Три направления развития платформы Java. Характерные особенности языка Java. Три принципа ООП. Пример. Достоинства и недостатки ООП. Классы и объекты. Свойства объектов. Пример.

- J2ME (Java 2 Micro Edition). Эта платформа специально ориентирована на встроенные приложения и программное обеспечение для специализированных пользовательских цифровых устройств. Разработчику при этом предоставляется некий минимальный объем API.
- J2SE (Java 2 Standard Edition). Данная платформа ориентирована в основном на персональные компьютеры и рабочие станции. Когда говорят о Java 2 или JDK 1.2, обычно имеют в виду именно платформу J2SE.
- J2EE (Java 2 Enterprise Edition). Платформа Java, специально предназначенная для создания приложений уровня предприятия.

Особенности языка Java:

- Простота. Синтаксис языка Java изначально представлял собой упрощённый вариант синтаксиса языка C++. С одной стороны, многие программисты C и C++ легко переходили на Java. С другой стороны, многие традиционно проблемные элементы были исключены, что значительно облегчило написание программ. Например, в языке отсутствуют указатели и адресная арифметика, деструкторы объектов, зато осуществляется автоматическая «сборка мусора». Правда, в настоящее время синтаксис языка усложняется в связи с введением в язык новых технологий и подходов к программированию, но при этом обеспечивается обратная совместимость.
- Объектная ориентированность. В центре внимания находятся объекты и их типы. Особенности Java, связанные с объектами, сравнимы с языком C++, однако существует ряд значительных различий (например, механизм множественного наследования в Java заменен более простой концепцией интерфейсов). В целом Java более строгий объектно-ориентированный язык, чем C++ и Object Pascal.
- Распределённость. В Java изначально присутствовали средства создания распределённых приложений. Причём это не только низкоуровневые средства работы с сокетами (на основе протоколов TCP и UDP), но и API для работы с протоколами более высоких уровней (например, класс URL позволяет передавать данные по протоколу HTTP). Более того, в Java с начальных версий присутствует технология RMI, позволяющая программам в ходе своей работы вызывать методы объектов, находящихся на других компьютерах.
- Надежность. Значительное внимание в Java уделяется раннему обнаружению возможных ошибок, контролю в процессе выполнения программы, а также устранению ситуаций, которые могут вызвать ошибки.
- Безопасность. Язык Java предназначен для создания программ, работающих в сети. По этой причине большое внимание при его создании было уделено безопасности: в язык встроена настраиваемая система обеспечения безопасности при выполнении кода. Например, эта система предотвращает намеренное переполнение стека выполняемой программы (один из распространенных способов атаки, используемых вирусами), повреждение участков памяти, находящихся за пределами пространства, выделенного процессу, несанкционированное чтение файлов и их модификация.
- Независимость от архитектуры компьютера (также называемая кросс-платформенностью). Компилятор генерирует объектный файл, формат которого

не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах, для ее работы необходима лишь исполняющая система Java.

- Переносимость. В отличии от языков C и C++, ни один из аспектов спецификации Java не зависит от реализации конкретной исполняющей системы. И размер основных типов данных, и арифметические операции над ними строго определены. Например, тип int в языке J ava всегда означает 32-х разрядное целое число. В языках C и C++ тип int может означать как 16-разрядное, так и 32-х разрядное целое число. Фиксированный размер числовых типов позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Бинарные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений, связанных с разным порядком следования байтов на разных платформах. Строковые данные сохраняются в стандартном формате Unicode.
- Многопоточность. Преимуществами многопоточности является более высокая производительность программ в реальном масштабе времени. Простота организации многопоточных вычислений делает язык Java привлекательным для разработки программного обеспечения серверов.
- Динамичность. Язык и основанные на нём технологии активно развиваются, появляются новые библиотеки и средства программирования. В ходе разработки программ на Java чаще всего встречаются файлы следующих типов.
  - Исходные файлы (с расширением java) содержат исходный код программ на Java. Ещё их называют модулями компиляции.
  - Файлы классов (с расширением class) содержат скомпилированные Java-программы – байтовые коды классов, определенные спецификацией Java.
  - Файлы архивов (с расширением jar) содержат наборы файлов, представленные в упакованном виде (в ZIP-формате). В файлы архивов обычно помещаются файлы классов, мультимедиа-файлы, файлы настроек и т.д. Java позволяет непосредственно запускать программы из JAR-файлов.

    Исходный файл на языке Java – это текстовый файл, содержащий в себе одно или несколько описаний классов. Компилятор Java предполагает, что исходный текст программ хранится в файлах с расширениями java. Получаемый в процессе компиляции байт-код для каждого класса записывается в отдельном выходном файле, с именем, совпадающим с именем класса, и расширением class. Байт-коды классов могут далее распространяться самостоятельно в виде пакетов, библиотек и готовых программ. Для выполнения откомпилированных программ требуется т.н. виртуальная машина Java (Java Virtual Machine, JVM). Это среда, обеспечивающая загрузку классов и выполнение их кода, а также функционирование стандартных служб Java. Реализации JVM существуют для различных программно-аппаратных платформ (например, Intel+Windows, Intel+MacOS и т.д.), причём реализации могут быть как бесплатными, так и платными. Именно наличие JVM позволяет обеспечить кросс-платформенность Java-программ. Изначально JVM преобразовывала байт-код в исполняемый машинный код в ходе интерпретации, т.е. Java сочетала черты компиляторов и интерпретаторов. Однако интерпретирование приводит к снижению производительности, поэтому сейчас применяется технология HotSpot, которая при запуске программы вызывает ещё один компилятор, преобразующий байт-код в инструкции процессора. Однако выполнение этого кода всё равно происходит в рамках JVM и при тесном взаимодействии с ней. Тем не менее, производительность Java-программ при этом значительно возрастает. В

настоящее время язык Java как таковой является достаточно сложным инструментом разработки, в который включено большое количество средств программирования, причём многие из них являются неотъемлемой частью языка. Изучение Java может вызывать некоторые затруднения по следующим причинам.

- Во-первых, многие возможности заложены в язык на уровне синтаксиса и оказываются взаимосвязанными. Так, чтобы полностью понять, как описываются методы в классах (вообще-то, базовый элемент синтаксиса в объектно-ориентированных языках), требуется понимание особенностей наследования, реализаций интерфейсов, многопоточного программирования и механизма обработки исключительных ситуаций.
- Во-вторых, в ходе развития языка в нём стали появляться элементы синтаксиса, которые выглядят очень просто, однако за ними скрываются достаточно сложные действия, понимание которых требует знания как аспектов языка, так и приёмов программирования в целом. Так, появившийся в Java5 цикл for в форме for-each (достаточно популярной благодаря другим языкам), достаточно естественный для массивов и строк, вообще говоря, основан на паттерне проектирования «итератор», множественном наследовании от интерфейсов и специальном API. В данном пособии для облегчения изучения применён следующий подход. Во-первых, при рассмотрении какого-либо вопроса внимание акцентируется на основных его элементах; при этом детали, связанные с другими темами, могут не упоминаться, или даваться без детального объяснения. Во-вторых, авторы будут придерживаться «хронологического» порядка изложения: сначала будут рассмотрены базовые элементы языка, появившиеся ещё в Java2, а уже потом новшества, появившиеся позднее. В итоге данное пособие скорее имеет характер не справочника, а материала для первоначального изучения.

Естественно, что полностью описать даже базовые, синтаксические элементы Java в рамках небольшой книги невозможно. Поэтому в данном пособии, первом в серии, изложены лишь основы языка, связанные с объектно-ориентированным программированием, причём в стандарте JavaSE4.

Основные принципы ООП

Традиционно называют три основных принципа ООП: инкапсуляцию, наследование и полиморфизм.

В отличие от процедурных языков программирования, где данные и инструкции по их обработке существуют отдельно друг от друга, в объектно-ориентированных языках данные и инструкции объединяются в одной сущности – в объекте. Правда, при этом есть небольшой парадокс: при написании программ вы описываете не конкретные объекты, а их классы, т.е. наборы однотипных объектов. Т.о. инкапсуляция – это объединение данных и методов их обработки в одну сущность, имеющую чёткие границы (собственно, буквальный перевод термина – «заключение в оболочку»). Такой подход приводит к тому, что элементы сущности внутри «оболочки» могут тесно взаимодействовать друг с другом, а вот снаружи «оболочки» доступ должен быть ограничен. Т.е. «оболочка» (как и, например, скорлупа яйца) решает одновременно две задачи: удерживает содержимое внутри как одно целое, а также не даёт проникнуть внутрь несанкционированным образом. При описании класса эта «оболочка» реализуется с помощью разграничения доступа к элементам класса. Поэтому

инкапсуляция – это сокрытие реализации класса и отделение его внутреннего представления от внешнего.
Рассмотрим в качестве примера инкапсуляции настольный персональный компьютер и его содержимое. Внутренние элементы компьютера разнообразны и тесно взаимодействуют друг с другом. При этом они заключены в корпус, не позволяющий вам вмешиваться во все внутренние процессы, но имеющий ряд разъёмов, позволяющих подключить оборудование. Разъёмы и протоколы передачи данных через эти разъёмы, по сути, и определяют внешний интерфейс (внешнее представление) компьютера. Особенности внутренней реализации при этом оказываются скрыты. Нарушение же инкапсуляции, как и везде, может привести к печальным последствиям: если оставить корпус компьютера открытым, туда может попасть какой-нибудь инородный предмет, благодаря которому работа компьютера будет затруднена или станет невозможна (например, не стоит вставлять дополнительную планку памяти в работающий компьютер). С другой стороны, некоторые специалисты держат у себя компьютеры без корпуса в разобранном виде (например, в качестве тестовых стендов для оборудования), что подчёркивает ещё одну особенность инкапсуляции: её иногда можно нарушать, но только если вы знаете, зачем вам это нужно, полностью контролируете ситуацию и понимаете последствия.
Название второго принципа – наследование – иногда приводит к проблемам понимания, т.к. в реальной жизни обычно наследуются объекты (например, фамильные драгоценности), да и наследниками тоже являются объекты (да, вы можете унаследовать цвет глаз вашей матери, но при этом и вы, и ваша мать остаются уникальными объектами). В ООП наследование происходит не между объектами, но между классами. Наследование – отношение между классами, при котором один класс использует структуру и поведение другого (одиночное наследование) или других (множественное наследование) классов. С одной стороны, наследование направлено на облегчение разработки и т.н. повторное использование кода. С другой стороны, при наследовании передаётся не только реализация, но и тип, и эти два аспекта наследования следует различать. При этом обычно класс-наследник как-то модифицирует, конкретизирует родительский класс. Рассмотрим снова компьютеры, а точнее, компьютеры как класс. Общими чертами, например, являются способность получать команды, выполнять вычисления и выводить результат. Наследный от него класс персональных компьютеров подразумевает, кроме уже упомянутого, типичные средства ввода и вывода, некоторые стандартные разъёмы. При этом другой наследный класс, например, суперкомпьютеры, может иметь совсем другие свойства, периферийные устройства и средства управления. Т.е. каждый из этих подклассов, сохраняя общие черты, по-своему конкретизирует особенности устройств. Ноутбуки, в свою очередь, можно считать наследниками персональных компьютеров, предъявляющими дополнительные требования к реализации, форме, весу и т.д. Но при этом они наследуют тип (внешнее представление) своего родительского класса: вряд ли кому нужен ноутбук без монитора, клавиатуры и хотя бы USB-разъёмов. Таким образом, благодаря наследованию классы образуют иерархию, каждый элемент которой дополняет и усложняет вышестоящие.
Обычно наибольшие затруднения в понимании вызывает третий принцип – полиморфизм, т.е. буквально «множественность формы». Формой в данном случае, опять же, является внешнее представление, а многими формами может обладать объект. Действительно, кроме внешн его представления своего собственного класса, объект также обладает, как минимум, внешними представлениями и всех родительских классов (его класс их унаследовал). Как следствие, объект может выступать в качестве экземпляра не только своего непосредственного класса, но и других классов.
Полиморфизм – это способность объекта соответствовать во время выполнения двум или

более возможным типам. Отличительной особенностью полиморфизма (по сравнению с двумя уже рассмотренными принципами) является то, что он говорит уже не только о классах, но и об объектах, существующих во время выполнения программы. Допустим, у вас есть конкретный нетбук (пусть класс нетбуков является дочерним классом ноутбуков). Как нетбук он очень лёгкий и сравнительно недорогой (т.е. ваш нетбук удовлетворяет требованиям класса нетбуков, «принимает их форму»). С другой стороны, он является портативным, объединяет в себе устройства ввода и вывода, т.е. «удовлетворяет форме» (или типу) ноутбуков. С третьей стороны, вы можете решать на нём те же задачи, что и на любом персональном компьютере: работать с текстами, графикой, выполнять вычисления и т.д. Т.е. ваш нетбук может выступать и в качестве экземпляра класса персональных компьютеров. А с четвёртой стороны, поскольку «внешняя оболочка» нетбука достаточно твёрдая, им можно забивать гвозди. Иначе говоря, нетбук также удовлетворяет типу «небольшие твёрдые предметы», объекты которого вполне допускают своё использование в качестве молотка. В данном случае особо примечателен тот факт, что конкретно это «внешнее представление» объекта было получено из другой иерархии наследования (даже ноутбуком забивать гвозди уже не так удобно). Т.е. особенную красоту полиморфизм приобретает при использовании именно множественного наследования. Но самым важным моментом является даже не то, что нетбук можно использовать в различных целях и различным образом, а то, что способ использования (т.е. используемая «форма» объекта, его тип) может меняться в течение его жизни. Произведён он был как нетбук (создан объект класса нетбуков), но вы можете что-то на нём посчитать (как на персональном компьютере), потом перенести его (как ноутбук), потом снова посчитать, а потом позабивать гвозди. Второй стороной полиморфизма является то, что если вам нужен объект некоторого типа, вам могут «подсунуть» любой объект, удовлетворяющий этому типу. Например, если вам понадобился доступ в интернет, ближайшей возможностью может оказаться ваш домашний персональный компьютер, компьютер в интернет-кафе, ноутбук вашего друга, или вообще какой-нибудь смартфон, который даже компьютером в классическом смысле слова не является (но в его внешнем представлении есть операции по доступу в интернет).

Достоинства ООП

Упрощение разработки обуславливается следующими особенностями.

- Разделение функциональности: чем больше и сложнее программная система, тем важнее становится разделение её на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от мелких деталей. Для этой цели классы являются достаточно удобным инструментом.
- Локализация кода: данные и операции над ними вместе образуют определенную сущность, и они не разносятся по всей программе, как это нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного

обеспечения.

- Инкапсуляция: обеспечивает некоторую модульность кода, что облегчает разделение выполнения задачи между несколькими программистами и обновление версий отдельных компонентов.

Возможность создания легко расширяемых систем обуславливается следующими факторами.

- Обработка разнородных структур данных: благодаря полиморфизму программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.

- Изменение поведения на этапе выполнения: существуют средства динамического (на этапе выполнения) связывания с объектами, поэтому один объект легко может быть заменен другим. Благодаря этому поведение программы, и даже её алгоритмы могут быть изменены на этапе выполнения.
- Полиморфизм и программирование в соответствии с типом: если программа написана таким образом, что используются только методы объектов, объявленные в типе имени переменной (не используются остальные методы действительного класса объекта), то благодаря наследованию и полиморфизму можно изменять и расширять программу, внося изменения локально, без переписывания основной части кода.

Недостатки ООП

- Неэффективность на этапе выполнения. Существует фактор, который влияет на время выполнения: это инкапсуляция данных. Рекомендуется не предоставлять прямой доступ к полям класса, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Кроме того, даже прямой доступ к полям объекта может требовать больших затрат, чем обращение к локальной переменной. Таким образом, возникают некие накладные расходы.
- Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет особый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.
- Излишняя избыточность. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода и время разработки.
- Психологическая сложность проектирования. Как не парадоксально (ведь многие понятия и подходы в ООП взяты из реальной жизни), но у программистов (особенно писавших на процедурных языках) часто возникают значительные сложности при переходе к объектным языкам.
- Техническая сложность документирования и проектирования. Документирование классов – задача более трудная, чем документирование процедур и модулей. Поскольку любой метод может быть переопред елен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод. Кроме того, само написание программы в виде набора классов порождает новые проблемы, обуславливаемые распределением функциональности между классами и выбором порядка и способа взаимодействия объектов классов в ходе решения задачи.

Классы и объекты

Как говорилось ранее, в основном написанное далее будет относиться к J2SE 1.4, пока не будет специально оговорено, что рассматривается JavaSE 5. Программы на языке Java

создаются из классов (classes). Однако основным видом сущностей в запущенной на выполнение программе являются объекты. Зная имя класса, можно создать нужное число его объектов (objects), или, как еще говорят, экземпляров класса. Объекты обладают следующими базовыми характеристиками.
Состояние объекта – значения, описывающие объект, определяются его полями.
Поведение объекта – действия, которые может выполнить объект, определяются его методами.
Сущность или уникальность объекта – позволяет различать объекты одного класса, даже если они обладают одинаковым состоянием и поведением.
Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково и имеют одинаковый набор свойств, описывающих состояние. Каждый объект сохраняет информацию о своем состоянии. Со временем состояние объекта может измениться, но только в результате вызовов методов (если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден).
Ниже приведен пример простого класса, названного Body, объекты которого будут описывать небесные тела.
Пример 1. Простой класс

```
class Body {
    public long idNum;
    public String name;
    public Body orbits;

    public static long nextId = 0;
}
```

В объявлении класса содержится ключевое слово class, за которым следует наименование класса и перечень его членов, заключенных в фигурные скобки. Объявление класса создает новый тип (type), так что ссылки на объекты этого типа в коде могут выглядеть так:

```
Body mercury;
```

Здесь mercury – переменная, способная хранить ссылку на объект типа Body. Выражение не создает объект, а только описывает ссылку, которая способна указывать на объект класса. На протяжении своего существования переменная mercury может ссылаться на множество различных объектов типа Body. Все эти объекты должны быть явным образом созданы.
Таким образом, классы определяют объекты (структуру и поведение). Объект при этом имеет тип, описываемый классом (например, у объекта можно вызвать методы, описанные в классе). Вообще говоря, к типам в Java относятся не только классы (но ещё и интерфейсы, о которых речь пойдёт позднее), поэтому далее в тексте слово «тип» будет употребляться в том случае, если речь идёт и о классах, и об интерфейсах.

Члены класса. Модификаторы объявления класса. Пакеты. Пространства имен. Модуль компиляции.

Члены класса

Класс может содержать члены (members) трех основных категорий.
Поля (fields) – переменные, относящиеся к классу и его объектам и в совокупности определяющие состояние класса или конкретного объекта.
Методы (methods) – именованные фрагменты исполняемого кода класса, обуславливающие особенности поведения объектов класса.
Вложенные типы – объявления классов или интерфейсов, размещенные в контексте объявлений других классов или интерфейсов.
Рассмотрение данного вида элементов классов выходит за рамки настоящего пособия.

Модификаторы объявления класса

В объявлении класса может употребляться ряд служебных слов-модификаторов, придающих классу дополнительные свойства. Эти ключевые слова указываются перед словом class при объявлении.
public. Модификатор public помечает класс признаком общедоступности. Он означает, что в любом коде можно объявлять ссылки на объекты класса и обращаться к его доступным членам. Если модификатор public не задан, класс будет доступен только в контексте пакета, которому принадлежит. Большинством инструментальных средств разработки Java выдвигается требование, чтобы объявление класса с модификатором public находилось в файле с тем же именем, которое присвоено классу, отсюда следует, что файл не может содержать более одного объявления класса, помеченного как public (количество не публичных классов может быть произвольным).
abstract. Класс, обозначенный модификатором abstract, трактуется как неполный, другими словами, создавать экземпляры такого класса запрещено. Подобное свойство класса обычно обусловлено наличием в его объявлении абстрактных методов (снабженных тем же модификатором abstract), которые должны быть реализованы в классах-наследниках.
final. Класс, определенный как final, не допускает наследования.
stictfp (strict floating point). Присутствие в объявлении класса модификатора stictfp означает, что операции с плавающей запятой, предусмотренные методами-членами класса, должны выполняться точно и единообразно всеми виртуальными машинами Java. В противном случае JVM оставляет за собой право использовать особенности конкретного процессора, на котором выполняется программа. Это, с одной стороны, обычно увеличивает скорость выполнения программы, но с другой стороны, при этом одна и та же программа на различных компьютерах может получить различные результаты.
Очевидно, что в объявлении класса не могут использоваться одновременно модификаторы final и abstract. Объявление способно содержать несколько модификаторов, порядок их указания несущественен.

Пакеты

Пакет – это комплект программного обеспечения, который может распространяться независимо и применяться при разработке приложений в сочетании с другими пакетами. Членами пакетов являются взаимосвязанные классы, интерфейсы, вложенные пакеты, а также дополнительные файлы ресурсов (например, графические), используемые в коде классов. Полезность применения пакетов обусловлена несколькими причинами.

- Пакеты позволяют группировать взаимосвязанные классы и интерфейсы в единое целое. Например, библиотечные классы, предназначенные для решения задач статистического анализа, целесообразно объединить в одном пакете.
- Пакеты разделяют пространство имён типов, позволяя избежать конфликтов идентификаторов. Вполне возможна ситуация, когда различные разработчики одинаково назовут свои классы. Если бы при этом пространство имён классов было единым, неизбежно бы возникали конфликты. При использовании пакетов имя класса должно быть уникальным только в рамках пакета.
- Пакеты обеспечивают дополнительные средства защиты элементов кода. Фрагменты кода внутри пакета могут взаимодействовать, используя права доступа, которыми не обладает любой внешний код. Более того, за счёт возможной вложенности пакетов друг в друга, возможно создание иерархий пакетов, с учётом приведённых выше полезных особенностей пакетов.

Каждый исходный файл, в котором размещены объявления классов и интерфейсов, относящихся, например, к пакету attr, должен содержать специальное объявление:

```
package attr;
```

Это объявление должно располагаться в начале текста файла, до объявления классов или интерфейсов. В пределах файла допускается задать только одно объявление package. Наименование любого типа, принадлежащего пакету, неявно0 снабжается префиксом имени этого пакета.

Пример 2. Объявление публичного класса в пакете

```
package skybodies;

public class Body {
    // ...
}
```

Если тип принудительно не объявлен как часть некоторого пакета, он располагается в анонимном, или безымянном пакете (unnamed). Способ оформления кода в виде анонимного пакета вполне приемлем, когда речь идёт об исполняемом приложении (или апплете), которое заведомо не предназначено для вызова из какой бы то ни было сторонней программы. Тексты классов, ориентированных на совместное использование, должны быть размещены в именованных пакетах. Об имени типа, в которое включён префикс названия пакета, отделённый символом точки, говорят как о полном имени типа (например, полным именем класса String является java.lang.String). Имя без указания пакета принято называть простым именем.

При написании кода, которому необходимо обращаться к членам определённого пакета, возможно использование двух подходов. Один из них состоит в употреблении полного имени нужного типа. Он удобен, если вам редко нужно использовать имя типа в коде программы. Однако если учесть, что имена пакетов бывают достаточно длинными, указание полного имени типа даже несколько раз может оказаться утомительным занятием. Другой подход связан с т.н. импортированием пакета или отдельной его части. Например, если вам необходимы элементы пакета attr, нужно поместить в верхней части текста с исходным кодом (после объявления package, если таковое имеется, но перед любыми другими строками) следующую инструкцию импорта:

```
import attr.*;
```
Теперь для ссылки на типы, принадлежащие пакету attr, можно использовать простые имена, например, Attributed. Команду импорта, в которой используется символ *, называют объявлением импорта по требованию (import on demand). Можно воспользоваться также инструкцией импорта единственного типа (single type import):
```
import attr.Attributed;
```
Процедура импортирования кода, принадлежащего текущему пакету, выполняется неявно, поэтому любые типы пакета допускают непосредственное обращение из кода других типов, объявленных в том же пакете.
Также существует один пакет, который всегда импортируется по умолчанию: java.lang. В данном пакете содержатся типы, лежащие в основе языка.
Важно понимать суть механизма импортирования в Java: единственное, что он позволяет делать – это использовать простые имена вместо полных, при этом не происходит никакого копирования исходного кода или байт-кода. Но компилятор, при обнаружении в тексте программы простого имени, которое не может быть найдено в текущем пакете, просмотрит предложения об импортировании в поисках импортированного типа с совпадающим простым именем, или импортированного пакета, в котором есть тип с совпадающим простым именем.

Понятие имени

Имена задаются посредством идентификаторов и указывают на элементы программы (типы, переменные, поля, методы и т.д.). Идентификатор является последовательностью цифр и букв (а с учётом ориентированности Java на Unicode – достаточно разнообразных букв), не может включать в себя символы-разделители (например, точку) и не может начинаться с цифры.
Задача управления именами элементов программы решается посредством двух механизмов. Во-первых, для элементов различных видов рассматриваются различные пространства имён (namespace). Во-вторых, имена, видимые в одной части программы, «скрываются» от других посредством соответствующих контекстов. Так, например, разделение пространств имён позволяет использовать один и тот же идентификатор для поля и метода класса, а средства контекстного сокрытия дают возможность пользоваться одним идентификатором для обозначения переменных-счётчиков всех циклов for.
Существует шесть различных пространств имён:

- пакеты,
- типы,
- поля,
- методы,
- локальные переменные и параметры,
- метки.

Разделение пространств имён обеспечивает гибкие возможности написания программ, но при неразумном использовании оно способен доставить и серьёзные неприятности (см. пример 3).
Пример 3. Душераздирающий, но синтаксически правильный код
```
package Reuse;

class Reuse {
    Reuse Reuse (Reuse Reuse) {
        Reuse:
            for(;;) {
                if (Reuse.Reuse(Reuse)== Reuse)
```

```
                    break Reuse;
                }
            return Reuse;
        }
}
```

Каждое объявление имени определяет контекст, в котором имя может использоваться. Например, контекстом параметра метода служит блок тела этого метода; контекст локальной переменной определяется границами блока, в котором она определена; контекст переменной цикла, объявленной в секции инициализации заголовка цикла for, распространяется на блок тела цикла. Имя нельзя использовать за пределами его контекста – например, один метод класса не способен обращаться к параметрам другого метода. Контексты, однако, могут быть вложенными, и код внутреннего контекста обладает правами доступа ко всем именам внешнего контекста. Например, разрешено обращаться из тела цикла for к локальным переменным, объявленным в пределах того же метода, где находится и цикл for. Подразумевается, что имена, объявленные во внешних контекстах, в некоторых случаях могут быть скрыты именами, принадлежащими внутренним контекстам. Например, имена полей класса способны перекрываться именами локальных переменных, унаследованные поля класса – полями, объявленными в текущем классе.

Не разрешается перекрывать имена во вложенных контекстах внутри одного блока кода. Это означает, что локальная переменная в теле метода не может обладать тем же именем, что и параметр метода; переменную цикла for не разрешается обозначать именем, которое принадлежит локальной переменной или параметру; во вложенном блоке нельзя повторно использовать имена, объявленные во внешнем блоке (см. пример 4).

Пример 4. Неверное объявление переменных во вложенных контекстах

```
{ int a = 0;
    {
        int a = 2; // неверно, переменная уже объявлена
        // …
    }
}
```

Впрочем, ничто не запрещает создавать в пределах блока различные (невложенные) циклы for или применять невложенные блоки с объявлениями одноимённых переменных.

Кроме того, в Java существуют общепринятые правила именования (они являются частью общепринятых правил по оформлению кода). Следование этим правилам, с одной стороны, не проверяется компилятором, но, с другой стороны, следование им облегчит работу с текстом программы и вам, и другим программистам. Пакеты всегда именуются только маленькими буквами. Пакеты, имена которых начинаются с java, являются стандартными пакетами языка (например java.lang, java.util). Пакеты, имена которых начинаются с javax, являются дополнительными пакетами, вошедшими в стандарт языка (например, javax.swing, javax.xml). Прочие пакеты, разрабатываемые компаниями и сообществами, принято именовать в соответствии с названием сайта разработчика. Так, если пакет dom разработан консорциумом W3C, имеющим сайт www.w3c.org, то именем пакета должно быть org.w3c.dom.

Типы именуются с большой буквы, новые слова обозначаются написанием с большой буквы, разделители не используются. Имя типа должно быть говорящим, аббревиатуры обычно используют, только если они общеизвестные. Для классов имена обычно обозначают категорию объектов (например, Student, ArrayIndexOutOfBoundException), а для интерфейсов – либо категорию, либо способность к чему-либо (например, Runnable).

Поля именуются с маленькой буквы, новые слова обозначаются написанием с большой буквы, разделители не используются (например, value, enabled, distanceFromShop).

Методы именуются с маленькой буквы, новые слова обозначаются написанием с большой буквы, разделители не используются (например, calculate, toString, concat). Если метод предназначен только для получения некоторого значения, то его имя должно начинаться со слова get (например, getHeight) или со слова is, если значение булево (например, isVisible).
Т.н. поля-константы (неизменяемые значения, хранящиеся в контексте класса) именуются только большими буквами, для разделения слов используется подчеркивание (например, PI, SIZE_MIN, SIZE_MAX).
Имена локальных переменных и параметров методов должны начинаться с маленькой буквы. В остальном правила именования не накладывают ограничений, но вы должны быть благоразумны: не стоит называть переменные так, чтобы нельзя было понять, что они хранят.

Понятие модуля компиляции

Модуль компиляции хранится в файле с расширением java и является единичной порцией входных данных для компилятора. Состоит из:

- объявления пакета (package MyPackage;);
- выражений импортирования (import java.lang.Double; import java.lang.*;);
- объявлений верхнего уровня (классов и интерфейсов).

Поля. Модификаторы доступа.

Поля

Переменные, объявленные в классе, называют полями (fields). Примером поля может служить переменная name класса Body, рассмотренного выше. Объявление поля состоит из модификаторов, наименования типа переменной, за которым следует её идентификатор и необязательная конструкция инициализации, позволяющая присвоить переменной некое исходное значение.

В примере 1 каждый объект класса Body обладает собственными копиями трех полей: типа long для хранения уникального номера, позволяющего различить объект среди ему подобных, типа String, содержащего ссылку на строку имени объекта, и типа Body, ссылающегося на другой объект того же типа, который представляет небесное тело, вокруг которого обращается текущее. Объект (экземпляр) класса обладает «личными» копиями полей, т.е. собственным – в общем случае, уникальным – состоянием. Поля объекта также называют переменными экземпляра. Например, изменение содержимого поля orbits, принадлежащего одному из объектов Body, не воздействует на одноименные поля других объектов того же типа. В конструкции объявления поля разрешено использовать дополнительные модификаторы, задающие определенные свойства поля:

- модификаторы доступа (будут рассмотрены ниже),
- static,
- final,
- transient – относится к проблеме сериализации (представления объекта в виде последовательности байтов данных) и будет рассмотрен в следующих пособиях,
- volatile – связан с вопросами синхронизации потоков вычислений и управления памятью, будет рассмотрен в следующих пособиях.

В объявлении поля одновременно не могут использоваться модификаторы final и volatile. Поле может быть инициализировано в конструкции объявления с помощью оператора присваивания и значения соответствующего типа. В примере класса Body поле nextID инициализируется нулем. В качестве выражения инициализации допускается применять не только константы, но и имена других полей, конструкции вызова методов или более сложные выражения, состоящие из всех названных элементов вместе. Требуется, чтобы тип выражения инициализации совпадал с типом поля; кроме того, если в выражении используется вызов метода, тот не должен генерировать объявляемые исключения, поскольку в данном случае их «некому» будет отловить и обработать. Ниже приведены примеры допустимых выражений инициализации.

Пример 5. Объявление и инициализация полей

```
double zero = 0.0;
double zeroCopy = zero;
double rootTwo = Math.sqrt(2);
```

Если поле явно не инициализировано, ему присваивается значение соответствующего типа, принятое по умолчанию (см. таблицу 1).

Таблица 1. Значения типов по умолчанию

Тип	Значение по умолчанию
boolean	false
char	\u0000
byte	0
short	0
int	0

long	0
float	0.0f
double	0.0
Ссылочные типы	null

В соответствии с принятым соглашением null выражает тот факт, что объект еще не создан или создан неправильно.
Иногда требуется, чтобы существовала единственная копия поля, общая для всех объектов класса. С этой целью в объявление поля вводится модификатор static (такие поля называют статическими полями или переменными класса). Статическое поле находится в пределах класса в единственном экземпляре, независимо от того, сколько объектов класса было создано и создавались ли таковые вообще. В примере 1 класса Body объявлено одно статическое поле, nextID, предназначенное для хранения очередного доступного для использования порядкового номера объекта. Далее мы увидим, что в каждом создаваемом объекте класса Body переменной idNum будет присваиваться текущее содержимое nextID с последующим увеличением значения nextID на единицу. Для создания объектов Body и ведения учета их порядковых номеров вполне достаточно одного экземпляра переменной nextID.
В контексте «родного» класса к значению статического поля можно обращаться напрямую, но если необходимо обратиться к нему извне, перед идентификатором поля следует указать не ссылку на объект, а имя класса. Например, значение nextID может быть выведено на экран следующим образом:

```
System.out.println(Body.nextID);
```

Собственно говоря, для иллюстрации возможностей доступа к статическим полям вполне подходит и поле out класса System. Для обращения к статическому члену класса разрешено также пользоваться ссылкой на объект этого класса:

```
System.out.println(mercury.nextID);
```

Такой возможностью, однако, не следует злоупотреблять, поскольку в подобном случае создается ложное впечатление, что переменная nextID является полем объекта mercury, а не членом класса Body в целом. При обращении к статическому члену посредством ссылки на объект компилятор определяет имя соответствующего класса, а значением ссылки как таковой может быть даже null.
Значение поля, снабженного модификатором final, после инициализации уже не может быть изменено, так как любая попытка присваивания переменной нового содержимого приводит к ошибке времени компиляции. Модификатор final применятся для обозначения постоянства некоторого свойства класса или объекта на протяжении всего его жизненного цикла. Такие поля исполняют функции именованных констант.
В языке Java часто возникает необходимость в поле, являющимся единственным для всех объектов класса и значение этого поля не может быть изменено. Такое поле называется константой класса и объявляется с помощью ключевых слов static final.
Если поле, помеченное как final, лишено инициализатора, его принято называть blank final. Наличие возможности подобного объявления полезно в тех случаях, когда для инициализации поля простого выражения недостаточно. Поле final должно быть инициализировано только единожды в ходе инициализации класса (в случае, если поле static) либо в процессе конструирования объекта класса (если поле не статическое). Компилятор проверяет, выполнена ли такая операция, и выдает сообщение об ошибке, если выявляет, что поле final не получило соответствующего исходного значения.

Управление доступом

Если бы любой член любого класса или объекта был бы способен напрямую обращаться к элементам другого произвольного класса или объекта, то восприятие, отладка и поддержка такого программного кода была бы крайне затруднительна. Одно из преимуществ ООП состоит в принципе инкапсуляции или сокрытия данных. Чтобы воплотить его в жизнь, необходимы средства языка, позволяющие регламентировать, кто обладает доступом к членам класса либо к типу как таковому. Такими средствами являются модификаторы доступа.

Все члены класса всегда доступны в контексте самого класса. Для указания видимости элементов класса за его пределами предназначены модификаторы доступа четырех различных категорий.

- private. Элементы класса, помеченные как private, доступны только в контексте этого класса.
- package (default). Элементы, объявленные без указания модификатора доступа (т.е. ключевого слова package нет), открыты для самого класса и классов, размещенных в том же пакете.
- protected. Элементы, обозначенные с помощью служебного слова protected, доступны в пределах «родного» класса, классов того же пакета и классов-наследников.
- public. Элементы, объявленные как public, открыты во всех случаях, когда доступен сам класс.

Модификаторы доступа private и protected применимы исключительно по отношению к членам классов, но не к самим классам или интерфейсам (если только последние не являются вложенными). Чтобы обеспечить возможность доступа к члену класса из некоторого блока кода, первым делом следует открыть для доступа класс, которому принадлежит член.

Важно понимать, что контроль доступа осуществляется на уровне классов (или интерфейсов), но не на уровне объектов. Это значит, что, например, даже статические методы класса могут получить прямой доступ даже к приватным полям объекта этого же класса (если, например, в метод была передана ссылка на объект).

