notify() или notifyAll().

Когда поток просыпается, ему необходимо изменить состояние монитора объекта, на котором проходило ожидание. Для этого поток переходит в состояние BLOCKED и только после этого возвращается в работоспособное состояние.

Поток переходит в «пассивное» состояние (TERMINATED), если вызваны методы interrupt(), stop() (deprecated-метод) или метод run() завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод interrupt() успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток в этот момент неработоспособен, например, находится в состоянии TIMED_WAITING, то метод инициирует исключение InterruptedException. Чтобы это не происходило, следует предварительно вызвать метод isInterrupted(), который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

Перечисление TimeUnit представляет различные единицы измерения времени. В TimeUnit реализован ряд методов по преобразованию между единицами измерения и по управлению операциями ожидания в потоках в этих единицах. Используется для информирования методов, работающих со временем, о том, как интерпретировать заданный параметр времени.

Перечисление TimeUnit может представлять время в семи размерностях-значениях: NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS.

Кроме методов преобразования единиц времени, представляют интерес методы управления потоками:

void timedWait(Object obj, long timeout) — выполняет метод wait(long time) для объекта obj класса Object, используя заданные единицы измерения;

void timedJoin(Thread thread, long timeout) — выполняет метод join(long time) на потоке thread, используя заданные единицы измерения;

void sleep(long timeout) — выполняет метод sleep(long time) класса Thread, используя заданные единицы измерения.

В альтернативной системе управления потоками разработан механизм исполнителей, функции которого заключаются в запуске отдельных потоков и их групп, а также в управлении ими: принудительной остановке, контроле числа работающих потоков и планирования их запуска.

Интерфейс Callable<V> представляет поток, возвращающий значение вызывающему потоку. Определяет один метод V call() throws Exception, в код реализации которого и следует поместить решаемую задачу. Результат выполнения метода V call() может быть получен после окончания работы через экземпляр класса Future<V>, методами V get() или V get(long timeout, TimeUnit unit). Эти методы останавливают выполнение потока, в котором они вызваны, поэтому вызывать их следует в момент, когда закончится выполнение потока Callable.

Определить этот интервал затруднительно, и вместо ускорения работы приложения можно получить обратный результат. Перед извлечением результатов работы потока Callable можно проверить, завершилась ли задача успешно или была отменена, методами isDone() и isCancelled() соответственно.

Пусть стоит задача посчитать сумму значений элементов целочисленного списка с помощью интерфейса Callable. Решить эту задачу с помощью Stream API очень просто:

```
List<Integer> listInt = List.of(1, 10, 100, 1_000, 10_000);
int sum1 = listInt.stream().mapToInt(x -> x).sum();
// or
int sum2 = listInt.stream().reduce(0, (x, y) -> x + y);
System.out.println(sum1 + " " + sum2);//output: 11111 11111
```

Но если список состоит из сотен миллиардов элементов, то решать такую задачу следует с применением потоков.

В Java 5 добавлен механизм управления заданиями, основанный на возможностях интерфейса Executor и его наследников ExecutorService, ScheduledExecutorService, включающих организацию запуска потоков и их групп, а также способы их планирования, управления, отслеживания прогресса и завершения асинхронных задач. Все эти задачи можно было сделать и раньше, но это выполнялось либо самим программистом, либо с привлечением сторонних библиотек. Поэтому был определен наиболее распространенный набор задач и реализован в возможностях ExecutorService.

Класс ExecutorService методом execute(Runnable task) запускает традиционные потоки, методы же submit(Callable<T> task) и submit(Runnable task) запускают потоки как с возвращаемым значением, так и классические. Несколько потоков можно запустить методом invokeAll(Collection<? extends Callable<T>> tasks). Метод shutdown() прекращает действие самого исполнителя после того, как все запущенные им ранее потоки отработают, и не даст запустить новые, сгенерировав при этом исключение RejectedExecutionException. Метод shutdownNow() останавливает работу сервиса и удаляет все запущенные на объекте ExecutorService задачи-потоки.

При использовании ExecutorService метод shutdown() обязателен к вызову, иначе приложение не завершит свою работу.

Вызов метода boolean awaitTermination(long timeout, TimeUnit unit) останавливает поток, в котором вызван, и по истечении времени возвращает true, если все потоки, запущенные объектом ExecutorService, завершили свою работу, или false — если нет. Статические методы класса Executors определяют правила запуска потоков: newSingleThreadExecutor() позволяет исполнителю запускать только один поток, newFixedThreadPool(int numThreads) — число потоков не более, чем указано в параметре numThreads, ставя другие потоки в очередь ожидания окончания уже запущенных потоков, newScheduledThreadPool() — запуск по расписанию.

Потоку можно назначить приоритет от 1 (константа Thread.MIN_PRIORITY) до 10 (Thread.MAX_PRIORITY) с помощью метода setPriority(int newPriority).

Получить значение приоритета потока можно с помощью метода getPriority().

Для успешной демонстрации работы с приоритетами в классах WalkThread и TalkThread, следует увеличить число итераций, например, с 7 до 777, а также убрать задержки по времени.

Поток с более высоким приоритетом в данном случае, как правило, монополизирует вывод на консоль.

Приостановить (задержать) выполнение потока можно с помощью метода static void sleep(int millis) класса Thread. Поток переходит в состояние TIMED_WAITING. Иной способ состоит в вызове метода static void yield(), который отдает квант времени другому потоку, не мешая самому потоку работать и предоставляя возможность виртуальной машине переместить его в конец очереди других потоков.

Метод join() блокирует работу потока, в котором он вызван до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания при обращении к методу join(long timeout). Такие же результаты позволяет получить замена метода join() класса Thread на метод timedJoin(Thread thread, long timeout) перечисления TimeUnit.

Вызов статического метода yield() для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, чтобы другие потоки могли выполнять свои действия. В некоторых ситуациях поток может отдавать квант времени самому себе или вообще ничего не делать. Например, в случае потока с высоким приоритетом после обработки части пакета данных, когда следующая еще не готова, стоит уступить часть времени другим потокам. Польза yield() и приоритетов потока в оптимизации выполнения и взаимодействия потоков может оказаться не просто сомнительной, но и просто отрицательной. Если требуется надежная остановка потока, то следует применить другой способ.

Потоки-демоны работают в фоновом режиме вместе с программой, но представляют собой функциональность, которая не является важной для основной логики программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения, а его деятельность заключается в косвенном обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода setDaemon(boolean value), вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет.

Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в объект (поток, файл и т.д.). Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова synchronized.

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора.

Монитор экземпляра может иметь только одного владельца. При попытке конкурирующего доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта, только после этого завладеть им и начать использование объекта-ресурса.

Каждый экземпляр любого класса имеет монитор. Final-методы wait(), wait(long inmillis), notify(), notifyAll() класса Object корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Статический synchronized метод захватывает монитор экземпляра класса Class, того класса, на котором он вызван. Существует в единственном экземпляре. Нестатический synchronized метод захватывает монитор экземпляра класса, на котором он вызван.

Механизм wait\notify, эти final-методы не могут переопределяться и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации IllegalMonitorStateException.

Метод wait(), вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект. Возвратить блокировку объекта потоку можно вызовом метода notify() для одного потока или notifyAll() для всех потоков. Если ожидающих потоков несколько, то после вызова метода notify() невозможно определить, какой поток из ожидающих потоков заблокирует

объект. Вызов может быть осуществлен только из другого потока, заблокировавшего, в свою очередь, тот же самый объект.

Синхронизация ресурса ключевым словом synchronized накладывает достаточно жесткие правила на освобождение этого ресурса.

Дополнительные гибкие реализации моделей синхронизации представляют:

• интерфейс Lock, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством интерфейса Condition;

• класс семафор ReentrantLock, добавляющий ранее не существующую функциональность по отказу от попытки блокировки объекта с возможностью многократного повторения запроса на блокировку и отказа от нее;

• класс ReentrantReadWriteLock позволяет изменять объект только одному потоку, а читать в это время — нескольким.

Интерфейс Lock расширяет возможности блокирующей синхронизации. Появилась возможность провести проверку возможности блокировки, установить время ожидания блокировки и определить условия ее прерывания. В том время как synchronized метод блокирует объект, на котором вызван, методы интерфейса Lock блокируют сам объект Lock.

Интерфейс также оптимизирует работу JVM с процессами конкурирования за освобождаемые ресурсы.

Интерфейс Lock представляет методы:

void lock() — получает блокировку экземпляра ReentrantLock. Если экземпляр блокирован другим потоком, то поток отключается и бездействует до освобождения экземпляра;

void unlock() — освобождает блокировку экземпляра Lock. Если текущий поток не является обладателем блокировки, генерируется исключение IllegalMonitorStateException.

При работе с блокировками ресурсов потоков может возникать ситуация deadlock. То есть потоки в своем взаимодействии остановились и никаким образом не могут продолжить свое выполнение.

Наглядный пример взаимной блокировки состоит в следующем: поток № 1 заблокировал объект A, поток № 2 заблокировал объект B. Далее поток № 1 пытается получить доступ к объекту B и блокируется, так как объект B уже занят потоком № 2. В свою очередь, поток № 2 пытается получить доступ к объекту A и блокируется, так как объект A уже занят потоком № 1. В итоге оба потока заблокировали друг друга.

Ситуации с необходимостью обмена объектами при их взаимном блокировании возникают, для решения разработан класс Exchanger, предоставляющий возможность безопасного обмена объектами, в том числе и синхронизированными. Функционал обмена представляет метод T exchange(T ob). Возвращаемый параметр метода — объект, который будет принят из другого потока, передаваемый параметр ob метода — собственный объект потока, который помещается в буфер обмена и будет отдан другому потоку. Процесс обмена завершится успешно только в случае, если два потока вызовут метод exchange() на одном и том же объекте Exchanger.

11. Базы данных. Работа с подключениями. Выборка данных

API JDBC (Java DataBase Connectivity) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В настоящий момент JDBC выделены три типа драйверов:

1. Драйвер, представляющий собой частично библиотеку Java, работающий через native библиотеки для взаимодействия с клиентом СУБД.

2. Драйвер только на основе Java, работающий по сетевому и независимому от СУБД протоколу, который, в свою очередь, подключается к клиенту СУБД.

3. Сетевой драйвер, состоящий только из библиотеки Java, работающий напрямую с клиентом СУБД.

Если приложение выполняется на сервере, который не предполагает установки клиента СУБД, то выбор производится между вторым и третьим типами. Причем третий тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер третьего типа будет более эффективным с точки зрения производительности.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных, для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Последовательность действий для выполнения первого запроса.

1. Подключение библиотеки с классом-драйвером базы данных.

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, поместив ее предварительно в папку /lib приложения или сервера приложений.

mysql-connector-java-[version]-bin.jar для СУБД MySQL, ojdbc[version].jar для СУБД Oracle.

2. Установка соединения с БД.

До появления JDBC 4.0 объект драйвера СУБД для консольных приложений нужно было регистрировать с помощью вызова:

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver()); // for MySQL
DriverManager.registerDriver(new oracle.jdbc.OracleDriver()); // for Oracle
```

или создавать явно

```
Class.forName("com.mysql.cj.jdbc.Driver");
Class.forName("oracle.jdbc.OracleDriver");
```

В настоящее время в большинстве случаев в этом нет необходимости, так как экземпляр драйвера загружается автоматически при попытке получения соединения с БД.

Для установки соединения с БД вызывается один из перегруженных статических методов getConnection() класса java.sql.DriverManager. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект java.sql.Connection. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов.

```
Connection connection = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/testphones","root", "pass");
Connection connection = DriverManager.getConnection(
"jdbc:oracle:thin:@//localhost:1521:testphones", "system", "pass");
```

В результате будет возвращен объект Connection и создано одно установленное соединение с БД, именуемой testphones. Класс DriverManager предоставляет средства для управления набором драйверов баз данных. С помощью метода getDrivers(), Stream<Driver> drivers() можно получить список всех доступных драйверов.

3. Создание объекта для передачи запросов.

После создания объекта Connection и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект java.sql.Statement, создаваемый вызовом метода createStatement() класса Connection.