Методы. Модификаторы доступа. Метод main.

Методы

Методы класса, как правило, содержат код, который способен адекватно оспринимать состояние объекта и изменять его. В некоторых классах предлагаются поля, помеченные модификатором public или protected, т.е. открытые для непосредственного обращения из стороннего программного кода, но в большинстве случаев подобный подход нельзя признать приемлемым. Большинство объектов предназначено для получения решений, которые нельзя свести к простым элементам данных.
Ниже рассмотрен текст метода main, который непосредственно вызывается и выполнятся виртуальной машиной Java, предусматривающий создание объекта класса Body и вывод на экран содержимого его полей.
Пример 13. Пример метода

```
class BodyPrint {
    public static void main (String[] args) {
        Body sun = new Body("Солнце", null);
        Body earth = new Body("Земля", sun);
        System.out.println("Тело " + earth.name +
            " вращается вокруг тела " + earth.orbits.name
            + " и обладает номером "+ earth.idnum);
    }
}
```

Конструкция объявления метода состоит из двух частей: заголовка метода и его тела. Заголовок в общем случае включает набор модификаторов, наименование типа возвращаемого значения, сигнатуру и предложение throws, перечисляющие классы исключений, которые могут генерироваться методом. Сигнатура метода состоит из наименования (идентификатора) метода и списка (возможно, пустого) типов параметров, заключенного в круглые скобки. В объявлении любого метода должны быть указаны, по меньшей мере, тип возвращаемого значения и сигнатура – модификаторы и список throws необязательны. Тело метода представляет собой набор выражений, ограниченный фигурными скобками.

Модификаторы методов

В объявлении метода применяются модификаторы следующих категорий.

- Модификаторы доступа (уже были рассмотрены ранее).
- abstract. Модификатором abstract помечаются методы, которые точно не определены в контексте текущего класса. В объявлении методов abstract отсутствует тело – оно заменятся символом точки с запятой. Предполагается, что абстрактные методы должны быть реализованы в некотором производном классе.
- static. Рассмотрены ниже.
- final. Методы, обозначенные как final, не допускают переопределения в производных классах.
- synchronized. Метод, помеченный как synchronized, обладает дополнительной семантикой, касающейся проблемы управления вычислительными потоками, одновременно выполняющимися в контексте программного приложения.
- native. Рассмотрены ниже.

- strictfp (strict floating point). Метод, объявленный как strictfp, гарантирует, что все предусмотренные им операции с плавающей запятой, будут выполняться точно и единообразно всеми виртуальными машинами Java. Признак strictfp, содержащийся в объявлении класса, неявно распространяется на все методы класса, манипулирующими числами с плавающей запятой.

Метод abstract не может быть одновременно помечен любым из модификаторов – static, final, synchronized, native или strictfp.

Статические методы

Метод, обозначенный как static, принадлежит контексту класса в целом, а не контекстам экземпляров этого класса. Поэтому подобные методы также принято называть методами класса. Статические методы обычно предназначаются для выполнения задач, общих для класса, например, для получения очередного доступного серийного номера объекта и т.п. Статический метод способен обращаться только к полям и другим методам класса, обозначенным модификатором static, поскольку для доступа к нестатическим членам необходимо наличие ссылки на конкретный объект класса, а так как возможность обращения к this отсутствует, в пределах статического метода подобная ссылка может быть получена только извне (например, передана как параметр метода).

Вызов метода

Если метод нужно вызвать извне объекта, то это делается путём указания ссылки на объект класса (скажем, reference) и наименования метода со списком аргументов (method(arguments)).
Ссылка на объект и наименование метода объекта разделяются оператором точки (.): reference.method(arguments);
В примере класса BodyPrint (пример 13) мы обращались к методу println(), используя статическую ссылку System.out и передавая в качестве аргумента объект типа String, сформированный с помощью оператора сцепления (+) из нескольких строковых операндов.
Каждый метод объявляется с указанием конкретного числа параметров простых или объектных типов. Java не допускает объявления методов с переменным числом параметров (появляется в JavaSE5), хотя подобное ограничение вполне преодолимо – достаточно передать методу в качестве аргумента ссылку на массив объектов. При обращении к методу вызывающий код обязан предоставить набор аргументов, соответствующих по количеству и совместимых по типам со списком параметров в объявлении метода.
В объявлении метода необходимо задавать наименование типа (простого или ссылочного) возвращаемого значения. Если семантика метода такова, что он не должен возвращать каких бы то ни было значений, вместо имени типа возвращаемого значения указывается служебное слово void.
Пример BodyPrint (пример 13) иллюстрирует типичную ситуацию, связанную с проверкой состояния объекта класса. Однако более предпочтительным следует считать решение, в котором вместо прямого обращения к полям объекта в классе предусмотрен некоторый метод, который способен возвратить строковое представление состояния.
Ниже представлен метод в составе класса Body, возвращающий ссылку на объект типа String, который описывает совокупность требуемых полей конкретного экземпляра Body.
Пример 14. Метод преобразования к строке

```
public String toString() {
    String desc = idNum + " (" + name + ") ";
    if (orbits != null)
```

```
            desc += "вращается вокруг " + orbits.toString();
        return desc;
}
```

Операторы + и +=, используемые в тексте метода, выполняют функции сцепления (конкатенации) строк. Сначала создаётся строка desc, содержащая номер небесного тела и его наименование. Если тело является спутником другого тела, в desc добавляется текстовое описание тела более высокого уровня, получаемое с помощью того же метода toString(). Подобный рекурсивный процесс повторяется несколько раз, если иерархия небесных тел сложна (например, Луна-Земля-Солнце) – где каждое предыдущее тело служит спутником следующего, – и завершается по достижении «самого» центрального тела системы, когда поле orbits соответствующего объекта класса Body содержит значение null. Наконец, с помощью команды return строка desc возвращается в то место кода, откуда был осуществлён вызов метода. Обратите внимание на возникший специфический вид рекурсии: формально метод toString() вызывает метод toString(), но при этом методы принадлежат различным объектам и выполняются в различных контекстах. Такая ситуация достаточно часто встречается в ООП.
Метод toString() объекта имеет специальное назначение – он вызывается для получения строкового представления состояния объекта в тех ситуациях, когда в выражениях конкатенации присутствует ссылка на сам объект. Рассмотрим следующие выражения:

```
System.out.println("Тело " +sun);
System.out.println("Тело " +earth);
```

Методы toString() объектов sun и earth будут вызваны неявно и обеспечат вывод на экран следующих результатов:
Тело 0 (Солнце)
Тело 1 (Земля) вращается вокруг 0 (Солнце)
Метод toString() заведомо присутствует в составе любого объекта, независимо от того, был ли он (метод) объявлен в соответствующем классе или нет, поскольку все классы Java наследуют класс Object, а в нём реализован метод toString. Класс Object и вопросы наследования классов будут рассмотрены далее.

Выполнение метода и возврат из него

При вызове метода управление передаётся из того места кода, откуда был осуществлён вызов, в тело метода. Выражения тела метода выполняются в порядке, предусмотренном его семантикой. Метод завершает работу и передаёт управление обратно в код-инициатор в результате возникновения одного из трёх возможных событий: выполнение команды return, достижения конца тела метода (только для методов, «возвращающих» void) или генерации исключения. Если метод предусматривает возврат какого-либо значения, им может быть только одно значение простого или объектного типа. Методы, которые по смыслу должны возвращать несколько значений, способны решить эту задачу одним из следующих способов:

- возвратить ссылку на объект, который содержит поля с требуемыми значениями;
- разместить результаты работы в одном или нескольких объектах, на которые указывают ссылки, переданные методу в качестве аргументов;
- возвратить массив объектов нужного типа;
- разместить результат в public-полях этого же объекта.

Предположим, например, что нам необходимо создать метод, который должен вернуть информацию о том, какие операции с конкретным банковским счётом разрешено выполнять его владельцу. Следует принять во внимание, что владелец счёта может быть наделён несколькими полномочиями (по выполнению приходных, расходных операций и т.п.) одновременно, поэтому метод должен обладать способностью

возвращать группу значений. Решение задачи можно начать с объявления класса Permissions, объекты которого позволяют хранить булевы значения, подтверждающие права владельца на выполнение тех или иных банковских операций.

Пример 15. Вспомогательный класс для передачи набора значений

```
public class Permissions {
    public boolean canDeposit,  // Приход
                   canWithdraw, // Расход
                   canClose;    // Закрытие счёта
}
```

А вот так может выглядеть метод, предусматривающий заполнение полей объекта типа Permissions и возврат его.

Пример 16. Возврат набора значений с помощью объекта вспомогательного класса

```
public class BankAccount {
    private long number; // Номер счёта
    private long balance; // Остаток на счёте
    public Permissions permissionsFor(Person who) {
        Permissions perm = new Permissions();
        perm.canDeposit = canDeposit(who);
        perm.canWithdraw = canWithdraw(who);
        perm.canClose = canClose(who);
        return perm;
    }
    // Объявление методов canDeposit и т.д…
}
```

Любой вызов метода, предусматривающего возврат некоторого значения, должен завершаться либо выполнением соответствующей команды return, передающей в код-инициатор значение, которое может быть присвоено переменной соответствующего типа, либо выбрасыванием объекта исключения. Методом permissionsFor не должен возвращаться, например, объект класса String, поскольку тот не удастся присвоить переменной типа Permissions (в таких ситуациях говорят о несовместимости типов). Но ничто не запрещает объявить в качестве наименования типа, возвращаемого методом permissionsFor, класс Object, совершенно не затрагивая выражения return, так как ссылка на объект Permissions может быть присвоена переменной типа Object (если вспомнить, что класс Object является базовым по отношению ко всем остальным классам Java).

Параметры метода

При вызове метода аргументы передаются «по значению». Другими словами, значения переменных-параметров в теле метода – это копии тех значений, которые код-инициатор передал методу в виде аргументов. Если, например, метод получает в качестве аргументов значение типа double, в теле метода соответствующий параметр сохраняет копию значения, и возможные изменения этой копии в процессе работы метода никоим образом не воздействуют на содержимое переменных в коде-инициаторе. Рассмотрим следующий пример.

Пример 17. Передача аргументов по значению (простой тип)

```
class PassByValue {
    public static void main (String[] args)  {
        double one = 1.0;
        System.out.println("до: one = " + one);
        halveIt(one);
        System.out.println("после: one = " + one);
    }
```

```
    public static void halveIt(double arg) {
        arg /= 2.0; //Разделить значение arg пополам
        System.out.println("половина: arg = " + arg);
    }
}
```

Ниже показан результат работы программы – операция деления значения параметра arg в методе halveIt() не влияет на содержимое переменной one в теле метода main():
до: one = 1.0
половина: arg = 0.5
после: one = 1.0
Следует заметить, что если аргумент представляет собой ссылку на объект, то «по значению» передаётся именно ссылка, но не объект как таковой. Таким образом, внутри тела метода можно присвоить ссылке другой объект, не воздействуя на содержимое исходной ссылки. Но если, пользуясь переданной ссылкой, попробовать изменить значение поля объекта, на который она указывает, или вызвать какой-либо из методов, изменяющих состояние объекта, тот действительно изменит своё состояние с точки зрения любого блока программы, в котором имеются ранее созданные ссылки на объект. Чтобы продемонстрировать названные нюансы, приведём пример.
Пример 18. Передача аргументов по значению (ссылочный тип)

```
class PassRef {
    public static void main (String[] args)  {
        Body venus = new Body ("Венера", null);
        System.out.println("до: " + venus);
        commonName(venus);
        System.out.println("после: " + venus);
    }

    public static void commonName (Body bodyRef) {
        bodyRef.name = "Утренняя звезда";
        bodyRef = null;
    }
}
```

Результат работы программы выглядит так:
до: 0 (Венера)
после: 0 (Утренняя звезда)
Обратите внимание на то, что состояние «внешнего» по отношению к методу commonName() объекта поддалось изменению «изнутри» этого метода; кроме того, переменная venus всё ещё сохранила ссылку на тот же объект класса Body, а в методе commonName() копия bodyRef ссылочной переменной venus получила другое значение null.
В этот момент две переменные, venus в main() и bodyRef в commonName(), указывают на один и тот же объект. Когда метод commonName() вносит изменения в поле bodyRef.name, модифицируется содержимое того же объекта, общего для двух ссылочных переменных. Когда же commonName() присваивает переменной bodyRef значение null, изменяется значение только самой переменной bodyRef; состояние ссылки venus остаётся прежним, поскольку параметр bodyRef – это копия переменной venus, переданная в качестве аргумента по значению. Единственный элемент данных, который в последнем случае подвергается воздействию – это параметр bodyRef как таковой (то же наблюдалось и в предыдущем примере, когда единственной «пострадавшей» стороной в процессе работы метода halveIt() оказался параметр arg). Если бы изменение bodyRef влияло на значение venus в main, строка «после:», выводимая на экран, содержала бы слово null.

Существует ещё одно средство языка, имеющее прямое отношение к обсуждаемой теме: в объявлении метода параметры позволено помечать модификатором final, имея в виду, что значение параметра не может быть изменено в ходе выполнения тела метода. Будь bodyRef объявлен как final, компилятор не позволил бы изменить его значение на null. Если логика программы обусловливает недопустимость изменения значения параметра, достаточно снабдить его признаком final.

Применение методов для управления доступом

Вариант класса Body, предусматривающий использование различных конструкторов, значительно легче применять, чем простую версию класса, в которой содержатся только поля данных, так как мы всегда уверены в том, что поле idNum получит верное значение без дополнительных воздействий извне. Но пользователи класса Body могут обратиться к полям построенного объекта напрямую, поскольку тоже поле idNum объявлено нами публичным, т.е. полностью открыто для доступа извне. Исходя из семантики класса Body содержимое переменой idNum позволяется только «читать». Для реализации модели данных, открытых только для чтения, существуют две возможности: либо обозначить поле как final (и тогда оно приобретёт свойство «только для чтения» на весь период жизни объекта), либо каким-то образом скрыть его от постороннего воздействия. Чтобы «скрыть» поле, достаточно снабдить его модификатором private и создать новый метод, который должен выполнять функцию посредника между полем и внешним кодом.

Пример 19. Метод доступа для чтения

```
class Body {
    private long idNum; // теперь уже private
    public String name = "<Без имени>";
    public Body orbits = null;

    private static long nextId = 0;

    Body() {        idNum = nextID++;
    }

    public long getID() {
       return idNum;
    }
}
```

Теперь пользователь, заинтересованный в получении порядкового номера небесного тела, должен обратиться к методу getID(), который возвращает требуемое значение. Для прикладной программы больше не существует способов изменения содержимого поля, оно фактически приобретает искомое свойство «только для чтения». Но возможности доступа к приватному полю со стороны методов класса Body остаются по-прежнему полными. Методы, управляющие доступом к внутренним полям данных класса, иногда так и называют – методами доступа (accessor methods). Рекомендуется обозначать поля модификатором private и включать в класс специальные методы для присваивания и считывания данных. Если пользователи обладают возможностью обращения к полю класса напрямую, вы не сможете проконтролировать, какие значения ими задаются и предусмотреть все возможные последствия изменения этих значений. Кроме того, поле, открытое для доступа, становится частью контракта (совокупность описаний методов, полей и соответствующей им семантики) класса – позже вы не сможете изменить реализацию класса, не заставив всех обладателей вашего продукта перекомпилировать собственные приложения. О методах, позволяющих задавать (set)

значения полей объекта и считывать (get) их, иногда говорят, что они определяют свойства (properties) объекта.
Возвращаясь к примеру класса Body, снабдим поля name и orbits модификаторами private и предоставим соответствующие методы доступа.
Пример 20. Методы доступа

```
class Body {
    private long idNum;
    private  String name = "<Без имени>";
    private Body orbits = null;

    private static long nextId = 0;

// объявления конструкторов опущены

    public long getId() {
        return idNum;
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
    public Body getOrbits() {
        return orbits;
    }
    public void setOrbits(Body orbitsAround) {
        orbits = orbitsAround;
    }
}
```

Модификатор final подчас применяется в качестве альтернативного средства управления доступом, предотвращающего возможность изменения содержимого полей объекта. Но не следует путать обеспечение устойчивости данных к изменениям с регулированием уровней доступа к данным. Если поле не допускает изменения значений, его следует объявить как final, независимо от того, кто и как может к нему обращаться. И наоборот, если поле не должно стать частью контракта класса, поле надлежит скрыть, не принимая во внимание, позволено модифицировать его содержимое или нет.
Теперь, когда все поля класса Body снабжены модификатором private, вернёмся к ранее высказанной мысли о том, что признак доступности данных – это атрибут класса, а не частного объекта.
Предположим, что одно небесное тело способно «захватить» другое и заставить последнее изменить свою орбиту. Соответствующий метод класса Body может выглядеть так, как показано в примере 21.
Пример 21. Метод с доступом к другому объекту того же класса

```
public void capture (Body victim) {
    victim.orbits = this; // Захват жертвы
}
```

Если бы доступ к полям данных регулировался на уровне объекта, метод захвата объекта capture(), будучи вызванным в одном объекте, не смог бы обратиться к приватному полю orbits другого объекта класса Body, victim (жертва). Но поскольку возможности доступа устанавливаются на уровне класса, код метода класса способен обращаться ко всем полям всех объектов этого класса – ему достаточно иметь в своём распоряжении ссылку на нужный объект (такую, как victim в нашем примере).

Ключевое слово this

В примере 9 упоминалось о том, как можно обеспечить явный вызов одного конструктора из тела другого. Для этого применялось выражение this(), размещённое в самом начале тела конструктора-инициализатора. Служебное слово this, выполняющее роль специальной объектной ссылки, можно использовать в теле нестатического метода для указания на текущий объект, которому этот метод «принадлежит». В рамках статических методов ссылки this не существует, поскольку они вызываются без указания конкретного экземпляра класса. Ссылка this часто используется и в роли носителя информации о текущем объекте, передаваемого в качестве аргумента другим методам. Предположим, что в теле метода требуется добавить текущий объект в список объектов, ожидающих выполнения определённого сервиса. Подобная операция могла бы выглядеть так:

```
service.add(this);
```

В методе capture класса Body ссылка this позволяет задать для поля orbits объекта victim значение, указывающее на текущий объект (пример 21). Ничто не запрещает явно вводить ссылку this перед идентификатором поля или конструкцией вызова метода в коде текущего объекта. Например, строка присваивания `name = bodyName;` в теле конструктора Body с двумя параметрами (пример 9) равноценна следующей:

```
this.name = bodyName;
```

В соответствии с устоявшейся традицией ссылку this указывают только в тех случаях, когда она действительно необходима – например, если наименование поля класса перекрывается в контексте метода именем локальной переменной или параметра. Таким образом, объявление упомянутого выше конструктора Body с двумя параметрами могло бы выглядеть следующим образом.

Пример 22. Применение ключевого слова this

```
Body (String name, Body orbits) {
    this();
    this.name = name;
    this.orbits = orbits;
}
```

Поля name и orbits класса Body в теле конструктора «перекрываются» одноимёнными параметрами. Чтобы получить доступ к полю name (а не к параметру метода name!!!) мы обязаны предпослать идентификатору name префикс «this.», чтобы явно указать на то, что нас интересует именно поле. Подобное умышленное перекрытие одних идентификаторов другими можно считать вполне приемлемой практикой – правда преимущественно при использовании в контексте конструкторов и методов доступа.

Перегруженные методы

Объявление каждого метода включает сигнатуру – сочетание наименования метода и списка типов его параметров. Методы класса должны различаться сигнатурами – они могут обладать одним и тем же именем, но количество и/или типы их параметров в таком случае совпадать не могут. Методы класса, имеющие одно и то же имя, называют перегруженными (overloaded) (такое имя получает несколько возможных толкований). При анализе вызова метода компилятор анализирует количество и типы аргументов и находит тот перегруженный метод, сигнатура которого в наибольшей мере отвечает ситуации. В примере 23 приведены тексты двух перегруженных методов orbitsAround() класса Body, один из которых возвращает значение true в том случае, когда текущее небесное тело вращается вокруг тела, указанного посредством ссылки на объект, а второй выполняет то же самое, только в качестве параметра принимает номер объекта.

Пример 23. Перегруженные методы

```
public boolean orbitsAround(Body other) {
    return (orbits == other);
}

public boolean orbitsAround(long id) {
    return (orbits != null && orbits.idNum == id);
}
```

В объявлении обоих методов указано по одному параметру, но их типы различны. Если при вызове orbitsAround() в качестве аргумента вводится выражение ссылки на объект класса Body, управление предаётся первому из методов – в нём содержимое параметра other сравнивается со значением поля orbits текущего объекта. Если же в конструкции вызова задаётся значение типа long, в действие вступает второй одноимённый метод, сопоставляющий значение поля idNum текущего объекта с содержимым параметра id. Если компилятор не в состоянии найти сигнатуры метода, соответствующей конструкции вызова, он генерирует сообщение об ошибке.

В состав сигнатур не входит тип возвращаемого значения и список объявляемых исключений, поэтому вы не сможете перегрузить методы, руководствуясь различиями только в этих компонентах объявления.

Метод main

Способы запуска программ на выполнение в большой степени зависят от особенностей той или иной операционной системы, но в любом случае, чтобы активизировать приложение, вы обязаны указать имя некоторого класса. При запуске программы система пытается обнаружить в указанном классе метод main() и передать ему управление.

В объявлении метода main должны присутствовать модификаторы public и static, а также служебное слово void, а в списке параметров – единственный параметр типа String[] (ссылка на массив ссылок на объекты типа String). Метод main() объявляется публичным затем, чтобы обратиться к нему мог каждый субъект (в данном случае, виртуальная машина Java) и статическим, чтобы его можно было вызвать без создания объекта класса. Метод main() имеет «возвращаемый тип» void, т.е. он не возвращает никаких значений. В примере 24 приведён метод main(), который выводит на экран значения переданных ему аргументов.

Пример 24. Получение аргументов из командной строки

```
class Echo {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print(args[i] + " ");
        System.out.println();
    }
}
```

Строковые аргументы, передаваемые методу main(), играют роль параметров командной строки, которые могут быть введены пользователем при старте программы. Например, из командной строки приложение Echo может быть запущено следующем образом:

```
java Echo Здесь был Вася
```

В данном случае слово java обозначает наименование программы-интерпретатора байт-кода, Echo соответствует имени класса, подлежащего выполнению, а остальные слова – это аргументы программы. Команда java находит откомпилированный код класса Echo, загружает его в виртуальную машину Java и вызывает метод Echo.main(),

передавая ему полученные извне параметры командной строки, которые сохраняются в массиве args объектов типа String. Результат работы программы выглядит так:
Здесь был Вася
Приложение как совокупность классов может содержать несколько методов main() – таковые допустимо объявлять в любом классе, входящем в приложение, но в каждом конкретном случае запуска приложения используется только один метод main() – он принадлежит классу, наименование которого указано в командной строке (как, например, Echo).

Методы native

Если в процессе реализации Java-проекта возникает необходимость в применении существующего кода, написанного на другом языке программирования, или использовании низкоуровневых функций для непосредственного обращения к компьютерной аппаратуре, существует возможность прибегнуть к так называемым native-методам, которые могут быть вызваны из среды приложения Java, но создаются на одном из «родных» (native) для платформы языков – как правило, C или C++.
Native-методы объявляются посредством модификатора native. Тело метода реализуется на другом языке и поэтому в объявлении заменяется символом точки с запятой. Ниже в качестве примера приведено объявление метода, который обращается к операционной системе за информацией об идентификационном номере процессора хост-компьютера:

```
public native int getCPUID();
```

Единственное отличие native-методов состоит в том, что они реализуются на другом языке программирования. В остальном, они подобны обычным методам, т.е. могут быть переопределены, перегружены и снабжены любыми модификаторами – final, static, synchronized, public, protected или private, кроме abstract или strictfp.
При обращении к методам native свойства переносимости и безопасности, присущие коду Java, будут утрачены. Практически невозможно использовать native-методы в Java-коде, который предназначен для загрузки из Internet или выполнения на удалённых компьютерах сети (примером являются апплеты).
Native-методы реализуются с помощью библиотек API, предлагаемых разработчиками виртуальных машин Java для тех или иных платформ. Одна из стандартных библиотек API, предназначенных для программистов, использующих язык C, носит название JNI – от Java Native Interface. Существуют библиотеки и для других языков. Описание подобных библиотек выходит за рамки данного пособия.

Создание объектов. Конструкторы. Блоки инициализации. Статическая инициализация.

Создание объектов

Создание объектов обычно происходит с помощью оператора new и конструктора класса. Например, объекты, представляющие небесные тела (экземпляры класса Body), могут создаваться и инициализироваться так, как показано в примере 6.

Пример 6. Создание объектов

```
Body sun = new Body();
sun.idNum = Body.nextID++;
sun.name = "Солнце";
sun.orbits = null;

Body earth = new Body();
earth.idNum = Body.nextID++;
earth.name = "Земля";
earth.orbits = sun;
```

Сначала объявляется переменная sun, предназначенная для хранения ссылки на объект типа Body. Подобное объявление не предполагает немедленного создания объекта – оно всего лишь определяет переменную конкретного типа. Объект, на который ссылается переменная sun, создается посредством оператора new. Создание ссылки и создание объекта – различные операции!!! Конструкция, предполагающая использование new – это наиболее употребительный способ создания объектов. Намереваясь создать объект с помощью оператора new, мы задаем имя соответствующего класса и перечисляем требуемые аргументы, если таковые предусмотрены. Исполняющая система выделяет область памяти, необходимую для размещения содержимого полей объекта, и инициализирует их значениями, принятыми по умолчанию. После этого отрабатывают блоки инициализации и конструкторы. По завершению процесса система возвращает ссылку на созданный объект. Если система не находит достаточного фрагмента свободной памяти, она обычно активизирует процесс сборки мусора, чтобы попробовать освободить занятые участки памяти. Если и далее нехватка памяти все ещё ощущается, оператор new генерирует исключение типа OutOfMemoryError.

Объекты, создаваемые с помощью оператора new, нигде в программе явно не удаляются. Виртуальная машина Java всю ответственность за очистку памяти берет на себя, используя механизм сборки мусора, подразумевающий, что недостижимые по ссылкам объекты автоматически уничтожаются без вмешательства прикладной программы. Если объект больше не нужен, вы должны просто перестать на него ссылаться.

Конструкторы

Создаваемый объект приобретает некоторое исходное состояние. Чтобы состояние оказалось корректным, можно предусмотреть явную инициализацию полей класса в момент их объявления либо положиться на значения, предлагаемые по умолчанию. Впрочем, часто случается, что для приведения объекта в требуемое исходное состояние простой инициализации недостаточно: например, для получения данных необходим дополнительный код либо операцию инициализации нельзя свести к простому присваиванию выражений.

Для достижения целей, выходящих за рамки потребностей простой инициализации, в составе класса предусмотрены специальные члены – конструкторы (constructors).

Конструктор – это блок выражений, которые используются для инициализации созданного объекта. Инициализация выполняется до того момента, когда оператор new вернет в вызывающий блок ссылку на объект. Конструкторы обладают тем же именем, что и класс, в составе которого они объявляются. При создании объекта класса следует указывать конструктор, который, подобно обычным методам класса, способен принимать любое (в том числе и нулевое) число аргументов, но в отличие от методов не может возвращать значения какого бы то ни было типа. Конструкторы вызываются после присваивания полям вновь созданного объекта значений по умолчанию и выполнения явных инструкций инициализации полей.

В дополненной версии класса Body, приведенной ниже, объект приводится в исходное состояние как с помощью выражений инициализации, так и посредством конструктора.

Пример 7. Инициализация с помощью конструктора и выражений инициализации

```
class Body {
    public long idNum;
    public String name = "<Без имени>";
    public Body orbits = null;

    private static long nextID = 0;

    Body() {
        idNum = nextID++;
    }
}
```

Объявление конструктора состоит из имени класса, за которым следует список (возможно, пустой) параметров, ограниченный круглыми скобками, и тело конструктора – блок выражений, заключенный в фигурные скобки.

Конструктор класса Body объявлен без параметров. Его назначение – обеспечивать уникальность значения поля idNum создаваемого объекта. В исходном варианте класса (пример 1) небольшая ошибка, связанная, например, с неаккуратно выполненной операцией присваивания значения полю idNum либо с отсутствием инструкций приращения содержимого поля nextID, могла бы привести к тому, что несколько объектов Body получили бы один и тот же порядковый номер. Это было бы ошибкой, так как значения idNum различных объектов класса по задумке должны быть уникальными. Передав ответственность за выбор верных значений idNum самому классу, мы раз и навсегда избавляемся от подобных ошибок. Теперь конструктор класса Body – это единственный субъект, который изменяет содержимое поля nextID и нуждается в доступе к нему. Поэтому мы должны обозначить переменную nextID модификатором private, чтобы предотвратить возможность обращения к ней за пределами класса. Сделав это, мы исключим один из потенциальных источников ошибок, грозящих будущим пользователям нашего класса.

Выражения инициализации переменных name и orbits в теле объявления класса присваивают последним некоторые допустимые и целесообразные значения. Теперь при создании объект автоматически приобретает исходный набор свойств, описывающих его состояние. Далее возможно изменить некоторые из них по своему усмотрению.

Пример 8. Использование конструктора

```
Body sun = new Body();   //  idNum = 0
sun.name = "Солнце";

Body earth = new Body();  //  idNum = 1
earth.name = "Земля";
earth.orbits = sun;
```

В процессе создания объекта с помощью оператора new конструктор класса Body вызывается после присваивания полям name и orbits предусмотренных нами выражений инициализации.
Рассмотренный выше случай – когда заранее, в момент написания программы известно имя астрономического объекта и вокруг какого небесного тела проходит орбита его движения – относительно редок. Вполне резонным будет создать еще один конструктор, который принимает значение имени объекта и ссылки на центральный объект в качестве аргументов (пример 9).
Пример 9. Конструктор с параметрами

```
Body (String bodyName, Body orbitsAround) {
    this();
    name = bodyName;
    orbits = orbitsAround;
}
```

В данном примере один конструктор класса обращается к другому посредством выражения this() – первой исполняемой инструкции в теле конструктора-инициатора. Подобное предложение называют явным вызовом конструктора. Если конструктор, к которому вы намереваетесь обратиться явно, предполагает задание аргументов, при вызове они должны быть переданы. Какой из конструкторов будет вызван – обусловливается количеством аргументов и набором их типов. В данном случае выражение this() означает вызов конструктора без параметров, позволяющего установить значение idNum объекта и увеличить текущее содержимое статического поля nextID на единицу. Обращение к this() дает возможность избежать повторения кода инициализации idNum и изменения nextID. Теперь код, предусматривающий создание объектов, становится существенно более простым.
Пример 10. Применение конструктора с параметрами

```
Body sun = new Body("Солнце", null);
Body earth = new Body("Земля", sun);
```

Версия конструктора, вызываемого в процессе выполнения оператора new, определяется структурой списка передаваемых аргументов.
Если применяется выражение явного вызова конструктора, оно должно быть первой исполняемой инструкцией в теле конструктора-инициализатора. Выражения, используемые в качестве аргументов вызова, не должны содержать ссылки на поля и методы текущего объекта – во всех случаях предполагается, что на этой стадии конструирование объекта еще не завершено.
Применение специализированных конструкторов может быть обусловлено следующими причинами.
1. Без помощи конструкторов с параметрами некоторые классы не в состоянии обеспечить свои объекты приемлемыми исходными значениями.
2. При использовании дополнительных конструкторов задача определения начальных свойств объектов упрощается (наглядный пример – конструктор класса Body с двумя параметрами).
3. Создание объекта зачастую связано с большими издержками из-за некорректного выбора инициализаторов. Например, если объект класса содержит таблицу и вычислительные затраты на ее конструирование велики, совершенно неразумно предусматривать инициализатор по умолчанию, заведомо зная, что позже придется отбросить результат его работы и повторить инициализацию, чтобы придать таблице свойства, нужные в конкретной ситуации. Целесообразнее сразу применить подходящий конструктор, который способен создать таблицу с требуемыми свойствами.

4. Если конструктор не помечен признаком public, круг субъектов, которые могут им воспользоваться для создания экземпляров класса, ограничивается. Можно, например, запретить программистам, использующим пакет, создавать объекты класса, предусмотрев для всех конструкторов класса признак доступа на уровне пакета.
Весьма широкое применение находят и конструкторы без параметров. Если, объявляя класс, вы не создали ни одного конструктора какой бы то ни было разновидности, компилятор автоматически включит в состав класса пустой конструктор без параметров. Подобный конструктор (его называют конструктором по умолчанию) создается только в том случае, если других конструкторов не существует, поскольку можно привести примеры классов, в которых применение конструктора без параметров нецелесообразно или вовсе невозможно. Конструктор по умолчанию получает тот же признак доступа, что и класс, для которого он создается: если объявление класса снабжено модификатором public, то и конструктор будет помечен как public. Если необходимы и конструктор без параметров, и один или несколько дополнительных конструкторов, первый должен быть создан явно.
В тексте конструкторов допускается упоминание объявляемых исключений (см. главу 3). Предложение throws помещается после списка параметров, непосредственно перед открывающей фигурной скобкой, которая отмечает начало тела конструктора. Если в объявлении конструктора присутствует предложение throws, любой метод, который косвенным образом обращается к конструктору при выполнении оператора new, обязан либо обеспечить «отлов» исключений упомянутых типов с помощью соответствующих предложений catch, либо перечислить эти типы в разделе throws собственного объявления.

Блоки инициализации

Еще один способ осуществления сложных операций по инициализации полей объекта связан с использованием т.н. блоков инициализации, которые представляют собой наборы выражений, заключенные в фигурные скобки и размещенные внутри класса вне объявления методов или конструкторов. Блок инициализации выполняется так же, как если бы он был расположен в верхней части тела любого конструктора. Если блоков инициализации несколько, они выполняются в порядке следования в тексте класса. Блок инициализации способен генерировать исключения, если их объявления перечислены в предложениях throws всех конструкторов класса. В примере 11 конструктор без параметров заменен равноценным блоком инициализации.
Пример 11. Блок инициализации класса

```
class Body {
    public long idNum;
    public String name = "<Без имени>";
    public Body orbits = null;

    private static long nextID = 0;

    {
        idNum = nextID++;
    }

    Body (String bodyName, Body orbitsAround) {
        name = bodyName;
        orbits = orbitsAround;
    }
}
```

Теперь конструктор с двумя параметрами не должен выполнять явный вызов конструктора без параметров, как раньше, поскольку необходимые действия совершит блок инициализации. Подобный вариант применения блоков инициализации нельзя назвать самым удачным – мы обратились к нему только для того, чтобы продемонстрировать синтаксис. На практике блоки инициализации используют для выполнения более сложных операций, когда отсутствует необходимость в объявлении конструкторов с параметрами и нет особых причин для обращения к конструктору без параметров. Основное назначение блоков инициализации связано с обеспечением корректного исходного состояния создаваемого объекта, но в реальности можно заставить их выполнять любые необходимые операции – компилятор не проверяет, что именно делается внутри блока инициализации.

Статическая инициализация

Статическим полям класса позволено иметь инициализаторы, но язык предусматривает возможность выполнения более сложных операций в контексте блока статической инициализации. Блок статической инициализации во многом подобен обычному блоку инициализации – отличия состоят в применении модификатора static, возможности обращения только к статическим членам класса и запрете на генерацию объявляемых исключений. Далее приведен пример статической инициализации небольшого массива простых чисел (простым называют натуральное число, которое без остатка делится только на единицу и само на себя).

Пример 12. Статический блок инициализации

```
class Primes {
    static int[] knownPrimes = new int[4];

    static {
        knownPrimes[0] = 2;
        for (int i = 1; i< knownPrimes.length; i++)
            knownPrimes[i] = nextPrime(i);
    }
    // объявление nextPrime...
}
```

Поля класса инициализируются последовательно – сначала выполняются первые по порядку выражение или блок инициализации, затем следующие и т.д. Операции статической инициализации осуществляются при загрузке класса в виртуальную машину. В примере 12 компилятор гарантирует, что массив knownPrimes будет создан еще до того, как начнут выполняться инструкции блока статической инициализации. Подобным образом обеспечивается и то, что к моменту возможного обращения к статическому полю класса извне оно будет проинициализировано корректно.

Комментарии. Простые типы. Массивы. Операторы. Циклы.

Комментарии

Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов «//» и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // это комментарий
```

Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст комментариев символами «/*» и закончив символами «*/». При этом весь текст между этими парами символов будет расценен как комментарий и компилятор его проигнорирует. В Java многострочные комментарии не могут быть вложенными друг в друга: компилятор будет считать, что комментарий заканчивается с первыми встреченными символами «*/».

Пример 25. Многострочный комментарий

```
/*
это тоже комментарий
*/
```

Третья, особая форма комментариев, предназначена для сервисной программы javadoc, которая использует компоненты Java-компилятора для автоматической генерации документации по типам. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением класса, метода или поля документирующий комментарий, нужно начать его с символов «/**». Заканчивается такой комментарий точно так же, как и обычный комментарий – символами «*/». Программа javadoc умеет различать в документирующих комментариях некоторые специальные элементы, имена которых начинаются с символа «@». Вот пример такого комментария.

Пример 26. Комментарии документирования

```
/**
* Этот класс умеет делать замечательные вещи.
* @see java.applet.Applet
* @version 1.2
*/
class CoolApplet extends Applet {
/**
* У этого метода два параметра:
* @param key - это имя параметра.
* @param value - это значение параметра с именем key.
*/
void put (String key, Object value) {
```

Тэг @see создаёт перекрёстную ссылку, указывающую на другой документ javadoc. Тэг @author позволяет включить в комментарий сведения об авторе кода. Тэг @version служит для обозначения версии программного обеспечения. Тэг @param служит для документирования одного параметра метода. Для обозначения каждого параметра надлежит задавать отдельный тэг @param. Первое слово, следующее за тэгом, трактуется как наименование параметра, а остальная часть строки – как его описание.

Служебные слова

Служебные слова не могут применяться в качестве идентификаторов, поскольку правила их употребления строго регламентированы в самом языке. В таблице 2 перечислены служебные слова языка программирования Java.
Таблица 2. Зарезервированные слова Java //заебатая такая таблица
abstract assert boolean break byte
case catch char class const
continue default do double else
extends final finally float for
goto if implements import instanceof
int interface long native new
package private protected public return
short static strictfp super switch
syncronized this throw throws transient
try void volatile while

Слова null, true, и false хотя и выглядят как служебные слова, формально относятся к числу литералов и поэтому в таблицу не включены. Отметим, что слова const и goto зарезервированы в Java, но не используются. Кроме этого, в Java есть «зарезервированные» имена методов (см. таблицу 3). Эти методы наследуются каждым классом из класса Object, поэтому не стоит называть ваши методы одним из этих имён, если вы не хотите переопределить базовую функциональность ваших объектов.
Таблица 3. Зарезервированные имена методов Java
clone equals finalize getClass hashCode
notify notifyAll toString wait

Идентификаторы

Идентификаторы используются для именования типов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами, которые будут описаны ниже. Java – язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE – различные идентификаторы.

Литералы

Константы в Java задаются их литеральным представлением. Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода, где это имеет смысл.

Целочисленные литералы

Целые числа – это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 – это целый литерал. В данном примере приведены десятичные числа, то есть именно те, которые мы повседневно используем вне мира компьютеров. Кроме десятичных, в качестве целочисленных литералов могут использоваться также восьмеричные и шестнадцатеричные числа. Java распознает восьмеричные числа по стоящему впереди нолю. Нормальные десятичные числа не могут начинаться с нуля, так что использование в программе внешне допустимого числа 09

приведет к сообщению об ошибке при трансляции, поскольку 9 не входит в диапазон 0..7, допустимый для знаков восьмеричного числа. Шестнадцатеричная константа различается по стоящим впереди символам 0x или 0X. Диапазон значений шестнадцатеричной цифры – 0..15, причем в качестве цифр для значений 10..15 используются буквы от A до F (или от a до f). С помощью шестнадцатеричных чисел вы можете в краткой и ясной форме представить значения, ориентированные на использование в компьютере, например, написав 0xffff вместо 65535.
Целочисленные литералы являются значениями типа int, которые в Java хранятся в 32-битовых словах. Если вам требуется значение, которое по модулю больше, чем приблизительно 2 миллиарда, необходимо воспользоваться константой типа long. При этом число будет храниться в 64-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву L, указав таким образом, что данное число относится к типу long. Например, 0x7fffffffffffffffL или 9223372036854775807L – это значение, наибольшее для числа типа long.

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо в экспоненциальном форматах. В обычном формате число состоит из некоторого количества цифр, стоящей после них десятичной точки и следующих за ней десятичных цифр дробной части. Например, 2.0, 3.14159 и .6667 – это допустимые значения чисел с плавающей точкой, записанных в стандартном формате.
В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом E или e. Примеры чисел в экспоненциальном формате: 6.022e23, 314159E-05, 2e+100.
В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа double. Если вам требуется константа типа float, справа к литералу надо приписать символ F или f. Если вы любите избыточные определения – можете добавлять к литералам типа double символ D или d. Значения используемого по умолчанию типа double хранятся в 64-битовом слове, менее точные значения типа float – в 32-битовых.

Логические литералы

У логической переменной может быть лишь два значения – true (истина) и false (ложь). Логические значения true и false не преобразуются ни в какое числовое представление. В отличие от C и C++, ключевое слово true в Java не равно 1, а false не равно 0. В Java эти значения могут присваиваться только переменным типа boolean либо использоваться в выражениях с логическими операторами.

Символьные литералы

Символы в Java кодируются индексами в таблице символов Unicode. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (' '). Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов: 'a', 'z', '@'. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей (см. таблицу 4).

Таблица 4. Управляющие последовательности символов
\ddd Восьмеричный символ (ddd)
\uxxxx Шестнадцатиричный символ UNICODE (xxxx)
\' Апостроф
\" Кавычка
\\ Обратная косая черта
\r Возврат каретки (carriage return)
\n Перевод строки (line feed, new line)
\f Перевод страницы (form feed)
\t Горизонтальная табуляция (tab)
\b Возврат на шаг (backspace)

Строковые литералы

Строковые литералы в Java выглядят точно также, как и во многих других языках – это произвольный текст, заключенный в пару двойных кавычек (" "). Хотя строчные литералы в Java реализованы весьма своеобразно (Java создает объект для каждой строки), внешне это никак не проявляется. Примеры строчных литералов:
"Hello World!"; "\" А это в кавычках\"". Все управляющие последовательности и восьмеричные и шестнадцатеричные формы записи, которые определены для символьных литералов, работают точно так же и в строках. Строчные литералы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода. В Java, в отличие от ряда других языков, нет управляющей последовательности для продолжения строкового литерала на новой строке.

Операторы

Оператор – это символ и набор символов, означающий необходимость совершения некоторой операции над предлагаемыми аргументами – операндами. Примером может служить бинарный (имеющий два операнда) оператор +, означающий сложение двух его операндов (чисел). Синтаксически операторы чаще всего размещаются между выражениями (например, между идентификаторами и литералами). Детально операторы будут рассмотрены далее, их перечень приведен в таблице 5.
Таблица 5. Операторы языка Java
+ += - -=
* *= / /=
| |= ^ ^=
& &= % %=
> >= < <=
! != ++ --
>> >>= << <<=
>>> >>>= && ||
== = ~ ?:
() instanceof [ ] .
new

Разделители

Лишь несколько групп символов, которые могут появляться в синтаксически правильной Java-программе, все еще остались неназванными. Это – простые разделители, которые влияют на внешний вид и функциональность программного кода (см. таблицу 6).

Задача разделителей сходна с задачей пробелов – они отделяют элементы программы друг от друга, но конкретные разделители могут применяться только в определённых местах. Некоторые из разделителей выглядят так же, как операторы: например, круглые скобки () являются разделителем, когда используются в заголовке метода, но являются оператором, когда используются при вызове метода. Важно отличать разделители и операторы!

Таблица 6. Разделители

Символы	Название	Для чего применяются
( )	круглые скобки	Выделяют списки параметров в объявлении и вызове метода, также используются для задания приоритета операций в выражениях, выделения выражений в операторах управления выполнением программы.
{ }	фигурные скобки	Содержат значения автоматически инициализируемых массивов, также используются для ограничения блока кода в классах, методах и составных инструкциях.
[ ]	квадратные скобки	Используются в объявлениях массивов.
;	точка с запятой	Разделяет выражения.
,	запятая	Разделяет идентификаторы в объявлениях переменных, также используется при перечислении выражений в заголовке цикла for.
.	точка	Отделяет имена пакетов от имен подпакетов и типов.

Переменные

Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса (поле), доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок, задающих составную инструкцию.

Простые типы

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирования. Переменные простых типов хранят не ссылки на объекты, но непосредственно значения (например, числовые). В Java различают восемь простых типов: byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы.

- Целочисленные. К ним относятся типы byte, short, int и long. Эти типы предназначены для целых чисел со знаком.
- Числовые с плавающей точкой – float и double. Они служат для представления чисел, имеющих дробную часть.
- Символьный тип char. Этот тип предназначен для представления символов, например, букв или цифр.
- Логический тип boolean. Это специальный тип, используемый для представления логических величин.

Java является языком со строгой типизацией. В языке отсутствует автоматическое приведение типа с сужением области определения. Несовпадение типов приводит не к предупреждению при компиляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

Целочисленные типы

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка – знаковые. Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа есть свои естественные области применения.
Исторически сложилось, что на ЭВМ различных архитектур порядок байтов в слове может различаться. Например, в архитектурах SPARC и PowerPC байты хранятся в прямом порядке (старший байт хранится раньше), а для микропроцессоров Intel x86 характерно хранение байтов в обратном порядке (первыми идут младшие байты).
Поскольку программы на Java обладают свойством переносимости (кроссплатформенности), программы «воспринимают» целочисленные значения единообразно, независимо от платформы.
В Java не следует отождествлять разрядность целочисленного типа с занимаемым им количеством памяти. JVM может использовать для ваших переменных то количество памяти, которое сочтёт нужным, лишь бы только их поведение соответствовало поведению типов, указанных вами. Ниже приведена таблица разрядностей и допустимых диапазонов для различных типов целых чисел.
Таблица 7. Разрядность и диапазон значений целых чисел

Имя	Разрядность	Диапазон
long	64	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
int	32	-2 147 483 648 .. 2 147 483 647
short	16	-32 768 .. 32 767
byte	8	-128 .. 127

byte

Тип byte – это знаковый 8-битовый тип. Его диапазон значений – от -128 до 127. Он лучше всего подходит для хранения произвольного набора байтов, загружаемого из сети или из файла.

```
byte b;
byte c = 0x55;
```

Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

short

short – это знаковый 16-битовый тип. Его диапазон – от -32768 до 32767.

```
short s;
short t = 0x55aa;
```

int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений – от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. Всякий раз, когда в одном выражении фигурируют

переменные типов byte, short, int и целые литералы, тип всего выражения перед завершением вычислений приводится к int.

```
int i;
int j = 0x55aa0000;
```

long

Тип long предназначен для представления 64-битовых чисел со знаком.

```
long m;
long n = Ox55aa000055aa0000;
```

Числовые типы с плавающей точкой

Числа с плавающей точкой, часто называемые также вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой float и double и операторов для работы с ними. Характеристики этих типов приведены в таблице 8.

Таблица 8. Разрядность и диапазон значений чисел с плавающей точкой

Имя	Разрядность	Диапазон
double	64	1.7e-308 .. 1.7e+ 308
float	32	3.4e-038 .. 3.4e+ 038

float

В переменных с обычной, или одинарной точностью, объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита. Обратите внимание на то, что после литерала указана буква F: в противном случае литерал будет иметь тип double, что приведёт к ошибке компиляции.

```
float f;
float f2 = 3.14F
```

double

В случае двойной точности, задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Все трансцендентные математические функции, такие, как sin, cos, sqrt, возвращают результат типа double.

```
double d;
double pi = 3.14159265358979323846;
```

Приведение типов

Приведение типов (type casting) – одно из неприятных свойств C++. Тем не менее, приведение типов сохранено и в языке Java. Иногда возникают ситуации, когда у вас есть величина какого-то определенного типа, а вам нужно ее присвоить переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов.

В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении значения переменной типа byte или short в переменную типа int. Это называется расширением (widening) или повышением (promotion), поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа int всегда достаточно для хранения чисел из диапазона, допустимого для типа byte, поэтому в подобных ситуациях оператора явного приведения типа не требуется.

Обратное, в большинстве случаев, неверно, поэтому для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Эту процедуру иногда называют сужением (narrowing), поскольку вы явно сообщаете транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного вам типа.
Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа int) к типу приемника (переменной типа byte). Если бы при такой операции целое значение выходило за границы допустимого для типа byte диапазона, оно было бы уменьшено путем деления по модулю на допустимый для byte диапазон (результат деления по модулю на число – это остаток от деления на это число).

```
int a = 100;
byte b = (byte) a;
```

Автоматическое преобразование типов в выражениях

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.
Пример 27. Автоматическое приведение целочисленных типов

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Результат промежуточного выражения (a * b) вполне может выйти за диапазон допустимых для типа byte значений. Именно поэтому Java автоматически повышает тип каждой части выражения до типа int, так что для промежуточного результата хватает места.
Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений компилятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе компиляции. В нем мы пытаемся записать значение 50 * 2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение об ошибке от транслятора – ведь при занесении int в byte может произойти потеря точности.

```
byte b = 50;
b = b * 2;
```

Корректный код будет выглядеть следующим образом.

```
byte b = 50;
b = (byte) (b* 2);
```

В итоге в b будет занесено корректное значение 100.
Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной – long, то и тип всего выражения тоже повышается до long. Не забывайте, что все целые литералы, в конце которых не стоит символ L (или 1), имеют тип int.
По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип double. Приведенная ниже программа показывает, как повышается тип каждой величины в выражении для достижения соответствия со вторым операндом каждого бинарного оператора.

Пример 28. Автоматическое приведение числовых типов

```
class Promote {
    public static void main (String[] args) {
        byte b = 42;
        char c = 'a';
       short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d =.1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b)+ " + "+ (i / c)+
                           " - " + (d * s));
        System.out.println("result = "+ result);
    }
}
```

Подвыражение f * b – это число типа float, умноженное на число типа byte. Поэтому его тип автоматически повышается до float. Тип следующего подвыражения i / c (int, деленный на char) повышается до int. Аналогично этому тип подвыражения d * s (double, умноженный на short) повышается до double. На следующем шаге вычислений мы имеем дело с тремя промежуточными результатами типов float, int и double. Сначала при сложении первых двух тип int повышается до float и получается результат типа float. При вычитании из него значения типа double тип результата повышается до double. Окончательный результат всего выражения – значение типа double.

Символьный тип

Unicode – это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов. Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа char в этом языке – 16 бит. Переменная этого типа может хранить любой из десятков тысяч символов интернационального набора символов Unicode. Диапазон типа char – от 0 до 65535.

```
char c;
char c2 = 0xf132;
char c3 = 'a';
char c4 = '\n';
```

Хотя величины типа char и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми числами. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной. В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к нему целое число, чтобы получить символьное представление нужной нам цифры.

Пример 29. Арифметические действия с символьными значениями

```
int three = 3;
char one = '1';
char four = (char) (three + one);
```

В результате выполнения этого кода в переменную four заносится символьное представление нужной нам цифры – '4'. Обратите внимание: тип переменной one в приведенном выше выражении повышается до типа int, так что перед занесением результата в переменную four приходится использовать оператор явного приведения типа.

Логический тип

В языке Java есть простой тип boolean, используемый для хранения логических значений. Переменные этого типа могут принимать всего два значения – true (истина) и false (ложь). Значения типа boolean возвращаются в качестве результата всеми операторами сравнения, например (a < b). Кроме того, boolean – это тип, требуемый всеми условными инструкциями, такими как if, while, do.

```
boolean done = false;
```

Массивы

Массивы (arrays) – это упорядоченные наборы элементов одного типа. Элементами массива могут служить значения как простых, так и ссылочных типов. В последнем случае элементами массива будут именно ссылки, в том числе и ссылки на другие массивы: массивы сами по себе являются объектами и наследуют от класса Object.
Объявление

```
int[] ia = new int[3];
```

создаёт ссылочную переменную с именем ia типа int[], соответствующего массиву значений типа int, и помещает в неё ссылку на созданный объект массива из трёх элементов типа int.
В объявлении ссылочной переменной типа массива размерность не указывается. В свою очередь количество элементов объекта массива задаётся при его создании посредством оператора new. Длина массива фиксируется в момент создания и в дальнейшем изменению не поддаётся. Впрочем, ссылочной переменной типа массива в любой момент может быть поставлен в соответствие другой массив того же типа с другой размерностью.
Доступ к элементам массива осуществляется по значениям их номеров-индексов. Если длина массива length, то первый элемент массива имеет индекс равный 0, а последний – length-1. Длину массива можно определить с помощью поля length объекта массива (которое неявно снабжено признаками public и final).
Обращение к элементу массива выполняется путём применения к ссылке на массив оператора доступа к элементу массива (квадратных скобок с заключённым в них выражением, результат которого трактуется как индекс в массиве). Чаще всего ссылка на массив задаётся путём указания имени переменной:

```
ia[10]
```

Однако ссылка может быть получена и другим образом. Например, если метод toArray() имеет возвращаемый тип Object[] (т.е. ссылка на массив ссылок на произвольные объекты), а index – целочисленная переменная, то возможна следующая конструкция:

```
toArray()[index + 5]
```

При каждом обращении к элементу массива по индексу исполняющая система Java проверяет, находится ли значение индекса в допустимых пределах, и генерирует исключение типа ArrayIndexOutOfBoudsException, если результат проверки ложен. Выражение индекса должно относиться к типу int – только этим и ограничивается максимальное количество элементов массива. Ниже приведён пример кода, в котором в цикле выводится на экран содержимое каждого элемента массива ia:
Пример 29. Вывод элементов массива в консоль

```
for (int i = 0; i < ia.length; i++)
    System.out.println(i+ ": " + ia[i]);
```

Массив нулевой длины (т.е. такой, в котором нет элементов) принято называть пустым. Обратите внимание, что ссылка на массив, равная значению null, и ссылка на пустой массив – это совершенно разные вещи. Пустой массив – это реальный массив, в котором попросту отсутствуют элементы. Пустой массив представляет собой удобную

альтернативу значению null при возврате из метода. Если метод способен возвращать null, прикладной код, в котором выполняется обращение к методу, должен сравнить возвращённое значение с null прежде, чем перейти к выполнению оставшихся операций. Если же метод возвращает массив (возможно, пустой), никакие дополнительные проверки не нужны – разумеется, помимо тех, которые касаются длины массива и должны выполняться в любом случае.

Допускается и иная форма объявления ссылок на массивы, в которой квадратные скобки ставятся после идентификатора массива:

```
int ia[] = new int[3];
```

Первый вариант синтаксиса считается более предпочтительным, поскольку формально типом переменной является именно int[].

Правила употребления в объявлениях массивов тех или иных модификаторов обычны. Существует единственная особенность, которую важно помнить: модификаторы применяются к ссылочной переменной, а не к массиву как таковому. Если в объявлении указан признак final, это значит только то, что ссылка на массив не может быть изменена, но это никак не запрещает изменять содержимого элементов массива. В Java нельзя указать никакие модификаторы (скажем, final или volatile) для элементов массива.

Многомерные массивы

В Java поддерживается возможность объявления т.н. многомерных массивов (multidimensional arrays), т.е. массивов, элементами которых служат другие массивы. Код, предусматривающий объявление двумерной матрицы и вывод на экран содержимого её элементов, может выглядеть, например, так, как показано в примере 30.

Пример 30. Создание и работа с двумерным массивом

```
float[][] mat = new float[4][4];
for (int y = 0; y < mat.length; y++) {
    for (int x = 0; x < mat[y].length; x++)
        System.out.print(mat[y][x] + " ");
    System.out.println();
}
```

Типом ссылки в приведённом примере является float[][]. С одной стороны, количество пар скобок означает размерность массива. С другой стороны, пара квадратных скобок означает, что ссылка имеет тип массива, а тип его элементов указан слева от квадратных скобок. Т.о. в нашем примере mat – это, чтобы быть точными, одномерный массив, элементы которого имеют тип float[], т.е. являются ссылками на одномерные массивы с элементами типа float. В свою очередь, конструкция mat[y][x] означает взятие элемента с номером x из массива, получаемого в результате вычисления выражения mat[y], т.е. из элемента с номером y массива mat.

При создании объекта массива должна быть указана, по меньшей мере, его первая, «самая левая», размерность. Другие размерности разрешается не задавать – в этом случае их придётся определить позже, создав объекты «вложенных» массивов. Указание в операторе new единовременно всех размерностей – это самый лаконичный способ создания массива, позволяющий избежать необходимости использования дополнительных операторов new. Выражение объявления и создания массива mat, приведённое в примере 30, равнозначно коду в примере 31. Обратите внимание на то, что в цикле происходит присвоение значений именно одномерного массива mat.

Пример 31. Отложенная инициализация вложенных массивов

```
float[][] mat = new float[4][];
for (int y = 0; y < mat.length; y++)
    mat[y] = new float[4];
```

Такая форма инициализации обладает тем преимуществом, что позволяет наряду с получением массивов с одинаковыми размерностями (скажем, 4x4) создавать и т.н. «непрямоугольные массивы», необходимые для хранения различных по длине последовательностей данных: представьте, что будет, если в цикле будет стоять следующее выражение:

```
  mat[y] = new float[y];
```

При создании объекта массива каждый его элемент получает значение, предусмотренное по умолчанию и зависящее от типа массива: 0 – для числовых типов, '\u0000' – для типа char, false – для boolean и null – для ссылочных типов. Объявляя массив ссылочного типа, мы на самом деле определяем массив переменных этого ссылочного типа, объекты необходимо создавать дополнительно (пример 32).

Пример 32. Создание и инициализация массива объектов

```
Attr[] attrs = new Attr[12];
for (int i = 0; i < attrs.length; i++)
    attrs[i] = new Attr(names[i], values[i]);
```

После выполнения первого выражения, содержащего оператор new, переменная attrs получит ссылку на массив из 12 переменных, которые инициализированы значением null. Объекты типа Attr как таковые будут созданы только в процессе выполнения цикла.

Массив может инициализироваться и не значениями по умолчанию с помощью конструкции в фигурных скобках, в которых перечислены через запятую значения элементов:

```
String[] dangers = {"Львы", "Тигры", "Медведи"};
```

Тот же самый результат может быть достигнут и просто поэлементной инициализацией:

```
String[] dangers = new String[3];
dangers[0] = "Львы";
dangers[1] = "Тигры";
dangers[2] = "Медведи";
```

Первая форма, предусматривающая задание списка инициализаторов в фигурных скобках, не требует явного использования оператора new – он вызывается косвенно исполняющей системой. Длина массива в этом случае определяется количеством значений инициализаторов.

Также допускается и возможность явного использования оператора new, но размерность всё равно следует опускать – она, как и раньше, определяется автоматически:

```
String[] dangers = new String[] {"Львы", "Тигры", "Медведи"};
```

Подобную форму объявления и инициализации массива разрешается применять в любом месте кода, где может находиться ссылка, например, в выражении вызова метода:

```
printStrings(new String[] {"раз", "два", "три"});
```

Массив без названия (т.е. без явно сохраненной ссылки), который создаётся таким образом, называют анонимным (anonymous).

Массивы массивов могут инициализироваться посредством вложенных последовательностей значений. В примере 33 приведено объявление массива, содержащего несколько первых строк т.н. треугольника Паскаля, где каждая строка описана собственным массивом значений.

Пример 33. Явная инициализация непрямоугольного двумерного массива

```
int[][] pascalsTriangle = {
    {1};
    {1, 1};
    {1, 2, 1};
    {1, 3, 3, 1};
    {1, 4, 6, 4, 1};
};
```

Индексы многомерных массивов следуют в порядке от внешнего к внутренним. Так, например, pascalsTriangle[0] ссылается на массив значений типа int, содержащий один

элемент, pascalsTriangle[1] указывает на массив с двумя элементами и т.д.
Класс System содержит метод arraycopy, который позволяет присваивать значения элементов одного массива другому, избегая необходимости использования вложенных циклов.

Операторы

Операторы в языке Java – это специальные символы и их последовательности, которые сообщают компилятору о том, что вы хотите выполнить действие с некоторыми операндами.
Некоторые операторы требуют только одного операнда, и их называют унарными. Одни операторы ставятся перед операндами и называются префиксными, другие – после, и их называют постфиксными. Большинство же операторов ставят между двумя операндами, такие операторы называются инфиксными бинарными операторами.
Существует тернарный оператор, работающий с тремя операндами.
Всего в Java более 40 операторов.
Арифметические операторы используются для вычислений так же, как в алгебре (см. таблицу 9). Допустимые операнды должны иметь числовые типы. Например, использовать эти операторы для работы с логическими типами нельзя, а для работы с типом char можно, поскольку в Java тип char относится к числовым.
Таблица 9. Арифметические операторы

Оператор	Результат
+	Сложение (также унарный плюс)
-	Вычитание (также унарный минус)
*	Умножение
/	Деление
%	Деление по модулю
++	Инкремент (префиксный и постфиксный)
--	Декремент (префиксный и постфиксный)
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Деление по модулю с присваиванием

Операторы арифметических действий

В примере 34 приведена простая программа, демонстрирующая использование арифметических операторов. Обратите внимание на то, что операторы работают как с целыми литералами, так и с переменными.
Пример 34. Использование арифметических операторов

```
class BasicMath {
    public static void main(String args[]) {
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = b - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
```

```
        System.out.println("e = " + e);
    }
}
```

Выполнив эту программу, вы должны получить следующий результат:

a = 2
b = 6
c = 1
d = 4
e = -4

Оператор деления по модулю

Оператор деления по модулю, или оператор mod, обозначается символом %. Этот оператор возвращает остаток от деления первого операнда на второй. В отличие от C++, оператор mod в Java работает не только с целыми, но и с вещественными типами. Программа в примере 35 иллюстрирует работу этого оператора.

Пример 35. Использование оператора деления по модулю

```
class Modulus {
    public static void main (String[] args) {
        int x = 42;
        double y = 42.3;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

Выполнив эту программу, вы получите следующий результат:

x mod 10 = 2
y mod 10 = 2.3

Операторы арифметических действий с присваиванием

Для каждого из арифметических операторов есть форма, в которой одновременно с заданной операцией выполняется присваивание. Пример 36 иллюстрирует использование таких операторов.

Пример 36. Использование арифметических операторов с присваиванием

```
class OpEquals {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = 3;
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

При запуске программы вы получите следующий результат:

a = 6
b = 8

c = 3

Целочисленная арифметика

Арифметические операции с целочисленными операндами выполняются в соответствии с правилами дополнения по модулю, т.е. если значение выходит за границы допустимого диапазона изменения числовых величин соответствующего типа (например, int или long), оно уменьшается по модулю, отвечающему размеру этого диапазона. Поэтому результаты целочисленных операций никогда не приводят к переполнению разрядной сетки (overflow) переменной или исчезновению значащих разрядов (потере значимости – underflow) в ней – разряды числа могут быть только перенесены.
При целочисленном делении остаток отбрасывается (например, 7/2 равно 3 и (-7)/2 равно -3). Результаты применения операторов деления и вычисления остатка по отношению к аргументам целых типов подчиняются следующему правилу:
(x/y)*y + x%y == x
Так что, 7%2 есть 1, а -7%2 есть -1. Если в качестве операнда делителя в выражениях целочисленного деления и вычисления остатка задаётся ноль, операция считается недопустимой и генерируется исключение типа ArithmeticException.

Арифметика с плавающей запятой

Арифметические операции над конечными операндами с плавающей запятой, удовлетворяющими диапазонам точности представления значений float и double, выполняется вполне предсказуемо. Правило присваивания знака результату также традиционно:
при умножении и делении чисел с одним знаком результат положителен, а если знаки различны – отрицателен.
Кроме того, в Java для вещественных типов допустимы значения «положительная бесконечность», «отрицательная бесконечность» и «неопределённость».
В процессе выполнения арифметических действий возможны переполнение разрядной сетки до бесконечности (результат превышает верхнюю границу диапазона изменения значений типа float или double) и потеря значимости – нередко до ноля (результат слишком мал для типа float или double).
В результате вычисления некорректных выражений (таких как, например, деление бесконечности на бесконечность) получается неопределённость – значение NaN («не число», not a number). Сложение двух бесконечностей даёт в результате такую же бесконечность, если их знаки одинаковы, и NaN – если знаки различны. При вычитании бесконечностей одного знака будет получено значение NaN; вычитание бесконечностей с разными знаками даёт в результате бесконечность с тем знаком, который имеется у левого операнда. Например, (∞ - (-∞)) есть ∞. Результат вычисления арифметического выражения, одним из операндов которого является NaN, всегда равен NaN. Переполнение даёт в итоге бесконечность соответствующего знака, а потеря значимости – значение (возможно, нулевое) соответствующего знака.
При выполнении арифметических операций с плавающей запятой также поддерживается отрицательный ноль, -0.0, который в контексте операторов сравнения равен положительному нолю, +0.0. Хотя оба ноля считаются равноценными, в конкретных выражениях они способны приводить к различным результатам. Так, например, результат вычисления выражения 1f/0f равен положительной бесконечности, а выражения 1f/-0f равен отрицательной бесконечности. Если итогом исчезновения значащих разрядов является -0.0 и если -0.0 == 0.0, каким образом можно выявить факт получения отрицательного ноля? Следует поместить тестируемое нулевое значение в выражение, где знак способен себя проявить, и проверить результат.

Пусть, например, x содержит нулевое значение; тогда выражение 1/x будет равно отрицательной бесконечности, если x – отрицательный ноль, и положительной бесконечности – в противном случае.
Правила выполнения операций с бесконечными величинами совпадают с теми, которые приняты в математике. Алгебраическое сложение любой конечной величины с бесконечностью даёт в результате ту же бесконечность. Например, (-∞ + x) равно -∞ при любом конечном значении x.
Бесконечные значения в Java-программе задаются с помощью констант POSITIVE_INFINITY (положительная бесконечность) и NEGATIVE_INFINITY (отрицательная бесконечность), объявленных в классах-оболочках Float и Double. Например, Double.NEGATIVE_INFINITY указывает на версию значения отрицательной бесконечности для типа double. Умножение бесконечности на ноль даёт в результате NaN. При умножении бесконечности на ненулевое конечное значение будет получена бесконечность соответствующего знака.
Операции деления и вычисления остатка в применении к аргументам с плавающей запятой способны давать в результате бесконечные значения или NaN, но никогда не приводят к выбрасыванию исключений. Результаты операций деления и вычисления остатка при различных комбинациях значений аргументов представлены в таблице 10.
Таблица 10. Возможные значения при вещественных операциях

x	y	x / y	x % y
Конечное значение	±0.0	±∞	NaN
Конечное значение	±∞	±0.0	x
±0.0	±0.0	NaN	NaN
±∞	Конечное значение	±∞	NaN
±∞	±∞	NaN	NaN

В остальных случаях оператор вычисления остатка от деления аргументов с плавающей запятой действует аналогично случаю, когда он применятся к целочисленным аргументам.

Инкремент и декремент

В языке Java также существуют операторы, называемые операторами инкремента и декремента (++ и --), являющиеся сокращенным вариантом записи для сложения или вычитания из операнда единицы. Эти операторы уникальны в том плане, что могут использоваться как в префиксной, так и в постфиксной форме. Пример 37 иллюстрирует использование этих операторов.
Пример 37. Префиксная и постфиксная формы инкремента и декремента

```
class IncDec {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = ++b;
        int d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

Обратите внимание на результат выполнения программы (он объясняет различие между постфиксной и префиксной формами):
a = 2
b = 3
c = 4
d = 1

Целочисленные битовые операторы

Для целых числовых типов данных определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. В таблице 11 перечислены такие операторы. Они позволяют работать с каждым битом как с самостоятельной величиной.

Таблица 11. Битовые операторы

Оператор	Результат
~	Побитовое унарное отрицание (NOT)
&	Побитовое И (AND)
|	Побитовое ИЛИ (OR)
^	Побитовое исключающее ИЛИ (XOR)
>>	Сдвиг вправо с заполнением битом знака
>>>	Сдвиг вправо с заполнением нолями
<<	сдвиг влево
&=	Побитовое И (AND) с присваиванием
|=	Побитовое ИЛИ (OR) с присваиванием
^=	Побитовое исключающее ИЛИ (XOR) с присваиванием
>>=	Сдвиг вправо с присваиванием и заполнением битом знака
>>>=	Сдвиг вправо с присваиванием и заполнением нолями
<<=	Сдвиг влево с присваиванием

Операторы битовой арифметики

В таблице 12 показано, как каждый из операторов битовой арифметики воздействует на возможные комбинации битов своих операндов. Пример 38 иллюстрирует использование этих операторов в программе на языке Java.

Пример 38. Операторы битовой арифметики

```
class Bitlogic {
    public static void main(String[] args) {
        String binary[] = { "0000", "0001", "0010",
                            "0011", "0100", "0101",
                            "0110", "0111", "1000",
                            "1001", "1010", "1011",
                            "1100", "1101", "1110",
                            "1111" };
        int a = 3;   // 0+2+1  или двоичное 0011
        int b = 6;   // 4+2+0 или двоичное 0110
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println("a = " + binary[a]);
        System.out.println("b = " + binary[b]);
```

```
        System.out.println("a | b = " + binary[c]);
        System.out.println("a & b = " + binary[d]);
        System.out.println("a ^ b = " + binary[e]);
        System.out.println("~a & b | a & ~b = " +
                           binary[f]);
        System.out.println("~a = " + binary[g]);
    }
}
```

Результат выполнения программы будет выглядеть следующим образом:
a = 0011
b = 0110
a | b = 0111
a & b = 0010
a ^ b = 0101
~a & b | a & ~b = 0101
~a = 1100

Оператор побитового сдвига влево

Оператор << выполняет сдвиг влево всех битов своего левого операнда на число позиций, заданное правым операндом. При этом часть битов в левых разрядах выходит за границы и теряется, а соответствующие правые позиции заполняются нолями. Ранее уже говорилось об автоматическом повышении типа всего выражения до int в том случае, если в выражении присутствуют операнды типа int или целых типов меньшего размера. Если же хотя бы один из операндов в выражении имеет тип long, то и тип всего выражения повышается до long.

Операторы побитового сдвига вправо

Оператор >> означает в языке Java побитовый сдвиг вправо. Он перемещает все биты своего левого операнда вправо на число позиций, заданное правым операндом. Когда биты левого операнда выдвигаются за самую правую позицию слова, они теряются. При сдвиге вправо освобождающиеся старшие (левые) разряды сдвигаемого числа заполняются предыдущим содержимым знакового разряда. Такое поведение называют расширением знакового разряда, а сам оператор – оператором арифметического сдвига (он сохраняет смысл деления числа на 2 и для положительных, и для отрицательных чисел).
В программе в примере 34 байтовое значение преобразуется в строку, содержащую его шестнадцатеричное представление. Обратите внимание: сдвинутое значение приходится маскировать, то есть логически умножать на значение 0x0f, для того, чтобы очистить заполняемые в результате расширения знака биты и понизить значение до пределов, допустимых при индексировании массива шестнадцатеричных цифр.
Пример 39. Арифметический сдвиг вправо

```
class HexByte {
    public static void main(String[] args) {
        char hex[] = {'0', '1', '2', '3',
                      '4', '5', '6', '7',
                      '8', '9', 'a', 'b',
                      'c', 'd', 'e', 'f'};
        byte b = (byte) 0xf1;
        System.out.println("b = 0x" +
```

```
                    hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    }
}
```

Результат работы программы будет следующим:

b = 0xf1

Часто требуется, чтобы при сдвиге вправо расширение знакового разряда не происходило, а освобождающиеся левые разряды просто заполнялись бы нолями. Для этого используют т.н. оператор логического сдвига вправо >>> (см. пример 40).