```
Statement statement = connection.createStatement();
```

Объект класса, реализующего интерфейс Statement, используется для прямого выполнения SQL-запроса. Могут применяться также объекты классов PreparedStatement и CallableStatement для выполнения подготовленных запросов и хранимых процедур. Оба этих интерфейса наследуют возможности интерфейса Statement.

Метод createStatement(int resultSetType, int resultSetConcurrency) позволяет установить условия прокрутки и изменения объекта ResultSet.

Параметр resultSetType со значением ResultSet.TYPE_FORWARD_ONLY позволяет продвигаться по объекту только от начала к концу и выставляется по умолчанию. Значение ResultSet.TYPE_SCROLL_INTENSIVE разрешает навигацию в обе стороны и не учитывает изменения от других пользователей и учитывает изменения от других пользователей после того, как ResultSet был получен.

Значение ResultSet.CONCUR_READ_ONLY параметра resultSetConcurrency позволяет только читать результат и устанавливается по умолчанию. Значение ResultSet.CONCUR_UPDATABLE создает ResultSet с возможностью изменения данных.

4. Выполнение запроса.

Созданный объект Statement можно использовать для выполнения запросов SQL, передавая их в один из методов:

ResultSet executeQuery(String sql) — выполняет запросы SELECT.

Результаты выборки из базы помещаются в объект ResultSet:

int executeUpdate(String sql) — выполняет запросы, изменяющие состояние базы INSERT, UPDATE, DELETE. Возвращает количество строк, задействованных запросом;

boolean execute(String sql) — применяется для выполнения произвольных запросов;

int[] executeBatch() — выполняет batch-команды, т.е группу запросов, как один запрос

// extract all data from the phonebook table

ResultSet resultSet = statement.executeQuery("SELECT idphonebook, lastname, phone FROM phonebook");

5. Обработка результатов запроса на выборку данных производится методами интерфейса ResultSet, где самыми распространенными являются next(), first(), previous(), last(), beforeFirst(), afterLast(), isFirst(), isLast(), absolute(int i) — методы навигации по строкам таблицы результатов, группа методов по доступу к информации по номеру позиции в записи вида String getString(int pos), а также аналогичные методы, начинающиеся с getType(int pos) (int getInt(int pos), float getFloat(int pos) и др.) и updateType(). Среди них следует выделить методы getClob(int pos) и getBlob(int pos), позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами.

Следует обратить внимание, что счет позиций в ResultSet начинается с «1», а не с «0», как в коллекциях и массивах.

Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа int getInt(String columnLabel), String getString(String columnLabel), Object getObject(String columnLabel) и подобными им.

Обновляемый набор данных позволяет обновлять, изменять и ResultSet, и информацию в таблице базы данных: updateRow(), insertRow(), updateString() и др.

При первом вызове метода next() указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение false.

6. Закрытие соединения.

connection.close(); // closes also Statement & ResultSet

Когда база больше не нужна, соединение должно быть закрыто. Для того, чтобы правильно пользоваться приведенными методами, программисту как минимум требуется знать SQL, способ организации конкретной БД, типы полей БД и др.

7. Выгрузка драйверов.

По завершении работы приложения следует выгрузить или дерегистрировать драйвер:

```
DriverManager.getDrivers().asIterator().forEachRemaining(driver -> {
try {
DriverManager.deregisterDriver(driver);
} catch (SQLException e) {
// log }});
```

12. Базы данных. Изменение и редактирование баз данных

СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта www.mysql.com.

Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем root и паролем, например, pass. Если планируется разворачивать реально работающее приложение, стоит исключить тривиальных пользователей сервера БД, иначе злоумышленники могут получить полный доступ к БД. Для запуска следует использовать команду из папки /mysql/bin: mysqld -nt –standalone

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания базы данных и ее таблиц используются команды языка SQL.

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем testphones и одной таблицей PHONEBOOK. Таблица должна содержать три поля: числовое (первичный ключ) — IDPHONEBOOK, символьное — LASTNAME и числовое — PHONE и несколько занесенных записей.

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение любых символов.

В простом приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие СУБД. Объект ResultSet также нуждается в закрытии, но его автоматически закрывает метод close() интерфейса Statement при закрытии или механизм AutoCloseable.

Класс Abonent, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

```
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {}
public class Abonent extends Entity {
private int id;
private String name;
private int phone;
public Abonent() {}
public Abonent(int id, String name, int phone) {
this.id = id;
this.name = name;
this.phone = phone;}
public int getId() {
return id;}
public void setId(int id) {
this.id = id;}
public String getName() {
return name;}
public void setName(String name) {
this.name = name;}
public int getPhone() {
return phone;}
public void setPhone(int phone) {
this.phone = phone;}
public String toString() {
final StringBuilder sb = new StringBuilder("Abonent{");
sb.append("id=").append(id).append(", name='").append(name).append('\'');
sb.append(", phone=").append(phone).append('}');
return sb.toString();}}
```

Параметры соединения можно задавать: с помощью прямой передачи значений в коде класса, а также с помощью файлов properties, json, yaml или xml.

Окончательный выбор производится в зависимости от конфигурации проекта.

Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Пусть класс ConnectorCreator использует файл ресурсов database.properties, в котором хранятся, как правило, такие параметры подключения к БД, как логин, пароль доступа и др.

```
db.driver=com.mysql.cj.jdbc.Driver
user = root
password = pass
poolsize = 32
db.url = jdbc:mysql://localhost:3306/testphones
useUnicode = true
encoding = UTF-8
useSSL = true
useJDBCCompliantTimezoneShift = true
useLegacyDatetimeCode = false
serverTimezone = UTC
serverSslCert = classpath:server.crt
```

Код класса ConnectionCreator может выглядеть следующим образом:

```
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
public class ConnectionCreator {
private static final Properties properties = new Properties();
private static final String DATABASE_URL;
static {
try {
properties.load(new FileReader("datares\\database.properties"));
String driverName = (String) properties.get("db.driver");
Class.forName(driverName);
} catch (ClassNotFoundException | IOException e) {
e.printStackTrace(); // fatal exception}
DATABASE_URL = (String) properties.get("db.url");}
private ConnectionCreator() {}
public static Connection createConnection() throws SQLException {
return DriverManager.getConnection(DATABASE_URL, properties);}}
```

В таком случае получение соединения с БД сведется к вызову:

```
Connection connection = ConnectionCreator.createConnection();
```

Класс ConnectorCreator лучше сделать синглтоном.

Объект ResultSet позволяет вставлять запись в базу данных без дополнительных запросов. Объект Statement нужно создавать с разрешением на изменение в базе данных.

```
try (Connection connection = DriverManager.getConnection(url, prop);
Statement statement = connection.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
ResultSet resultSet = statement.executeQuery(
"SELECT idphonebook, lastname, phone FROM phonebook");
resultSet.moveToInsertRow(); // insert row
resultSet.updateInt(1, 77);
```

```
resultSet.updateString(2, "Bahdanovich");
resultSet.updateInt(3, 222322);
resultSet.insertRow();
resultSet.moveToCurrentRow();
} catch (SQLException e) {
e.printStackTrace();}}}
```

Изменения в базу данных также легко вносятся с помощью возможностей ResultSet:

```
while (resultSet.next()) {
int id = resultSet.getInt(1);
if (id == 2) {
resultSet.updateInt("phone", 550055); // update row
resultSet.updateRow();}}
```

В результате у записи с IDPHONEBOOK=2 номер телефона будет заменен как в ResultSet, так и в базе данных.

Существует целый ряд методов интерфейсов ResultSetMetaData и DatabaseMetaData для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для записей подобных методов нет.

Получить объект ResultSetMetaData можно следующим образом:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

Некоторые методы интерфейса ResultSetMetaData:

int getColumnCount() — возвращает число столбцов набора результатов объекта ResultSet;

String getColumnName(int column) — возвращает имя указанного столбца;

String getColumnTypeName(int column) — возвращает тип данных указанного столбца.

Если добавить следующий код к примеру

```
System.out.println("ColumnCount: " + rsMetaData.getColumnCount());
System.out.println("ColumnName: " + rsMetaData.getColumnName(1));
System.out.println("ColumnTypeName: " + rsMetaData.getColumnTypeName(1));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(1));
System.out.println("ColumnName: " + rsMetaData.getColumnName(2));
System.out.println("ColumnTypeName: " + rsMetaData.getColumnTypeName(2));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(2));
```

то в консоль будет выведено:

```
ColumnCount: 3
ColumnName: idphonebook
ColumnTypeName: INT
isAutoIncrement: true
ColumnName: lastname
ColumnTypeName: VARCHAR
isAutoIncrement: false
```

Получить объект DatabaseMetaData можно следующим образом:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

Некоторые методы обширного интерфейса DatabaseMetaData:

String getDatabaseProductName() — возвращает название СУБД;

String getDatabaseProductVersion() — номер версии СУБД;

String getDriverName() — имя драйвера JDBC;

String getUserName() — имя пользователя БД;

String getURL() — местонахождение источника данных;

ResultSet getTables() — набор типов таблиц, доступных для данной БД.

Если добавить следующий код

```
System.out.println("DatabaseName: " + dbMetaData.getDatabaseProductName());
System.out.println("DatabaseVersion: " + dbMetaData.getDatabaseProductVersion());
```

System.out.println("UserName: " + dbMetaData.getUserName());
System.out.println("URL: " + dbMetaData.getURL());
то в консоль будет выведено:
DatabaseName: MySQL
DatabaseVersion: 5.7.15-log
UserName: root@localhost
URL: jdbc:mysql://localhost:3306/testphones

Для представления запросов существует еще два типа объектов PreparedStatement и CallableStatement. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании PreparedStatement невозможен sql injection attacks. То есть, если существует возможность прямой передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект PreparedStatement.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод prepareStatement(String sql) интерфейса Connection, возвращающий объект PreparedStatement.

String sql = "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)";
PreparedStatement statement = connection.prepareStatement(sql);

Установка входных значений конкретных параметров этого объекта производится с помощью методов setString(int index, String x), setInt(int index, int x) и подобных им, после чего и осуществляется непосредственное выполнение запроса методами int executeUpdate(), ResultSet executeQuery().

```
import java.sql.*;
public class PreparedMain {
public static void main(String[] args) {
try(Connection connection = ConnectionCreator.createConnection()){
String sql
= "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES (?, ?, ?)";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setInt(1, 43);
statement.setString(2, "Skaryna");
statement.setInt(3, 990077);
int rowsUpdate = statement.executeUpdate();
System.out.println(rowsUpdate);
} catch (SQLException e) {
e.printStackTrace();
}}}
```

Так как данный оператор предварительно подготовлен, то он выполняется быстрее обычных операторов, ему соответствующих. Оценить преимущества во времени можно, выполнив большое число повторяемых запросов с предварительной подготовкой запроса и без нее, и сравнив время выполнения.

Как известно, таблицы базы данных обычно содержат столбец, помеченный как первичный ключ, значение которого уникально у каждой записи таблицы. Для первичных ключей разработаны различные механизмы автогенерации уникального значения. При добавлении записи в базу данных разработчику можно не знать об этих правилах, и тогда запрос на добавление данных не должен содержать информации о первичном ключе. Значение первичного ключа для этой записи будет сгенерировано базой данных автоматически.

Значение же сгенерированного ключа может понадобиться сразу же. Выполнять для его получения запрос SELECT неэкономично. Метод getGeneratedKeys() возвратит значение ключа. Объект PreparedStatement должен быть создан с параметром Statement.RETURN_GENERATED_KEYS.