Пример 40. Логический сдвиг вправо

```
class ByteUShift {
    public static void main(String[] args) {
        char hex[] = {'0', '1', '2', '3',
                      '4', '5', '6', '7',
                      '8', '9', 'a', 'b',
                      'c', 'd', 'e','f'};
        byte b = (byte) 0xf1;
        byte c = (byte) (b >> 4);
        byte d = (byte) (b >> 4);
        byte e = (byte) ((b & 0xff) >> 4);
        System.out.println(" b = 0x" +
               hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
        System.out.println(" b >> 4 =  0x" +
               hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
        System.out.println("b >>> 4 = 0x" +
               hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
        System.out.println("(b & 0xff) >> 4 = 0x" +
               hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
    }
}
```

Для этого примера переменную b можно было бы инициализировать произвольным отрицательным числом, мы использовали число с шестнадцатеричным представлением 0xf1. Переменной c присваивается результат знакового сдвига b вправо на 4 разряда. Как и ожидалось, расширение знакового разряда приводит к тому, что 0xf1 превращается в 0xff. Затем в переменную d заносится результат беззнакового сдвига b вправо на 4 разряда. Можно было бы ожидать, что в результате d содержит 0x0f, однако на деле мы снова получаем 0xff. Это – результат расширения знакового разряда, выполненного при автоматическом повышении типа переменной b до int перед операцией сдвига вправо. Наконец, в выражении для переменной e нам удается добиться желаемого результата – значения 2. Для этого нам пришлось перед сдвигом вправо логически умножить значение переменной b на маску 0xff, очистив таким образом старшие разряды, заполненные при автоматическом повышении типа. Обратите внимание на то, что при этом уже нет необходимости использовать беззнаковый сдвиг вправо, поскольку мы знаем состояние знакового бита после операции AND.

b = 0xf1

b >> 4 = 0xff

b >>> 4 = 0xff

b & 0xff) >> 4 = 0x0f

Операторы битовой арифметики с присваиванием

Так же, как и в случае арифметических операторов, у всех бинарных битовых операторов есть родственная форма, позволяющая автоматически присваивать результат операции левому операнду. В программе в примере 41 создаются несколько

целочисленных переменных, с которыми с помощью операторов, указанных выше, выполняются различные операции.
Пример 41. Побитовые операции с присваиванием

```
class OpBitEquals {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = 3;
        a |= 4;
```

ожидалось, расширение знакового разряда приводит к тому, что 0xf1 превращается в 0xff. Затем в переменную d заносится результат беззнакового сдвига b вправо на 4 разряда. Можно было бы ожидать, что в результате d содержит 0x0f, однако на деле мы снова получаем 0xff. Это – результат расширения знакового разряда, выполненного при автоматическом повышении типа переменной b до int перед операцией сдвига вправо. Наконец, в выражении для переменной e нам удается добиться желаемого результата – значения 2. Для этого нам пришлось перед сдвигом вправо логически умножить значение переменной b на маску 0xff, очистив таким образом старшие разряды, заполненные при автоматическом повышении типа. Обратите внимание на то, что при этом уже нет необходимости использовать беззнаковый сдвиг вправо, поскольку мы знаем состояние знакового бита после операции AND.
b = 0xf1
b >> 4 = 0xff
b >>> 4 = 0xff
b & 0xff) >> 4 = 0x0f

Операторы битовой арифметики с присваиванием

Так же, как и в случае арифметических операторов, у всех бинарных битовых операторов есть родственная форма, позволяющая автоматически присваивать результат операции левому операнду. В программе в примере 41 создаются несколько целочисленных переменных, с которыми с помощью операторов, указанных выше, выполняются различные операции.
Пример 41. Побитовые операции с присваиванием

```
class OpBitEquals {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = 3;
        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Результат работы программы будет следующим:
a = 3
b = 1
c = 6

Операторы сравнения

Для того, чтобы можно сравнивать два значения, в Java имеется набор операторов, описывающих отношения и равенство. Список таких операторов приведен в таблице 13.
Таблица 13. Операторы сравнения
Оператор Результат
== равно
!= не равно
> больше
< меньше
>= больше или равно
<= меньше или равно

Обратите внимание: в языке Java, так же, как в C и C++ проверка на равенство обозначается последовательностью (==). Один знак (=) – это оператор присваивания. Значения любых типов, включая целые и вещественные числа, символы, логические значения и ссылки, можно сравнивать, исполь-зуя оператор проверки на равенство == и неравенство !=. Эти операторы можно применять, если их операнды приводимы по типу, т.е. можно сравнивать ссылки со ссылками, любые числа с любыми числами, но нельзя сравнить ссылку с числом.
Результатом выполнения любого оператора сравнения является логическое значение (т.е. истина или ложь).

Логические операторы

Логические операторы (см. таблицу 14) применимы к операндам типа boolean. Некоторые из них сходны с уже знакомыми вам операторами, но при применении к логическим выражениям они могут иметь немного другой смысл. Все бинарные логические операторы воспринимают в качестве операндов два значения типа boolean и возвращают результат того же типа.
Таблица 14. Булевы логические операторы
Оператор Результат
& Логическое И (AND)
| Логическое ИЛИ (OR)
^ Логическое исключающее ИЛИ (XOR)
|| Оператор OR быстрой оценки выражений (short circuit OR)
&& Оператор AND быстрой оценки выражений (short circuit AND)
! Логическое унарное отрицание (NOT)
&= И (AND) с присваиванием
|= ИЛИ (OR) с присваиванием
^= Исключающее ИЛИ (XOR) с присваиванием
== Равно
!= Не равно
?: Тернарный оператор if-then-else

Программа в примере 42 практически полностью повторяет уже знакомый вам пример класса BitLogic. Только на этот раз мы работаем с булевыми логическими значениями.
Пример 42. Логические операторы

```
class BoolLogic {
    public static void main(String[] args) {
        boolean a = true;
        boolean b = false;
```

```
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a|b = " + c);
        System.out.println("a&b = " + d);
        System.out.println("a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println("!a = " + g);
    }
}
```

Результат выполнения программы будет следующим:

a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false

Существуют два интересных дополнения к набору логических операторов: это альтернативные версии операторов AND и OR, служащие для быстрой оценки логических выражений. Если первый операнд оператора OR имеет значение true, то независимо от значения второго операнда результатом операции будет величина true. Аналогично, в случае оператора AND, если первый операнд – false, то значение второго операнда на результат не влияет – он всегда будет равен false. Если вы используете операторы && и || вместо обычных форм & и |, то Java не производит вычисление правого операнда логического выражения, если ответ ясен из значения левого операнда. Общепринятой практикой является использование операторов && и || практически во всех случаях оценки булевых логических выражений.

Оператор проверки соответствия типу

Оператор instanceof проверяет, принадлежит ли объект некоторому типу. В левой части выражения instanceof указывается ссылка, а в правой – имя класса или интерфейса. Оператор instanceof возвращает true, если выражение левой части совместимо с типом, название которого указано в правой части, и false – в противном случае. Следует заметить, что данный оператор не позволяет выяснить реальный класс, экземпляром которого является объект. С применением оператора instanceof это возможно только если заранее известна иерархия классов, но код программы при этом будет несколько специфичен.

Условный оператор

Общая форма условного оператора такова:

<выражение1> ? <выражение2> : <выражение3>

В качестве первого операнда – <выражение1> – может быть использовано любое выражение, результатом которого является значение типа boolean. Если результат равен true, то выполняется оператор, заданный вторым операндом, то есть, <выражение2>.

Если же первый операнд равен false, то выполняется третий операнд – <выражение3>. Второй и третий операнды, то есть <выражение2> и <выражение3>, должны возвращать значения совместимых типов и не должны иметь тип void.
В программе в примере 43 тернарный оператор используется для проверки делителя перед выполнением операции деления. В случае нулевого делителя возвращается значение 0.
Пример 43. Тернарный условный оператор

```
class Ternary {
    public static void main(String[] args) {
        int a = 42;
        int b = 2;
        int c = 99;
        int d = 0;
        int e = (b == 0) ? 0 : (a / b);
        int f = (d == 0) ? 0 : (c / d);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("a / b = " + e);
        System.out.println("c / d = " + f);
    }
}
```

При выполнении этой программы исключительной ситуации деления на ноль не возникает и выводятся следующие результаты:

```
a = 42
b = 2
c = 99
 d = 0
a / b = 21
c / d = 0
```

Приоритеты операторов

В Java действует определенный порядок, или приоритет операций. В алгебре у умножения и деления более высокий приоритет, чем у сложения и вычитания. В программировании также приходится следить и за приоритетами операций. В таблице 16 указаны в порядке убывания приоритеты всех операций языка Java. Операторы, обладающие одинаковым приоритетом, указаны в пределах одной строки.
Таблица 16. Группы операторов в порядке убывания приоритета

Постфиксные операторы [] . (params) expr++ expr--
Унарные операторы ++expr --expr +expr -expr ~ !
Операторы создания объектов и преобразования типов new (type)expr
Операторы умножения, деления и вычисления остатка * / %
Операторы сложения и вычитания + -
Операторы побитового сдвига << >> >>>
Операторы сравнения < > >= <=
instanceof
Операторы равенства и неравенства == !=
И (AND) &
Исключающее ИЛИ (XOR) ^
Включающее ИЛИ (OR) |
Условное И (AND) &&
Условное ИЛИ (OR) ||

Условный оператор ?:
Операторы присваивания = += -= *= /= %= >>= <<= >>>= &= ^= |=

В таблицы приведены некоторые операторы, которые не обсуждались в разделе 2.10. Круглые скобки с параметрами (params) означают вызов метода с указанием его параметров. Квадратные скобки [] используются для получения доступа к элементам массивов. Оператор . (точка) используется для выделения элементов из ссылки на объект. Круглые скобки с указанием в них типа (type) означают явное приведение типа. Оператор new предназначен для создания новых объектов. Также возможно использование в выражениях круглых скобок для явного указания границ выражений и урегулирования порядка их выполнения. Вы всегда можете добавить в выражение несколько пар скобок, если у вас есть сомнения по поводу порядка вычислений или вам просто хочется сделать свои код более читаемым.
Рассмотрим следующее выражение:
a >> b + 3
Какому из двух выражений, a >> (b + 3) или (a >> b) + 3,
соответствует эта строка? Поскольку у оператора сложения более высокий приоритет, чем у оператора сдвига, правильный ответ – a >> (b + a). Так что если вам требуется выполнить операцию (a >> b)+ 3, без скобок не обойтись.

Управление выполнением метода

Средства управления порядком выполнения инструкций в Java почти идентичны средствам, используемым в C и C++.

Завершение работы метода

Раньше в примерах преимущественно использовался метод main(), не возвращающий значений (формально он имеет возвращаемый тип void). В коде метода можно спользовать инструкцию return, выполнение которой приведет к немедленному завершению работы метода и передаче управления коду, вызвавшему этот метод. Пример 44 иллюстрирует использование инструкции return для немедленного возврата управления, в данном случае – исполняющей среде Java.
Пример 44. Завершение работы метода

```
class ReturnDemo {
    public static void main(String[] args) {
        boolean t = true;
        System.out.println("Before the return");
        //Перед оператором return
        if (t) return;
        System.out.println("This won't execute");
        //Это не будет выполнено
    }
}
```

Зачем в этом примере использована инструкция if (t)? Дело в том, что если бы этого оператора не было, компилятор Java знал бы, что последний вызов println() никогда не будет выполнен. Такие случаи в Java считаются ошибками, поэтому без инструкции if откомпилировать этот пример нам бы не удалось. Заметим, что использование подобных «обманывающих компилятор» инструкций в реальном программировании нежелательно.

Ветвление

В общей форме инструкция ветвления записывается следующим образом:
if (<логическое выражение>) <инструкция1>;
[else <инструкция2>;]
Раздел else, выполняющийся в случае, если логическое выражение принимает значение false, необязателен. На месте любой из инструкций может стоять составная инструкция, заключенная в фигурные скобки. Логическое выражение – это любое выражение, возвращающее значение типа boolean.
В примере 45 приведена программа, в которой ветвление используется для определения того, к какому времени года относится тот или иной месяц.
Пример 45. Ветвления

```
class IfElse {
    public static void main(String[] args) {
        int month = 4;
        String season;
        if (month == 12 || month == 1 || month == 2) {
            season = "Winter";
        }
        else if (month >2 && month <= 5) {
            season = "Spring";
        }
        else if (month > 5 && month <= 8) {
            season = "Summer";
        }
        else if (month > 8 && month <= 11) {
            season = "Autumn";
        }
        else {
            season = "Bogus Month";
        }
        System.out.println("April is in " + season);
    }
}
```

После выполнения программы вы должны получить следующий результат:
April is in Spring

Циклы

Любой цикл можно разделить на 4 части – инициализацию, тело, итерацию и условие завершения. В зависимости от того, какие части вам нужны, вы можете использовать тот или иной вид цикла. В Java есть три циклические конструкции: while (цикл с предусловием), do-while (цикл с постусловием) и for (итерационный цикл).

while

Этот цикл многократно выполняется до тех пор, пока значение логического выражения равно true. Общая форма цикла с предусловием такова:
[инициализация;]
while (условие_продолжения) {
  тело;
[итерация;]
}

Инициализация и итерация необязательны. В примере 46 цикл с предусловием применяется для вывода 10 строк в консоль.

Пример 46. Цикл с предусловием

```
class WhileDemo {
    public static void main(String[] args) {
        int n = 10;
        while (n > 0) {
            System.out.println("tick " + n);
            n--;
        }
    }
}
```

do-while

Иногда возникает потребность выполнить тело цикла, по меньшей мере, один раз – даже в том случае, когда логическое выражение с самого начала принимает значение false. Для таких случаев в Java используется циклическая конструкция do-while. Её общая форма записи такова:

[инициализация;]
do {
  тело;
  [итерация;]
} while (условие_продолжения);

В примере 47 тело цикла выполняется до первой проверки условия завершения. Это позволяет совместить код итерации с условием завершения.

Пример 47. Цикл с постусловием

```
class DoWhile {
    public static void main(String[] args) {
        int n = 10;
        do {
            System.out.println("tick " + n);
        } while (--n < 0);
    }
}
```

for

В виде цикла предусмотрены места для всех четырех его частей.

Общая форма итерационного цикла for такова:

for (инициализация; условие_продолжения; итерация) тело;

Любой цикл, записанный с помощью оператора for, можно записать в виде цикла while, и наоборот. Если начальные условия таковы, что при входе в цикл условие продолжения не выполнено, то инструкции тела не выполняются ни одного раза. В канонической форме цикла for происходит увеличение целого значения счетчика с минимального значения до определенного предела (см. пример 48).

Пример 48. Простой итерационный цикл

```
class ForDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++)
            System.out.println("i = " + i);
    }
}
```

В примере 49 приведён вариант программы, ведущей обратный отсчет.

Пример 49. Итерационный цикл

```
class ForTick {
    public static void main(String[] args) {
        for (int n = 10; n > 0; n--)
            System.out.println("tick " + n);
    }
}
```

Обратите внимание – переменные можно объявлять внутри раздела инициализации инструкции for. Переменная, объявленная внутри for, действует в пределах этой инструкции.

Пример 50 показывает, как можно использовать цикл for при работе с массивами.

Пример 50. Обращение к элементам массива в итерационном цикле

```
class Months {
    static String months[] = {
        "January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December"};
    static int monthdays[] = {31, 28, 31, 30, 31,
        30, 31, 31, 30, 31, 30, 31};
    static String spring = "Spring";
    static String summer = "Summer";
    static String autumn = "Autumn";
    static String winter = "Winter";
    static String seasons[] = {winter, winter, spring,
        spring, spring, summer, summer, summer, autumn,
        autumn, autumn, winter };
    public static void main(String[] args) {
        for (int month = 0; month < 12; month++) {
            System.out.println(months[month] + " is a " +
                seasons[month] + " month with " +
                monthdays[month] + " days.");
        }
    }
}
```

При выполнении эта программа выводит следующие строки:

January is a Winter month with 31 days.
February is a Winter month with 28 days.
March is a Spring month with 31 days.
April is a Spring month with 30 days.
May is a Spring month with 31 days.
June is a Summer month with 30 days.
July is a Summer month with 31 days.
August is a Summer month with 31 days.
September is a Autumn month with 30 days.
October is a Autumn month with 31 days.
November is a Autumn month with 30 days.
December a Winter month with 31 days.

Иногда возникают ситуации, когда разделы инициализации или итерации цикла for требуют нескольких действий. Поскольку составную инструкцию в фигурных скобках в заголовок цикла for вставлять нельзя, Java предоставляет альтернативный путь. Применение запятой (,) для разделения нескольких действий допускается только внутри круглых скобок оператора for. В примере 51 показан простой пример цикла for, в котором в разделах инициализации и итерации выполняется несколько действий.

Пример 51. Итерационный цикл с несколькими действиями при инициализации и

итерации

```
class Comma {
    public static void main(String[] args) {
        int a;
        byte b;
        for (a = 1, b = 4; a < b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
        }
    }
}
```

Вывод этой программы показывает, что цикл выполняется всего два раза:

a = 1
b = 4
a = 2
b = 3

Обратите внимание на то, что в примере 51 объявление переменных цикла вынесено из блока инициализации самого цикла. Связано это с тем, что если бы в блоке инициализации было записано int a =1, byte b = 4, то знак запятой был бы неоднозначен: это разделение инструкций в блоке инициализации или разделение объявления переменных типа int? Поэтому в Java запрещено объявлять в блоке инициализации цикла for переменные различных типов. Впрочем, объявлять много переменных одного типа допускается.

Прерывание блоков инструкций

Инструкция прерывания break означает прекращение выполнения блока инструкций и передачу управления инструкции, следующей за данным блоком. С помощью break можно прервать не только текущий блок, но и указанный блок. Для этого блоку следует назначить имя с помощью метки и указать имя прерываемого блока после инструкции break. Естественно, прервать блок, который в настоящий момент не выполняется, нельзя (это было бы замаскированной версией инструкции безусловного перехода, применение которого в современных языках высокого уровня запрещено).

В программе в примере 52 имеется три вложенных блока, и у каждого своя уникальная метка. Инструкция break, стоящая во внутреннем блоке, приводит к переходу к инструкции, следующей за блоком b. При этом пропускаются два вызова println().

Пример 52. Именованная форма инструкции прерывания

```
class Break {
    public static void main(String[] args) {
        boolean t = true;
        a: {
            b: {
                c: {
                    // Перед break
                    System.out.println(
                        "Before the break");
                    if (t)
                        break b;
                    // Не будет выполнено
                    System.out.println(
                        "This won't execute");
```

```
                }
                // Не будет выполнено
                System.out.println("This won't execute");
            }
            // После b
            System.out.println("This is after b");
        }
    }
}
```

В результате исполнения программы вы получите следующий результат:

```
Before the break
This is after b
```

Инструкцию break можно применять не только к составным инструкциям, как было показано в примере 52, но и к циклам и блокам переключателей (будут рассмотрены позже), причём циклы тоже могут получать имена с помощью меток.

В некоторых ситуациях возникает потребность досрочно перейти к выполнению следующей итерации цикла, проигнорировав часть операторов его тела, ещё не выполненных в текущей итерации. Для этой цели в Java предусмотрена инструкция continue. В примере 53 она используется для того, чтобы в каждой строке печатались два числа: если индекс чётный, цикл продолжается без вывода символа новой строки.

Пример 53. Инструкция перехода к следующему витку цикла

```
class ContinueDemo {
    public static void main(String args[]) {
        for (int i=0; i < 10; i++) {
            System.out.print(i + " ");
            if (i % 2 == 0) continue;
            System.out.println();
        }
    }
}
```

Результат выполнения этой программы следующий:

```
0 1
2 3
4 5
6 7
8 9
```

Как и в случае инструкции break, в continue можно задавать метку, указывающую, в каком из вложенных циклов вы хотите досрочно прекратить выполнение текущей итерации. В примере 54 приведена программа, использующая инструкцию continue с меткой для вывода треугольного фрагмента таблицы умножения для чисел от 0 до 9.

Пример 54. Инструкция перехода к следующему витку цикла с меткой

```
class ContinueLabel {
    public static void main(String[] args) {
        outer: for (int i=0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                if (j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
        }
```

```
    }
  }
```
Инструкция continue в этой программе приводит к завершению внутреннего цикла со счетчиком j и переходу к очередной итерации внешнего цикла со счетчиком i. В процессе работы эта программа выводит следующие строки:

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81

Блок переключателей

Инструкция switch обеспечивает ясный способ переключения между различными частями программного кода в зависимости от значения одной переменной или выражения. Общая форма этой инструкции такова:

```
switch (выражение) {
   case значение1: инструкции1;
   case значение2: инструкции2;
   case значениеM: инструкцииM;
   default: инструкции;
}
```

Результатом вычисления выражения может быть только значение типа int и более узких (byte, short и char). При этом каждое из значений, указанных в предложениях case, должно быть совместимо по типу с выражением в switch. Все эти значения должны быть уникальными литералами или именованными константами

При выполнении вычисляется значение выражения, и оно сравнивается со значениями в предложениях case. Если значения совпадают, выполняются инструкции, начиная с инструкции после двоеточия в соответствующем предложении case. Если же значению выражения не соответствует ни одно из предложений case, управление передается коду, расположенному после ключевого слова default. Отметим, что предложение default необязательно. В случае, когда ни одно из предложений case не соответствует значению выражения и в switch отсутствует предложение default, выполнение программы продолжается с инструкции, следующей за switch. Внутри инструкции switch применение break без метки приводит к переходу на инструкцию, следующую за инструкцией switch. Если break отсутствует, после текущего раздела case будет выполняться инструкция следующего предложения case.

Иногда бывает удобно иметь в операторе switch несколько смежных разделов case, не разделенных оператором break (см. пример 55).

Пример 55. Блок переключателей

```
class SwitchSeason {
    public static void main(String[] args) {
        int month = 4;
        String season;
        switch (month) {
            case 12: // проходит дальше
            case 1:  // проходит дальше
```

```
            case 2:
                season = "Winter";
                break;
            case 3:  // проходит дальше
            case 4:  // проходит дальше
            case 5:
                season = "Spring";
                break;
            case 6:  // проходит дальше
            case 7:  // проходит дальше
            case 8:
                season = "Summer";
                break;
             case 9:  // проходит дальше
             case 10: // проходит дальше
             case 11:
                 season = "Autumn";
                 break;
             default:
                 season = "Bogus Month";
        }
        System.out.println("April is in the "
                           + season + ".");
    }
}
```

В примере 56 приведена программа, где оператор switch используется для передачи управления в соответствии с различными кодами символов во входной строке: программа подсчитывает число строк, слов и символов в текстовой строке.

Пример 56. Блок переключателей

```
class WordCount {
    static String text = "Now is the tifne\n" +
                         "for all good men\n" +
                         "to come to the aid\n" +
                         "of their country\n"+
                         "and pay their due taxes\n";
    static int len = text.length();
    public static void main(String[] args) {
        boolean inWord = false;
        int numChars = 0;
        int numWords = 0;
        int numLines = 0;
        char c;
        for (int i=0; i < len; i++) {
            c = text.charAt(i);
            numChars++;
            switch (c) {
                case '\n': numLines++; // проходит дальше
                case '\t': // проходит дальше
                case ' ' :
                    if (inWord) {
                        numWords++;
                        inWord = false;
                    }
```

```
                    break;
                default: inWord = true;
            }
        }
        System.out.println("\t" + numLines +"\t" +
                           numWords + "\t" + numChars);
    }
}
```

В этой программе для подсчета слов использовано несколько концепций, относящихся к обработке строк. В результате исполнения программы вы получите следующий результат:

```
5       21      94
```

Исключения. Родительский класс исключений. Выбрасывание исключений. Объявляемые и необъявляемые исключения. Пример. Синхронные и асинхронные исключения. Пример. Предложение throws. try, catch и finally.

Общие сведения об исключениях

В процессе выполнения программные приложения встречаются с ошибочными ситуациями различной степени тяжести – например, отсутствует файл или недоступен сетевой адрес. Многие программисты, разрабатывая код, даже не пытаются выявить все возможные ошибочные условия, и на то есть веские основания: если бы при вызове каждого метода программа анализировала все потенциальные источники ошибок, код приобрел бы настолько запутанную форму, которая просто исключила бы возможность его восприятия и дальнейшего развития. Особенно неприятно в такой ситуации то, что в программе оказывается перемешанным код, ради которого она и писалась, и код, отвечающий за анализ и ликвидацию последствий ошибок. С другой стороны, хорошая программа должна продолжать корректно работать при любых обстоятельствах. Таким образом, необходим компромисс между стремлением обеспечить корректность работы программы и вполне естественным желанием сохранить основную программу «незапятнанной» разнообразными проверками.
Механизм исключений (exceptions) предлагает простой способ обработки возникающих ошибок с сохранением удобочитаемости кода. При этом информацию об ошибке можно получить непосредственно, без введения и использования специальных переменных-флагов и анализа различных побочных эффектов. Исключения дают возможность представить ошибки, которые могут сопутствовать выполнению кода метода, в виде зримой части контракта этого метода.
Список подобных исключений доступен программисту, использующему код метода, он проверяется компилятором и при необходимости принимается во внимание теми производными классами, в которых переопределяется метод.
Исключение выбрасывается (throw), если в процессе выполнения кода встречается ошибочная ситуация. Затем исключение отлавливается (catch) отвечающим за него блоком кода. Исключения, не подвергшиеся обработке, приводят к завершению потока вычислений, в котором они возникли. При этом вызывается, если он был установлен, обработчик неотловленных исключений самого потока инструкций, или обработчик в группе потоков, которой принадлежит завершающийся поток. Обычно обработчики по умолчанию только выводят в консоль так называемый путь прохождения ошибки (stack trace), в который включаются имена всех методов, через которые «прошла» ошибка в поисках своего обработчика. Т.о. путь прохождения ошибки позволяет проследить порядок вызова методов, приведший к ошибке, и непосредственно метод и строку, где ошибка возникла. Умение читать такие сообщения очень важно при отладке, т.к. вид ошибки, место и обстоятельства её возникновения помогут вам понять, что же именно произошло.
Исключения – это объекты. Все типы исключений должны наследовать от класса Throwable или от одного из классов, производных от него. Класс Throwable содержит поле типа String, предназначенное для хранения информации об исключении. Обычно новые типы исключений создаются на основе класса Exception, производного от Throwable.

JVM также способна выбрасывать объекты исключений, обычно это исключения двух основных семейств типов: производных от ошибки времени исполнения (класс RuntimeException), либо производных от серьёзных ошибок (класс Error). Исключения типа Error свидетельствуют о весьма серьёзных ошибках, которые обычно не имеет смысла отлавливать, а их последствия, как правило, не поддаются преодолению.
С формальной точки зрения вполне допустимо расширять классы RuntimeException или Error с целью создания собственных необъявляемых исключений, т.е. таких исключений, которые могут быть сгенерированы без предварительного объявления их в предложении throws в методе. Уместность наследования от того или иного класса исключений определяется исходя из логики объявляемых и необъявляемых исключений, а также из задач создаваемого класса исключений.
Большинство типов RuntimeException и Error поддерживает, самое меньшее, по два конструктора: первый не предусматривает задания параметров, а второй в качестве параметра принимает объект String с содержательным описанием природы ошибки. Строка описания может быть получена с помощью метода getMessage(). Примером необъявляемых исключений, наследующих от RuntimeException, могут служить ArithmeticException (возникает при некорректных арифметических действиях, например, при целочисленном делении на ноль) и IndexOutOfBoundsException (возникает при выходе за границы чего-либо, например, при выходе за границы массива возникнет наследное от него исключение ArrayIndexOutOfBoundsException). В свою очередь, примером исключений, наследующих от Error, могут служить LinkageError (возникает, если класс зависит от других классов, которые были изменены после компиляции зависимого класса, например, при попытке загрузить принципиально некорректный байт-код возникнет наследующее от LinkageError исключение ClassFormatError) и VirtualMachineError (возникает в случае ошибок JVM, например, наследующее от него исключение OutOfMemoryException возникает при невозможности создать новый объект по причине нехватки памяти).
Когда говорят об исключениях, чаще всего имеют в виду объявляемые исключения. В случае применения объявляемых исключений компилятор получает возможность проверить, действительно ли метод генерирует только те исключения, которые указаны в его объявлении. Все исключения классов, наследующих Exception, относятся к категории объявляемых. Исключения времени выполнения и объекты ошибок классов RuntimeException, Error и производных от них принадлежат к группе необъявляемых исключений.
Объявляемые исключения служат для представления ситуаций, которые, несмотря на их «исключительный» характер, вполне предсказуемы, и поскольку они происходят, их необходимо уметь преодолевать. Необъявляемые исключения отражают события, которые связаны с ошибками в логике кода и поэтому не могут быть успешно преодолены во время выполнения программы. Например, исключение типа IndexOutOfBoundsException, генерируемое при попытке доступа к элементу массива с неверным значением индекса, уведомляет вас о том, что программа некорректно вычисляет индекс или неправильно проверяет значение, используемое в качестве индекса. Ошибки такого рода должны быть исправлены в исходном коде программы. Если принять во внимание, что ошибку легко допустить в любой строке кода, то будет неразумно пытаться объявлять и отлавливать все исключения, поэтому многие из них и остаются в категории «необъявляемых».
Иногда бывает полезно иметь в своем распоряжении более полную информацию о природе ошибке, не только ту единственную строку текста, которая предусмотрена классом Exception. В таких случаях класс Exception может быть расширен с добавлением недостающих элементов данных, которые обычно передаются конструктору в виде аргументов (см. пример 57).

Предположим, что есть метод, который выполняет замещение текущего значения указанного атрибута новым. Если атрибут с заданным именем не существует, следует предусмотреть действия по выбрасыванию соответствующего исключения, так как вполне резонно полагать, что метод способен иметь дело только с существующими атрибутами. Желательно, чтобы объект исключения располагал наименованием атрибута, виновного в происшедшем. Рассмотрим объявление класса NoSuchAttributeException, помогающего решить задачу.

Пример 57. Класс исключения

```
public class NoSuchAttributeException extends Exception {
    public String attrName;

    public NoSuchAttributeException(String name) {
        super ("Атрибут с именем  \"" + name +
               "\" не найден");
        attrName = name;
    }
}
```

В состав класса NoSuchAttributeException, производного от Exception, добавлены конструктор, принимающий в качестве параметра строку наименования атрибута, и публичное поле, предназначенное для ее хранения. В теле конструктора вызывается конструктор базового класса, которому передается строка описания ошибки. С точки зрения кода, выполняющего обработку ошибок, рассмотренный тип исключений достаточно полезен, так как он содержит и внятное описание ошибки, и наименование атрибута, отсутствие которого привело к ее возникновению. Возможность добавления данных – это только один довод в пользу создания новых типов исключений. Другой, не менее важный, связан с тем, что тип исключения как таковой – это неотъемлемая часть информации об ошибке, поскольку исключения «отлавливаются» в соответствии с их типами. По этой причине класс NoSuchAttributeException стоило создать даже в том случае, если бы нам и не требовались дополнительные поля данных. Вообще говоря, новый тип исключений целесообразно создавать в тех случаях, когда желательно приобрести возможность «отлова» одних исключений, игнорируя при этом другие.

При создании новых типов исключений важно уделять внимание тому, от какого класса исключения вы наследуете, поскольку при этом вы определяете, по сути, подвид существовавшего ранее исключения. Поэтому важно также иметь представление об уже существующих видах исключений, особенно стандартных.

Инструкция throw

Исключения могут быть выброшены методами и операторами, которые вы используете в своей программе. Кроме того, вы можете выбросить исключение в своём коде, для этого используется инструкция throw, имеющая следующий формат:

throw <Выражение>;

где <Выражение> обязано возвращать ссылку на объект типа Throwable. В примере 58 показана генерация описанного ранее исключения NoSuchAttributeException.

Пример 58. Явное выбрасывание исключения

```
public void replaceValue(String name, Object newValue)
                         throws NoSuchAttributeException {
    Attr attr = find(name);
    if (attr == null)
        throw new NoSuchAttributeException(name);
    attr.setValue(newValue);
}
```

В теле метода replaceValue() сначала вызывается метод, выполняющий поиск атрибута по заданному имени. Если поиск не удался, объект исключения типа NoSuchAttributeException вначале создается, с передачей его конструктору строки имени атрибута, а затем выбрасывается. Исключение – это объект, поэтому, прежде всего, его следует явно создать. Если же атрибут с указанным именем существует, его текущее значение заменяется новым.

При выбрасывании исключения выполнение инструкции или выражения, приведшего к исключению, завершается аварийно. Это также приводит к последовательному аварийному завершению всех блоков и методов, участвующих в цепочке вызовов, вплоть до точки программного кода, в которой исключение отлавливается. Если исключение остается необработанным, соответствующий поток вычислений останавливается, но предварительно предоставляется возможность каким-то образом обслужить исключение (с помощью обработчиков исключений потоков и групп потоков). Если исключение возникло, все действия, предусмотренные в программном коде после инструкции или выражения, вызвавших исключение, далее не выполняются. Если исключение было выброшено в процессе вычисления левостороннего операнда выражения, никакие составляющие правостороннего операнда больше не анализируются. Аналогично, если появление исключения спровоцировал левый аргумент в любой части выражения, все аргументы справа от него игнорируются. Следующим будет выполнен либо «завершающий» блок finally, либо блок catch, обрабатывающий исключение.

Результатом работы инструкции throw служит синхронное (synchronous) исключение – то же самое происходит, например, при целочисленном делении на ноль: исключение возникает как непосредственный итог выполнения определенной инструкции, причем неважно, выбрасывается ли оно явно, командой throw, либо косвенно, исполняющей системой, обнаружившей, например, факт деления на ноль. Асинхронное (asynchronous) исключение также возникает при выполнении инструкции, но причина возникновения исключения находится не в этой инструкции.

Асинхронные исключения проявляются в двух ситуациях. Первая связана с внутренними ошибками виртуальной машины Java – такие исключения считаются асинхронными, поскольку их появление провоцируется кодом виртуальной машины, а не инструкциями приложения. Понятно, что с подобными ошибками автору прикладной программы в подавляющем большинстве случаев справиться не удастся. Примером такой ситуации может служить нехватка памяти при создании объектов: место выбрасывания исключения OutOfMemoryException будет определяться, по сути, не выполняемой инструкцией создания объекта, а количеством памяти JVM, указываемом при запуске виртуальной машины.

Ко второй группе причин возникновения асинхронных исключений относится использование устаревших и не рекомендованных для применения методов (например, Thread.stop), а также подобных им разрешённых методов (например, stopThread) из состава Java TM Virtual Machine Debug Interface (JVMDI) – native-интерфейса виртуальной машины, позволяющего проверять состояние работающих приложений Java и управлять ими. Такие методы предоставляют возможность генерирования асинхронных исключений различных категорий (объявляемых и необъявляемых) в любой момент на протяжении цикла выполнения конкретного потока вычислений. Подобные механизмы рискованны и опасны просто по своей внутренней природе, поэтому их применение в большинстве случаев не поощряется.

Предложения throws

В примере 58 в объявлении метода replaceValue() ясно указано, какие объявленные исключения он способен генерировать. Язык требует предоставления подобной информации, поскольку сведения об исключениях, которые могут быть выброшены

методом, важны в такой же степени, как и данные о типе возвращаемого им значения. Поэтому логично в объявлении метода указывать не только тип возвращаемого значения, но и список возможных исключений.

Объявляемые методом исключения указываются с помощью предложения throws в виде списка наименований типов исключений, разделяемых символом запятой. В списке обязаны присутствовать те объявляемые исключения, которые генерируются (тем или иным способом) в методе, но не обрабатываются в его теле. Метод вправе выбрасывать исключения типов, производных от тех, которые объявлены в предложении throws: язык позволяет полиморфным образом использовать объекты производных типов в любом контексте, где предусмотрено задание объектов базового типа. Более того, метод способен генерировать исключения нескольких различных классов, производных от одного базового и при этом обнародовать в throws только один базовый класс. Следует, однако, иметь в виду, что при использовании такого подхода вы лишите программистов, которые будут обращаться к методу, ценной информации о том, какие именно исключения расширенных типов способен выбрасывать метод. С точки зрения удобства использования и возможностей развития кода предложение throws должно быть настолько полным и точным в деталях, насколько это приемлемо в конкретной ситуации.

Часть контракта метода, определяемая предложением throws, строго проверяется на этапе компиляции: метод вправе выбрасывать объявляемые исключения только тех типов, которые явно названы. Выбрасывание объявляемых исключений других типов – как непосредственное, с помощью инструкции throw, так и косвенное, вследствие обращения к другим методам, запрещается. Если в объявлении метода вообще отсутствует предложение throws, это не значит, что не могут быть выброшены какие-либо исключения – речь идёт только о невозможности генерирования объявляемых исключений.

Поскольку типы объявляемых исключений должны быть указаны в предложении throws, отсюда следует, что фрагменты кода, не находящиеся в методах с предложением throws, не вправе выбрасывать объявляемые исключения ни явно, ни косвенно (либо обязаны их явно отлавливать и обрабатывать). Такими фрагментами кода являются блоки инициализации классов (в них можно явно отловить исключение) и инициализаторы полей (никак не могут отлавливать исключения). Однако в действительности блоки инициализации и инициализаторы полей могут выбрасывать объявляемые исключения, но только если блоки и поля не являются статическими, а типы исключений объявлены во всех конструкторах класса. Статические инициализаторы, действительно, никак не могут выбрасывать объявляемые исключения.

Вызывая метод, в определении которого содержится предложение throws с перечнем объявляемых исключений, можете поступить одним из трёх способов:

- отловить исключения и обработать их;
- отловить исключения и вместо них сгенерировать исключения таких типов, которые указаны в предложении throws текущего метода;
- объявить соответствующие исключения в собственном предложении throws и позволить им «пройти» через код незамеченными.

Предложения throws и переопределение методов

При переопределении унаследованного метода или реализации метода абстрактного класса не позволяется задавать в предложении throws нового метода больше объявляемых исключений, чем в исходном методе.

Исключением может быть ситуация, когда список объявляемых исключений дополняется классом, родительский класс которого был объявлен в родительском методе, т.е.

формально новый тип исключения при этом не добавляется. Вообще говоря, переопределенный метод должен сохранить контракт родительского: с одной стороны, он может и не объявлять исключения, объявленные в родительском методе, с другой стороны, он не может объявить тип исключения, который не может быть приведен к одному из типов, объявленных в родительском методе.

Предложения throws и методы native

В объявление native-метода может быть включено предложение throws, которое заставляет код, вызывающий метод, отлавливать или переопределять указанные объявляемые исключения. Однако реализация native-методов находится вне компетенции компилятора Java, который поэтому не в состоянии проверить, действительно ли код метода выбрасывает только те исключения, которые им объявлены.

Блок try-catch-finally

Как уже говорилось, при возникновении исключений виртуальная машина начинает поиск обработчика исключения по цепочке вызова методов (от места возникновения исключения к основному методу потока инструкций). Обработчики исключений указываются с помощью блока try-catch-finally, синтаксис которого выглядит так:

```
try {
    Инструкции – тело блока
} catch (Тип_исключения_1 идентификатор_1) {
    Инструкции – обработка исключения
} catch (Тип_исключения_2 идентификатор_2) {
    Инструкции – обработка исключения
...
} finally {
    Инструкции
}
```

Собственно, блок try-catch-finally и позволяет достигнуть компромисса между точностью работы программы и сложностью кода, о котором говорилось ранее: блок отделяет основной код программы, ради которого она пишется, от обработки всех возможных исключений.

В тело блока try помещается основной код, который выполняется до момента возникновения исключительной ситуации либо до благополучного достижения конца кода блока. Если в процессе работы выбрасывается исключение – либо непосредственно, командой throw, либо косвенно, при вызове метода или оператора – работа блока try прекращается и система последовательно проверяет все предложения catch, пытаясь найти среди них то, тип которого допускает присваивание объекта выброшенного исключения. Если искомое предложение catch найдено, ему в качестве аргумента передаётся объект исключения, после чего выполняется блок кода этого предложения. Другим предложениям catch управление не передаётся. Если соответствующее предложение catch не найдено, исключение передаётся последовательно в блоки try, находящиеся в методах, вызвавших данный метод, в каковых блоках может быть найден требуемый код catch.

Конструкция try-catch-finally может содержать любое количество предложений catch, либо не содержать таковых вовсе, а каждое предложение catch способно отлавливать исключения различных типов. Предложение catch чем-то напоминает «встроенный метод», обладающий единственным параметром типа исключения, подлежащего обработке. Код catch способен выполнить некие восстановительные операции, выбросить собственное исключение, вручая полномочия по обработке ошибки внешнему коду, либо

осуществить всё необходимое, а затем передать управление инструкции, следующей за try (после выполнения блока finally, если таковой имеется).
Задание в предложении catch параметра, относящегося к общему типу исключений (таковому, например, как Exception) – это обычно не очень правильный выбор, поскольку такое предложение способно отлавливать любые исключения, а не только те, которые относятся к исключениям производных типов, рассматриваемым в конкретной ситуации.
Предложение catch, содержащее в качестве параметра один из базовых типов исключений, не может быть расположено перед теми catch, в которых заданы соответствующие производные типы. Предложения catch просматриваются системой поочерёдно, поэтому размещение в начале последовательности тех из них, которые относятся к базовым типам, воспрепятствует передаче управления остальным и приведёт к ошибке компиляции (см. пример 59).
Пример 59. Предложения catch

```
class SuperException extends Exception {
    // Базовый класс исключений
}
class SubException extends SuperException {
    // Производный класс исключений
}
class BadCatch {
    public void goodTry() {
        // Последовательность предложений catch неверна
        try {
            throw new SubException();
        } catch (SuperException superRef) {
            // Ловит и SuperException, и SubException
        } catch (SubException subRef) {
            // Код недостижим
        }
    }
}
```

Конкретный блок try в ходе выполнения может сгенерировать только одно исключение. Если в предложениях catch или finally выбрасываются другие исключения, предложения catch текущей конструкции try-catch-finally заново не проверяются. Но ничто не запрещает обрабатывать такие исключения с помощью вложенных конструкций try-catch-finally. Если после блока try присутствует предложение finally, его код выполняется по завершении работы остальных фрагментов кода try. Это происходит независимо от того, каким образом протекал процесс вычислений – успешно, с выбрасыванием исключения либо с передачей управления посредством команды return или break.
Обычно блок finally используется для осуществления операций очистки внутреннего состояния объекта или высвобождения ресурсов, не связанных со свободно распределяемой памятью, таких как, например, дескрипторы открытых файлов, хранимые в локальных переменных. Предложения finally используются и в тех случаях, когда необходимо осуществить ряд завершающих операций после выполнения инструкций break, continue или return. Поэтому нередко применяются такие конструкции try, в которых отсутствуют предложения catch.
При передаче управления в блок finally предварительно сохраняется информация о причине окончания выполнения блока try – код был исчерпан естественным образом либо завершен принудительно с помощью одной из управляющих инструкций, таких как return, или сгенерированного исключения. Эта информация восстанавливается в момент выхода из блока finally. Если, однако, код finally обладает собственными полномочиями по передаче управления во внешний блок посредством инструкций break или

return либо выбрасывания исключения, исходная причина «забывается» и замещается новой (см. пример 60).

Пример 60. Блок finally

```
try {
    // Что-то присходит
    return 1;
} finally {
    return 2;
}
```

При выполнении кодом try команды return блоку finally передаётся управление и информация о причине завершения try – инструкция return возвратила значение 1. Код finally, в свою очередь, также выполняет команду return, но теперь она возвращает другое значение – 2, и исходное «намерение» системы заменяется новым. Если в процессе выполнения кода try будет выброшено исключение, итогом всё равно окажется, разумеется, инструкция return 2. Но если бы блок finally не содержал этой инструкции, первоначальная цель – «возвратить с помощью команды return значение 1» – была бы реализована. Важно понимать, что любое исключение имеет формальную и реальную причину возникновения. Например, формальной причиной возникновения исключения FileNotFoundException является отсутствие файла с указанным именем при попытке его открытия. Но фактической причиной может быть имя файла, неверно введённое пользователем, и лишь переданное в метод чтения из файла в виде аргумента. В такой ситуации метод чтения из файла, конечно, может отловить исключение, но делать это не имеет смысла: «нести ответственность» может только тот метод, который получил имя файла от пользователя и передал его методу чтения. Таким образом, исключение следует обрабатывать там, где это можно сделать и имеет смысл делать.

Интерфейсы. Модификаторы в объявлениях интерфейсов. Пример простого интерфейса. Объявление интерфейса. Константы и методы в интерфейсах.

Базовая единица программы, написанной на языке Java – это класс (class), но базовый инструмент ООП – это тип (type). Класс определяет тип, но также определяет и его реализацию. В частности, это приводит к тому, что при наследовании классов передаются и тип, и реализация. Это, в свою очередь, приводит к проблеме множественного наследования классов: если у двух классов-родителей есть методы с одинаковой сигнатурой, какой из этих методов достанется классу-потомку?.. Именно поэтому в Java отсутствует множественное наследование классов.
Нетрудно заметить, что проблема множественного наследования возникает только благодаря наследованию реализации. Поэтому было бы полезно иметь средство, позволяющее наследовать только тип, без реализации. Это позволит создавать деревья наследующих типов, альтернативные дереву классов, и значительно расширить возможности полиморфизма.
Интерфейс (interface) позволяет описать тип в абстрактной форме – в виде набора заголовков методов и объявлений констант, которые все вместе образуют контракт типа. Интерфейс описывает тип, но не содержит блоков реализации. Поэтому экземпляры интерфейсов создавать нельзя. Эта возможность предоставляется производным классам, которые способны реализовать (implements) один или несколько интерфейсов. Таким образом в Java допускается возможность множественного наследования типов и единичного наследования реализации – классу позволено непосредственно наследовать только один базовый класс, но произвольное количество интерфейсов. Все классы, которые расширяются конкретным классом, и все реализуемые им интерфейсы в совокупности называют базовыми типами (supertypes). Новый класс – с точки зрения базовых типов – принято называть производным типом (subtype).
Производный тип «содержит в себе» все унаследованные им базовые типы, поэтому ссылка на объект производного типа может быть полиморфным образом использована в любом контексте, где требуется ссылка на любой из базовых типов (класс или интерфейс).
Объявление интерфейса создаёт новый тип точно так же, как и объявление класса. Наименование интерфейса можно употреблять в качестве имени типа в объявлении любой переменной, и такой переменной допускается присваивать ссылки на объекты соответствующих производных типов.
Обычно интерфейсы создаются для определения категории объектов, у которых есть набор определяемых интерфейсом методов, либо для определения свойства объекта, способности его к чему-либо (выражающейся в наличии у объекта специальных методов, или даже иногда в самом факте реализации интерфейса). В первом случае название интерфейса обычно совпадает с названием категории: например, List – объект, хранящий в себе упорядоченный набор других объектов, или ChatSequence – объект, содержащий в том или ином виде последовательность символов. Во втором случае способность к чему-то обозначается с помощью суффикса «able». В составе стандартных пакетов Java есть немало таких интерфейсов:

- Cloneable: объекты этого типа поддерживают операцию клонирования;
- Comparable: объекты допускают упорядочивание и поэтому могут сравниваться;

- Runnable: соответствующие объекты содержат код, способный выполняться в виде независимого потока вычислений;
- Serializable: объекты этого типа могут быть преобразованы в последовательность байтов с целью сохранения на носителях или переноса в среду другой виртуальной машины, а затем при необходимости восстановлены в исходном виде.

Пример простого интерфейса

Рассмотрим в качестве примера интерфейс Comparable. Этот интерфейс может быть реализован в любом классе, объекты которого поддерживают некий «естественный порядок». Интерфейс содержит единственный метод и выглядит следующим образом.

Пример 69. Интерфейс Comparable

```
public interface Comparable {
    int compareTo(Object o);
}
```

Объявление интерфейса сродни объявлению класса за тем исключением, что вместо служебного слова class используется слово interface. Помимо этого существуют определённые правила объявления членов интерфейса, о которых будет сказано чуть позже. В примере метод compareTo() в качестве аргумента принимает ссылку на объект типа Object и сравнивает его с текущим объектом, возвращая отрицательное, положительное или нулевое целое число, если, соответственно, текущий объект меньше или больше аргумента, или равен ему. Если два объекта взаимно несопоставимы (обычно вследствие несовместимости их типов, например, Integer заведомо нельзя сравнивать со String), выбрасывается исключение типа ClassCastException.

Вспомним класс, служащий для представления данных о небесных телах. Естественный порядок тел, являющихся спутниками одного и того же «центрального» светила, может быть определён в зависимости от радиуса орбиты их вращения. Соответствующая версия объявления класса Body будет выглядеть, например, так (см. пример 70).

Пример 70. Реализация интерфейса Comparable

```
class Body implements Comparable {
    // Объявления полей опущены
    // Задаётся в процессе конструирования
    int orbitalDistance = ...;
    public int compareTo(Object o) {
        Body other = (Body) o;
        if (orbits == other.orbits)
           return orbitalDistance - other.orbitalDistance;
        else
           throw new IllegalArgumentException(
                     "Неверное значение orbits");
    }
}
```

Интерфейсные типы, реализуемые классом, в его объявлении перечисляются после служебного слова implements (конструкция implements целиком размещается после предложения extends, если таковое имеется, но перед блоком тела класса). Все указанные в объявлении интерфейсы называют базовыми интерфейсами (superinterfaces) класса. Класс обязан обеспечить реализацию всех методов, определённых в унаследованных базовых интерфейсах, иначе в его объявление следует включить модификатор abstract, имея в виду, что конкретной реализацией в будущем «займутся» какие-либо конкретные (неабстрактные) производные классы.

В примере класса Body (пример 70), как и во многих других возможных реализациях метода compareTo(), используется оператор явного преобразования типов, который заменяет ссылку на объект Object ссылкой на объект класса Body. Преобразование типов осуществляется перед операцией сравнения. Если ссылка, переданная в качестве аргумента, указывает на объект, не относящийся к классу Body и поэтому не поддерживающий подобное преобразование, генерируется исключение типа ClassCastException. Затем вычисляется разность радиусов орбит двух тел и возвращается результат. Если тела не являются спутниками одного и того же «центрального» небесного объекта, они не могут быть сопоставлены, и поэтому выбрасывается исключение типа IllegalArgumentException.

Интерфейсы, как и классы, вводят новые наименования типов, которые могут быть использованы в объявлениях переменных:

```
Comparable obj;
```

Действительно, мощь интерфейсов во многом обусловлена именно тем обстоятельством, что зачастую вместо переменных конкретных типов гораздо удобнее и выгоднее объявлять переменные соответствующих интерфейсных типов. Например, целесообразно определить общий метод сортировки таким образом, чтобы он оказался способен сортировать элементы любого массива объектов типа Comparable (разумеется, объекты должны быть сопоставимы), независимо от того, какому конкретному классу они принадлежат (см. пример 71).

Пример 71. Применение ссылок интерфейсных типов

```
class Sorter {
    public static Comparable[] sort(Comparable[] list){
        // Детали реализации...
        return list;
    }
}
```

Посредством ссылки на интерфейсный тип, однако, разрешается обращаться только к членам соответствующего интерфейса. Например, следующий фрагмент кода породит ошибку компиляции (пример 72):

Пример 72. Неверное использование интерфейсной ссылки

```
Comparable obj = new Body();
String name = obj.getName(); // Неверно:
// в составе интерфейса Comparable нет метода getName()
```

Если необходимо интерпретировать obj как объект класса Body, следует применить соответствующий оператор преобразования типов.

Существует, правда, единственное исключение из этого правила: ссылку на любой интерфейс позволяется трактовать как ссылку на объект класса Object, поскольку Object – это базовый класс по отношению ко всем классам, поэтому его методы есть у любого объекта (или, говоря иначе, его контракту удовлетворяет любой объект). Поэтому следующее выражение вполне допустимо:

```
String desc = obj.toString();
```

Здесь ссылка на интерфейсный тип Comparable неявно преобразуется в ссылку на тип Object.

Объявление интерфейса

В объявлении интерфейса содержится служебное слово interface, за которым следует наименование интерфейса и перечень его членов, заключенный в фигурные скобки.

В составе интерфейса могут присутствовать члены трёх категорий:

- константы (поля);
- методы;

- вложенные классы и интерфейсы.

Все члены интерфейса по умолчанию обладают признаком public, но модификатор public принято опускать. Применение других модификаторов доступа по отношению к членам интерфейса лишено особого смысла.

Константы в интерфейсах

В состав интерфейса могут быть включены объявления именованных констант. Такие константы определяются как поля, но им неявно присваиваются признаки public, static и final – и вновь, по традиции, соответствующие модификаторы принято не указывать. Поля должны быть снабжены соответствующими инициализаторами – использование полей с отложенной инициализацией (blank final) не разрешается.

Поскольку в объявлении интерфейса отсутствует какой-либо код реализации, здесь запрещено объявление «нормальных» полей – в противном случае они послужили бы частью контракта интерфейса, которая «диктовала» бы производным классам те или иные правила реализации, а это противоречит самой сути интерфейса. Тем не менее, в интерфейсах позволено определять именованные константы, поскольку подчас они бывают весьма полезны при проектировании типов. Например, объявление интерфейса, описывающего возможные уровни красноречия человека, могло бы выглядеть так (пример 73).

Пример 73. Константы в интерфейсах

```
interface Verbose {
    int SILENT = 0; // Безмолвный
    int TERSE = 1; // Немногословный
    int NORM = 2; // Нормальный
    int VERVOSE = 3; // Многоречивый

    void setVerbosity(int level);
    int getVerbosity();
}
```

В метод setVerbosity() вместо «безмолвных» чисел удобнее передавать «красноречивые» константы SILENT, TERSE, NORM или VERVOSE, смысл которых определён в самих их названиях. Если интерфейс должен содержать данные, поддающиеся изменению и позволяющие совместное использование, этого можно добиться, применяя именованные константы, которые ссылаются на объекты с необходимыми полями данных. Для создания такого рода объектов уместно использовать вложенные классы, рассмотрение которых выходит за рамки настоящего пособия.

Методы в интерфейсах

Методы, объявляемые в составе интерфейса, неявно получают признак abstract, поскольку не содержат (и не могут содержать) блок тела. По этой причине блок тела в объявлении заменяется символом точки с запятой. В соответствии с принятым соглашением модификатор abstract в объявлении метода не указывается. В объявлении метода, принадлежащего интерфейсу, запрещается задавать и какие-либо другие модификаторы. Все методы неявно помечаются признаком public, поэтому использование других модификаторов доступа не допускается (public также обычно исключается).

В объявлениях методов нельзя употреблять модификаторы, имеющие отношение к особенностям реализации – такие как native, synchronized или strictfp – поскольку интерфейс по определению не способен регламентировать правила реализации.

Методы запрещено помечать модификатором final, так как в их объявлениях вообще отсутствуют какие-либо блоки реализации, которые можно было бы трактовать как завершённые. Кроме того, методы интерфейсов не способны быть статическими, поскольку модификаторы static и abstract нельзя использовать одновременно. Разумеется, внутри объявлений классов, реализующих интерфейс, те же (переопределённые) методы могут быть снабжены любыми подходящими модификаторами.

Модификаторы в объявлениях интерфейсов

В объявлении интерфейсов разрешается использовать следующие модификаторы.

- public. Публичный интерфейс открыт для доступа извне. Как и в случае с объявлением классов, модификатор public по отношению к интерфейсу разрешается использовать только тогда, когда интерфейс описан в файле с тем же именем. При отсутствии этого модификатора, интерфейс получает признак доступа уровня пакета.
- abstract. Каждый интерфейс по определению абстрактен, поскольку все его методы не содержат блоков реализации. И вновь, в соответствии с принятым соглашением, модификатор abstract в объявлении интерфейса опускается.
- strictfp. Объявление интерфейса, помеченного признаком strictfp, предполагает, что все операции с плавающей запятой, которые позже могут быть предусмотрены в реализующем коде, должны выполняться точно и единообразно всеми виртуальными машинами Java. При этом не предполагается, что каждый метод интерфейса неявно получает тот же признак strictfp, поскольку этот вопрос относится к компетенции классов, обеспечивающих конкретную реализацию.

Расширение интерфейсов. Наследование и сокрытие констант. Наследование, переопределение и перегрузка методов. Пустые интерфейсы. Пример. Отличия абстрактного класса от интерфейса.

Расширение интерфейсов

Для интерфейсов также допустимо наследование, выражающееся в наследовании типа (без реализации). Говорят, что интерфейсы допускают расширение; в этом случае в объявлении производного интерфейса используется служебное слово extends. Интерфейсы, в отличие от классов, способны расширять более одного интерфейса (пример 74).

Пример 74. Расширение интерфейсов

```
public interface SerializableRunnable
                 extends java.io.Serializable, Runnable {
    // ...
}
```

Интерфейс SerializableRunnable одновременно наследует от интерфейсов java.io.Serializable и Runnable. Это означает, что все методы и константы, определённые в каждом родительском интерфейсе, становятся, наряду с собственными методами и константами, частью контракта нового интерфейса SerializableRunnable. Наследуемые интерфейсы называют базовыми (superinterfaces) по отношению к новому интерфейсу, который, в свою очередь, является производным (subinterface), или расширенным, интерфейсом относительно базовых.

Наследование и сокрытие констант

Производный интерфейс наследует все константы, объявленные в базовых интерфейсах. Если в производном интерфейсе объявлена константа с тем же именем, что и унаследованная, то, независимо от их типов, новая константа «скрывает» старую (те же правила справедливы и в отношении унаследованных полей классов).

Ссылка на константу посредством простого имени в контексте производного интерфейса или класса, который реализует этот интерфейс, означает обращение к константе, принадлежащей производному, а не базовому интерфейсу. Константа, унаследованная от базового интерфейса, всё ещё доступна, если при ссылке на неё указывать полное имя, т.е. название интерфейса, сопровождаемое оператором точки и идентификатором самой константы. Точно так же обычно выполняется обращение к статическим членам классов.

Пример 75. Сокрытие констант при расширении интерфейсов

```
interface X {
    int VAL = 1;
}
interface Y extends X {
    int VAL = 2;
    int SUM = VAL + X.VAL;
}
```

Интерфейс Y содержит объявления двух констант – VAL и SUM. Чтобы обратиться к скрытой константе VAL, унаследованной из интерфейса X, необходимо указать её полное имя – X.VAL. Во внешнем коде для ссылки на константы интерфейса Y можно использо-

вать обычную форму, принятую при обращении к статическим членам класса – Y.VAL и Y.SUM. Разумеется, посредством выражения X.VAL вы получите доступ и к константе VAL интерфейса X.
Если интерфейс Y реализуется каким-либо классом, константы интерфейса Y с точки зрения этого класса будут выглядеть как члены класса. Например, в контексте класса, объявление которого выглядит как class Z implements Y {}, вполне допустимо следующее выражение:

```
System.out.println(«Z.VAL=» + Z.VAL + «Z.SUM=» + Z.SUM);
```

Но при этом отсутствует возможность обращения посредством Z к X.VAL. Однако, если ссылка на объект Z существует, адресовать X.VAL удаётся с помощью оператора преобразования типов:

```
Z z = new Z();
System.out.println("z.VAL=" + z.VAL + ", ((Y)z).VAL=" +
((Y)z).VAL + ", ((X)z).VAL=" + ((X)z).VAL);
```

Результат работы кода будет выглядеть так:
z.VAL=2, ((Y)z).VAL=2, ((X)z).VAL=1
Мы вновь наблюдаем ту же картину, что и при использовании статических полей в расширенных классах. Таким образом, не имеет значения, откуда унаследовано статическое поле – из базового класса или базового интерфейса.
Если интерфейс наследует несколько констант с одним и тем же именем, использование этого имени без дополнительных разъяснений чревато недоразумениями и приводит к ошибке компиляции. Продолжим пример, касающийся интерфейсов X и Y, и предположим, что объявлены ещё два интерфейса (пример 76).
Пример 76. Проблема конфликта констант при множественном наследовании

```
interface C {
    String VAL = "Интерфейс C";
}
interface D extends X, C {}
```

Что теперь может означать выражение D.VAL – обращение к целочисленной константе X.VAL или строковой константе C.VAL? В подобных случаях мы обязаны явно оговаривать свои намерения – достаточно прямо написать X.VAL или C.VAL. Класс, который реализует более одного интерфейса, либо расширяет базовый класс и при этом реализует интерфейсы, приводит нас к тем же трудностям, связанным с сокрытием данных и неоднозначностью имён, что и интерфейс, наследующий несколько других интерфейсов. Наличие в классе собственных статических полей обусловливает сокрытие одноимённых унаследованных полей базовых классов или интерфейсов, и обычные ссылки на такие поля будут выглядеть двусмысленно.

Наследование, переопределение и перегрузка методов

Производный интерфейс наследует все методы базовых интерфейсов. Если метод, объявленный в производном интерфейсе, обладает теми же сигнатурой и типом возвращаемого значения, что и унаследованный метод, новое объявление переопределяет любое и все аналогичные объявления унаследованных методов. Переопределение в интерфейсах, в отличие от переопределения в классах, не несёт какой-либо семантической нагрузки – интерфейс в результате наследования будет в итоге содержать одно объявление метода, и в любом классе, реализующем интерфейс, может присутствовать только одна реализация этого метода.
Если интерфейс наследует более одного метода с одной и той же сигнатурой, или класс реализует несколько интерфейсов, содержащих метод с одинаковой сигнатурой, всё равно можно говорить о том, что существует только один такой метод – а именно тот, конкретный вариант кода которого в конечном итоге представлен в классе,

реализующем интерфейс. В данном случае возможен конфликт контрактов, если одноименные методы в различных интерфейсах несут различную семантическую нагрузку. Такая ошибка не может быть выявлена компилятором и относится к разряду серьёзных ошибок проектирования.
Как и при переопределении методов в процессе расширения класса, методу, переопределённому при наследовании интерфейса, не позволяется декларировать больше объявленных исключений, чем предусмотрено в исходном объявлении соответствующего метода. Если наследуются (без переопределения) два или более метода и их объявления различаются только предложением throws, при реализации этого метода должны учитываться все перечисленные в предложениях throws объявленные исключения.
Если в составе интерфейса объявлен метод с тем же именем, но иным списком параметров, нежели в унаследованном интерфейсе, имеет место перегрузка метода. Класс, реализующий интерфейс, должен предоставить конкретные варианты кода для каждой из перегруженных форм метода. Если объявленный в интерфейсе метод отличается от унаследованного только типом возвращаемого значения, при компиляции будет выдано сообщение об ошибке.

Пустые интерфейсы

Некоторые интерфейсы не содержат объявлений каких-либо методов, а просто обозначают некоторое свойство или общий признак принадлежности будущих классов к некоторой группе. Примером такого интерфейса – их принято называть пустыми (empty), или интерфейсами-маркерами (marker interface) – может служить Cloneable, в составе которого отсутствуют объявления каких бы то ни было методов и констант. Интерфейс Cloneable, будучи реализованным в классе, относит этот класс к числу тех, в которых поддерживается механизм клонирования. К числу интерфейсов-маркеров, помимо упомянутого Cloneable, относится также интерфейс Serializable. Пустые интерфейсы способны оказывать серьёзное влияние на поведение производных классов – вспомните, например, о Cloneable.

Абстрактный класс или интерфейс?

Существует два главных различия между интерфейсами и абстрактными классами.

- Интерфейсы обеспечивают инструментарий множественного наследования, производный класс способен наследовать одновременно несколько интерфейсов. Класс может расширять единственный базовый класс, даже если тот содержит только абстрактные методы.
- Абстрактный класс частично может быть реализован, он вправе содержать члены, помеченные как protected, и/или static и т.п. Структура интерфейса ограничена объявлениями публичных констант и методов, без какой бы то ни было реализации.

Указанные различия обычно обусловливают выбор средств, наиболее предпочтительных в конкретных обстоятельствах. Если возможность множественного наследования важна, следует обратиться к интерфейсам. Абстрактные классы предлагают реализацию – частичную или даже полную – и поэтому цель может быть достигнута посредством простого наследования вместо необходимости реализации «с ноля». Кроме того, абстрактный класс способен управлять реализацией некоторых методов, обозначая их как final.

Потоки данных. Байтовые потоки. Базовые абстрактные классы байтовых потоков. Символьные потоки. Базовые абстрактные классы символьных потоков. Примеры байтовых и символьных классов потоков. Стандартные потоки. InputStreamReader и OutputStreamWriter.

Ввод/вывод

Функции ввода-вывода информации, реализованные в составе стандартного **пакета java.io,** определены в терминах потоков данных (streams). **Потоки данных** – это упорядоченные последовательности данных, которым соответствует определённый источник (для потоков ввода) или получатель (для потоков вывода). Большинство типов потоков данных поддерживает методы определённых базовых интерфейсов и абстрактных классов с некоторыми дополнениями.
Пакет java.io охватывает определения типов двух основных разновидностей – символьных потоков и байтовых потоков. Под **символьными потоками** понимают последовательности 16-битовых символов Unicode, а каждому байту отвечает порция данных длиной 8 бит. Ввод-вывод может быть либо текстовым, либо бинарным. Функции текстового ввода-вывода способны обращаться с потоками символов, поддающихся восприятию человеком (например, с исходными текстами программ), а средства обработки бинарных данных имеют дело с информацией, хранящейся в двоичном виде (например, с битовыми представлениями графических изображений).
Символьные потоки используются в операциях текстового ввода-вывода, а байтовые – при работе с бинарными данными. Байтовые потоки принято называть **потоками ввода** и **потоками вывода**, а **символьные – потоками чтения** и **потоками записи.**
Почти для каждого потока ввода существует соответствующий поток вывода, а для большинства потоков ввода и вывода определены символьные потоки чтения и записи, обладающие схожими функциональными чертами, и наоборот.
Классы и интерфейсы из состава пакета java.io можно условно разделить на пять обширных групп:

1) типы общего назначения, служащие для построения различных типов байтовых и символьных потоков: потоки ввода и вывода, чтения и записи, а также классы, позволяющие выполнять взаимные преобразования между ними
2) классы, определяющие разновидности потоков: фильтрованные потоки, буферизованные потоки, канальные потоки, а также специальные подвиды таких потоков, такие как поток чтения строк с сохранением нумерации и лексический анализатор потока чтения
3) специализированные потоковые классы и интерфейсы для ввода и вывода значений простых типов и строк
4) классы и интерфейсы для обработки файловых данных в стиле, не зависящем от особенностей платформы
5) классы и интерфейсы, поддерживающие механизм сериализации объектов
Объекты класса IOException используются многими методами ввода-вывода в качестве сигнала о возникновении исключительной ситуации

Байтовые поток

Существует ряд особенностей, присущих всем байтовым потокам данных: например, все они поддерживают механизм открытия и закрытия. Поток открывается при создании объекта класса и, оставаясь открытым, позволяет осуществлять операции чтения и записи данных. **Поток закрывается при вызове метода close() объекта.** Закрытие потока служит цели высвобождения системных ресурсов, которые используются потоком. Если поток явно не закрывается, он продолжает существовать и расходовать системные ресурсы. Потоки следует закрывать сразу после завершения их использования.

Абстрактный класс InputStream

В составе абстрактного класса InputStream объявлены методы, обеспечивающие выполнение функций ввода байтовых данных из определённого источника. **InputStream является базовым классом по отношению к потокам ввода, определённых в пакете java.io,** и обладает методами, перечисленными ниже.

- **public abstract int read() throws IOException**
  Вводит один байт данных и возвращает его в виде целого числа из диапазона от 0 до 255, иными словами, байтовое значение трактуется как целое без знака. Если байтов, готовых для ввода, не существует ввиду достижения конца потока, возвращается значение -1
- **public int read(byte[] buf, int offset, int count) throws IOException**
  Вводит байты данных и сохраняет их в части массива buf, максимальное количество байтов, подлежащих вводу, определяется значением count (если 0 – ввод не производится и возвращается 0). Байты заносятся в элементы массива, начиная с buf[offset] и до buf[offset+count-1]. Метод возвращает количество фактически введённых байтов. Если ввиду достижения конца потока не введён ни один байт, возвращается -1. Если первый байт не может быть введён по причине, не связанной с достижением конца потока (например, поток уже закрыт), выбрасывается исключение типа IOException
- **public int read(byte[] buf) throws IOException**
  Метод аналогичен предыдущему при условии read(buf, 0, buf.length)
- **public long skip(long count) throws IOException**
  Пропускает, самое большее, count байтов либо часть последовательности байтов до конца потока. Возвращает количество фактически пропущенных байтов. Если значение count отрицательно, операция не выполняется
- **public int available() throws IOException**
  Возвращает количество байтов, которые могут быть введены или пропущены, не вызывая блокировки. Реализация метода, предлагаемая по умолчанию, возвращает значение 0
- **public void close() throws IOException**
  Закрывает ранее открытый поток ввода. Метод используется для высвобождения ресурсов, связанных с потоком. Обращение к закрытому потоку приводит к выбрасыванию исключения типа IOException, но повторное закрытие потока эффектов не вызывает. В реализации метода, предлагаемой по умолчанию, никакие действия не предусмотрены

При реализации InputStream требуется, чтобы производный класс обеспечил практическое воплощение варианта метода read(), осуществляющего ввод единственного байта, поскольку остальные версии read() основываются на первой. Во многих случаях, однако, производительность операций может быть улучшена за счёт переопределения и других методов. Исходные версии методов available() и close() нуждаются в переопределении в соответствии с потребностями конкретных типов потоков.

Пример 1

```
import java.io.*;
 class CountBytes {
     public static void main(String[] args) throws IOException {
          InputStream in;
          if (args.length == 0)
               in = System.in;
          else
               in = new FileInputStream(args[0]);
          int total = 0;
          while (in.read() != -1)
               total++;
         System.out.println(total + " байтов");
     }}
```

Абстрактный класс OutputStream

Абстрактный класс OutputStream во многом аналогичен классу InputStream; он предлагает модель вывода байтовых данных в объект - получатель. Методы класса OutputStream перечислены ниже.

- **public abstract void write(int b) throws IOException**
  Выводит значение b в виде байта. В качестве параметра передаётся именно int, поскольку значение часто является результатом выполнения арифметических операций над аргументами byte. Выражение, операнды которого относятся к типу byte, обладает типом int, поэтому объявление параметра в виде int означает возможность избежать необходимости использования операций явного преобразования результатов вычисления выражения к типу byte. Следует заметить, однако, что выводится только 8 младших битов результат типа int
  public void write(byte[] buf, int offset, int count) throws IOException
  Выводит count байтов массива buf, начиная с элемента buf[offset]
- **public void write(byte[] buf) throws IOException**
  Метод аналогичен предыдущему при условии write(buf, 0, buf.length)
- **public void flush() throws IOException**
  Осуществляет сброс потока. Единственный вызов flush приводит к сбросу всех буферов в цепочке взаимосвязанных потоков. Если поток не относится к категории буферизированных, реализация метода, предлагаемая по умолчанию, не предусматривает выполнения каких бы то ни было действий
- **public void close() throws IOException**
  Закрывает ранее открытый поток вывода. Метод используется для высвобождения ресурсов, связанных с потоком. Любые попытки обращения к закрытому потоку приводят к выбрасыванию исключения типа IOException, но повторное закрытие потока эффектов не вызывает. В реализации метода, предлагаемой по умолчанию, никакие действия не предусмотрены

При реализации OutputStream требуется, чтобы производный класс обеспечил практическое воплощение варианта метода write(), осуществляющего вывод единственного байта, поскольку остальные версии write основываются на первой.

Символьные потоки

Абстрактными классами, обеспечивающими чтение и запись символьных потоков, являются Reader и Writer. В каждом из них предусмотрены методы, схожие с «собратьями» из состава классов, представляющих байтовые потоки.
Классы символьных потоков появились в составе Java после реализации байтовых потоков и имели целью обеспечение полной поддержки таблиц символов Unicode. Как и в

случае байтовых потоков, по окончании использования символьного потока его следует закрывать, чтобы высвободить используемые им системные ресурсы.

Абстрактный класс Reader

Абстрактный класс Reader обеспечивает поддержку символьного потока чтения аналогично тому, как это делает InputStream, реализующий модель байтового потока ввода, и методы Reader схожи с теми, которые объявлены в составе InputStream.

- **public int read() throws IOException**
  Считывает один символ данных и возвращает его в виде целого числа из диапазона от 0 до 65535. Если символов, готовых для чтения, не существует ввиду достижения конца потока, возвращается значение -1. Метод выполняет блокировку до тех пор, пока имеются доступные для чтения символы, не достигнут конец потока либо не выброшено исключение
- **public abstract int read(char[] buf, int offset, int count) throws IOException**
  Считывает символы и сохраняет их в части массива buf, максимальное количество символов, подлежащих чтению, определяется значением count (если 0 – чтение не производится и возвращается 0). Символы заносятся в элементы массива, начиная с buf[offset] и до buf[offset+count-1]. Метод возвращает количество фактически считанных символов. Если ввиду достижения конца потока не считан ни один символ, возвращается -1. Если первый символ не может быть считан по причине, не связанной с достижением конца потока (например, поток уже закрыт), выбрасывается исключение типа IOException
- **public int read(char[] buf) throws IOException**
  Метод аналогичен предыдущему при условии read(buf, 0, buf.length)
- **public long skip(long count) throws IOException**
  Пропускает, самое большее, count символов либо часть последовательности символов до конца потока. Возвращает количество фактически пропущенных символов. Значение count не должно быть отрицательным
- **public boolean ready() throws IOException**
  Возвращает true, если поток готов для чтения данных, т.е. в нём существует хотя бы один доступный символ. Результат, равный false, не свидетельствует о том, что очередной вызов read приведёт к возникновению блокировки, поскольку в промежутке между обращениями к ready и read в поток может поступить очередная порция данных
- **public abstract void close() throws IOException**
  Закрывает ранее открытый поток чтения. Метод используется для высвобождения ресурсов, связанных с потоком. Любые попытки обращения к закрытому потоку приводят к выбрасыванию исключения типа IOException, но повторное закрытие потока эффектов не вызывает

При реализации Reader требуется, чтобы производный класс обеспечил практическое воплощение варианта метода read(), осуществляющего чтение данных в массив символов, и версии метода close(). Во многих случаях, однако, производительность операций может быть улучшена за счёт переопределения и других методов.

Различия Reader и InputStream

В Reader базовым абстрактным методом чтения является вариант метода read(), осуществляющий чтение в массив типа char, а остальные версии основываются на базовой. В классе InputStream в качестве основного объявлен метод read(), выполняющий ввод одного байта.
Классы, производные от Reader, обязаны предоставить реализацию абстрактного метода close(): в отличие от наследования пустой реализации; многим классам потоков, как правило, необходимо «знать», был ли закрыт поток или нет, так что метод close обычно подлежит переопределению.
Наконец, в классе InputStream предусмотрен метод available(), позволяющий выяснить, какой объём данных готов для ввода, а класс Reader предоставляет метод ready(), свидетельствующий о том, существуют ли такие данные или нет.

Пример 2

```
import java.io.*;
class CountSpace {
     public static void main(String[] args) throws IOException {
          Reader in;
          if (args.length == 0)
               in =new InputStreamReader(System.in);
          else
               in = new FileReader(args[0]);
          int ch;
          int total;
          int spaces = 0;
         for (total = 0; (ch = in.read()) != -1; total++) {
             if (Character.isWhitespace((char) ch))
                 spaces++;
         }
         System.out.println(total + " символов, " + spaces + "
пробелов ");
     }}
```

Абстрактный класс Writer

Абстрактный класс Writer обеспечивает поддержку символьного потока записи аналогично тому, как это делает OutputStream, реализующий модель байтового потока вывода, и многие методы Writer схожи с теми, которые объявлены в составе OutputStream, однако, помимо этого, в классе Writer предусмотрены некоторые другие полезные версии метода write.

- **public void write(int ch) throws IOException**
  Записывает ch в виде символа. Символ передаётся как значение int, но записывается только 16 битов последнего
- **public abstract void write(char[] buf, int offset, int count) throws IOException**
  Записывает count символов массива buf, начиная с элемента buf[offset]
- **public void write(char[] buf) throws IOException**
  Метод аналогичен предыдущему при условии write(buf, 0, buf.length)
- **public void write(String str, int offset, int count) throws IOException**
  Записывает count символов строки str, начиная с символа str.charAt(offset)
- **public void write(String str) throws IOException**
  Метод аналогичен предыдущему при условии write(str, 0, str.length())
- **public abstract void flush() throws IOException**
  Осуществляет сброс потока. Единственный вызов flush приводит к сбросу всех буферов в цепочке взаимосвязанных потоков. Если поток не относится к категории буферизированных, никакие действия не выполняются

- **public abstract void close() throws IOException**
  Закрывает ранее открытый поток записи, выполняя при необходимости его сброс. Метод используется для высвобождения ресурсов, связанных с потоком. Любые попытки обращения к закрытому потоку приводят к выбрасыванию исключения типа IOException, но повторное закрытие потока эффектов не вызывает

Классы, производные от Writer, обязаны обеспечить реализацию варианта метода write(), связанного с записью символов из части массива, а также методов close() и flush(). Все остальные методы Writer основаны на этих трёх. **Это отличает Writer от OutputStream**, в котором в качестве базового метода вывода предусмотрен вариант write(), осуществляющий вывод одного байта, а для методов close() и flush() предложены реализации по умолчанию. Производительность операций может быть улучшена за счёт переопределения в производных классах и других методов

Символьные и стандартные потоки

Стандартные потоки
- System.in
- System.out
- System.err

существовали в Java ещё до появления в языке символьных потоков. Поэтому стандартные потоки относятся к категории байтовых.

InputStreamReader и OutputStreamWriter

Эти классы обеспечивают возможность преобразования байтовых потоков ввода в символьные потоки чтения и символьных потоков записи в байтовые потоки вывода соответственно, с учётом заданной кодировки символов или кодировки, принятой по умолчанию в конкретной локальной системе. Объекту InputStreamReader в качестве источника передаётся байтовый поток ввода, и InputStreamReader обеспечивает чтение соответствующих символов Unicode. Объекту OutputStreamWriter в качестве получателя передаётся байтовый поток вывода, и OutputStreamWriter сохраняет в нём байтовые представления символов Unicode.
Методы read() класса InputStreamReader обеспечивают ввод байтов из заданного потока InputStream и преобразование их в символы с использованием соответствующей кодировки. Аналогично, методы write() класса OutputStreamWriter получают переданные символы, преобразуют их в байты, используя соответствующую кодировку, и выводят в заданный поток OutputStream. В обоих классах при закрытии потока-преобразователя также закрывается и связанный с ним байтовый поток. Такое поведение не всегда желательно (например, при преобразовании данных из стандартных потоков).
Классы FileReader и FileWriter являются расширенными версиями классов InputStreamReader и OutputStreamWriter соответственно и обеспечивают возможности обработки файловых данных с поддержкой Unicode и локальных стандартов кодировки символов.
Классов ReaderInputStream и WriterOutputStream, которые могли бы обеспечить трансляцию символьных потоков в байтовые и наоборот, **не существует.**

Потоки Filter

Классы семейства фильтрованных потоков Filter – **FilterInputStream, FilterOutputStream, FilterReader и FilterWriter** – позволяют объединять потоки в цепочки для получения составных потоков, обладающих расширенным набором функций. Каждый фильтрованный поток данных привязывается к другому потоку, которому передаёт полномочия по фактическому выполнению операций ввода или вывода. Позволяется связывать в единую цепочку любое количество байтовых (символьных) Filter-потоков ввода (чтения). Источником исходных данных может быть поток, не относящийся к семейству потоков Filter. Подобным образом могут соединяться в цепочки и Filter-потоки вывода (записи). Все потоки вывода (записи) в цепочке, начиная с первого и заканчивая предпоследним, должны относиться к классу Filter, но последним может быть объект любого потокового класса вывода (записи).
Не все классы семейства Filter в действительности осуществляют изменение данных. Некоторые просто привносят дополнительные черты поведения, а другие предлагают новые интерфейсы взаимодействия с потоками.

Потоки Buffered

Классы семейства буферизованных потоков Buffered – **BufferedInputStream, BufferedOutputStream, BufferedReader и BufferedWriter** – осуществляют буферизацию данных, позволяющую избежать необходимости обращения к источнику (получателю) данных при выполнении каждой отдельной операции read() или write(). Эти классы часто используются в сочетании с потоками семейства File: доступ к данным на диске выполняется намного медленнее, чем к информации в буфере памяти, и средства буферизации помогают снизить потребность в обращениях к диску.
Когда метод read() потока Buffered, вызываемый для ввода (чтения) данных, обнаруживает, что буфер потока пуст, он вызывает одноимённый метод потока источника, заполняет буфер настолько большой порцией данных, насколько это возможно и возвращает запрошенные данные из буфера. При очередных обращениях к методу read() объекта Buffered данные будут извлекаться из буфера, пока его содержимое не исчерпается, и в этот момент объекту вновь придётся вызвать read() потока-источника. Процесс повторяется до тех пор, пока не иссякнет источник данных.
Buffered – потоки вывода (записи) ведут себя аналогичным образом. Когда write() заполняет буфер данных, вызывается одноимённый метод потока-получателя, освобождающий буфер. Такой механизм буферизации помогает превратить последовательность запросов на вывод (запись) небольших порций данных, адресуемых объекту Buffered, в единственный вызов метода write() потока-получателя.

Потоки Piped

Канальные потоки, определяемые классами семейства Piped - **PipedInputStream, PipedOutputStream, PipedReader и PipedWriter** – используются в виде пар ввода-вывода (записи-чтения). Данные, переданные в поток вывода (записи), служат источником для потока ввода (чтения). Каналы реализуют механизм обмена данными между различными потоками вычислений.
Единственный безопасный способ обращения с потоками данных Piped связан с использованием двух потоков вычислений: один из них осуществляет вывод данных, а другой – их ввод. Когда буфер канала полностью заполняется, попытка вывода данных в него приводит к блокированию соответствующего потока вычислений. Если операции вывода и ввода выполняются одним потоком вычислений, тот будет блокирован постоянно. Поток вычислений, осуществляющий ввод, блокируется, если буфер канала пуст.

Чтобы избежать опасности «вечного» блокирования одного потока вычислений, когда его «собрат» на другом конце канала прекращает работу, каждый канал отслеживает подлинность и работоспособность потоков вычислений, обращавшихся к нему с целью вывода и ввода данных последними. Канал, прежде чем блокировать текущий поток вычислений, проверяет, «жив» ли поток на другом конце канала. Если обнаруживается, что работа противоположного потока вычислений прекращена, текущий поток генерирует исключение типа IOException.
Потоки Piped должны быть связаны друг с другом. Это можно сделать при создании потоков, передав конструктору потока PipedReader в качестве аргумента ссылку на поток PipedWriter (и наоборот, порядок несущественен) или позднее, используя метод connect(). При попытках использования потоков Piped до их соединения или связывания ранее соединённых потоков выбрасывается исключение типа IOException.

Байтовые потоки ByteArray

Массивы байтов, размещённые в оперативной памяти, могут выступать в роли источника или получателя данных при работе с потоками семейства ByteArray. Объект класса **ByteArrayInputStream** использует массив типа byte в качестве источника данных. В составе класса **ByteArrayOutputStream** предусмотрены средства динамического наращивания объёма массива типа byte, получающего выводимые данные.

Символьные потоки CharArray и String

Символьные потоки семейства CharArray (**CharArrayReader и CharArrayWriter**) аналогичны по назначению потокам ByteArray – они позволяют в качестве источника или получателя данных использовать массивы типа char, размещённые в памяти. В составе класса CharArrayWriter предусмотрены средства динамического наращивания объёма массива типа char, получающего выводимые данные. Объект класса **StringReader** позволяет считывать символы из строки String. Объект класса **StringWriter** позволяет записывать символы в буфер, который может интерпретироваться как объект типа String или StringBuffer.

Потоки Print

Классы семейства Print – **PrintStream и PrintWriter** – содержат объявление ряда методов, которые упрощают задачу вывода (записи) в поток значений простых типов и объектов в удобочитаемом текстовом формате. В потоковых классах Print реализованы перегруженные версии методов print() и println() для вывода следующих значений типов – char, char[], int, long, float, double, Object, String, boolean. Эти методы гораздо удобнее для применения, нежели обычные методы потокового вывода (записи) write(). Метод println() добавляет в конец порции выводимых данных, переданной в виде параметра, признак завершения строки (без параметров просто завершает текущую строку).
Каждый из потоков Print действует как поток Filter, и поэтому в процессе вывода данные могут быть подвержены дополнительной фильтрации. Класс PrintStream работает с байтовыми потоками, а PrintWriter – с символьными. Поскольку чаще возникает потребность в записи символов, в обычных ситуациях следует пользоваться средствами класса PrintWriter.
Одна из важных характеристик поведения потоков Print связана с тем, что ни один из реализованных в них методов вывода (записи) не выбрасывает исключения типа IOException. Если при передаче данных «внутреннему» потоку возникает ошибка, методы завершают выполнение нормальным образом. Проверить наличие ошибки можно с помощью вызова метода checkError(), возвращающего результат типа boolean.

Класс StreamTokenizer

Задачи лексического анализа потока данных относятся к числу традиционных и в составе пакета java.io представлен класс StreamTokenizer, позволяющий решать некоторые из них. Поток разбивается на лексемы с помощью объекта StreamTokenizer, конструктор которого принимает в качестве параметра объект типа Reader, выполняющий функцию источника данных; объект StreamTokenizer действует в соответствии с заданными параметрами сканирования данных. На каждой итерации цикла сканирования вызывается метод nextToken(), который возвращает очередную считанную из потока лексему и информацию о её типе, присваивая эти данные полям объекта StreamTokenizer.
Класс StreamTokenizer ориентирован преимущественно на анализ текстов, представляющих код, написанный на каком-либо языке программирования; класс нельзя отнести к числу инструментов лексического анализа общего назначения.
Когда метод nextToken распознаёт лексему, он возвращает её тип в виде значения и присваивает последнее полю **ttype.**
Существует четыре типа лексем, воспринимаемых методом **nextToken():**

- **TT_WORD**; обнаружено слово; оно сохраняется в поле **sval** типа String
- **TT_NUMBER;** обнаружено число; оно сохраняется в поле **nval** типа double; распознаются только десятичные числа с плавающей запятой (с десятичной точкой или без таковой) в нормальной нотации (анализатор не распознает ни 3.4e79 как число с плавающей запятой, ни 0xffff как шестнадцатеричное число)
- **TT_EOL;** обнаружен признак завершения строки
- **TT_EOF;** достигнут конец файла

Подразумевается, что текст, подлежащий анализу, состоит из байтов, значение которых относятся к диапазону от \u0000 до \u00FF, - корректность распознавания символов Unicode, не принадлежащих указанному интервалу, не гарантируется. Поток ввода включает специальные и обычные символы. К категории специальных относятся те символы, которые анализатор трактует специальным образом, а именно: символы пробела, символы, обозначающие числа и слова, и т.д. Любой другой символ воспринимается как обычный. Когда очередным символом потока является обычный символ, в качестве его типа возвращается значение этого символа.

Пример 3

```
static double sumStream(Reader in) throws IOException {
     StreamTokenizer nums = new StreamTokenizer(in);
     double result = 0.0;
     while (nums.nextToken() != StreamTokenizer.TT_EOF) {
          if (nums.ttype == StreamTokenizer.TT_NUMBER)
               result += nums.nval;
     }
     return result;
}
```

Байтовые потоки Data

В интерфейсах DataInput и DataOutput определены методы потокового ввода-вывода данных простых типов, а классы DataInputStream и DataOutputStream обеспечивают реализацию интерфейсов, предлагаемую по умолчанию. Интерфейсы, предусматривающие потоковый ввод и вывод бинарных данных, обладают почти одинаковой структурой.
Ниже перечислены методы семейств read и write, обеспечивающих ввод и вывод данных каждого из простых типов:

readBoolean (writeBoolean), readChar (writeChar), readByte (writeByte), readShort (writeShort), readInt (writeInt), readLong (writeLong), readFloat (writeFloat), readDouble (writeDouble), readUTF (writeUTF)
Методы интерфейса DataInput обычно реагируют на событие достижения конца потока, выбрасывая исключение типа EOFException, производного от IOException.

Для каждого интерфейса Data существует определённый поток Data. Кроме того, имеется класс RandomAccessFile, который реализует одновременно оба интерфейса Data – ввода и вывода. Каждый класс Data является расширением соответствующего класса Filter, так что потоки Data могут использоваться в целях фильтрации данных других потоков. В составе каждого класса Data есть конструкторы, в качестве параметра принимающие ссылки на другой подходящий поток ввода или вывода.
Примеры реализации соответствующих методов приведены в примере 4.

Пример 4

```
public static void writeData(double[] data, String file) throws
IOException {
          OutputStream fout = new FileOutputStream(file);
              DataOutputStream out = new DataOutputStream(fout);
              out.writeInt(data.length);
             for (int i =0; i < data.length; i++)
                  out.writeDouble(data[i]);
            out.close();
}

public static double[] readData(String file) throws IOException {
              InputStream fin = new FileInputStream(file);
                      DataInputStream in = new DataInputStream(fin);
                      double[] data = new double[in.readInt()];
                  for (int i =0; i < data.length; i++)
                      data[i] = in.readDouble();
                  in.close();
                  return data;
}
```

Класс File

Класс File реализует ряд полезных средств для работы с файлами и директориями: создание, проверка атрибутов, удаление, переименование. В составе класса File также объявлены методы, которые позволяют создавать временные файлы, используемые, например, для хранения промежуточных результатов и удаляемые при завершении работы программы.

Потоки File

Потоковые классы семейства File – FileInputStream, FileOutputStream, FileReader и FileWriter – позволяют трактовать файл как поток, предназначенный для ввода(чтения) или вывода(записи) данных. В составе каждого из типов предусмотрены три конструктора, принимающие в качестве параметра одно из следующих значений:

- строку String, задающую имя файла
- объект класса File, указывающий на файл
- объект FileDescriptor

Объект FileDescriptor служит для представления сущности, описывающей открытый файл и определяемой особенностями применяемой операционной системы. Если при

использовании байтового или символьного потоков ввода (чтения), обнаруживается, что файл не существует, выбрасывается исключение типа FileNotFoundException. При доступе к файлу во всех случаях проверяется наличие соответствующих полномочий: если необходимые права на обращение к файлу отсутствуют, генерируется исключение типа SecurityException. Два первых конструктора при построении байтового или символьного потоков вывода(записи) предусматривают создание файла, если такого не существует, при наличии файла его содержимое усекается.

Сериализация объектов. Подготовка классов к сериализации. Порядок сериализации и десериализации. Настройка механизма сериализации. Контроль версий объектов.

Сериализация объектов

Одной из существенных возможностей многих реальных приложений является их способность преобразовывать объекты в байтовые потоки, которые могут передаваться в пределах сети - например, с целью использования в рамках технологии удалённого вызова метода, - сохраняться на диске в виде файлов, а затем при необходимости восстанавливаться в форме «живых» объектов. **Процесс преобразования содержимого объекта в поток байтов принято называть сериализацией объекта, а обратное преобразование – восстановление объекта из данных потока – десериализацией.** Потоки Object – **ObjectOutputStream** и **ObjectInputStream** - позволяют осуществлять сериализацию и десериализацию объектов. ObjectInputStream и ObjectOutputStream, помимо данных стандартных классов (простых типов, строк и их массивов) позволяют вводить и выводить графы объектов.
Под термином *граф объекта* имеется в виду, что когда содержимое объекта выводится в поток ObjectOutputStream средствами методов writeObject(), в потоки сохраняются наборы байтов, представляющие и текущий объект, и все другие объекты, на который тот ссылается. Поскольку данные об объекте, подвергшимся сериализации, представляются в форме байтов, в семействе потоков Object отсутствуют символьные разновидности Reader и Writer. Результатом десериализации последовательности байтов, представляющих ранее сериализованный объект, из потока ObjectInputStream с помощью методов readObject служит граф объекта, равнозначный исходному.

Пример 5

```
// Сериализация
LinkedList ls = new LinkedList();
FileOutputStream fileOut = new FileOutputStream("list");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(ls);

// Десериализация
FileInputStream fileIn = new FileInputStream("list");
ObjectInputStream in = new ObjectInputStream(fileIn);
LinkedList ls = (LinkedList) in.readObject();
```

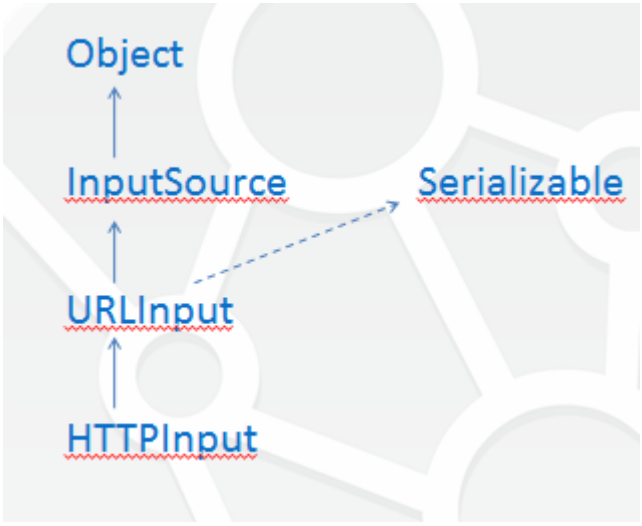
Подготовка классов к сериализации

Необходимым условием возможности осуществления сериализации объектов класса средствами потока ObjectOutputStream является реализация классом интерфейса-маркера Serializable. Поддержка классом интерфейса Serializable свидетельствует о том, что объекты класса готовы к сериализации.
По умолчанию процесс сериализации заключается в сериализации каждого поля объекта, которое не обозначено как **transient** или **static.** Данные простых типов и строки сохраняются с помощью того же механизма, какой поддерживается потоками DataOutputStream, а сериализация объектов выполняется средствами метода **writeObject().**

Схема сериализации, предусмотренная по умолчанию, предполагает, что все поля-объекты, подлежащие сериализации, должны указывать на типы, в свою очередь поддерживающие возможность сериализации. Кроме того, требуется, чтобы класс, базовый по отношению к рассматриваемому, либо обладал конструктором без параметров (чтобы он мог быть вызван в процессе десериализации), либо сам в свою очередь обеспечивал реализацию интерфейса Serializable. В большинстве ситуаций для обеспечения возможности сериализации объектов класса вполне достаточно тех мер, которые предусмотрены схемой сериализации, предлагаемой по умолчанию, и проблема исчерпывается простым упоминанием интерфейса Serializable в объявлении класса.
Схема десериализации, предлагаемая по умолчанию, предусматривает считывание из потока байтовых данных, сохранённых в процессе сериализации. Статические поля класса остаются в неприкосновенности – в ходе загрузки класса будут выполнены все обычные процедуры инициализации, и статические поля получат требуемые исходные значения. Каждому полю transient в восстановленном объекте присваивается значение по умолчанию, соответствующее типу этого поля.
Иногда встречаются классы, которые сами по себе допускают возможность сериализации, но их отдельные элементы – нет. Любая попытка осуществления сериализации объекта, не поддерживающего эту возможность, приводит к выбрасыванию исключения типа NotSerializableException.

Порядок сериализации и десериализации

Каждый класс несёт ответственность за обеспечение надлежащей сериализации состояния собственных объектов, т.е. содержимого их полей.
Сериализация и десериализация объектов выполняется в нисходящем порядке по древовидной иерархии типов – от первого из классов, поддерживающих интерфейс Serializable, до классов более частных типов. Указанный порядок редко играет весомую роль при сериализации, но приобретает существенное значение в процессе десериализации.
Рассмотрим следующую иерархию типов, которой подчиняется класс HTTPInput



При десериализации объекта HTTPInput поток ObjectInputStream вначале выделяет память для нового объекта, а затем находит в иерархии типов ближайший из классов, поддерживающий интерфейс Serializable, - в нашем случае – URLInput. Далее поток вызывает конструктор без параметров для класса, базового по отношению к найденному (ближайшего из числа тех, которые не поддерживают интерфейс Serializable) – в данном случае, InputSource. Если должно быть сохранено иное состояние части объекта, относящейся к базовому классу (InputSource), ответственность за сериализацию и

десериализацию этого состояния возлагается на URLInput. Если же базовый класс, не обеспечивающий сериализацию, обладает собственным значимым состоянием, то почти наверняка придётся отступить от схемы сериализации, предлагаемой по умолчанию, и осуществить дополнительную настройку ближайшего из классов, поддерживающих механизм сериализации. Если такой класс непосредственно наследует класс Object, процедура настройки окажется простой, поскольку Object не обладает каким-либо особым состоянием, подлежащим сохранению и последующему восстановлению. Как только ближайший из классов, поддерживающих механизм сериализации, завершает восстановление данных того подмножества полей, которые относятся к базовому классу, он приступает к воссозданию своего собственного состояния, считывая данные из потока. Затем поток ObjectInputStream осуществляет просмотр иерархии объектов в нисходящем порядке и выполняет десериализацию каждого объекта посредством вызовов readObject(). По достижении нижнего уровня дерева иерархии процесс десериализации исходного объекта полностью завершается. В ходе десериализации объекта могут быть найдены ссылки на другие ранее сериализованные объекты.
Эти объекты подвергаются десериализации в том порядке, в котором они обнаруживаются. Прежде чем будут выполнены любые из рассмотренных операций десериализации, соответствующие классы должны быть загружены.
В процессе загрузки выполняется поиск класса с требуемым именем и проверка версии этого класса. Если подлинность и соответствие версии класса установлены, тот должен быть загружен. Если же класс не найден либо не может быть загружен по иным причинам, метод readObject() выбрасывает исключение типа ClassNotFoundException.

Настройка механизма сериализации

Схема сериализации/десериализации, предлагаемая по умолчанию, вполне работоспособна во многих ситуациях, но, увы, не во всех. Для некоторых классов обычные средства десериализации могут оказаться неадекватными или неэффективными. Руководствуясь подобными соображениями, создатели Java предусмотрели возможность включать в состав класса специальные private – версии методов writeObject() и readObject(). Методы активизируются потоками ObjectOutputStream и ObjectInputStream соответственно в тот момент, когда необходимо осуществить сериализацию и десериализацию объекта. Подобные переопределённые варианты методов writeObject() и readObject() вызываются только в контексте объектов соответствующих классов, и методы несут ответственность исключительно за состояние объекта класса как такового, в том числе и в части, унаследованной от базового класса, не поддерживающего механизм сериализации.
Если в составе класса реализованы собственные варианты методов writeObject() и readObject(), они не должны обращаться к одноимённым методам базового класса
private void writeObject(ObjectOutputStream out) throws IOException
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
В объявлении метода writeObject() обозначена вероятность выбрасывания им исключения типа IOException, так как оно может быть сгенерировано любым из «низкоуровневых» методов write(), которые будут здесь вызываться, и если подобное произойдёт, процесс сериализации должен быть завершён аварийным образом. В предложении throws объявления метода readObject() также следует упомянуть исключение IOException – оно может быть выброшено вызываемыми методами read(), а в такой ситуации процесс десериализации следует остановить. Помимо того, метод readObject() способен генерировать и исключение типа ClassNotFoundException, так как в общем случае при десериализации полей текущего объекта может потребоваться загрузка и других классов.
Существует ограничение, которое следует принять во внимание, приступая к настройке механизма сериализации: нельзя присваивать значение полю final в теле метода

readObject(), поскольку поля final могут быть проинициализированы только в блоках инициализации или конструкторах. Механизм сериализации, предусмотренный по умолчанию, однако, способен «обходить» это ограничение – он вполне нормально работает и с теми классами, в составе которых имеются поля final.

Контроль версий объектов

Реализация классов со временем изменяется. Если подобное происходит в промежутке времени между сериализацией и десериализацией объекта этого класса, поток ObjectInputStream способен обнаружить факт внесения изменений. При сохранении объекта вместе с ним записывается уникальный идентификатор номера версии – 64-битовое значение типа long. По умолчанию идентификатор создаётся в виде хеш-кода, построенного на основе информации об именах класса, его членов и базовых интерфейсов. Изменение таких данных служит сигналом о возможной несовместимости версий класса.

При вводе данных об объекте из потока ObjectInputStream считывается также и идентификатор номера версии. Затем предпринимается попытка загрузки соответствующего класса. Если требуемый класс не найден либо идентификатор загруженного класса не совпадает с тем, который считан из потока, метод readObject() выбрасывает исключение типа InvalidClassException.

Если успешно загружены все нужные классы и выявлено совпадение всех идентификаторов, объект, как принято считать, может быть подвергнут десериализации. Таким образом, получается, что любое, даже самое незначительное изменение в структуре класса способно привести к несовместимости версий.

Но во многих случаях модификация классов носит отнюдь не радикальный характер. Существует возможность сохранить совместимость класса, снабжённого дополнительными несущественными нововведениями, с данными ранее сериализованных объектов. Для этого следует принудительно объявить в составе класса специальное поле, содержащее значение идентификатора номера версии, например:

private static final long serialVersionUID = ...;

Конкретное значение идентификатора рассчитывается средствами используемой среды разработки Java. Ничто не мешает использовать в качестве идентификатора и любое произвольное значение, если оно фиксируется в самой первой версии класса, но это не самое лучшее решение, так как вы почти наверняка не сможете столь же тщательно сгенерировать идентификатор, как это делают стандартные инструменты и обеспечить его уникальность, чтобы предотвратить возможные конфликты с хеш-кодами других классов.

Теперь, когда поток ObjectInputStream находит описание требуемого класса и сопоставляет его идентификатор с тем, который сохранён вместе с сериализованными данными, проблема несовместимости не возникнет, даже если реализация класса на самом деле подверглась изменению.

Расширенный класс. Конструкторы расширенных классов. Порядок выполнения конструкторов. Перегрузка и переопределение методов. Совместимость. Явное преобразование типов.

Понятие наследования

Одно из основных преимуществ ООП состоит в возможности наследования или расширения функционального потенциала базового (родительского) класса при построении на его основе производного (дочернего) класса. Объект нового класса допускается применять в любом контексте, где предусмотрено использование экземпляров исходного класса. Последняя возможность носит название **полиморфизма** и означает, что определённый класс способен восприниматься либо сам по себе, либо в качестве любого из исходных классов, которые он наследует. Набор методов и полей класса, открытых для свободного доступа извне, в совокупности с описанием их назначения называют **контрактом класса.**
Существуют две формы практического воплощения концепции наследования:

- **наследование контракта или типа**, в результате чего производный класс получает тип базового и поэтому может быть использован полиморфным образом во всех случаях, где допустимо применение базового класса
- **наследование способов реализации** – производный класс приобретает функциональные характеристики базового в виде набора доступных полей и методов

Очень часто инструменты наследования используются для обеспечения возможности специализации, когда производный класс становится специализированной версией базового класса. В процессе наследования в новом классе может быть предусмотрено, например, изменение способа реализации какого-то метода с целью достижения большей эффективности. При решении задачи расширения класса обе названные выше формы наследования всегда рассматриваются совместно.

Расширенный класс. Конструкторы расширенных классов

Класс может расширить только один класс

```
class MyClass1 {
}
class MyClass2 extends MyClass1{
}
```

Объект расширенного (производного) класса содержит поля, унаследованные от базового класса, и собственные переменные состояния. Чтобы создать объект расширенного класса, надлежит корректно проинициализировать оба набора переменных.
Конструктор производного класса способен обращаться к полям базового, но только базовый класс «осведомлён» о том, как их следует инициализировать, чтобы гарантировать безупречное выполнение контракта. Конструкторы расширенного класса обязаны передавать полномочия по инициализации унаследованных полей, явно или косвенно обращаясь к услугам конструкторов базового класса. Конструктор расширенного класса может напрямую обращаться к конструкторам базового класса посредством ещё одного способа явного вызова конструкторов, предусматривающего применение служебного слова **super.**
Конструкторы расширенных классов могут вызывать друг друга по ключевому слову **this.**

Конструкторы не относятся к категории методов и не наследуются.
Если в первой строке тела конструктора производного класса не вызывается ни другой конструктор того же класса, ни конструктор базового класса, компилятор обеспечивает автоматическое обращение к «базовому» конструктору без аргументов – такой вызов осуществляется в первую очередь, прежде всех других исполняемых выражений тела конструктора. Если же в составе базового класса отсутствует конструктор без параметров, в самом начале тела конструктора производного класса обязана явно присутствовать инструкция вызова какого-либо другого конструктора.
Иногда довольно полезной оказывается ситуация, когда конструкторы расширенного класса обеспечивают передачу аргументов в конструкторы базового класса.
Достаточно общим можно назвать и вариант расширенного класса, в котором нет конструкторов с теми же сигнатурами, что и в базовом.

Порядок выполнения конструкторов

В процессе создания объекта расширенного класса виртуальная машина выделяет память для хранения всех его полей, включая и те, которые унаследованы от базового класса, и последние получают исходные значения по умолчанию, отвечающие их типам. Далее процесс можно разделить на следующие стадии:

- вызов конструктора базового класса
- присваивание исходных значений полям объекта посредством выполнения соответствующих выражений и блоков инициализации
- выполнение инструкций, предусмотренных в теле конструктора

В первую очередь выполняется явное или косвенное обращение к конструктору базового класса. Если осуществляется явный вызов конструктора того же производного класса с помощью ссылки this(), подобная цепочка вызовов конструкторов этого класса выполняется до тех пор, пока не будет найдена инструкция явного или косвенного обращения к конструктору базового класса, после чего таковой и вызывается.
Конструктор базового класса выполняется в соответствии с той же последовательностью стадий – процесс продолжается рекурсивно, завершаясь по достижении конструктора класса Object, поскольку далее не существует конструкторов классов более высокого уровня иерархии. В выражениях, выполняемых в процессе вызова конструктора базового класса, не допускается присутствие ссылок на любые члены текущего объекта.
На второй стадии процесса обрабатываются выражения и блоки инициализации полей в порядке их объявления. В этот момент ссылки на другие члены объекта допустимы, если таковые уже полностью определены. Наконец, выполняются выражения тела конструктора.
Если текущий конструктор был вызван явно, по его завершении управление передаётся в тело конструктора-инициатора, где выполняются оставшиеся инструкции. Процесс повторяется до тех пор, пока управление не будет передано обратно в тело конструктора «исходного» производного класса, название которого было указано в выражении new.
Если во время процесса конструирования выбрасывается исключение, виртуальная машина завершает выполнение выражения new, генерируя то же исключение, и не возвращает ожидаемую ссылку на объект. Поскольку выражение явного вызова конструктора текущего или базового класса должно быть первым в теле конструктора-инициатора, отловить исключение, выброшенное вызванным конструктором, невозможно, иначе существовала бы и вероятность создания объектов с неверным исходным состоянием.
Методы, предназначенные для вызова на стадии конструирования объекта, должны быть спроектированы особенно тщательно, с учётом вышеназванных соображений. В конструкторах следует избегать вызовов методов, допускающих переопределение, т.е. всех тех, которые не помечены как private, static или final.

Пример 1