```
String sql = "INSERT INTO phonebook(lastname, phone) VALUES (?, ?)";
PreparedStatement statement = connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
statement.setString(1, "Kalinouski");
statement.setInt(2, 186300);
statement.executeUpdate();
ResultSet resultSet = statement.getGeneratedKeys();
if(resultSet.next()) {
int key = resultSet.getInt(1);
System.out.println(key);
}
```

Механизм автогенерации также удобен во избежание ошибки дублирования ключа. Пользователь может пытаться добавить запись с уже существующим в базе данных значением. Или возможна неточность порядка значения. Например, все значения ключей в базе двадцатизначные, а пользователь добавил двузначное значение. Ошибки это не вызовет, но внешне такие ключи будут выглядеть несогласованно.

Интерфейс CallableStatement имеет более широкие возможности, чем интерфейс PreparedStatement, поэтому обеспечивает выполнение хранимых процедур.

Хранимая процедура — это, в общем случае, именованная последовательность команд SQL, рассматриваемых как единое целое, и выполняющаяся в адресном пространстве процессов СУБД, которую можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных. Для создания объекта CallableStatement вызывается метод prepareCall() объекта Connection.

Интерфейс CallableStatement позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса в том, что CallableStatement способен обрабатывать не только входные (IN) параметры, но выходящие (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован с помощью метода registerOutParameter(). После установки входных и выходных параметров вызываются методы execute(), executeQuery() или executeUpdate().

Пусть в СУБД MySQL существует хранимая процедура findlastname, которая по уникальному номеру телефона для каждой записи в таблице phonebook будет возвращать соответствующее ему имя:

```
CREATE DEFINER=`root`@`localhost`
PROCEDURE `findlastname`(IN p_phone INT, OUT p_lastname VARCHAR(40))
BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код вида:

```
final String SQL = "{call findlastname (?, ?)}";
CallableStatement statement = connection.prepareCall(SQL);
statement.setInt(1, 654321);
statement.registerOutParameter(2, java.sql.Types.VARCHAR);
statement.execute();
String lastName = statement.getString(2);
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL, как одну единицу.

```
import java.sql.*;
import java.util.Arrays;
public class BatchMain {
public static void main(String[] args) {
Connection connection = null;
try {
connection = ConnectionCreator.createConnection();
connection.setAutoCommit(false); // turn off autocommit
Statement statement = connection.createStatement();
statement.addBatch("INSERT INTO phonebook VALUES (92, 'Sapega', 112211)");
statement.addBatch("INSERT INTO location VALUES (201, 'Minsk')");
statement.addBatch("INSERT INTO location VALUES (202, 'Lviv')");
// submit a batch of update commands for execution
int[] updateCounts = statement.executeBatch();
connection.commit();
System.out.println(Arrays.toString(updateCounts));
} catch (SQLException e) {
try {
if(connection != null) {
connection.rollback();}
} catch (SQLException ex) {
ex.printStackTrace();}
} finally {
try {
if(connection != null) { // turn on autocommit
connection.setAutoCommit(true);}
} catch (SQLException e) {
e.printStackTrace();
}try {
if(connection != null) {
connection.close();
}} catch (SQLException e) {
e.printStackTrace();}}}}
```

Чтобы запустить это приложение, необходимо в базу данных testphones добавить таблицу location со столбцами idcity и city.

Удалить одну из команд нельзя, можно лишь очистить полностью методом clearBatch().

Если используется объект PreparedStatement, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод executeBatch() интерфейса PreparedStatement возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует список объектов типа Abonent со стандартным набором геттеров и сеттеров для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов execute() или executeUpdate() становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
List<Abonent> abonents = new ArrayList<>();
// filling abonents
```

```
PreparedStatement statement = connection.prepareStatement("INSERT INTO phonebook VALUES(?,?,?)");
for (Abonent abonent : abonents) {
statement.setInt(1, abonent.getId());
statement.setString(2, abonent.getName());
statement.setInt(3, abonent.getPhone());
statement.addBatch();}
int[] updateCounts = statement.executeBatch();
} catch (SQLException throwables) {
throwables.printStackTrace();}
```

При проектировании информационных систем возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к утрате информации или к финансовым потерям. Простейшим примером может служить ситуация с перечислением денег с одного счета на другой. Если сбой произошел в тот момент, когда операция снятия денег с одного счета уже произведена, а операция зачисления на другой счет еще не произведена, то система, допускающая такие ситуации, должна быть признана не отвечающей требованиям заказчика. Или должны выполняться обе операции, или не выполняться вовсе. Такие две операции трактуют как одну и называют транзакцией.

Транзакция, или деловая операция, определяется как единица работы, обладающая свойствами ACID:

– Атомарность — две или более операций, выполняются все или не выполняется ни одна.

– Согласованность — при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.

– Изолированность — во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.

– Долговечность — все изменения, произведенные с данными во время транзакции, обязательно сохраняются, например, в базе данных, что позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор COMMIT. В API JDBC эта операция выполняется по умолчанию после каждого вызова методов executeQuery() и executeUpdate(). Если же необходимо сгруппировать запросы и только после этого выполнить операцию COMMIT, сначала вызывается метод setAutoCommit(boolean param) интерфейса Connection с параметром false, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций. После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция COMMIT не будет выполнена непосредственно. Подтверждает выполнение SQL-запросов метод commit() интерфейса Connection, в результате действия которого все изменения таблицы производятся как одно логическое действие. Если же транзакция не выполнена, то методом rollback() отменяются действия всех запросов SQL, начиная от последнего вызова commit().

В следующем примере информация добавляется в таблицу в режиме действия транзакции, подтвердить или отменить действия которой можно, снимая или добавляя комментарий в строках вызова методов commit() и rollback().

Схематически транзакцию можно представить в виде:

```
Connection connection = null;
try { connection = / take connection /
connection.setAutoCommit(false); // 1
Statement statement = connection.createStatement();
statement.executeUpdate("some update/insert/delete/select query_1"); // 2
statement.executeUpdate("some update/insert/delete/select query_2"); // 2
```

```
connection.commit(); // 3a
} catch (SQLException e) {
connection.rollback(); // 3b
//log or throw
} finally {
connection.setAutoCommit(true); // 4}
```

Цифрами обозначена последовательность действий.

Для транзакций существует несколько типов чтения:

– грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена.

– Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

– неповторяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

– фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса Connection (по возрастанию уровня ограничения):

– TRANSACTION_NONE — информирует о том, что драйвер не поддерживает транзакции;

– TRANSACTION_READ_UNCOMMITTED — позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, неповторяющееся и фантомное чтения;

– TRANSACTION_READ_COMMITTED — означает, что любое изменение, сделанное в транзакции, не видно вне ее, пока она не сохранена, что предотвращает грязное чтение, но разрешает неповторяющееся и фантомное;

– TRANSACTION_REPEATABLE_READ — запрещает грязное и неповторяющееся чтение, но фантомное разрешено;

– TRANSACTION_SERIALIZABLE — определяет, что грязное, неповторяющееся и фантомное чтения запрещены.

Метод boolean supportsTransactionIsolationLevel(int level) интерфейса DatabaseMetaData определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса Connection определяют доступ к уровню изоляции:

int getTransactionIsolation() — возвращает текущий уровень изоляции;

void setTransactionIsolation(int level) — устанавливает необходимый уровень.

Точки сохранения, представляемые классом java.sql.Savepoint, дают дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции. Фактически это транзакция внутри транзакции. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных.

Подтвердить Savepoint невозможно, все точки сохранения автоматически подтверждаются с подтверждением всей транзакции. Таким образом, если произойдет ошибка, можно вызвать метод rollback(Savepoint point) для отмены всех изменений, которые были сделаны после точки сохранения. Метод boolean supportsSavepoints() интерфейса DatabaseMetaData используется для того, чтобы определить, поддерживают ли точки

сохранения сама СУБД и ее драйвер JDBC. Методы setSavepoint(String name) и setSavepoint() возвращают объект Savepoint интерфейса Connection и используются для установки именованной или неименованной точки сохранения во время текущей транзакции.

При этом новая транзакция будет начата, если в момент вызова setSavepoint() не будет активной транзакции.

Savepoint используется при сложном взаимодействии с базой данных в пределах одного логического действия операции. Например, покупка авиабилета.

На первой странице заполняется форма с данными покупателя, на второй — условия багажа и страховки, на третьей — внесение информации о банковской карточке. Если же пользователь решил вернуться на шаг назад, то с применением Savepoint можно отменить только этот шаг, не отменяя все действия целиком.

13. Шаблоны проектирования GRASP

GRASP (англ. general responsibility assignment software patterns — общие шаблоны распределения ответственностей; также существует английское слово "grasp" — «контроль, хватка») — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственностей классам и объектам.

Описано 9 таких шаблонов: каждый помогает решить некоторую проблему, возникающую как в объектно-ориентированном анализе, так и в практически любом проекте по разработке программного обеспечения. Таким образом, шаблоны «G.R.A.S.P.» — хорошо документированные, стандартизированные и проверенные временем принципы объектно-ориентированного анализа, а не попытка привнести что-то принципиально новое.

Краткая характеристика девяти шаблонов:

1. Информационный эксперт (Information Expert)

Шаблон определяет базовый принцип распределения ответственностей:

Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту.

Этот шаблон — самый очевидный и важный из девяти. Если его не учесть — получится спагетти-код, в котором трудно разобраться.

Локализация же ответственностей, проводимая согласно шаблону:

Повышает инкапсуляцию, простоту восприятия, готовность компонентов к повторному использованию, но снижает степень зацепления.

2. Создатель (Creator)

Проблема: Кто отвечает за создание объекта некоторого класса A?

Решение: Назначить классу B обязанность создавать объекты класса A, если класс B:

содержит(contains) или агрегирует(aggregate) объекты A;

записывает(records) объекты A;

активно использует объекты A;

обладает данными для инициализации объектов A

Можно сказать, что шаблон «Creator» - это интерпретация шаблона «Information Expert» (смотрите пункт № 1) в контексте создания объектов.

Большинство порождающих шаблонов проектирования так или иначе выводятся или опираются на шаблон «Creator».

3. Контроллер (Controller)

Отвечает за операции, запросы которых приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление);

Не выполняет работу самостоятельно, а делегирует компетентным исполнителям;

Может представлять собой:

Систему в целом;

Подсистему;

Корневой объект;

Устройство.

4. Низкое зацепление (Low Coupling)

См. также: Зацепление (программирование)

«Степень зацепления» — мера неотрывности элемента от других элементов (либо мера данных, имеющихся у него о них).

«Слабое» зацепление является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать.

«Слабое» зацепление — распределение ответственностей и данных, обеспечивающее взаимную независимость классов. Класс со «слабым» зацеплением:

Имеет слабую зависимость от других классов;

Не зависит от внешних изменений (изменение в одном классе оказывает слабое влияние на другие классы);

Прост для повторного использования.

5. Высокая связность (High Cohesion)

См. также: Связность (программирование)

Высокая связность класса — это оценочная модель, направленная на удержание объектов должным образом сфокусированными, управляемыми и понятными. Высокая связность обычно используется для поддержания низкого зацепления. Высокая связность означает, что обязанности данного элемента тесно связаны и сфокусированы. Разбиение программ на классы и подсистемы является примером деятельности, которая увеличивает связность системы.

И наоборот, низкая связность — это ситуация, при которой данный элемент имеет слишком много несвязанных обязанностей. Элементы с низкой связностью часто страдают от того, что их трудно понять, трудно использовать, трудно поддерживать.

Связность класса — мера сфокусированности предметных областей его методов:

«Высокая» связность — сфокусированные подсистемы (предметная область определена, управляема и понятна);

«Низкая» связность — абстрактные подсистемы, затруднены:

Восприятие;

Повторное использование;

Поддержка;

Устойчивость к внешним изменениям.

6. Полиморфизм (Polymorphism)

См. также: Полиморфизм (информатика)

Устройство и поведение системы:

Определяется данными;

Задано полиморфными операциями её интерфейса.

Пример: Адаптация коммерческой системы к многообразию систем учёта налогов может быть обеспечена через внешний интерфейс объектов-адаптеров (см. также: Шаблон «Адаптеры»).