```
class X {
      protected int xMask = 0x00ff;
      protected int fullMask;
      public X() {
            fullMask = xMask;
      }
      public int mask (int orig) {
            return (orig & fullMask);
      }
}
class Y extends X {
      protected int yMask = 0xff00;
      public Y() {
            fullMask |= yMask;
      }}
```

Этап	Что происходит	xMask	yMask	fullMask
0	**Полям присвоены значения по умолчанию**	**0**	**0**	**0**
1	**Вызван конструктор Y**	**0**	**0**	**0**
2	**Вызван конструктор X**	**0**	**0**	**0**
3	**Вызван конструктор Object**	**0**	**0**	**0**
4	**Проинициализированы поля X**	**0x00ff**	**0**	**0**
5	**Выполнен конструктор X**	**0x00ff**	**0**	**0x00ff**
6	**Проинициализированы поля Y**	**0x00ff**	**0xff00**	**0x00ff**
7	**Выполнен конструктор Y**	**0x00ff**	**0xff00**	**0xffff**

Наследование и переопределение членов

Перегружая унаследованный **метод** базового класса, мы просто добавляем в объявление производного класса новый метод с тем же именем, но другой сигнатурой.
Переопределяя метод, мы изменяем его реализацию, так что при обращении к методу объекта производного класса будет вызвана именно новая версия метода, а не «старая», принадлежащая базовому классу.

При переопределении метода в производном классе его наименование и сигнатура должны оставаться теми же, что объявлены в базовом классе. Переопределённые методы обладают собственными признаками доступа.
В производном классе разрешается изменять уровень доступа к унаследованному методу базового класса, но только в сторону повышения. Уменьшение возможности доступа к методу относительно того уровня, который определён в базовом классе, нарушает контракт базового класса, и экземпляр производного класса теперь не может быть использован в контексте, предусматривающем применение базового. При переопределении метода позволяется изменять и другие модификаторы. Признаками synchronized, native и strictfp разрешено манипулировать совершенно свободно, поскольку они относятся исключительно к особенностям внутренней реализации метода.
Переопределённый метод может быть помечен как final, но тот, который подвергается переопределению – нет. Метод экземпляра производного класса не может обладать той же сигнатурой, что и статический унаследованный метод, и наоборот. Переопределённый метод в производном классе, однако, может быть снабжён модификатором abstract даже в том случае, если в базовом этого предусмотрено не было.
Допускаются и различия методов базового и производного классов, касающиеся предложения throws, если только каждый из типов исключений, перечисленных в объявлении переопределённого метода, совпадает с одним из тех, которые заданы в объявлении «базового» метода, либо является производным от любого из них. Другими словами, каждый их типов исключений в объявлении throws переопределённого метода должен быть полиморфным образом совместим хотя бы с одним из типов, указанных в объявлении метода, который принадлежит базовому классу. Это означает, что предложение throws переопределённого метода может содержать меньше типов, чем перечислено в объявлении метода базового класса, либо больше специфических производных типов, либо то и другое одновременно. В переопределённом методе допускается и отсутствие предложения throws – в этом случае предполагается, что метод не должен генерировать **объявляемые исключения.**

Совместимость

В любой ситуации, когда значение выражения присваивается некоторой переменной, тип выражения должен быть совместим с типом переменной. Если говорить о ссылочных типах, это означает, что выражение должно относиться к тому же типу, что и переменная, либо к соответствующему производному типу. Например, любой метод, для которого предусмотрена передача аргумента типа Attr, способен воспринимать и аргумент типа ColorAttr, если ColorAttr является производным классом базового класса Attr. Подобное принято называть **совместимостью присваивания.** Но обратное утверждение не верно: не удастся ни явно присвоить значение типа Attr переменной класса ColorAttr, ни передать аргумент типа Attr методу, в котором объявлен параметр класса ColorAttr. Те же правила действуют и в отношении выражений, которые возвращаются из тела метода командой return: тип конкретного возвращаемого значения должен быть совместим с тем, который значится в объявлении метода.
Значение null – это специальный случай: null позволено присваивать переменным всех ссылочных типов, включая и массивы.
О типах, находящихся на более высоких ступеньках иерархии классов, говорят как о **более широких, или менее конкретных**, по сравнению с теми, которые расположены ниже. Соответственно, производные типы называют **более узкими, или более конкретными**, нежели их «прародители». Когда в выражении, предусматривающем использование объекта базового класса, применяется объект производного класса, имеет место преобразование с расширением типа.

Подобная операция, допустимость которой проверяется на этапе компиляции, приводит к тому, что объект производного типа интерпретируется программой как объект базового класса. Программисту в этом случае не нужно предпринимать никаких дополнительных усилий.
Обратное действие, когда ссылка на объект базового класса преобразуется в ссылку на объект производного класса, называют **преобразованием с сужением типа.** В этом случае следует явно применить оператор преобразования типов.

Явное преобразование типов

Оператор преобразования типов позволяет сообщить компилятору, что конкретное выражение следует трактовать таким образом, будто оно относится к тому типу, который явно указан. Оператор может быть применён в любых ситуациях, но его обычное употребление связано всё-таки с преобразованиями, предусматривающими сужение типа
That sref = (That) this;
More mref = (More) sref;
Преобразование с расширением типа нередко обозначают терминами **преобразование вверх или безопасное преобразование**, поскольку тип, расположенный на низкой ступени иерархии, приводится к классу более высокого уровня, а подобная операция заведомо допустима. Преобразование с сужением типа соответственно называют **преобразованием вниз** – воздействию подвергается объект базового типа, который должен быть интерпретирован как объект производного типа. Преобразование вниз – это ещё и небезопасное преобразование, поскольку его результат в общем случае может быть неверен. Когда компилятор, анализируя исходный текст программы, встречает выражение явного преобразования типов, он всегда проверяет корректность операции. Если выясняется, что операция некорректна ещё на этапе компиляции, выдаётся соответствующее сообщение об ошибке. Если же компилятор не в состоянии сразу подтвердить возможность преобразования или опровергнуть её, он добавляет в код дополнительные инструкции, призванные проверить код во время его выполнения. Когда подобный контроль даёт отрицательный результат, генерируется исключение типа ClassCastException.

Сокрытие полей. Доступ к унаследованным членам. Возможность доступа и переопределение. Сокрытие статических членов. Служебное слово super. Проверка типа.

Сокрытие полей. Доступ к унаследованным членам

Поля класса не допускают переопределения – их можно только скрыть. Когда в производном классе объявляется поле с тем же именем, что и в базовом, прежнее продолжает существовать, но перестаёт быть доступным, если обращаться к нему непосредственно по имени. Чтобы сослаться на одноимённое поле, принадлежащее базовому классу, следует воспользоваться служебным словом **super** либо сослаться на текущий объект с приведением типа к базовому классу.

Если вызывается метод посредством ссылки на объект, выбор одной из возможных альтернатив обусловливается фактическим классом объекта. Если же осуществляется обращение к полю, в рассмотрение принимается объявленный тип ссылки.

Пример 2

```
class SuperShow {
     public String str = "SuperStr";
     public void show() {
            System.out.println("Super.show: "+str);
     }}
 class ExtendShow extends SuperShow{
     public String str = "ExtendStr";
     public void show() {
            System.out.println("Extend.show: "+str);
     }
     public static void main (String[] args) {
           ExtendShow ext = new ExtendShow();
           SuperShow sup = ext;
           sup.show();
           ext.show();
           System.out.println("sup.str: "+sup.str);
           System.out.println("ext.str: "+ ext.str);
     }
}
```

Результат:

Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str: SuperStr
ext.str: ExtendStr

Механизм переопределения методов позволяет расширять существующий код, наделяя его новыми специализированными функциями, которые не были изначально предусмотрены автором исходного базового класса. Но если речь заходит о полях, трудно представить ситуацию, когда их сокрытие можно было бы считать безусловно полезным. Сокрытие полей формально позволено с той целью, чтобы разработчики существующих базовых классов могли свободно добавлять в них новые поля public и protected, не рискуя причинить вред всем возможным «наследникам». Если бы язык запрещал использование

одноимённых полей в базовом и производном классах, добавление нового поля в существующий базовый класс оказывалось бы чревато проблемами, поскольку в некотором производном классе, возможно, уже объявлено поле с тем же именем.
Важно понимать, что переопределение методов является фундаментальной особенностью объектно-ориентированных языков, и способы вызова методов родительских классов являются важным средством инкапсуляции и повторного использования кода, в то время как сокрытие полей возникает лишь как неудобный побочный эффект от возможности назначить произвольное имя для поля в дочернем классе, и весь механизм доступа к скрытым полям предназначен для преодоления последствий этого побочного эффекта.

Возможность доступа и переопределение

Метод может быть переопределён только в том случае, если он доступен. Метод, помеченный как private, не «виден» за пределами класса. Если вдруг в производном классе «объявится» метод с теми же сигнатурой и типом возвращаемого значения, что и метод private базового класса, это будут два совершенно разных и не связанных между собой метода – метод производного класса в этой ситуации нельзя квалифицировать как переопределённый. Коротко говоря, обращение к методу private всегда приводит к вызову метода, объявленного в текущем классе.

Сокрытие статических членов

Статические члены класса не могут быть переопределены, поскольку всегда скрыты – как поля, так и методы. Обращение к каждому статическому полю или методу чаще всего осуществляется посредством задания имени класса, в котором они объявлены. Из факта сокрытия статического поля или метода в результате объявления одноимённого члена производного класса вытекает одно следствие – если для доступа к статическому члену используется ссылка, то, как и в ситуации с сокрытием полей, при выборе подходящей альтернативы компилятор учитывает объявленный тип ссылки, а не тип объекта, на который она указывает.

Служебное слово super

Служебное слово super может быть использовано в теле любого нестатического члена класса. При доступе к полю или вызове метода выражение super действует как ссылка на текущий объект, представленный как экземпляр базового класса.
Применение super – это единственный вариант, когда выбор метода обусловливается типом ссылки. Вызов super.method() всегда означает обращение к методу method базового класса – никакие возможные переопределённые реализации метода в производных классах в расчёт не принимаются.

Пример 3

```
class That {
  protected String nm() {
          return "That";
     }}
 class More extends That {
     protected String nm() {
          return "More";
     }
     protected void printNM() {
          That sref = (That) this;
          System.out.println("this.nm() = " + this.nm());
```

```
            System.out.println("sref.nm() = " + sref.nm());
            System.out.println("super.nm() = " + super.nm());
    }}
```

Результат:
this.nm() = More
sref.nm() = More
super.nm() = That

Проверка типа

Часто возникает задача определения принадлежности объекта тому или иному типу, и решить её позволяет оператор instanceof, возвращающий в результате вычисления значение true, если выражение левой части совместимо с типом, название которого указано в правой части, и false – в противном случае. Следует иметь в виду, что null нельзя причислить к какому бы то ни было типу, и поэтому результат применения instanceof по отношению к null всегда равен false.
Используя instanceof, мы можем загодя убедиться в правомерности преобразования с сужением типа, которое хотим выполнить, и избежать возникновения исключительных ситуаций.

Пример 4

```
if (sref instanceof More)
    mref = (More)sref;
```

Заметим, что мы всё ещё обязаны применять оператор преобразования типов, чтобы сообщить компилятору о наших истинных намерениях.

Методы и классы final. Методы и классы abstract. Класс Object. Методы класса Object. Клонирование объектов.

Методы и классы final

Обозначая метод класса модификатором final, мы имеем в виду, что ни один производный класс не в состоянии переопределить этот метод, изменив его внутреннюю реализацию. Класс в целом может быть помечен как final:

```
final class NoExtending {
     //…
}
```

Класс, помеченный как final, не поддаётся наследованию и все его методы косвенным образом приобретают свойство final. Применение признака final в объявлениях классов и методов способно повысить уровень безопасности кода. Во многих случаях для достижения достаточного уровня безопасности кода вовсе нет необходимости обозначать весь класс как final – вполне возможно сохранить способность класса к расширению, пометив модификатором final только его «критические» структурные элементы. В этом случае вы оставите в неприкосновенности основные функции класса и одновременно разрешите его наследование с добавлением новых членов, но без переопределения «старых».

Разумеется, поля, к которым обращается код методов final, должны быть в свою очередь помечены как final или private, так как в противном случае любой производный класс получит возможность изменить их содержимое, воздействуя на поведение соответствующих методов.

Ещё один эффект применения модификатора final связан с упрощением задачи оптимизации кода, решаемой компилятором. Когда вызывается метод, не помеченный как final, исполняющая система определяет фактический класс объекта, связывает вызов с наиболее подходящим кодом из группы перегруженных методов и передаёт управление этому коду. Но если метод, например, getName(), обозначен как final, операция обращения к нему упрощается. В самом простом случае, подобном тому, который касается getName(), компилятор может заменить вызов метода кодом его тела. Такой механизм носит название **встраивания кода.** При этом два следующих выражения выполняются совершенно одинаково:

```
System.out.println ("id = " + rose.name);
System.out.println ("id = " + rose.getName());
```

Та же схема оптимизации может быть применена компилятором и по отношению к методам private и static, так как и они не допускают переопределения. Использование модификатора final в объявлениях классов способствует также повышению эффективности некоторых операций проверки типов. В этом случае многие подобные операции могут быть выполнены уже на стадии компиляции и поэтому потенциальные ошибки выявляются гораздо раньше. Если компилятор встречает в исходном тексте ссылку на класс final, он может быть «уверен», что соответствующий объект относится именно к тому типу, который указан. Компилятор в состоянии сразу определить место, занимаемое классом в общей иерархии классов, и проверить, верно тот используется или нет. Если модификатор final не применяется, соответствующие проверки осуществляются только на стадии выполнения программы.

Методы и классы abstract

Вы можете объявить абстрактный класс, представив только часть его реализации и заранее предусмотрев возможность переопределения всех или некоторых методов в производных классах. До сих пор мы имели дело с **конкретными классами,** в которых каждый метод был объявлен полностью.
Абстрактные классы находят широкое применение в тех случаях, когда, например, некоторые признаки поведения класса приемлемы для всех его объектов, но при этом существуют и такие функциональные особенности, которые целесообразно реализовать только в определённых производных классах, а не в базовом классе непосредственно. Объявления подобных классов снабжают модификатором abstract, тем же признаком помечают и методы класса, в объявлении которого отсутствует блок тела. Если необходимо создать класс, в котором все методы должны быть абстрактными, возможно, имеет смысл воспользоваться объявлением интерфейса. Во многих случаях код, относящийся к сфере компетенции самого базового абстрактного класса, - это хороший претендент на приобретение статуса final, гарантирующего, что контракт класса не будет нарушен.
Любой класс, содержащий методы abstract, сам должен быть обозначен как abstract. Абстрактный метод должен быть переопределён в любом производном классе, если только тот не помечен как abstract. В любом производном классе позволено переопределять конкретные методы базового класса, помечая их признаком abstract.
Объекты абстрактного класса нельзя создавать, поскольку заведомо известно, что какие-то его методы, которые могут быть вызваны прикладной программой, не реализованы.

Класс Object

Класс Object находится на вершине иерархии классов Java. Object явно или косвенно наследуется всеми классами, поэтому переменная типа Object способна указывать на объект любого типа, будь то экземпляр какого-либо класса или массив.
Любой переменной типа Object нельзя непосредственно присваивать значения простых типов (таких как int, boolean и т.п.), но эти ограничения легко обойти, «запаковав» значения в объекты соответствующих классов-оболочек (Integer, Boolean и др.), которые будут описаны далее. В составе класса Object определена целая группа методов, которые наследуются всеми производными классами. Эти методы можно условно разделить на две категории – прикладные методы и методы, обеспечивающие поддержку многопоточных вычислений.

Методы класса Object

- **public boolean equals(Object obj)**
  Проверяет, равны ли текущий объект и объект, на который указывает ссылка obj, переданная в качестве параметра, и возвращает значение true, если факт равенства установлен, и false – в противном случае. Если необходимо проверить, указывают ли две ссылки на один и тот же объект, следует применять операторы == или !=. Метод equals сопоставляет содержимое объектов. В исходной реализации метода equals, предусмотренной в классе Object, предполагается, что объект равен только самому себе, т.е.удовлетворяет условию this ==obj.
- **public int hashCode()**
  Возвращает значение хеш-кода текущего объекта – числа, используемого для быстрого сравнения объектов. Каждый объект обладает собственным хеш-кодом, который находит применение в хеш-таблицах. В реализации по умолчанию

предусмотрен возврат значения, которое, как правило, различно для разных объектов. Значение кода используется в процессе сохранения объекта в одной из хеш-коллекций. Если объект не изменял свое состояние, то значение хэш-кода не должно изменяться.

- **protected Object clone() throws CloneNotSupportedException**
  Возвращает клон текущего объекта. Клон – это новый объект, являющийся копией текущего.
- **public final Class getClass()**
  Возвращает объект типа Class, который представляет информацию о классе текущего объекта на этапе выполнения программы.
- **protected void finalize() throws Throwable**
  Позволяет выполнить необходимые операции очистки состояния объекта до того момента, когда объект будет уничтожен в процессе сборки мусора.
- **public String toString()**
  Возвращает строковое представление объекта. Метод toString() вызывается неявно, когда ссылка на объект употребляется в качестве операнда в контексте выражений конкатенации строк с помощью оператора +. Версия метода toString(), реализованная в классе Object, по умолчанию возвращает строку, содержащую наименование класса, которому принадлежит текущий объект, символ @ и шестнадцатеричное представление хеш-кода объекта.

hashCode() и equals()

Оба метода, hashCode() и equals(), должны быть переопределены, если необходимо придать понятию равенства объектов иной смысл, отличный от того, который предлагается классом Object. По умолчанию считается, что любые два различных объекта «не равны», т.е. метод equals() возвращает значение false, и их хеш-коды, как правило, различны. Если версия метода equals(), реализованная в некоем классе, допускает «равенство» двух различных объектов, их хеш-коды, возвращаемые соответствующими вариантами метода hashCode(), также должны быть равны. **Если хэш-коды объектов одинаковы, то это еще не значит, что объекты эквивалентны.** Механизм хеширования тесно связан с методом equals(), возвращающим true, если в хеш-таблице найден ключ с заданным значением. **Изменение реализации в классе метода equals() влечет за собой изменение реализации метода hashCode().**

Клонирование объектов

Результатом клонирования является копия объекта.
Массивы поддерживают операцию клонирования:
int[] arrayCopy = (int []) array.clone();
В классе Object метод clone() является защищенным. Метод clone() реализуется в конкретном классе. Никто не гарантирует того, что результатом его выполнения будет копия объекта, и даже того, что новый объект будет того же класса.
Однако существует ряд соглашений, регламентирующих реализацию метода clone():

- Класс должен реализовывать интерфейс-маркер (пустой интерфейс) Cloneable. Метод Object.clone() проверяет, реализован ли в классе, которому принадлежит текущий объект, интерфейс Cloneable, и выбрасывает исключение типа CloneNotSupportedException, если ответ отрицательный
- Класс должен переопределять метод clone(). Результатом работы метода Object.clone() является точная копия объекта, т.е. Object.clone() обеспечивает простое клонирование - копирование всех полей исходного объекта в объект-копию и по завершении работы возвращается ссылка на созданный объект-

копию. Метод вполне работоспособен во многих ситуациях, но при определённых обстоятельствах приходится его переопределять и дополнять

- Результат клонирования должен быть получен вызовом super.clone()

Пример 5

```
public Object clone() {
    Object result = null;
    try {
        result = super.clone();
    } catch (CloneNotSupportedException ex) { }
    return result;
}
```

Простого клонирования может быть недостаточно, в этом случае применяется **глубокое клонирование.** В процессе глубокого клонирования соответствующие методы clone вызываются для каждого объекта, обозначенного переменной-полем, и каждого элемента массива объектов. Процесс носит рекурсивный характер – клонированию подвергаются все объекты, служащие членами других объектов, начиная от текущего. Т.е. если объект содержит ссылки на агрегированные объекты, после процедуры простого клонирования необходимо создать и их копии тоже.

Пример 6

```
public Object clone() {
  Object result = null;
  try {
    result = super.clone();
    result.a = (...) a.clone();
    ...
  } catch (CloneNotSupportedException ex) { }
  return result;
}
```

Рефлексия. Возможности и участники механизма рефлексии. Получение представления класса. Возможности класса Class. Передача параметров в методы. Создание экземпляров классов. Вызов методов. Вызов статического метода.

Рефлексия и ее возможности

Рефлексия – возможность программы анализировать саму себя, взаимодействуя с виртуальной машиной Java (JVM).
Возможности механизма рефлексии:

- Загрузка типов во время исполнения программы
- Исследование структуры типов и их элементов
- Создание экземпляров классов
- Вызов методов
- Загрузка классов из набора байтов

Участники механизма рефлексии

- **Класс java.lang.Class**
  - Класс является метаклассом по отношению к другим типам
  - Экземпляры класса **Class** описывают классы и интерфейсы выполняемого приложения
  - Методы класса **Class** позволяют исследовать содержимое описываемого класса и его свойства
- **Класс java.lang.ClassLoader**
  - Реализует механизмы загрузки классов
- **Пакет java.lang.reflect**
  Содержит ряд дополнительных и вспомогательных классов:
  - **Field** – описывает поле объекта
  - **Method** – описывает метод объекта
  - **Constructor** – описывает конструктор объекта
  - **Modifier** – инкапсулирует работу с модификаторами
  - **Array** – инкапсулирует работу с массивами

Получение представления класса

- **Метод Class Object.getClass()**
  Возвращает ссылку на представление класса, экземпляром которого является объект
- **Псевдополе Object.class**
  Ссылка на представление указанного класса
- **Метод static Class Class.forName()**
  Возвращает ссылку на представление класса, имя которого указывается параметром типа String

Возможности класса Class

- **static Class forName(String name)**
  Загрузка класса в JVM по его имени
- **boolean isInterface()**
- **boolean isLocalClass()**
  Определение вида типа
- **Class getSuperclass()**
- **Class[] getInterfaces()**
  Получение родительских типов
- **Class[] getClasses()**
  Получение вложенных типов
- **Class Class.getDeclaringClass()**
  Для вложенных типов возвращает ссылку на объект Class внешнего типа
- **Object newInstance()**
  Создание объекта
- **Field[] getFields()**
- **Field getField(...)**
  Получение списка всех public полей и конкретного поля по имени
- **Method[] getMethods()**
- **Method getMethod(...)**
  Получение списка всех public методов и конкретного метода по имени и списку типов параметров
- **Constructor[] getConstructors()**
- **Constructor getConstructor(...)**
  Получение списка всех конструкторов и конкретного конструктора по списку типов параметров

Пример 1

```
import java.lang.reflect.*;
class ListMethods {
     public static void main(String[] argv)   throws
ClassNotFoundException {
            Class c = Class.forName(argv[0]);
            Constructor[] cons = c.getConstructors( );
            printList("Constructors", cons);
            Method[] meths = c.getMethods( );
            printList("Methods", meths);
            Field[] fields = c.getFields();
          printList("Fields", fields);
     }
     static void printList(String s, Object[] o) {
           System.out.println( s );
           for (int i = 0; i < o.length; i++)
               System.out.println(o[i].toString( ));
     }
}
```

Передача параметров в методы

Поскольку на момент написания программы типы и даже количество параметров неизвестно, используется другой подход:

- Ссылки на все параметры в порядке их следования помещаются в массив типа **Object**
- Если параметр имеет примитивный тип, то в массив помещается ссылка на экземпляр класса-оболочки соответствующего типа, содержащий необходимое значение

Возвращается всегда тип **Object**

- Для ссылочного типа используется приведение типа или рефлексивное исследование
- Для примитивных типов возвращается ссылка на экземпляр класса-оболочки, содержащий возвращенное значение

Создание экземпляров классов

- **Object Class.newInstance()**
  Возвращает ссылку на новый экземпляр класса, используется конструктор по умолчанию
- **Object Constructor.newInstance(Object[] initArgs)**
  Возвращает ссылку на новый экземпляр класса, с использованием конструктора и указанными параметрами конструктора

Вызов методов

- **Прямой вызов,** если на момент написания кода известен класс-предок загружаемого класса
- **Вызов через экземпляр класса Method**
  Object Method.invoke(Object obj, Object[] args)
  - obj – ссылка на объект, у которого должен быть вызван метод, принято передавать null, если метод статический
  - args – список параметров для вызова методов

Пример 2

```
import java.lang.reflect.*;
public class Main {
      public static void main(String[] args) {
             if (args.length == 3) {
                try {
                   Class c = Class.forName(args[0]);
                 Method m = c.getMethod(args[1], new Class [] {Double.TYPE});
                 Double val = Double.valueOf(args[2]);
                 Object res = m.invoke(null, new Object [] {val});
                 System.out.println(res.toString());
              } catch (ClassNotFoundException e)  { System.out.println("Класс
не найден"); }
                catch (NoSuchMethodException e) { System.out.println("Метод
не найден"); }
                catch (IllegalAccessException e)       {
System.out.println("Метод недоступен");
                      } catch (InvocationTargetException e) {
                        System.out.println("При вызове возникло исключение");
              }
          }  }  }
```

Статический импорт. Автоупаковка и автораспаковка (автобоксинг). Аргументы переменной длины. Улучшенный цикл for.

Статический импорт

- Имеется:
  ```
  hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
  ```
- Хотелось бы:
  ```
  hypot = sqrt(pow(side1, 2) + pow(side2, 2));
  ```
- **Импорт элемента типа**
  import static pkg.TypeName.staticMemberName;
  import static java.lang.Math.sqrt;
  import static java.lang.Math.pow;
- **Импорт всех элементов типа**
  import static pkg.TypeName.*;
  import static java.lang.Math.;*

Помимо импорта статических членов классов и интерфейсов, определенных в API языка Java, вы можете использовать это средство для импорта статических членов классов и интерфейсов, созданных вами.

Особенности статического импорта

- Повышает удобство написания программ и уменьшает объем кода
- Уменьшает удобство чтения программ
- Приводит к конфликтам имен

Вывод: рекомендуется к использованию только когда действительно необходим (статический импорт желателен в тех случаях, когда статический член используется многократно).

Автоупаковка/автораспаковка

- Имеется:
  ```
  List list = new LinkedList();
  list.add(new Integer(1));
  list.add(new Integer(10));
  ```
- Хотелось бы:
  ```
  List list = new LinkedList();
  list.add(1);
  list.add(10);
  ```

Автоупаковка – процесс автоматической инкапсуляции данных простого типа в экземпляр соответствующего ему класса-обертки в случаях, когда требуется значение ссылочного типа.
Автораспаковка – процесс автоматического извлечения примитивного значения из объекта-упаковки в случаях, когда требуется значение примитивного типа

```
List list = new LinkedList();
list.add(1);
int b = (Integer)list.get(0) + 10;
```

С появлением Java 2 v5.0 стало возможным использование объекта типа Boolean для управления оператором if и любыми операторами цикла языка Java:

```
Boolean b;
//
while  (b)   {  //
```
Кроме удобства, которое предоставляет механизм автоупаковки/распаковки, он может помочь в предупреждении ошибок. Так как автоупаковка всегда создает правильный объект, а автораспаковка всегда извлекает надлежащее значение, не возникает ситуаций для формирования неверного типа объекта или значения. В тех случаях, когда вам нужен тип, отличающийся от созданного автоматическим процессом, вы и сейчас можете упаковывать и распаковывать значения вручную так, как делали это раньше.

Особенности автоупаковки

- Происходит при присваивании, вычислении выражений и при передаче параметров
- **Объекты создаются без использования ключевого слова new**
  `Integer i = 15;`
- Ошибочно полагать, что примитивные типы не нужны (для хранения элементарных типов данных из-за более высокой производительности применяются базовые типы, а не объекты)
- Автоупаковка требует существенных ресурсов
- Злоупотреблять автоупаковкой не стоит

Аргументы переменной длины

- Имеется:
  ```
  int s1 = sum(new int[] {1, 2});
  int s2 = sum(new int[] {1, 2, 3});
  ```
- Хотелось бы:
  ```
  int s1 = sum(1, 2);
  int s2 = sum(1, 2, 3, 4);
  ```

Пример 1
```
int sum(int ... a) {
  int s = 0;
  for (int i = 0; i < a.length; i++)
    s += a[i];
return s;
}
int s2 = sum(1, 2, 3);
int s1 = sum(new int[] {1, 2});
```

Особенности переменного количества аргументов

- Внутри там все равно находится массив
- Аргумент переменной длины в методе может быть только один
- В список параметров метода могут быть включены "обычные" (обязательно указываемые при вызове метода) параметры наряду с параметром переменной длины
- В этом случае аргумент переменной длины должен быть последним в списке аргументов метода
- В сочетании с перегрузкой методов аргумент переменной длины способен приводить к ошибкам компиляции ввиду неоднозначности кода
  ```
  static void vaTest(int …v) { //...
  static void vaTest(int n, int …v)  { //...
  vaTest(1)
  ```

Улучшенный цикл for

- Имеется:
```
int[] nums = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < 5; i++)
     sum += nums[i];
```
- Хотелось бы:
```
int[] nums = {1, 2, 3, 4, 5};
int sum = 0;
for (int x: nums)
    sum += x;
```
- **Общая форма записи:**
  for (type iterVar : iterableObj) statement
  Где **type** – тип элемента, **iterVar** – переменная цикла, **iterableObj** – объект с элементами, **statement** – тело цикла

Работа улучшенного цикла for

- В каждом витке цикла «извлекается» очередной элемент агрегата
- Ссылка на него (для ссылочных типов) или значение (для примитивных) помещается в переменную цикла
- Тип переменной цикла должен допускать присвоение элементов агрегата
- Цикл выполняется до тех пор, пока не будут перебраны все элементы агрегата

Пример 13
```
int sum = 0;
int nums[][] = new int[3][5];
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 5; j++)
    nums[i][j] = (i + 1) * (j + 1);
for (int[] x: nums)
  for (int y: x)
    sum += y;
```

Особенности улучшенного цикла for

- По сути это внутренний итератор
- Переменная цикла доступна только для чтения
- Порядок обхода в целом не определен
- Нет доступа к соседним элементам
- Агрегат обязан реализовывать интерфейс **java.lang.Iterable<T>**
- Этот интерфейс содержит лишь один элемент
  Iterator<T> iterator()
- Данный, вроде бы знакомый, интерфейс, тоже претерпел некоторые изменения:
  - boolean hasNext()
  - void remove()
  - T next()
- Область применения обобщенного цикла for «несколько уже», чем у «необобщенной» версии

- Зато для этого класса задач (поиск среднего арифметического, поиск минимального или максимального значения в наборе, поиск дублирующих значений и т. д.) синтаксис обобщенного цикла существенно удобнее

Настраиваемые типы и их особенности. Ограниченные типы. Метасимвольный аргумент. Метасимвол с ограничениями. Настраиваемые методы, конструкторы, интерфейсы. Примеры.

Настраиваемые типы

- Имеется:
```
List list= new LinkedList ();
list.add(10);
list.add(5);
list.add((Integer)list.get(0) + (Integer)list.get(1));
```
- Хотелось бы:
```
List<Integer> list= new LinkedList<Integer>();
list.add(10);
list.add(5);
list.add(list.get(0) + list.get(1));
```

- Позволяют создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр
- Класс, интерфейс или метод, работающие с параметризованными типами, называются **настраиваемыми**
- Позволяют создавать более компактный код, чем универсальные (обобщенные) типы
- Обеспечивают автоматическую проверку и приведение типов
- Позволяют создавать хороший повторно используемый код

Пример 2
```
class Gen<T> {
    T ob;
    Gen(T o) {
        ob = o;
    }
    T getob() {
        return ob;
    }
    void showType() {
        System.out.println("Type of T is " +
ob.getClass().getName());
     }
}

class GenDemo {
    public static void main(String args[]) {
           Gen<Integer> iOb;
           iOb = new Gen<Integer>(88);
           iOb.showType();
           int v = iOb.getob();
           System.out.println("value: " + v);
```

```
            System.out.println();
            Gen<String> strOb = new Gen<String>("Generics Test");
            strOb.showType();
            String str = strOb.getob();
            System.out.println("value: " + str);
}
}
```

Результат:
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test

Общий синтаксис:

- **Объявление настраиваемого типа**
  class имяКласса<список-параметров-типа> {...}

class Generic2<T, E> {...}

- **Создание ссылки и объекта настраиваемого типа**
  имяКласса<список-параметров-типа> имяПеременной = new имяКласса<список-параметров-типа>(список-аргументов);

Generic2<Integer, String> gObj = new Generic2<Integer, String>(10, "ok");

Ограниченные типы

При объявлении параметра типа вы можете задать верхнюю границу, определяющую суперкласс, от которого должны быть унаследованы все аргументы типа. Такое ограничение устанавливается с помощью ключевого слова extends при описании параметра типа

<T extends superclass>

Приведенное объявление указывает на то, что параметр T можно заменить только типом superclass или его подклассами (производными от него классами). Таким образом, superclass задает верхнюю границу включительно.

Пример 3

```
class Stats<T extends Number> {
  T[] nums;
  Stats(T[] o) {nums = o;}
  double average() {
    double sum = 0.0;
    for(int i = 0; i < nums.length; i++)
      sum += nums[i].doubleValue();
    return sum / nums.length;
  }
}

class BoundsDemo {
     public static void main(String args[]) {
           Integer inums[] = { 1, 2, 3, 4, 5 };
           Stats<Integer> iob = new Stats<Integer>(inums);
           double v = iob.average();
           System.out.println("iob average is " + v);
           Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
           Stats<Double> dob = new Stats<Double>(dnums);
```

```
            double w = dob.average();
            System.out.println("dob average is " + w);
            // Эти строки не будут компилироваться,так как String
            // не является подклассом суперкласса Number
            // String strs[]   =   { "1", "2", "3", "4", "5"  };
            // Stats<String> strob = new Stats<String>(strs);
            // double x = strob.average() ;
            // System.out.printlnf"strob average is  "  + v);
            }
}
```

Результат:
Average is 3.0
Average is 3.3

- Как имя типа может быть указан интерфейс
- Как имя типа может быть указан ранее введенный параметр

Метасимвольный аргумент

Пример 4

```
boolean sameAvg(Stats<T> ob) {
if ((average) == ob.average())
  return true;
return false;
}
```

К сожалению, приведенный пример будет обрабатывать только те объекты класса Stats, у которых тип такой же, как у объекта, вызвавшего метод. Например, если метод вызывает объект типа Stats<Integer>, параметр ob должен тоже быть типа Stats<Integer>. Для создания универсального метода вы должны использовать другую функциональную возможность средств настройки типов — метасимвольный аргумент, или символьную маску. **Метасимвольный аргумент задается знаком ? и представляет неизвестный тип.**

Пример 5

```
boolean sameAvg(Stats<?> ob) {
  if ((average) == ob.average())
    return true;
  return false;
}
```

Важно понять, что метасимвол не влияет на тип создаваемого объекта класса Stats. Тип определяется ключевым словом extends в объявлении класса Stats. Метасимвол, или маска, обеспечивает совместимость любых допустимых объектов типа Stats.

Метасимвол с ограничениями

Ограниченный метасимвол, или маска, задает как верхнюю, так и нижнюю границы аргумента типа. Это позволяет ограничить набор типов объектов, которые будут обрабатываться методом.

- **Ограничение сверху:**
  <? extends super>
  Оно констатирует, что маска ? соответствует типу super и любому производному от него классу
  Тип super допускается
- **Ограничение снизу**

<? super sub>
В этом случае, допустимыми аргументами считаются только суперклассы класса, заданного как sub
Тип sub не допускается

Настраиваемые методы

Как было показано ранее, методы в настраиваемых классах могут использовать параметр типа и таким образом автоматически становятся настраиваемыми по отношению к этому параметру типа. Однако можно и объявить настраиваемый метод с одним или несколькими собственными параметрами типа. Более того, есть возможность создать настраиваемый метод внутри ненастраиваемого класса.

Пример 6

```
class GenMethDemo {
          // Определяет, является ли объект элементом массива
     static <T, V extends T> boolean isIn(T x, V[] y) {
           for(int i=0; i < y.length; i++)
                if(x.equals(y[i])) return true;
           return false;
     }
      public static void main(String args[]) {
          // Использует метод isin() для объектов типа Integer
          Integer nums[] = { 1, 2, 3, 4, 5 };
          if(isIn(2, nums))
              System.out.println("2 is in nums");

if(!isIn(7, nums))
      System.out.println("7 is not in nums");
  System.out.println();
 // Использует метод isIn() для объектов типа String
 String strs[] = { "one", "two", "three",  "four", "five" };
     if(isIn("two", strs))
        System.out.println("two is in strs");
     if(!isIn("seven", strs))
        System.out.println("seven is not in strs");
     //Не откомпилируется, поскольку типы не совместимы
   //   if(isln("two", nums})
   //   System.out.println("two is in strs”);
  }  }
```

Результат:
2 is in nums
7 is not in nums
two is in strs
seven is not in strs

Далее приведена синтаксическая запись для настраиваемого метода:
<type-param-list> ret-type meth-name(param-list) { //...
type-param-list всегда представляет собой разделенный запятыми список параметров типа. Обратите внимание на то, что у настраиваемых методов этот список предшествует типу значения, возвращаемого методом.

Ряд особенностей

Конструкторы так же могут быть настраиваемыми (даже если сам класс – нет)

Пример 7