7. Чистая выдумка (Pure Fabrication)

Не относится к предметной области, но:

Уменьшает зацепление;

Повышает связность;

Упрощает повторное использование.

«Pure Fabrication» отражает концепцию сервисов в модели предметно-ориентированного проектирования.

8. Перенаправление (Indirection)

См. также: Посредник (шаблон проектирования)

Слабое зацепление между элементами системы (и возможность повторного использования) обеспечивается назначением промежуточного объекта их посредником.

Пример: В архитектуре Model-View-Controller, контроллер (англ. controller) ослабляет зацепление данных (англ. model) с их представлением (англ. view).

9. Устойчивость к изменениям (Protected Variations)

Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами. Поведение может варьироваться лишь через создание другой реализации интерфейса.

14. Шаблоны проектирования GoF

Шаблон проектирования (паттерн, от англ. design pattern) — повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные.

На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы.

Алгоритмы по своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи.

История

В 1970-е годы архитектор Кристофер Александр составил набор шаблонов проектирования. В области архитектуры эта идея не получила такого развития, как позже в области программной разработки.

В 1987 году Кент Бэк (Kent Beck) и Вард Каннингем (Ward Cunningham) взяли идеи Александра и разработали шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке Smalltalk.

В 1988 году Эрих Гамма (Erich Gamma) начал писать докторскую диссертацию при Цюрихском университете об общей переносимости этой методики на разработку программ.

В 1989—1991 годах Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на C++ и опубликовал в 1991 году книгу «Advanced C++ Idioms».

В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides) публикует книгу Design Patterns — Elements of Reusable Object-Oriented Software. В этой книге описаны 23 шаблона проектирования. Также команда авторов этой книги известна общественности под названием «Банда четырёх» (англ. Gang of Four, часто сокращается до GoF). Именно эта книга стала причиной роста популярности шаблонов проектирования.

Плюсы

В сравнении с полностью самостоятельным проектированием, шаблоны обладают рядом преимуществ. Основная польза от использования шаблонов состоит в снижении сложности разработки за счёт готовых абстракций для решения целого класса проблем. Шаблон даёт решению своё имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация деталей решений: модулей, элементов проекта, — снижается количество ошибок.

Применение шаблонов концептуально сродни использованию готовых библиотек кода. Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова. Набор шаблонов помогает разработчику выбрать возможный, наиболее подходящий вариант проектирования.]

Минусы

Хотя легкое изменение кода под известный шаблон может упростить понимание кода, по мнению Стива Макконнелла, с применением шаблонов могут быть связаны две сложности. Во-первых, слепое следование некоторому выбранному шаблону может привести к усложнению программы. Во-вторых, у разработчика может возникнуть желание попробовать некоторый шаблон в деле без особых оснований (см Золотой молоток).]

Многие шаблоны проектирования в объектно-ориентированном проектировании можно рассматривать как идиоматическое воспроизведение элементов функциональных языков. Питер Норвиг утверждает, что 16 из 23 шаблонов, описанных в книге «Банды четырёх», в динамически-типизируемых языках реализуются существенно проще, чем в C++, либо оказываются незаметны. Пол Грэхэм считает саму идею шаблонов проектирования — антипаттерном, сигналом о том, что система не обладает достаточным уровнем абстракции, и необходима её тщательная переработка. Нетрудно видеть, что само определение шаблона как «готового решения, но не прямого обращения к библиотеке» по сути означает отказ от повторного использования в пользу дублирования. Это, очевидно, может быть неизбежным для сложных систем при использовании языков, не поддерживающих комбинаторы и полиморфизм типов, и это в принципе может быть исключено в языках, обладающих свойством гомоиконичности (хотя и не обязательно эффективно), так как любой шаблон может быть реализован в виде исполнимого кода.

15. Порождающие и структурные шаблоны проектирования

Шаблоны бывают следующих трех видов:

— Порождающие.

— Структурные.

— Поведенческие.

Если говорить простыми словами, то это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов.

Порождающие шаблоны — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Существуют следующие порождающие шаблоны:

- простая фабрика (Simple Factory);
- фабричный метод (Factory Method);
- абстрактная фабрика (Abstract Factory);
- строитель (Builder);
- прототип (Prototype);
- одиночка (Singleton).

Простая фабрика (Simple Factory). В объектно-ориентированном программировании (ООП), фабрика — это объект для создания других объектов. Формально фабрика — это функция или метод, который возвращает объекты изменяющегося прототипа или класса из некоторого вызова метода, который считается «новым».

Пример из жизни: Представьте, что вам надо построить дом, и вам нужны двери. Было бы глупо каждый раз, когда вам нужны двери, надевать вашу столярную форму и начинать делать дверь. Вместо этого вы делаете её на фабрике.

Простыми словами: Простая фабрика генерирует экземпляр для клиента, не раскрывая никакой логики.

Когда использовать: Когда создание объекта — это не просто несколько присвоений, а какая-то логика, тогда имеет смысл создать отдельную фабрику вместо повторения одного и того же кода повсюду.

Фабричный метод (Fabric Method) — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Пример из жизни: Рассмотрим пример с менеджером по найму. Невозможно одному человеку провести собеседования со всеми кандидатами на все вакансии. В зависимости от вакансии он должен распределить этапы собеседования между разными людьми.

Простыми словами: Менеджер предоставляет способ делегирования логики создания экземпляра дочерним классам.

Когда использовать: Полезен, когда есть некоторая общая обработка в классе, но необходимый подкласс динамически определяется во время выполнения. Иными словами, когда клиент не знает, какой именно подкласс ему может понадобиться.

Абстрактная фабрика — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Пример из жизни: Расширим наш пример про двери из простой фабрики. В зависимости от ваших нужд вам понадобится деревянная дверь из одного магазина, железная дверь — из другого или пластиковая — из третьего. Кроме того, вам понадобится соответствующий специалист: столяр для деревянной двери, сварщик для железной двери и так далее. Как вы можете заметить, тут есть зависимость между дверьми.

Простыми словами: Фабрика фабрик. Фабрика, которая группирует индивидуальные, но связанные/зависимые фабрики без указания их конкретных классов.

Когда использовать: Когда есть взаимосвязанные зависимости с не очень простой логикой создания.

Строитель — порождающий шаблон проектирования, который предоставляет способ создания составного объекта. Предназначен для решения проблемы антипаттерна «Телескопический конструктор».

Пример из жизни: Представьте, что вы пришли в McDonalds и заказали конкретный продукт, например, БигМак, и вам готовят его без лишних вопросов. Это пример простой фабрики. Но есть случаи, когда логика создания может включать в себя больше шагов. Например, вы хотите индивидуальный сэндвич в Subway: у вас есть несколько вариантов того, как он будет сделан. Какой хлеб вы хотите? Какие соусы использовать? Какой сыр? В таких случаях на помощь приходит шаблон «Строитель».

Простыми словами: Шаблон позволяет вам создавать различные виды объекта, избегая засорения конструктора. Он полезен, когда может быть несколько видов объекта или когда необходимо множество шагов, связанных с его созданием.

Когда использовать: Когда может быть несколько видов объекта и надо избежать «телескопического конструктора». Главное отличие от «фабрики» — это то, что она используется, когда создание занимает один шаг, а «строитель» применяется при множестве шагов.

Прототип (Prototype) задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Он позволяет уйти от реализации и позволяет следовать принципу «программирование через интерфейсы». В

качестве возвращающего типа указывается интерфейс / абстрактный класс на вершине иерархии, а классы-наследники могут подставить туда наследника, реализующего этот тип.

Пример из жизни: Помните Долли? Овечка, которая была клонирована. Не будем углубляться, главное — это то, что здесь все вращается вокруг клонирования.

Простыми словами: Прототип создает объект, основанный на существующем объекте при помощи клонирования.

Когда использовать: Когда необходим объект, похожий на существующий объект, либо когда создание будет дороже клонирования.

Одиночка — порождающий шаблон проектирования, гарантирующий, что в однопроцессном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

Пример из жизни: В стране одновременно может быть только один президент. Один и тот же президент должен действовать, когда того требуют обстоятельства. Президент здесь является одиночкой.

Простыми словами: Обеспечивает тот факт, что создаваемый объект является единственным объектом своего класса.

Пример презентационного материала для проведения занятия

Рабочая среда Java

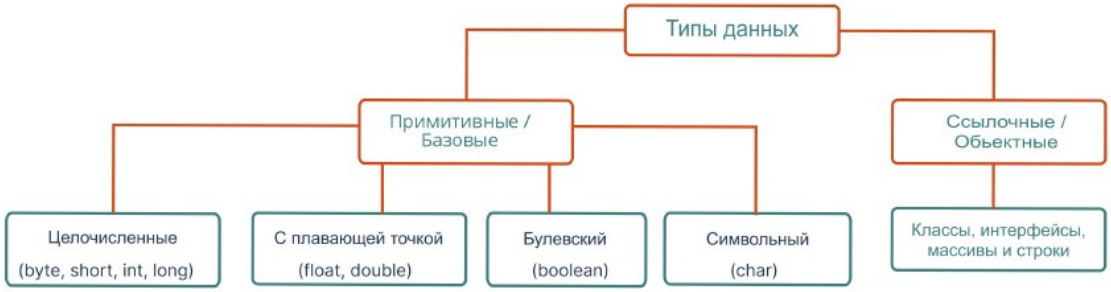
Платформы

- ***Платформа Java (Java Platform)*** – программная среда, в которой работают приложения Java
- Существуют версии платформы Java для различных ОС (Windows, Linux, Solaris, Mac OS)
- Включает в свой состав:
  - *Java Virtual Machine (JVM) –виртуальная машина Java* –программа, интерпретирующая приложения Java
  - *Java API* - библиотека программных компонентов (классов и интерфейсов), реализующих стандартный функционал

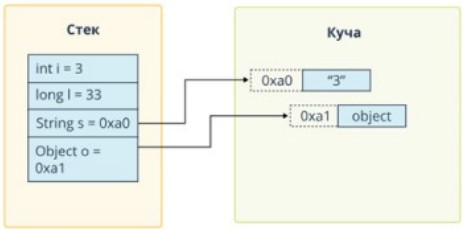
- ***Java Platform, Standard Edition (Java SE)*** – платформа широкого назначения для рабочих станций
- ***Java Platform, Enterprise Edition (Java EE)*** – платформа для корпоративных приложений и приложений интернет
- ***Java Platform, Micro Edition (Java ME)*** – платформа для устройств с ограниченными ресурсами и мобильных устройств
- ***Java Card*** – платформа для смарт-карт

Кафедра автоматизированных систем обработки информации http:// gsu.by

Синтаксис JAVA




```
public class Example {
    public static void main(String[] args) {
        int i = 3;
        long l = 33L;
        String s = "3";
        Object o = new Object();
    }
}
```



Кафедра автоматизированных систем обработки информации http:// gsu.by

Наследование

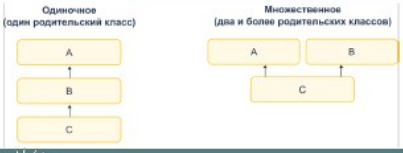
Наследование

Наследование – механизм объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. В Java множественное наследование запрещено.

```
class A {
   public int field1;
   public void method() {
      /* ... */
   }
}
```

```
class B extends A {
   public int field2;
}
```

```
public static void main(String[] args) {
   B b = new B();
   b.field1 = 5;
   b.field2 = 8;
   b.method();
}
```



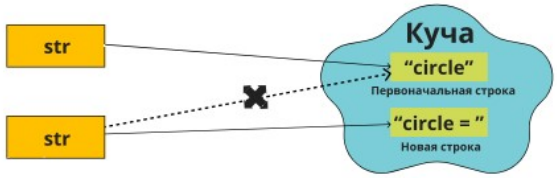
Конструкторы класса String

Строка – это последовательность символов. В Java строки реализованы с помощью трех основных классов модуля *java.base* пакета *java.lang*, таких как: **String, StringBuilder, StringBuffer.** Для форматирования и обработки строк применяются классы **Formatter, Pattern, Matcher.**

Основные конструкторы класса **String**:

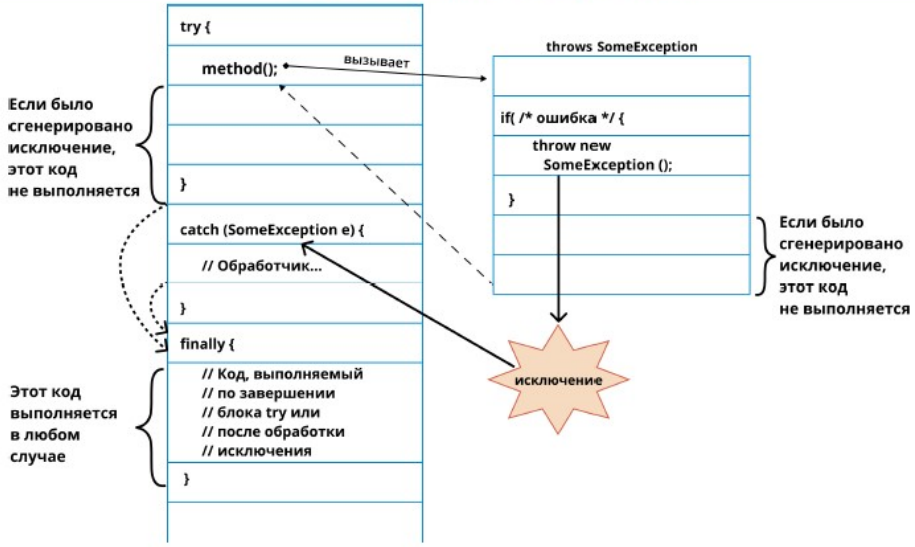
- String();
- String(String str);
- String(char[] unicodechar);
- String(byte[] bytes);
- String(StringBuffer sbuf);
- String(StringBuilder sbuild);

Каждый раз, когда редактируется содержимое строки, на самом деле создается новая строка (новый объект) с содержимым модифицированной строки.



https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/String.html

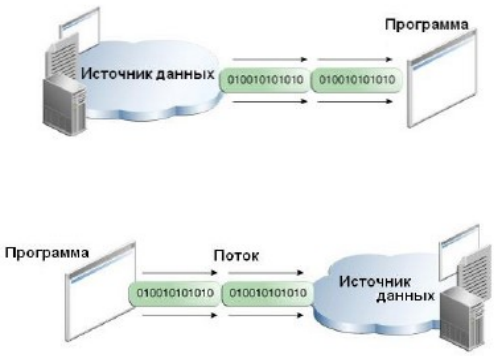
Обработчик завершения



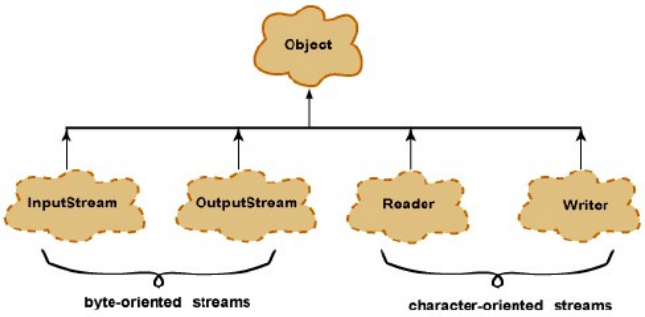
Один блок **try** может содержать и блок **finally**, и блок **catch**. Таким образом блок **try** контролируется на возникновение исключения. Также это гарантирует освобождение ресурсов при любых условиях выхода.

Потоки ввода-вывода

Потоки ввода/вывода - это абстракция, которая представляет собой последовательность передаваемых или принимаемых данных. Обеспечиваются библиотеками классов java.io, java.nio.



Все потоки ввода-вывода можно разделить на **символьные** и **байтовые** потоки, для организации каждого из этих потоков существует своя иерархия классов.

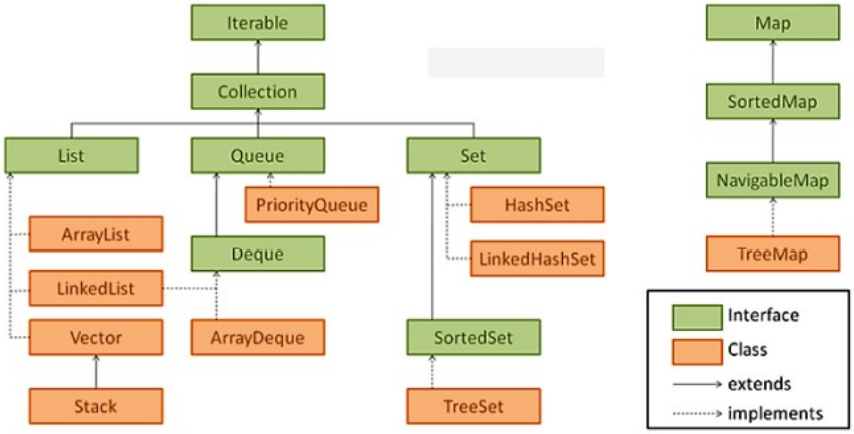


java.util. Collection Framework

Коллекции – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним.

Коллекции представляют собой реализацию абстрактных типов (структур) данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

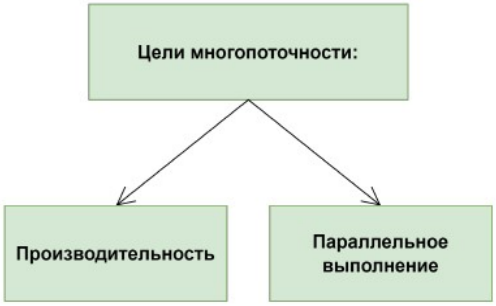


Понятие Multithreading

Потоки управления позволяют выполнять различные задачи параллельно.

Когда следует использовать потоки управления?

- при выполнении длительной операции;
- необходимо ускорить выполнение операции;
- при использовании оконных приложений.



Java Database Connectivity

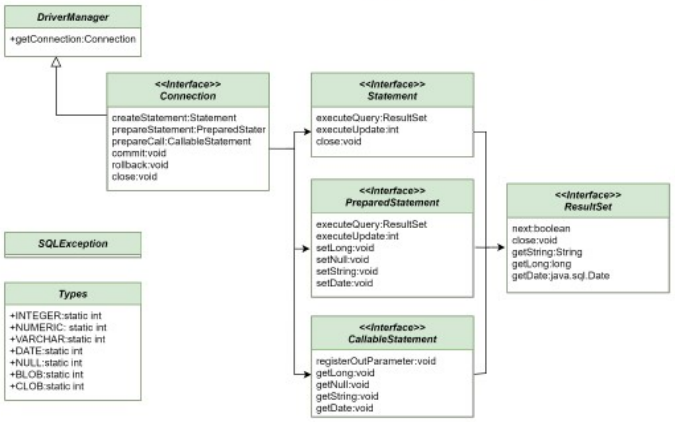
Использование **JDBC**. Порядок действий.

1. Загрузка класса драйвера базы данных.
2. Установка соединения с БД.
3. Создание объекта для передачи запросов.
4. Выполнение запроса.
5. Обработка результатов выполнения запроса.
6. Закрытие соединения.

Что может JDBC?

- Устанавливать соединение с БД
- Отсылать SQL-запросы
- Обрабатывать результаты

Основные интерфейсы и классы JDBC



3 ВОПРОСЫ ДЛЯ САМОКОНТРОЛЯ

1. Расшифровать аббревиатуры JVM, JDK и JRE. Кто что включает и как взаимодействует.
2. Объяснить различия между терминами «объект» и «ссылка на объект».
3. Какие области памяти использует Java для размещения примитивных типов, объектов, ссылок.
4. Какие примитивные типы Java существуют, как создать переменные примитивных типов?
5. Объяснить, что такое явное и неявное приведение типов, привести примеры, когда такое преобразование имеет место.
6. Что такое классы-оболочки, для чего они предназначены?
7. Объяснить разницу между примитивными и ссылочными типами данных.
8. Что такое autoboxing и unboxing?
9. Дать определение таким понятиям как «класс» и «объект».
10. Что такое конструктор? Как отличить конструктор от любого другого метода? Сколько конструкторов может быть в классе?
11. Что такое конструктор по умолчанию? Может ли в классе совсем не быть конструкторов? Объяснить, какую роль выполняет оператор this() в конструкторе?
12. Какова процедура инициализации полей класса и полей экземпляра класса?
13. Когда инициализируются поля класса, а когда — поля экземпляров класса?
14. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?
15. Дать определение перегрузке методов. Чем удобна перегрузка методов?
16. Указать, какие методы могут перегружаться, и какими методами они могут быть перегружены?
17. Что такое финальные поля, какие поля можно объявить со модификатором final?
18. Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?
19. Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?
20. Что такое модификатор static?
21. Generics. Что это такое и для чего применяются. Во что превращается во время компиляции и выполнения?

22. Что такое enum?
23. Принципы ООП.
24. Правила переопределения метода **boolean equals(Object o).**
25. Зачем переопределять методы **hashCode()** и **equals()** одновременно?
26. Написать метод **equals()** для класса, содержащего одно поле типа String.
27. Для чего используется ключевое слово **final**?
28. Чем является класс Object? Перечислить известные методы класса Object, указать их назначение.
29. Для чего используется наследование классов в java-программе? Привести пример наследования. Поля и методы, помеченные модификатором доступа private, наследуются?
30. Что в конструкторе класса делает оператор super()?
31. Возможно ли в одном конструкторе использовать операторы super() и this()?
32. Что такое переопределение методов? Зачем оно нужно? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?
33. Определить правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?
34. Какие свойства имеют финальные методы и финальные классы? Зачем их использовать?
35. Какие применяются правила приведения типов при наследовании. Записать примеры явного и неявного преобразования ссылочных типов. Объяснить, какие ошибки могут возникать при явном преобразовании ссылочных типов.
36. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?
37. Для чего служит интерфейс Cloneable? Как правильно переопределить метод clone() класса Object, для того чтобы объект мог создавать свои адекватные копии?
38. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?
39. Могут ли перечисления реализовывать интерфейсы или содержать абстрактные методы? Могут ли перечисления содержать статические методы?
40. Что такое параметризованные классы? Для чего они необходимы?
41. Для чего используется параметризация <? extends Type>, <? super Type>?

42. Что такое внутренние, вложенные и анонимные классы? Как определить классы такого вида? Как создать объекты классов такого вида.
43. Перечислить возможности доступа к членам внешнего класса, которым наделены вложенные классы?
44. Перечислить возможности доступа к членам внешнего класса, которым наделены внутренние классы?
45. Перечислить возможности доступа к членам внешнего класса, которым наделены анонимные классы?
46. Могут ли классы внутри классов быть базовыми, производными или реализующими интерфейсы?
47. Что такое интерфейс? Как определить и реализовать интерфейс в java-программе?
48. Можно ли описывать в интерфейсе конструкторы и создавать объекты?
49. Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?
50. Какие модификаторы по умолчанию имеют поля интрефейса?
51. Какие модификаторы по умолчанию имеют методы интрефейса?
52. Чем отличается абстрактный класс от интерфейса?
53. Когда применять интерфейс логичнее, а когда абстрактный класс?
54. Бывают ли интерфейсы без методов? Для чего?
55. Могут ли классы внутри классов реализовывать интерфейсы?
56. Возможно ли анонимный класс создать на основе реализации интерфейса?
57. Дать определение функционального интерфейса.
58. Что такое лямбда-выражение? Его структура.
59. Интерфейс Comparator. Его назначение.
60. Как сортировать список с применением лямбда-выражений? С помощью какого интерфейса?
61. Как создать объект класса String, какие существуют конструкторы класса String? Что такое строковый литерал? Что значит «упрощенное создание объекта String»?
62. Можно ли изменить состояние объекта типа String? Что происходит при попытке изменения состояния объекта типа String? Можно ли наследоваться от класса String? Почему строковые объекты immutable?
63. Что такое пул литералов? Как строки заносятся в пул литералов? Как занести строку в пул литералов и как получить ссылку на строку, хранящуюся в пуле литералов
64. В чем отличие объектов классов StringBuilder и StringBuffer от объектов класса String?
65. Что представляет собой регулярное выражение?
66. Что такое метасимволы регулярного выражения?
67. Какие существуют классы символов регулярных выражений? Что такое квантификаторы?

68. Какие классы Java работают с регулярными выражениями?
69. Что такое группы в регулярных выражениях? Как нумеруются группы? Что представляет собой группа номер «0»?
70. Что для программы является исключительной ситуацией? Какие существуют способы обработки ошибок в программах?
71. Что такое исключение для Java-программы? Что значит «программа генерировала\выбросила исключение»?
72. Привести иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?
73. Объяснить работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному блоку try?
74. Можно ли вкладывать блоки try друг в друга, можно ли вложить блок try в catch или finally? Как происходит обработка исключений, выброшенных внутренним блоком try, если среди его блоков catch нет подходящего?
75. Как работает блок try с ресурсами?
76. Указать правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченное исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок catch выбрасывать иные исключения?.
77. Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не будет вызван? Может ли блок finally выбрасывать исключения? Может ли блок finally выполниться дважды?
78. Как генерировать исключение вручную? Объекты каких классов могут быть генерированы в качестве исключений? Можно ли генерировать два исключения одновременно?
79. Объяснить, как работают операторы throw и throws. В чем их отличия?
80. Объяснить правила реализации секции throws при переопределении метода и при описании конструкторов производного класса.
81. Как ведет себя блок throws при работе с проверяемыми и непроверяемыми исключениями?
82. Как создать собственные классы исключений?
83. Что такое поток данных? Какие потоки данных существуют в Java?
84. Привести иерархию потоков ввода-вывода в Java.
85. Какие классы байтовых потоков ввода-вывода существуют?
86. Какие классы символьных потоков ввода-вывода существуют?
87. Как работают методы read() и write() базовых классов иерархии символьных и байтовых потоков?
88. Для чего используются классы BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter?

89. Для чего применяются классы InputStreamReader и OutputStreamWriter?
90. Что такое сериализация, для чего нужна, когда применяется?
91. Что такое десериализация?
92. Ключевое слово transient, для чего нужно?
93. Назвать основные интерфейсы коллекций. Какие бывают коллекции?
94. В чем особенности разных видов коллекций? Когда и какие коллекции следует применять?
95. Сравнить ArrayList и LinkedList.
96. Как устроены HashSet, TreeMap, TreeSet.
97. Принцип работы и реализации HashMap.
98. Особенности интерфейса Set.
99. Как добавляются объекты в HashSet?
100. Какими способами можно отсортировать коллекцию?
101. Как правильно удалить элемент из коллекции при итерации в цикле?
102. Что происходит при добавлении в ArrayList нового элемента и как это реализовано?
103. Если в коллекцию часто добавлять элементы и удалять, какую лучше использовать? Почему? Как они устроены?
104. Что такое поток выполнения? Состояния потока.
105. Как создать поток? Какими тремя способами можно создать поток, запустить его, прервать (завершить, убить)?
106. Жизненный цикл потока.
107. Как выполнить набор команд в отдельном потоке?
108. Как работают методы wait() и notify()/notifyAll()? Каков будет результат, если на ресурсе вызвать метод notify(), не вызвав до этого соответствующий ему wait()?
109. Чем отличается работа метода wait() с параметром и без параметра?
110. Как работает метод yield()? Чем отличаются методы Thread.sleep() и Thread.yield()?
111. Как можно осуществить приостановку/возобновление работы потоков?
112. Чем отличаются методы Thread.sleep() и object.wait()?
113. Как работает метод join()?
114. Зачем нужны потоки-демоны? Как создать поток-демон? Когда следует применять потоки-демоны?
115. На что влияет приоритет потока? Как потоку установить приоритет?
116. Что такое синхронизация? Зачем она нужна? Для чего нужно ключевое слово synchronized? Какие методы синхронизации существуют? Какими средствами достигается?
117. Что такое монитор объекта? Кто и как с ним может работать? Объяснить, что значит поток, взявший монитор, может взять его повторно?
118. Отличия работы synchronized от Lock?

119. Есть ли у Lock механизм, аналогичный механизму wait\notify у synchronized?
120. Что такое deadlock? Нарисовать схему, как это происходит. Как избежать этой ситуации?
121. Что такое JDBC? Перечислить основные классы и интерфейсы, входящие в состав JDBC, указать их назначение. Какие еще существуют технологии Java, работающие с БД?
122. Привести алгоритм получения соединения с базой данных, выполнения запроса и обработки результатов. Как загрузить драйвер базы данных и что он собой представляет?
123. Нужно ли регистрировать драйвер БД? Если да, то как это сделать?
124. Как правильно закрыть Connection?
125. Чем отличается Statement от PreparedStatement?
126. Зачем нужен CallableStatement? Как выполняется вызов хранимых процедур из Java-программы?
127. Чем отличаются метод executeUpdate() от executeQuery()?
128. Для чего JDBC использует объекты типа ResultSet?
129. Привести определение транзакции, commit и rollback.
130. Что такое точка сохранения и как ее создать? Как откатить транзакцию до точки сохранения или до предыдущего commit?
131. Что означает термин «метаданные»? Какую информацию предоставляют объекты классов DatabaseMetaData, ResultsSetMetaData и для чего она может быть использована?

4 ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

1 Основы синтаксиса. Java Code Convention
2 Наследование и интерфейсы
3 Обработка исключений
4 Работа с файлами. Сериализация
5 Коллекции в Java
6 Работа с XML документами
7 Работа с JSON документами
8 Работа с базами данных

Лабораторная работа №1 Основы синтаксиса. Java Code Convention
Задание: создать программу для вывода текстовой информации в консоль. С помощью командной строки вызвать компилятор для компиляции программы, затем интерпретатор для выполнения программы.

Лабораторная работа №2 Наследование и интерфейсы
Задание: создать консольное приложение согласно варианту, удовлетворяющее следующим требованиям:

- использовать возможности ООП: классы, наследование, полиморфизм, инкапсуляция;
- каждый класс должен иметь отражающее смысл название и информативный состав;
- наследование должно применяться только тогда, когда это имеет смысл;
- при кодировании должны быть использованы соглашения об оформлении кода java code convention;
- классы должны быть грамотно разложены по пакетам;
- консольное меню должно быть минимальным;
- для хранения параметров инициализации можно использовать файлы.

Лабораторная работа №3 Обработка исключений
Задание: выполнить лабораторную работу на основе индивидуальных заданий. Предусмотреть обработку исключений, возникающих при нехватке памяти, отсутствии требуемой записи (объекта) в файле, недопустимом значении поля и т.д. Реализуйте собственные обработчики исключений и исключения ввода/вывода.

Лабораторная работа №4 Работа с файлами. Сериализация
Задание: выполнить лабораторную работу на основе индивидуальных заданий, контролируя состояние потоков ввода/вывода. При возникновении ошибок, связанных с корректностью выполнения математических операций, генерировать и обрабатывать исключительные ситуации. При выполнении заданий для вывода результатов создавать новую директорию и файл средствами класса File.

Лабораторная работа №5 Коллекции в Java
Задание: разработать программу согласно варианту с использованием коллекций. При выполнении задания необходимо сопровождать все реализованные процедуры и функции набором тестовых входных и выходных данных и описаниями к ним.

Лабораторная работа №6 Работа с XML документами
Задание: создать файл XML и соответствующую ему схему XSD. При разработке XSD использовать простые и комплексные типы, перечисления, шаблоны и предельные значения. Создать приложение для разбора XML-документа и инициализации коллекции объектов информацией из XML-файла. Для разбора использовать SAX, DOM и StAX парсеры. Для сортировки объектов использовать интерфейс Comparator. Произвести проверку XML-документа с привлечением XSD. Определить метод, производящий преобразование разработанного XML-документа в документ, указанный в каждом задании.

Лабораторная работа №7 Работа с JSON документами
Задание: создать файл JSON. Используя процесс сериализации и десериализации объектов, произвести запись и чтение данных в/из файла.

Лабораторная работа №8 Работа с базами данных
Задание: разработать программу в соответствии с вариантом. В каждом из заданий необходимо выполнить следующие действия:

- организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение;
- создать БД. Привести таблицы к одной из нормированных форм;
- создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов;
- создать класс на модификацию информации.

Примеры реализации лабораторных работ:

Ход работы. вычислить сколько раз каждая буква встречается в тексте. Для реализаии задачи необходимо использовать коллекцию HashMap. Перебор элементов коллекции необходимо производить через вложенный класс Map.Entry. В качестве тестового значения выберем строку «лабораторная работа». Далее, с помощью цикла произведем перебор всех символов строки и проверим является ли символ буквой с помощью метода Character.isLetter. Если да, то проверим, содержит ли коллекция указанный символ в качестве ключа, т.к. ключи в коллекции – уникальные значения, то в случае повторного содержания в строе такого символа, в коллекции он будет перезаписываться. Если такой ключ существует, в значении этого элемента инкрементируем значение, что в дальнейшем покажет нам количество вхождений данного

символа в строку. Как результат работы программы выведим всю коллекцию в консоль.

Листинг программы

```
package by.gsu.lab3;
import java.util.HashMap;
import java.util.Map.Entry;
public class Main {
public static void main(String[] args) {
String txt = " лабораторная работа ";
HashMap<Character, Integer> map = new HashMap<Character, Integer>(40);
for (int i = 0; i < txt.length(); ++i) {
char c = txt.charAt(i);
//проверяем является ли символ буквой
if (Character.isLetter(c)) {
if (map.containsKey(c)) {
map.put(c, map.get(c) + 1);
} else {
map.put(c, 1); }}}
//вывод на экран букв с частотой их появления
for (Entry<Character, Integer> entry : map.entrySet()) {
System.out.println("буква: " + entry.getKey() + " кол - во: " + entry.getValue()); }}}
```

Результат:

```
буква: т кол - во: 2
буква: я кол - во: 1
буква: а кол - во: 5
буква: б кол - во: 2
буква: л кол - во: 1
буква: н кол - во: 1
буква: о кол - во: 3
```

5 ТЕСТОВЫЕ ЗАДАНИЯ (примеры)

1

Баллов: 1

Когда программист указывает тип элементов массива и его название, он:

Выберите один ответ.

- ○ Создает массив
- ○ Инициализирует массив
- ○ Выделяет память под массив
- ○ Объявляет массив

2

Баллов: 1

Какой класс коллекции позволяет наращивать и сокращать размер, предоставляет индексный доступ к элементам?

Выберите один ответ.

- ○ java.util.LinkedHashSet
- ○ java.util.HashSet
- ○ java.util.ArrayList

3

Баллов: 1

Какими характеристиками обладает любая переменная? Выберите 2 варианта ответа.

Выберите по крайней мере один ответ:

- ☐ Вариативность
- ☐ Вложенность
- ☐ Область видимости
- ☐ Время жизни

4

Баллов: 1

Что будет результатом компиляции данного кода?

```
class ExceptionOne extends Exception {}
class ExceptionTwo extends ExceptionOne {}
abstract class Abstract {
    abstract void method() throws ExceptionOne;
}
public class Main extends Abstract {
    static int a,b,c,d;
    @Override
    void method() throws ExceptionTwo {
        throw new ExceptionTwo();
    }
    public static void main(String[] args) {
        Main main = new Main();
        try {
            main.method();
            a++;
        }
        catch (ExceptionTwo ex) {
            b++;
        }
        catch (ExceptionOne ex) {
            c++;
        }
        finally {
            d = a + b + c;
        }
        System.out.println(a + " " + b + " " + c + " " + d);
    }
}
```

Выберите один ответ.

- ○ 0 1 1 2
- ○ compile error
- ○ 0 1 0 1
- ○ 1 0 0 1
- ○ 0 0 1 1

5

Баллов: 1

Что будет результатом компиляции и выполнения следующего кода?

```
public class MultipleReturn {
    public int getInt() {
        try {
            String[] students = {"Marry", "Paul"};
            System.out.println(students[5]);
        } catch (Exception e) {
            return 10;
        } finally {
            return 20;
        }
    }
    public static void main(String[] args) {
        MultipleReturn var = new MultipleReturn();
        System.out.println(var.getInt());
    }
}
```

Выберите один ответ.

- 20;
- 0;
- 10;
- Ошибка компиляции;

6

Баллов: 1

Что будет результатом компиляции и выполнения следующего кода?

```
class ParentClass {
    void parentMethod(int i) {
        System.out.println("parentMethod ParentClass" + i);
    }
}
class ChildClass extends ParentClass{
    public void parentMethod(int i) {
        System.out.println("parentMethod ChildClass" + i);
    }
    public void childMethod(int i) {
        System.out.println("childMethod ChildClass" + i);
    }
    public static void main(String args[]) {
        ParentClass quest = new ChildClass ();   // 1
        quest.parentMethod(1);   // 2
        quest.childMethod(1);   // 3
    }
}
```

Выберите один ответ.

- parentMethod ChildClass 1
- parentMethod ChildClass 1
- Compilation error in line 1
- Compilation error in line 3
- childMethod ChildClass 1
- Compilation error in line 2
- parentMethod ParentClass 1

7

Баллов: 1

Каково предназначение BufferedOutputStream?

Выберите один ответ.

- используется для буферизации вывода
- преобразовывает входной поток в считывающий класс
- используется при просмотре файлов для компилятора
- преобразовывает выходной поток в записывающий класс

8

Баллов: 1

Какое предназначение у поточного класса LineNumberInputStream?

Выберите один ответ.

- все варианты не верны
- следит за количеством считанных из потока строк
- посылка данных в файл на диске
- для предотвращения физического чтения устройств при каждом новом запросе данных

9

Баллов: 1

Что будет выведено в результате при компиляции и запуске данного кода?

```
StringBuilder sb1 = new StringBuilder("I like Java.");//1
StringBuilder sb2 = new StringBuilder(sb1);//2
System.out.println(sb1.equals(sb2));
```

Выберите один ответ.

- false
- ошибка компиляции в строке 2
- ошибка компиляции в строке 1
- true

10

Баллов: 1

Что будет результатом выполнения следующего фрагмента кода?

```
String[] strArray = new String[] {"One", "Two", "Three"};
strArray[2] = null;
for (String val : strArray)
System.out.print(val + ", ");
```

Выберите один ответ.

- Возникнет ошибка времени выполнения
- One, Two, null,
- One, Two, Three,
- Возникнет ошибка компиляции
- One, null, Three,

11

Баллов: 1

Что будет результатом компиляции и выполнения следующего кода?

```
public class BoxPrinter<T> {
    private T val;
    public BoxPrinter(T val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return "[" + val + "]";
    }
    public static void main(String[] args) {
        BoxPrinter<String> var = new BoxPrinter<>("ASOI");
        System.out.println(var);
        BoxPrinter<Integer> var1 = new BoxPrinter<>(10);
        System.out.println(var1);
    }
}
```

Выберите один ответ.

- Ошибка времени исполнения.
- Ошибка компиляции;
- [null]
[null]
- [ASOI]
[10]

12

Баллов: 1

Что такое класс с точки зрения программирования?

Выберите один ответ.

- Определение структуры и поведения некоторой сущности
- Определение взаимодействия между сущностями одного типа
- Определение поведения некоторой сущности
- Определение структуры некоторой сущности

13

Баллов: 1

Какой класс используется для буферизации ввода?

Выберите один ответ.

- RandomAccessFile
- ByteArrayOutputStream
- DataInputStream
- BufferedInputStream

14

Баллов: 1

Что будет результатом компиляции и выполнения следующего кода?

```
public enum Animals {
    DOG("woof"), CAT("meow"), FISH("burble");                    //Строка 1
    String sound;                                                //Строка 2
    Animals(String sound) {                                      //Строка 3
        this.sound = sound;
    }
}
class Main{
    public static void main(String[] args) {
        System.out.println(Animals.DOG.sound + " " + Animals.FISH.sound); //Строка 4
    }
}
```

Выберите один ответ.

- Ошибка компиляции в //Строка 1
- Ошибка компиляции в //Строка 4
- Ошибка компиляции в //Строка 2
- woof burble;
- Ошибка компиляции в //Строка 3

15 Баллов: 1

Что будет результатом компиляции и выполнения следующего кода?

```
public enum CoffeeSize {BIG, HUGE, OVERWHELMING}
class Main{
    public static void main(String[] args) {
        System.out.println(CoffeeSize);
    }
}
```

Выберите один ответ.

- ○ Ошибка времени выполнения
- ○ Ошибка компиляции;
- ○ BIG
- ○ BIG, HUGE, OVERWHELMING
- ○ Что-то похожее на: [LCoffeeSize;@58ceff1

16 Баллов: 1

Выберите типы вложенных классов.

Выберите по крайней мере один ответ:

- ☐ Локальные классы
- ☐ Статические вложенные классы
- ☐ Вплетенные классы
- ☐ Мультиклассы
- ☐ Анонимные классы
- ☐ Внутренние классы

17 Баллов: 1

В стеке хранится лишь адрес ячейки памяти, которая находится в куче для:

Выберите один ответ.

- ○ Примитивной переменной
- ○ Любого типа переменных
- ○ Ссылочной переменной

18 Баллов: 1

Какой метод класса InputStream сбрасывает указатель в ранее установленную метку?

Выберите один ответ.

- ○ reset()
- ○ release()
- ○ drop()
- ○ skip()
- ○ slip()

19 Баллов: 1

Какое из утверждений об операторе switch верное?

Выберите один ответ.

- ○ В операторе switch должна быть только одна ветка case.
- ○ В операторе switch всегда должна быть ветка default.
- ○ Символьный литерал может использоваться как ключ в ветке case.
- ○ В операторе switch ветка default указывается после всех веток case.

20 Баллов: 1

Какие из указанных действий приведут к тому, что поток переходит в состояние "TERMINATED"? Выберите два варианта ответа.

Выберите по крайней мере один ответ:

- ☐ окончание выполнения метода run();
- ☐ вызов метода wait() с параметром null.
- ☐ вызов метода stop();
- ☐ вызов метода sleep() без параметра;
- ☐ вызов метода notifyAll();

Учреждение образования
«Гомельский государственный университет имени Франциска Скорины»

УТВЕРЖДАЮ
Проректор по учебной работе
ГГУ имени Ф. Скорины
_____ И.В. Семченко
04.08.2017
(дата утверждения)
Регистрационный № УД-34-2017-14 / уч.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Учебная программа учреждения высшего образования по учебной дисциплине для специальности:

1 – 53 01 02 Автоматизированные системы обработки информации

Согласовано
Зам. директора по производству
ЧУЗЯ - Гомель - Парк
В.Я. Архипенко

2017 г.

Учебная программа составлена на основе: образовательного стандарта ОСВО 1-53 01 02 2013 г. и учебного плана УВО, регистрационный № 1-53 01 13, дата утверждения 25.08.2013

СОСТАВИТЕЛЬ:
М.С. Данильченко, старший преподаватель кафедры АСОИ

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:

Кафедрой автоматизированных систем обработки информации
(протокол № 9 от 11.04.2017);

Научно-методическим советом Учреждения образования «Гомельский государственный университет имени Франциска Скорины».
(протокол № 8 от 07.06.2017).

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Необходимость изучения дисциплины «Шаблоны проектирования» заключается в том, что она является основой для освоения учащимися современных приемов программирования и походов к решению типовых задач.

Целью дисциплины является формирование понятий в области объектно-ориентированного программирования (ООП) и получение навыков работы с основными шаблонами проектирования.

Задачами дисциплины являются:

– усвоение студентами основных понятий и принципов ООП;

– овладение навыками написания программ на объектно-ориентированных языках программирования;

– овладение подходами к выбору архитектурных решений в разработке программного обеспечения;

– формирование навыков применения шаблонов проектирования для решения задач широко спектра.

В результате изучения учебной дисциплины студент должен:

знать:

– современное состояние одного из алгоритмических языков высокого уровня;

– основные структуры данных и методы работы с ними;

– базовые шаблоны проектирования, применяемые для решения типовых задач программирования.

уметь:

– решать типовые задачи ООП с использованием шаблонов проектирования;

– применять знания шаблонов проектирования при решении не тривиальных задач ООП.

владеть:

– современными средствами разработки приложений с использованием объектно-ориентированных языков программирования;

– навыками разработки простейших web-приложений;

– навыками отладки программ.

Специалист должен обладать следующими видами компетенций:

1 Требования к академическим компетенциям специалиста:

АК-1 Уметь применять базовые научно-теоретические знания для решения теоретических и практических задач.

АК-2. Владеть системным и сравнительным анализом.

АК-3. Владеть исследовательскими навыками.

АК-4. Уметь работать самостоятельно.

АК-5. Быть способным порождать новые идеи (обладать креативностью).

АК-11. Применять соответствующий физико-математический аппарат, методы математического анализа и моделирования, теоретического и

экспериментального исследования в физике, химии, экологии для решения проблем, возникших в ходе профессиональной деятельности.

АК-12. Владеть основными методами, способами и средствами получения, хранения, переработки информации, наличием навыков работы с компьютером как средством управления информацией.

2 Требования к профессиональным компетенциям специалиста:

ПК-11. Проводить анализ эффективности функционирования систем обработки информации и выявлять узкие места по производительности и надежности.

ПК-12. Выявлять и устранять уязвимость систем обработки информации к угрозам безопасности.

ПК-13. Выполнять реконфигурацию баз данных и системного программного обеспечения под условия применения.

ПК-14. Выполнять обновление системного и прикладного программного обеспечения.

ПК-15. Выявлять актуальные проблемы развития и совершенствования информационных технологий.

ПК-16. Выполнять экспертизу проектов средств реализации информационных технологий.

ПК-17. Выделять области эффективного применения различных методов и средств реализации информационных технологий.

ПК-18. Консультировать потребителей по вопросам выбора эффективных методов решения задач, связанных с представлением, хранением, отображением, передачей и аналитической обработкой информации.

Изучение дисциплины компонента учреждения образования «Шаблоны проектирования» предусмотрено учебным планом подготовки специалистов специальности 1-53 01 02 – «Автоматизированные системы обработки информации».

Дисциплина компонента учреждения образования «Шаблоны проектирования» изучается студентами 3 курса дневной формы обучения специальности 1-53 01 02 - «Автоматизированные системы обработки информации», студентами 3,4 курсов заочной интегрированной формы обучения на основе среднего специального образования и студентами 2,3 курсов заочной сокращенной формы обучения.

Дневная форма обучения: всего часов по плану – 156; аудиторное количество часов – 64, из них: лекции – 30, лабораторные занятия – 34.

Форма отчётности – экзамен в 5 семестре.

Заочная форма обучения: всего часов по плану – 158, аудиторное количество часов – 16, из них: лекции – 10, лабораторных работ – 6.

Форма отчетности – экзамен в 7 семестре.

Заочная форма обучения (интегрированная форма обучения на основе среднего специального образования): всего часов по плану – 158, аудиторное количество часов – 16, из них: лекции – 10, лабораторных работ – 6.

Форма отчетности – экзамен в 5 семестре.

1 СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Тема 1. ООП. Парадигмы ООП. Классы и объекты

Основы объектно-ориентированного программирования. Представление об инкапсуляции, полиморфизме и наследовании. Подходы к разработке приложений с использованием принципов ООП. Основы работы в IDE.

Тема 2. Основы синтаксиса Java. Java Code Convention

Основы синтаксических конструкций Java. Область видимости переменных и методов. Структура методов и классов. Основные типы данных. Написание консольных приложений. Работа с основными потоками ввода-вывода. Циклы. Логические операторы. Основные математические методы и операторы.

Тема 3. Наследование в Java. Интерфейсы

Наследование классов. Приведение ссылочных типов. Абстрактные классы. Реализация и перегрузка методов. Наследование методов родительского класса. Интерфейсы. Описание методов и параметров интерфейса. Реализация интерфейса. Интерфейс Comparable.

Тема 4. Обработка строк

Тип данных String. Конструкторы и методы типа String. Форматированный вывод строк. Обработка строковых параметров. Классы StringBuilder и StringBuffer. Локализация строковых ресурсов, работа с локалями.

Тема 5. Исключения и ошибки

Класс Exception. Класс Error. Обработка исключений. Передача исключений. Реализация собственных исключений. Передача сообщений об ошибках. Блок finally.

Тема 6. Работа с файлами. Сериализация

Класс File. Бинарные потоки ввода InputStream и его наследники. Символьные потоки ввода InputStreamReader. Буферные потоки ввода BufferedReader. Бинарные потоки вывода OutputStream и его наследники. Символьные потоки вывода Writer. Сериализация объектов и интерфейс Serializable. Обработка исключений сериализации-десериализации. Особенности сериализации полей transient и static.

Тема 7. Коллекции в Java

Понятие о коллекции. Списки List, ArrayList. Set, TreeSet. Коллекции "ключ-значение" Map, HashMap. Интерфейс Comparable и метод compateTo. Сортировка коллекций. Сравнение производительности работы коллекций различных классов. Обработка данных в коллекциях. Класс Iterator.

Тема 8. Работа с XML документами

Основы синтаксиса XML. Структурные элементы XML документа. DOMParser и особенности его реализации. SAXParser и особенности его реализации. Интерфейс ContentHandler. StAXParser. DTD и XSD схемы проверки XML документа. Основы синтаксиса DTD и XSD схем.

Тема 9. Работа с JSON документами

Основы синтаксиса JSON. Обработка JSON стандартными средствами Java. Классы JSONArray и JSONObject. Сторонние библиотеки обработки JSON. Библиотека Gson.

Тема 10. Работа с многопоточными приложениями

Понятие многопоточного приложения. Жизненный цикл потоков. Проверка состояния потока. Класс Thread и переопределение метода run(). Интерфейс Runnable. Сервисные потоки и особенности их жизненного цикла. Распределенные вычисление в пределах одного приложения.

Тема 11. Базы данных. Работа с подключениями. Выборка данных

Основные понятия баз данных. Типы данных. Локальные и серверные базы данных. Основы синтаксиса SQL. Запросы SELECT. Сортировка результатов запроса выборки. Группировка данных в запросе. Условия выборки. Объединение запросов. Создание подключений к базе данных средствами Java. Менеджмент подключений

Тема 12. Базы данных. Изменение и редактирование баз данных

Запросы Create Table, Alter Table. Запросы Insert, Update, Delete. Использование PreparedStatement. Понятие транзакции, работа с транзакциями в Java. Подтверждение и откат изменений в транзакции. Точки сохранения транзакции и отмена изменений до точки сохранения.

Тема 13. Шаблоны проектирования GRASP

Понятие шаблонов проектирования. Шаблон Expert. Шаблон Creator. Шаблон Low Coupling. Шаблон High Cohesion. Шаблон Controller.

Тема 14. Шаблоны проектирования GoF

Понятие о шаблонах GoF. Особенности применения шаблонов. Стандартные задачи решаемые шаблонами.

Тема 15. Порождающие и структурные шаблоны проектирования

Понятие о порождающих и структурных шаблонах проектирования. Шаблон Factory. Шаблон AbstractFactor. Шаблон Builder. Шаблон Singleton. Шаблон Bridge. Шаблон Decorator.

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА (дневная форма обучения)

Номер раздела, темы, занятия	Название раздела, темы, занятия; перечень изучаемых вопросов	Количество аудиторных Часов				Кол-во часов УСР	Формы контроля знаний
		лекции	практические занятия	лабораторные занятия	Иное		
1	2	3	4	5	6	7	8
1	Введение 1 Объектно-ориентированное программирование. 2 Парадигмы ООП. 3 Классы и объекты.	**2**		–			
2	Основы синтаксиса Java. Java Code Convention 1 Параметры и методы. Модификаторы области видимости. 2 Операторы цикла и ветвления. 3 Требования к оформлению кода.			**2**		**2**	Защита отчета по лабораторной работе
3	Наследование в Java. Интерфейсы 1 Правила и механизмы наследования. 2 Понятие интерфейса. 3 Правила реализации интерфейсов.	**2**		**4**			Защита отчета по лабораторной работе
4	Обработка строк 1 Класс String. 2 Форматированный вывод строк. 3 Классы StringBuffer и StringBuilder. 4 Локализация приложений			–		**2**	

1	2	3	4	5	6	7	8
5	Исключения и ошибки 1 Класс Error 2 Класс Exception и его наследники 3 Генерация и обработка исключений 4 Создание собственных классов исключений	**2**		**4**			Защита отчета по лабораторной работе
6	Работа с файлами. Сериализация 1 Класс File 2 Потоки ввода/вывода 3 Сериализация. Интерфейс Serializable			**4**		**2**	Защита отчета по лабораторной работе
7	Коллекции в Java 1 Списки 2 Множества 3 Коллекции ключ-значение	**2**		**4**			Защита отчета по лабораторной работе
8	Работа с XML документами 1 Структура XML документа. 2 DOM, SAX и StAX. 3 DTD и XSD схемы.	**2**		**4**			Защита отчета по лабораторной работе
9	Работа с JSON документами 1 Структура JSON документа 2 JSONArray и JSONObject 3 Библиотека GSON			**4**		**2**	Защита отчета по лабораторной работе
10	Работа с многопоточными приложениями 1 Жизненный цикл потока. 2 Класс Thread. Интерфейс Runnable 3 Методы управления потоками. 4 Синхронизация потоков выполнения.	**2**		**4**			Защита отчета по лабораторной работе
11	Базы данных. Работа с подключениями. Выборка данных 1 JDBC. 2 Методы для работы с выборкой данных. 3 Обработка результатов выполнения запросов.	**2**		**2**			

12	Базы данных. Изменение и редактирование баз данных 1 Обработка запросов изменения структуры БД. 2 Запросы изменения таблиц.	**2**		**2**			
13	Шаблоны проектирования GRASP 1 Назначение шаблонов GRASP. 2 Реализация в Java.			**-**		**2**	
14	Шаблоны проектирования GoF 1 Назначение шаблонов GoF. 2 Реализация в Java.	**2**		**-**			
15	Порождающие и структурные шаблоны проектирования 1 Назначение шаблонов. 2 Реализация в Java.	**2**		**-**			
	Всего	**30**		**34**			Экзамен 5 семестр

Старший преподаватель М.С. Данильченко

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА
(заочная форма обучения)

Номер раздела, темы, занятия	Название раздела, темы, занятия; перечень изучаемых вопросов	Количество аудиторных часов					Кол-во часов УСР	Формы контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
1	Введение 1 Объектно-ориентированное программирование. 2 Парадигмы ООП. 3 Классы и объекты.	2						
	Основы синтаксиса Java. Java Code Convention 1 Параметры и методы. Модификаторы области видимости. 2 Операторы цикла и ветвления. 3 Требования к оформлению кода.	**2**			**2**			Защита отчета по лабораторной работе
	Наследование в Java. Интерфейсы 1 Правила и механизмы наследования. 2 Понятие интерфейса. 3 Правила реализации интерфейсов.	**2**			**2**			Защита отчета по лабораторной работе
	Работа с файлами. Сериализация 1 Класс File 2 Потоки ввода/вывода 3 Сериализация. Интерфейс Serializable	**2**			**2**			Защита отчета по лабораторной работе
	Коллекции в Java 1 Списки 2 Множества 3 Коллекции ключ-значение Форматы хранения файлов.	**2**						
	Итого по дисциплине	**10**			**6**			Экзамен 7 семестр

Старший преподаватель М.С. Данильченко

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА
(заочная интегрированная форма обучения на основе среднего специального образования)

Номер раздела, темы, занятия	Название раздела, темы, занятия; перечень изучаемых вопросов	Количество аудиторных часов					Кол-во часов УСР	Формы контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
1	Введение 1 Объектно-ориентированное программирование. 2 Парадигмы ООП. 3 Классы и объекты.	2						
2	Основы синтаксиса Java. Java Code Convention 1 Параметры и методы. Модификаторы области видимости. 2 Операторы цикла и ветвления. 3 Требования к оформлению кода.	**2**			**2**			Защита отчета по лабораторной работе
3	Наследование в Java. Интерфейсы 1 Правила и механизмы наследования. 2 Понятие интерфейса. 3 Правила реализации интерфейсов.	**2**			**2**			Защита отчета по лабораторной работе
4	Работа с файлами. Сериализация 1 Класс File 2 Потоки ввода/вывода 3 Сериализация. Интерфейс Serializable	**2**			**2**			Защита отчета по лабораторной работе
5	Коллекции в Java	**2**						

	1 Списки 2 Множества 3 Коллекции ключ-значение Форматы хранения файлов.							
	Итого по дисциплине	**10**			**6**			Экзамен 5 семестр

Старший преподаватель М.С. Данильченко

ИНФОРМАЦИОННО - МЕТОДИЧЕСКАЯ ЧАСТЬ

Формы контроля знаний

- Реферат.
- Лабораторные работы.
- Экзамен.

Перечень лабораторных работ

1. Основы синтаксиса. Java Code Convention
2. Наследование и интерфейсы
3. Обработка исключений
4. Работа с файлами. Сериализация
5. Коллекции в Java
6. Работа с XML документами
7. Работа с JSON документами
8. Работа с базами данных

Перечень программного обеспечения

1 Java Development Kit
2 Eclipse IDE
3 MySQL
4 Apache Tomcat

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

ОСНОВНАЯ

1. И.Н. Блинов, В.С. Романчик Java. Промышленное программирование. Практическое пособие. - Минск. Универсал пресс, 2007.
2. Нотон П. JAVA:Справ.руководство :Пер.с англ./Под ред.А.Тихонова.-М.:БИНОМ:Восточ.Кн.Компания,1996:Восточ.Кн.Компания.-447с..-(Club Computer).
3. Патрик Нотон, Герберт Шилдт Полный справочник по Java .- McGraw-Hill,1997, Издательство "Диалектика",1997.
4. Майкл Эферган Java: справочник .- QUE Corporation, 1997, Издательство "Питер Ком", 1998.
5. Джо Вебер Технология Java в подлиннике .- QUE Corporation, 1996, "BHV-Санкт-Петербург",1997.
6. Джейсон Мейнджер Java: Основы программирования .- McGraw-Hill,Inc.,1996, Издательская группа BHV, Киев,1997.
7. И.Ю.Баженова Язык программирования Java .- АО "Диалог-МИФИ", 1997.
8. Майкл Томас, Пратик Пател, Алан Хадсон, Доналд Болл(мл.) Секреты программирования для Internet на Java .- Ventana Press, Ventana Communications Group, U.S.A.,1996, Издательство "Питер Пресс", 1997.
9. Кен Арнольд, Джеймс Гослинг Язык программирования Java .- Addison-Wesley Longman,U.S.A.,1996, Издательство "Питер-Пресс", 1997.

ДОПОЛНИТЕЛЬНАЯ

1. Дэвид Флэнэген Java in a Nutshell .- O'Reilly & Associates, Inc., 1997, Издательская группа BHV, Киев, 1998
2. Нейл Бартлетт, Алекс Лесли, Стив Симкин Программирование на Java. Путеводитель .- The Coriolis Group,Inc.,1996, Издательство НИПФ "ДиаСофт Лтд.",1996
3. Крис Джамса Библиотека программиста Java .- Jamsa Press, 1996, ООО "Попурри", 1996
4. Аарон И.Волш Основы программирования на Java для World Wide Web .- IDG Books Worldwide,Inc.,1996, Издательство "Диалектика",1996