```
class GenCons {
  private double val;
    <T extends Number> GenCons(T arg) {
    val = arg.doubleValue();
  }
  void showval() {
    System.out.println("val: " + val);
  }
}

class GenConsDemo {
  public static void main(String args[]) {
       GenCons test = new GenCons(100);
       GenCons test2 = new GenCons(123.5F);
       test.showval();
       test2.showval();
   }
}
```

Результат:
val: 100.0
val: 123.5

Интерфейсы так же могут быть настраиваемыми

Пример 8

```
interface MinMax<T extends Comparable<T>> {
   T min();
   T max();
}
// Теперь реализуем MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T>
    T[] vals;
     MyClass(T[] o) { vals = o; }

 public T min() {
    T v = vals[0];
     for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0) v = vals[i];
     return v;   }
public T max() {
    T v = vals[0];
     for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) > 0) v = vals[i];
     return v;   }
}

 class GenIFDemo {
     public static void main(String args[]) {
          Integer inums[] = {3, 6, 2, 8, 6 };
```

```
        Character chs[] = {'b', 'r', 'p', 'w' };
        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs);
        System.out.println("Max value in inums: " + iob.max());
        System.out.println("Min value in inums: " + iob.min());
        System.out.println("Max value in chs: " + cob.max());
        System.out.println("Min value in chs: " + cob.min());
} }
```

Результат:
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b

- ```
  // Это неправильно!
  class MyClass<T extends Comparable<T>>
  implements MinMax<T extends Comparable<T>>
  ```
- ```
  // Это тоже неправильно!
  class MyClass implements MinMax<T> { //
  ```

Вообще, если класс реализует настраиваемый интерфейс, этот класс также должен быть настраиваемым, по крайней мере, до той степени, которая обеспечивает получение параметра типа и передачу его в интерфейс.
Применение настраиваемого интерфейса обладает двумя преимуществами:

- Во-первых, интерфейс можно реализовать для данных разных типов
- Во-вторых, у вас появляется возможность наложить ограничения (т. е. установить границы) на типы данных, для которых может быть реализован интерфейс. В случае интерфейса MinMax, например, только типы, реализующие интерфейс Comparable, могут передаваться для замены параметра T

Далее приведена обобщенная синтаксическая конструкция для описания настраиваемого интерфейса:
interface interface-name<type-param-list> {//...
В данной записи type-param-list — это разделенный запятыми список параметров типа. Когда настраиваемый интерфейс реализуется, вы должны заменить его списком аргументов типа
class class-name<type-param-list>
implements interface-name<type-param-list> {//..

Нельзя создавать объекты, используя типы-параметры:
Пример 9

```
class Gen<T> {
  T ob;
  Gen() {
    ob = new T(); //Неверно
  }
}
```

Статические члены класса не могут использовать его параметры типа
Пример 10

```
class Wrong<T> {
   // Неверно
```

```
   static T ob;
  // Неверно
   static T getob() {
      return ob;
   }
}
```

- **Настраиваемый класс не может расширять класс Throwable (нельзя создавать настраиваемые классы исключений)**
- **От настраиваемых типов можно наследовать, есть ряд особенностей**
- **Нельзя создать экземпляр массива, у которого базовый тип задан с помощью параметра типа**

Пример 11

```
class Generic<T> {
    T[] vals; // Верно
    Generic(T[] nums) {
        //vals = new T[10]; //Неверно
        vals = nums; // Верно
     } }
```

- **Нельзя создать массив из ссылок на объекты конкретной версии настраиваемого типа**

  ```
  Generic<Integer>[] gens = new Generic<Integer>[10];
  //Неверно
  Generic<?>[] gens = new Generic<?>[10]; // Верно
  ```
- **И как же это работает?**
- **Механизм стирания:**
  - **В реальном байт-коде никаких настраиваемых типов в целом-то и нет…**
  - **Вся информация о настраиваемых типах удаляется на стадии компиляции**
  - **Именно компилятор осуществляет контроль безопасности приведения типов**
  - **А внутри после компиляции все те же «обобщенные» классы, явные приведения типов и т.д.**
- **Возможны ошибки неоднозначности**

Пример 12

```
class MyGenClass<T, V> {
    T ob1;
    V ob2;
    // Неверно
    void set(T o) {
         ob1 = o;
    }
    void set(V o) {
         ob2 = o;
    }
}
```

Перечислимые типы. Метаданные.

Перечислимые типы

В простейшей форме **перечислимый тип — это список именованных констант.** Перечислимый тип создается с помощью нового ключевого слова **enum.**

- Имеется:

```
class Apple {
        public static final int JONATHAN = 0;
        public static final int GOLDENDEL = 1;
        public static final int REDDEL = 2;
        public static final int WINESAP = 3;
        public static final int CORTLAND = 4;
}
```

- Хотелось бы:

```
enum Apple {
     Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

- Перечислимый тип Apple. Идентификаторы Jonathan, GoldenDel и так далее называются **константами перечислимого типа**
- Каждая объявляется неявно общедоступным, статическим членом класса Apple
- После того как вы описали перечислимый тип, можете создать переменную этого типа
- Но, несмотря на то что перечислимые типы определены как классы, вы не можете создать экземпляр типа enum с помощью операции new
- Вместо этого вы объявляете и используете переменную перечислимого типа почти так же, как вы поступаете с переменными одного из базовых типов, таких как int или char

```
Apple ap;
ap = Apple.RedDel;
```

- Можно проверить равенство двух констант перечислимого типа с помощью операции отношения = =

```
if(ap = = Apple.GoldenDel)    //
```

- Значение перечислимого типа можно использовать для управления оператором switch

```
switch(ap) {
       case Jonathan: //...
      case Winsap: //...
 default: //...
}
```

Во все перечислимые типы автоматически включены два метода - values() и valueOf()Ж

- **public static enumType[] values()**
  возвращает ссылку на массив ссылок на все константы перечислимого типа

```
Apple[] allApples = Apple.values();
```

- **public static enumType valueOf(String str)**
  возвращает константу перечислимого типа, имя которой соответствует указанной строке, иначе выбрасывает исключение

```
Apple ap = Apple.valueOf("Jonathan");
```

Замечания

- Перечислимый тип в языке Java — это класс
- Несмотря на то, что вы не можете инициализировать переменную типа enum с помощью операции new, у перечислимого типа много функциональных возможностей таких же, как у других классов
- К примеру, вы можете создать для него конструкторы, добавить поля и методы и даже реализовать интерфейсы
- Конструктор вызывается отдельно для каждой константы
- Перечислимый тип не может наследовать другой класс, но все перечислимые типы автоматически наследуют от класса **java.lang.Enum**
- Тип enum не может быть суперклассом (не может расширяться)
- Клонировать экземпляры перечислимых типов нельзя, сравнивать и выполнять прочие стандартные операции – можно

Пример 14

```
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winsap(15), Cortland(8);
  private int price;
  Apple(int p) {
    price = p;
  }
  Apple() {
    price = -1;
  }
  int getPrice() {
    return price;
  }
}
```

Метаданные

В основе механизма метаданных лежат так называемые аннотации. **Аннотация – это «интерфейс» специфического вида, позволяющий задавать описания классов и их элементов.**

```
@interface MyAnnotation {
  String str();
  int val();
}
```

- Члены-методы имеют, скорее, смысл полей
- Тела этих методов будут создаваться автоматически Java
- Аннотациями можно снабжать классы, методы, поля, параметры, константы перечислимых типов и аннотации
- В любом случае аннотация предшествует объявлению
- Все аннотации наследуют интерфейс **java.lang.annotation.Annotation**

```
@MyAnnotation(str = "Example", val= 100)
public static void myMeth() {...}
```

Правила сохранения аннотаций

- Правила сохранения аннотаций определяют, в какой момент аннотации будут уничтожены
- В языке Java описаны три таких правила — source, class и runtime, включенные в перечислимый тип **java.lang.annotation.RetentionPolicy**

- ***SOURCE*** – аннотация, заданная с правилом сохранения source, существует только в исходном тексте программы и отбрасывается во время компиляции
- ***CLASS*** – аннотация, заданная с правилом сохранения class, помещается в файл .class в процессе компиляции. Но она не доступна в JVM во время выполнения
- ***RUNTIME*** – аннотация, заданная с правилом сохранения runtime, помещается в файл .class в процессе компиляции и доступна в JVM во время выполнения.
- Правило RUNTIME предлагает максимальную продолжительность существования аннотации
- Правило сохранения для аннотации задается с помощью одной из встроенных аннотаций @Retention
  @Retention (retention-policy)
- В этой записи **retention-policy** задается одной из описанных ранее констант перечислимого типа
- Если для аннотации не задано правило сохранения, **по умолчанию задается правило *CLASS***
  ```
  @Retention(RetentionPolicy.RUNTIME)
  @interface myAnnotation { String str(); int val(); }
  ```
- Получение информации об аннотациях производится средствами рефлексии
- Вы можете получить конкретную аннотацию, связанную с объектом, с помощью вызова метода **getAnnotation()** или использовать метод **getAnnotations()** для получения массива всех аннотаций
- Методы **getAnnotation()** и **getAnnotations()**, определены в новом интерфейсе **AnnotatedElement**, который включен в **пакет java.lang.reflect**
- Этот интерфейс поддерживает рефлексию для аннотаций и реализован в классах **Method**, **Field**, **Constructor**, **Class** и **Package**

Использование значений по умолчанию

Вы можете передавать значения по умолчанию методам-членам аннотаций, которые будут использоваться, если не задано значение при вставке аннотации. Значение по умолчанию указывается с помощью ключевого слова default в объявлении метода-члена:
type member() default value;
Значение value должно иметь тип, совместимый с типом type.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
  String str() default "Testing";
  int val() default 9000; }
@MyAnnotation()
@MyAnnotation(str = "some string");
@MyAnnotation(val = 100);
@MyAnnotation(str = "Testing", val = 100);
```

Аннотации-маркеры

Аннотация-маркер — это специальный тип аннотации, не содержащий методов-членов. Единственная цель такой аннотации — пометить объявление. В этом случае достаточно присутствия аннотации. Лучше всего для проверки наличия аннотации-маркера использовать метод **isAnnotationPresent()**, который определен в интерфейсе **AnnotatedElement.**

```
@interface MyMarker {}
```

Одночленная аннотация

Одночленная аннотация — это еще один специальный тип аннотации, содержащий единственный метод-член. Она применяется так же, как и обычная аннотация, за исключением того, что для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, вы можете просто указать значение для этого метода-члена, когда создается аннотация; при этом не нужно указывать имя метода-члена. Но для того чтобы воспользоваться краткой формой, **следует для метода-члена использовать имя value.**

```
@interface MySingle {
  int value(); //Имя только такое!
}
@MySingle(100)
```

Вы можете применять синтаксическую запись одночленной аннотации, когда создаете аннотацию, у которой есть и другие методы-члены, но для них должны быть заданы значения по умолчанию.

```
@interface SomeAnno {
  int value();
  int xyz() default 0;
}
```

Если Вы хотите использовать значение по умолчанию для метода xyz(), можно создать аннотацию типа @SomeAnno, применяя синтаксическую запись для одночленной аннотации с указанием значения для метода value().

```
@SomeAnno(88)
```

В этом случае в метод xyz() передается нулевое значение по умолчанию, а метод value() получает значение 88. Конечно, передача в метод xyz() значения, отличного от значения по умолчанию, потребует явного задания имен обоих методов-членов.

```
@SomeAnno (value = 88, xyz = 99)
```

Встроенные аннотации

- **@Retention**
  Применяется к аннотациям, позволяет задать правило сохранения
- **@Documented**
  Применяется к аннотациям, указывает, что она должна быть документирована
- **@Target**
  Применяется к аннотациям, позволяет указать типы объектов, к которым данная аннотация может применяться
- **@Inherited**
  Применяется к аннотациям классов, указывает, что данная аннотация будет унаследована потомками класса
- **@Override**
  Применяется к методам, указывает, что метод обязан переопределять метод родительского класса
- **@Deprecated**
  Указывает на то, что объявление является устаревшим или вышедшим из употребления
- **@SupressWarnings**
  Указывает на то, что указанные виды предупреждений компилятора не будут показываться. Эти предупреждения задаются с помощью имен в строковой форме

Особенности аннотаций

- Аннотация не может наследовать другую аннотацию
- Все методы, объявляемые в аннотации, не должны иметь параметров
- Все методы, объявляемые в аннотации должны возвращать один из перечисленных далее типов:
  - примитивный тип *(int, double)*
  - объект типа String
  - объект типа Class
  - перечислимый тип *(enum)*
  - другой тип аннотации
  - массив элементов одного из вышеперечисленных типов
- В аннотациях нельзя задавать ключевое слово throws
- Аннотации не могут быть настраиваемыми, т.е. они не могут принимать параметры типа

Классы-обертки примитивных типов. Класс Math. Класс String и класс StringBuffer. Класс Arrays. Классы для работы со временем и локализацией. java.util.Random. Коллекции. Интерфейс Collection. Класс Collections. Синхронизированные и неизменяемые обертки.

Классы-обертки примитивных типов

Значения примитивных типов не могут быть непосредственно использованы в контексте, где требуется ссылка. Ссылочное представление значений примитивных типов является основной задачей т.н. классов-оберток. Экземпляр такого класса хранит внутри значение примитивного типа и предоставляет доступ к этому значению. Для каждого примитивного типа Java существует свой класс-обертка. Такой класс является неизменяемым (т.е. для изменения значения необходимо создавать новый объект), к тому же имеет атрибут final – от него нельзя наследовать класс. Все классы-обертки (кроме Void) реализуют интерфейс Serializable, поэтому объекты любого (кроме Void) класса-обертки могут быть сериализованы.

Класс-обертка	**Примитивный тип**
Boolean	**boolean**
Byte	**byte**
Character	**char**
Double	**double**
Float	**float**
Integer	**int**
Long	**long**
Number	

Short	**short**
Void	

При этом классы обертки числовых типов – Byte, Short, Integer, Long, Float, Double наследуются от одного класса – Number. В нем содержится код, общий (часть реализована посредством абстрактных методов) для всех классов-оберток числовых типов. Класс Void является исключением, так как в Java нет значений, которые можно было бы заключить в соответствующую «обертку», в составе класса нет методов, и он не позволяет создавать объекты. Он нужен только для получения ссылки на объект Class, соответствующий void. Эта ссылка представлена статической константой TYPE. В Java не поддерживается тип void – это служебное слово используется в конструкциях объявлений методов и свидетельствует об отсутствии возвращаемых значений. Класс Void применятся только в механизме рефлексии.

Процесс инкапсуляции значения в объект называется **упаковкой**

```
Integer iOb = new Integer(100);
```

Процесс извлечения значения из оболочки типа называется **распаковкой**

```
int i = iOb.intValue();
```

Для получения значения типа long из объекта iOb следует вызвать метод iOb.longValue(). Таким образом, можно распаковать значение в переменную простого типа, отличающегося от типа оболочки.

Наполнение классов-оберток

- **Константы типов**
  Integer.MAX_VALUE, Double.NaN
- **Конструкторы:**
  - **принимающий в качестве параметра значение простого типа и создающий объект соответствующего класса-обертки**
    Float(float value)
  - **преобразующий содержимое параметра типа String в исходное значение объекта (за исключением класса Character, в составе которого такого конструктора нет)**
    Float(String s)
- **Методы получения значения**
  public type typeValue() – возвращает значение простого типа type, соответствующее содержимому текущего объекта класса-оболочки
  new Integer(6).intValue() вернет 6
- **Методы преобразования типов**
  public static type parseType(String str, int radix) – преобразует строку str в числовое значение заданного типа type с учетом указанного основания системы счисления radix. Если содержимое str не поддается преобразованию, либо значение radix выходит за границы допустимого диапазона, выбрасывается исключение типа NumberFormatException
  Integer.parseInt("1010",2) вернет 10
  Integer.parseInt("10") вернет 10
- **Методы проверки состояния и вида значения**
  public int compareTo(Type other) – возвращает значение меньшее (большее) нуля или равное нулю, если содержимое текущего объекта соответственно меньше (больше) значения other того же типа Type или равно ему. Метод не определен в

классе Boolean
new Integer(5).compareTo(4) вернет 1
public boolean isInfinite(double val) – возвращает true, если значение val соответствует положительной или отрицательной бесконечности
Double.isInfinite(5.6) вернет false

- **Специальные методы, обусловленные спецификой типа**
  public static String toHexString(type val) – возвращает строковое представление заданного значения в виде последовательности шестнадцатеричных цифр без знака
  Integer.toHexString(10) вернет строку «a»

Класс Math

- Предназначен для выполнения простых математических операций
- Не имеет явного конструктора
- Является final-классом
- Все методы являются статическими
- Не гарантирует повторяемости результатов (в отличие от класса StrictMath)

Наполнение класса Math

- Константы **E** и **PI**
- Функция взятия модуля **abs()**
- Функции максимума и минимума **max()**, **min()**
- Функции округления **round()**, **rint()**
- Функции ближайшего целого **ceil()**, **floor()**
- Тригонометрические функции **sin()**, **cos()**, **tan()**, **asin()**, **acos()**, **atan()**
- Функции перевода **toDegrees()**, **toRadians()**
- Функции степени **pow()**, **exp()**, **log()**, **sqrt()**
- Случайное значение **random()**

Работа со строками. Класс String

Экземпляры этого класса можно создавать без ключевого слова new, посредством задания литералов (например, «Привет») либо при использовании операторов + или += для сцепления содержимого двух объектов String с образованием новой строки. Каждый строковый литерал порождает экземпляр String. Значение любого типа может быть приведено к строке. Значение объекта класса String не может быть изменено без порождения нового объекта. Класс String реализует операции для строки в целом.

Наполнение класса String

- **Строковое представление**
  valueOf(type) – метод для преобразования в строку соответствующего типа
  copyValueOf(char[] data) – преобразует в String массив значений char
- **Преобразование типов**
  getBytes() – возвращает массив байтов текущей строки, преобразованных в соответствии с кодировкой символов и региональным стандартом, принятыми по умолчанию
  getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) – копирует символы из фрагмента текущей строки, заданного значениями начальной srcBegin и конечной srcEnd позиций, в указанный массив dst значений char, начиная с элемента

dst[dstBegin]. Элемент исходной строки с индексом, равным srcEnd, не копируется
toCharArray() – преобразует строку в массив типа char

- **Сравнение**
  compareTo(String str) – возвращает значение типа int, меньшее нуля, равное нулю или большее нуля, если соответственно строка, представляемая текущим объектом, меньше строки, переданной в качестве параметра метода, равна ей или больше нее
  compareToIgnoreCase(String str) – выполняет сравнение без учета регистра символов
  equals(Object o) – возвращает true, если переданный аргумент типа String ссылается на строку с таким же набором символов, который хранится и в текущей строке, т.е. обе строки обладают одинаковой длиной и в точности теми же символами Unicode, расположенными в одинаковом порядке. Если аргумент относится к типу, отличному от String, либо обладает другим содержимым, String.equals возвращает false
  equalsIgnoreCase(String str) – служит для сопоставления строк при условии пренебрежения регистром символов
  intern() – возвращает новую строку с тем же содержимым, что и в текущем объекте String
- **Выделение элементов**
  charAt(int index) – возвращает значение типа char, отвечающее символу строки, который расположен на указанной позиции
- **Операции над всей строкой**
  concat(String str) – возвращает новую строку, которая служит результатом сцепления (конкатенации) двух строк
  replace(char oldchar, char newchar) – возвращает новый объект String, в котором все экземпляры символа oldchar заменены символами newchar
  toLowerCase() – возвращает новый объект String, в котором каждый символ заменен аналогом в нижнем регистре (если таковой предусмотрен в региональном стандарте, применяемом по умолчанию)
  toUpperCase()– возвращает новый объект String, в котором каждый символ заменен аналогом в верхнем регистре (если таковой предусмотрен в региональном стандарте, применяемом по умолчанию)
  trim() – возвращает новый объект String, в котором удалены все начальные и конечные символы пробелов
- **Проверка содержимого строки**
  endsWith(String suffix) – возвращает true, если текущая строка завершается последовательностью символов suffix
  indexOf(char ch) – возвращает значение позиции первого экземпляра символа ch
  lastIndexOf(char ch) – возвращает значение позиции последнего экземпляра символа ch
  length() – возвращает целочисленное значение, равное количеству символов в стороке
  regionMatches(int start, String other, int ostart, int count) – возвращает true, если каждый символ заданного фрагмента текущей строки совпадает с соответствующим символом фрагмента строки, переданной посредством ссылки other. Анализу подвергаются часть текущей строки, начиная с символа на позиции start, и фрагмент строки other, начиная с позиции ostart. Количество символов, участвующих в сравнении, задается значением count
  startsWith(String prefix, int start) – возвращает true, если в текущей строке, начиная с позиции start, расположена последовательность символов prefix

Класс StringBuffer

Если бы класс String, представляющий объекты строк, не допускающих изменения, был единственным доступным средством, в процессе обработки строковых данных для хранения промежуточных результатов пришлось бы создавать новые объекты String. На самом деле компилятор действует более эффективно. В процессе вычисления промежуточных результатов он использует объект класса StringBuffer и создает объект класса String только когда это действительно необходимо. StringBuffer реализует методы модификации строки без порождения нового объекта.

- Реализует операции с элементами строки по отдельности
- Используется по умолчанию при конкатенации строк
- Для хранения строк использует буфер переменного объема

Класс StringBuffer во многом подобен классу String, он поддерживает ряд одноименных методов с теми же контрактами. Но StringBuffer не является производным от String и обратное утверждение также неверно. Эти классы совершенно независимы – они оба наследуют класс Object.

Наполнение класса StringBuffer

- **Добавление фрагментов**
  append(...) – позволяет преобразовать значение, переданное в качестве параметра, в строку и присоединить результат в конец
  insert(...) – позволяет преобразовать значение, переданное в качестве второго параметра, в строку и вставить результат в позицию, указанную в качестве первого параметра
- **Поиск вхождений**
  indexOf(String str) – возвращает значение первой позиции, с которой начинается заданная строка str
  lastIndexOf(String str) – возвращает значение последней позиции, с которой начинается заданная строка str
- **Состояние буфера**
  capacity() – возвращает значение текущей емкости буфера
  ensureCapacity(int minimum) – проверяет емкость буфера и при необходимости увеличивает ее до значения minimum
- **Извлечение подстрок**
  charAt(int index) – возвращает значение типа char, отвечающее символу строки, который расположен на указанной позиции
  getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) – копирует символы текущей строки StringBuffer в заданный массив dst. При этом символы строки из диапазона, заданного параметрами srcBegin и srcEnd (символ на позиции srcEnd не учитывается), присваиваются элементам массива, начиная с dst[dstBegin]
  reverse() – перестраивает порядок следования символов строке StringBuffer на обратный
- **Модификация строки**
  delete(int start, int end) – удаляет сегмент строки с позиции start до позиции end (за исключением символа end)
  deleteCharAt(int index) – удаляет символ на заданной позиции
  replace(int start, int end, String str) – заменяет фрагмент строки, задаваемый позициями начала (start) и конца (end, этот символ не учитывается), содержимым параметра str. Буфер удлиняется или сокращается в зависимости от того, что длиннее – строка str или заменяемый диапазон символов
  setCharAt(int index, char ch) – позволяет заменить символ в указанной позиции

строки
setLength(int newLength) – выполняет усечение или удлинение строки до указанной длины

Пакет java.text

- Содержит классы, предназначенные для реализации преобразований в строки и обратно с учетом форматирования и локализации
- Использование этих классов позволяет создавать приложения, не зависящие от особенностей локализации

Пакет java.util

- Классы для работы со временем
- Классы для работы с локализацией
- Классы для работы с массивами
- Классы и интерфейсы коллекций
- Прочие вспомогательные классы и интерфейсы

Классы для работы со временем

- **Date**
  Отражает дату и время с точностью до миллисекунд. Не рекомендуется к использованию
- **Calendar** и сопутствующие
  Содержит константы и методы для работы с датой и временем с учетом особенностей локализации
- **Timer**
  Позволяет создавать задания для более позднего запуска (с использованием потоков инструкций)

Классы для работы с локализацией

- **Locale**
  Содержит константы и методы для работы с языками и особенностями регионов
- **TimeZone**
  Содержит методы для работы с часовыми поясами
- **SimpleTimeZone**
  Реализует **TimeZone** для Григорианского календаря

java.util.Randon

Экземпляр класса является отдельным генератором псевдослучайных чисел (ГПСЧ). Различные ГПСЧ позволяют формировать некоррелированные последовательности. «Основание» имеет размерность 48bit.

- Методы получения ПСЧ:
  nextBoolean(), nextByte(), nextDouble(), nextFloat(), nextInt(),
  nextLong() – возвращает очередное псевдослучайное равномерно распределенное значение типа long, принадлежащее интервалу от Long.MIN_VALUE до Lpng.MAX_VALUE включительно
  nextGaussian() – возвращает очередное псевдослучайное значение типа double, распределенное по закону Гаусса (нормальное распределение)

- Метод настройки
  setSeed(long seed) – изменяет исходное значение последовательности псевдослучайных чисел, генерируемых текущим объектом Random, на seed. Обращение к методу приводит к тому, что текущая последовательность сбрасывается и генератор начинает отсчет чисел с заданного значения seed

Класс Arrays

Класс Arrays предлагает ряд полезных статических методов, предназначенных для работы с массивами:

- **sort()** – сортирует массив
- **binarySearch()** – осуществляет поиск заданного ключа в отсортированном массиве. Возвращает индекс найденного ключа либо отрицательное значение подходящей позиции вероятной вставки ключа, если ключ не найден
- **equals()** – возвращает true, если две ссылки на массивы, переданные в качестве параметров, указывают на один и тот же объект, обе равны null либо массивы обладают одинаковыми длиной и содержимым
- **fill()** – присваивает всем элементам массива одно и тоже значение
- **asList()** – представляет массив в виде объекта-списка

Коллекции

Коллекции (контейнеры) – хранилища, поддерживающие разнообразные способы накопления и упорядочивания объектов с целью обеспечения возможностей эффективного доступа к ним. В Java коллекции разделены на интерфейсы, абстрагирующие общие принципы работы с коллекциями, и классы, реализующие конкретную функциональность. Не все методы, заявленные в интерфейсах, должны в действительности реализовываться классами. Часть методов может просто выбрасывать исключение UnsupportedOperationException.

Интерфейс Collection

- Является образующим для интерфейсов коллекций
- Определяет базовую функциональность любой коллекции
- Подразумевает добавление, удаление, выбор элементов в коллекции
- Допускает дубликаты и пустые элементы

Методы интерфейса Collection

- **Добавление элементов**
  boolean add(Object o) – проверяет, содержит ли коллекция элемент o, добавляет его в коллекцию и возвращает true, если при выполнении операции потребовалось изменение коллекции. Если коллекция позволяет хранение одинаковых элементов, метод всегда возвращает true. Если наличие элементов-дубликатов недопустимо и соответствующий элемент уже присутствует в коллекции, возвращается fase
  boolean addAll(Collection c) – добавляет каждый элемент коллекции c в текущую коллекцию, возвращает true, если любая операция добавления требует изменения текущей коллекции

- **Исключение элементов**
  boolean remove(Object o) – удаляет из коллекции один экземпляр элемента o и

возвращает true, если при выполнении операции потребовалось изменение коллекции (т.е. если элемент действительно присутствовал в коллекции). Если значение o равно null, true возвращается в том случае, когда в составе коллекции существовал элемент null
boolean removeAll(Collection c) - удаляет каждый элемент коллекции c из текущей коллекцию, возвращает true, если любая операция удаления требует изменения текущей коллекции
boolean retainAll(Collection c) - удаляет из текущей коллекции все элементы, отсутствующие в коллекции c, возвращает true, если любая операция удаления требует изменения текущей коллекции.
void clear() – очищает коллекцию, т.е. удаляет все ее элементы

- **Состояние коллекции**
  boolean contains(Object o) – возвращает true, если коллекция содержит элемент o, т.е. если в коллекции имеется такой элемент, для которого метод equals, сравнивающий его с элементом o, возвращает true. Если значение o равно null, метод возвращает true в том случае, когда в коллекции существует элемент, равный null
  boolean containsAll(Collection c) – возвращает true, если текущая коллекция содержит все элементы коллекции c
  boolean isEmpty() – возвращает true, если коллекция пуста
  int size() – возвращает размер коллекции, т.е. количество элементов, которые она содержит в данный момент. Возвращаемое значение ограничено сверху величиной Integer.MAX_VALUE – даже в том случае, если в коллекции хранится большее число элементов
- **Вспомогательные методы**
  Object[] toArray() – возвращает новый массив, содержащий ссылки на все элементы коллекции
  Iterator iterator() – возвращает объект итератора, позволяющий получать последовательно расположенные элементы коллекции

Класс Collections

В его составе определен целый ряд статических методов прикладного назначения, позволяющих оперировать объектами коллекций. Инструменты класса можно условно поделить на две обширные группы: методы, обеспечивающие создание и поддержку объектов оберток (оболочек) коллекций, и прочие методы. Обертка дает возможность дополнить свойства «внутреннего» объекта-коллекции – синхронизировать доступ к нему или запретить использование тех методов объекта, которые изменяют содержимое коллекции.

Синхронизированные обертки

Объект синхронизированной обертки активизирует методы «внутренней» коллекции только после выполнения необходимых действий по синхронизации доступа к ее содержимому. Чтобы получить объект синхронизированной обертки, следует передать ссылку на объект коллекции одному из статических методов класса Collections: **synchronizedCollection(), synchronizedSet(), synchronizedSortedSet(), synchronizedList(), synchronizedMap()** или **synchronizedSortedMap().**

Каждый из перечисленных методов возвращает объект обертки коллекции соответствующего типа, методы которого полностью синхронизированы, что гарантирует безопасность их использования в многопоточной среде.

Неизменяемые обертки

В составе класса Collections имеются статические методы, возвращающие для заданных объектов коллекций неизменяемые обертки: **unmodifiableCollection(), unmodifiableSet(), unmodifiableSortedSet(), unmodifiableList(), unmodifiableMap(), unmodifiableSortedMap().** Объекты-обертки названных типов свободно «пропускают» обращенные к «внутренней» коллекции вызовы методов, которые не изменяют ее содержимого, но активизация любого из методов, предполагающих воздействие на содержимое, приводит к выбрасыванию исключения UnsupportedOperationException.

Прикладные методы класса Collections

- **min(Collection coll), max(Collection coll)** – возвращает элемент коллекции coll, обладающий «наименьшим» или «наибольшим» значением в соответствии с критерием естественного упорядочивания элементов этой коллекции
- **reverse(List list)** – изменяет порядок размещения элементов в списке list на противоположный
- **shuffle(List list)** – «тасует» элементы списка list случайным образом. Все перестановки элементов осуществляются с приблизительно равной вероятностью
- **fill(List list, Object elem)** – заменяет каждый элемент списка list значением elem
- **copy(List dst, List src)** – копирует каждый элемент списка-источника src в список получатель dst. Если список dst слишком мал, чтобы вместить все элементы списка src, выбрасывается исключение типа IndexOutOfBoundsException
- **nCopies(int n, Object elem)** – возвращает список, не допускающий изменения, который содержит n-элементов со значением, равным elem
- **sort(List list)** – сортирует список list в соответствии с правилом естественного упорядочения его элементов
- **binarySearch(List list, Object key)** - использует алгоритм бинарного поиска для отыскания в списке list объекта-ключа key и возвращает индекс найденного объекта. Элементы списка должны располагаться в соответствии с правилом естественного упорядочения. Если объект не найден, возвращается отрицательное значение, соответствующее подходящей позиции вероятной вставки объекта
- **Прочие прикладные методы**

Интерфейс Set. Интерфейс List. Интерфейс Iterator. Интерфейс Map. Классы коллекций.

Интерфейс Set

- Расширяет интерфейс Collection
- Не разрешает наличие дубликатов
- Допускается наличие только одной ссылки null
- Объекты коллекции должны реализовывать метод equals()

Интерфейс List

- Расширяет интерфейс Collection
- Подразумевает хранение упорядоченной последовательности объектов
- Порядок хранения определяется порядком добавления элементов
- Позволяет обращаться к элементам по их номеру

Специальные методы интерфейса List

- **Адресное добавление**
  void add(int index, Object o) – вставляет элемент o на index-ю позицию в списке, сдвигая каждый последующий элемент на одну позицию в направлении конца списка
- **Адресные операции с элементами**
  Object get(int index) – возвращает элемент, занимающий index-ю позицию в списке
  Object set(int index, Object o) – возвращает содержимое элемента, расположенного на index-й позиции в списке, и замещает его в списке значением o
  Object remove(int index) – возвращает содержимое элемента, занимающего index-ю позицию в списке, и удаляет этот элемент из списка, сдвигая каждый последующий элемент на одну позицию в направлении начала списка
- **Операции поиска**
  int indexOf(Object o) – возвращает индекс первого объекта в списке, равного(equals) параметру o, либо null, если o равен null. Возвращает -1, если совпадения не обнаружены
  int lastIndexOf(Object o) - возвращает индекс последнего объекта в списке, равного(equals) параметру o, либо null, если o равен null. Возвращает -1, если совпадения не обнаружены
- **Специальные операции**
  List subList(int from, int to) – возвращает частичный список или курсор текущего списка; в курсор включаются элементы списка, занимающие позиции от from до to, кроме элемента с индексом to. Содержимое курсора поддерживается коллекцией в актуальном состоянии: изменения, вносимые в курсор, отображаются в текущем списке. Изменения, которые претерпевает текущий список, однако, не обязательно «видимы» посредством курсора и могут приводить к неопределенным результатам
  ListIterator listIterator() – возвращает объект ListIterator, который позволяет осуществлять пошаговое перемещение по списку, начиная с элемента с индексом 0

Интерфейс Iterator

- Позволяет работать с коллекцией как с набором элементов

- Получать следующий объект **Object next()**
- Проверять наличие следующего объекта **boolean hasNext()**
- Исключать объект из коллекции **void remove()**

Интерфейс Map

- Не расширяет интерфейс Collection
- Подразумевает хранение набора объектов парами ключ/значение
- Ключи должны быть уникальными
- Порядок следования пар ключ/значение не определен
- Имеет расширение SortedMap, требующее упорядоченности по значениям ключей

Методы интерфейса Map

- **Добавление объектов**
  Object put(Object key, Object value) – устанавливает соответствие между объектом-ключом key и объектом-значением value в текущей коллекции. Если коллекция уже содержит значение, соответствующее key, значение возвращается и заменяется в коллекции содержимым параметра value. Если в коллекции нет пары key/value, та помещается в коллекцию и возвращается null; метод может вернуть null и в том случае, если ключ key прежде существовал и ссылался на значение null
  void putAll(Map t) – помещает в коллекцию содержимое другой коллекции t
- **Исключение объектов**
  Object remove(Object key) – удаляет из коллекции соответствие для заданного ключа key. Возвращаемое значение обладает той же семантикой, что и в случае метода put()
  void clear() – очищает коллекцию, удаляя из нее все пары соответствий
- **Состояние**
  boolean containsValue(Object value) – возвращает true, если коллекция содержит по меньшей мере один экземпляр заданного значения value
  boolean containsKey(Object key) – возвращает true, если коллекция содержит значение для заданного ключа key
  int size() – возвращает размер коллекции соответствий, т.е. количество хранящихся в ней пар ключ/значение. Возвращаемое значение ограничено сверху величиной Integer.MAX_VALUE – даже в том случае, если в коллекции хранится большее число элементов-пар
  boolean isEmpty() – возвращает true, если коллекция пуста
- **Доступ к объекту по ключу**
  Object get(Object key) – возвращает объект-значение, которому соответствует заданный ключ key, либо null, если соответствие не найдено; null также может быть возвращено в случае, если коллекция допускает хранение значений null и ключ key явно указывает на null
- **Преобразование типа**
  Set entrySet() – возвращает объект Set, элементами которого служат объекты типа Map.Entry, представляющие отдельные пары соответствий текущей коллекции. Map.Entry – это вложенный интерфейс, обдадающий методами, которые позволяют манипулировать парами соотвествий
  Set keySet() – возвращает объект Set, элементами которого служат ключи текущей коллекции соответствий
  Collection values() – возвращает объект Collection, элементами которого служат значения текущей коллекции соответствий

Классы коллекций

- **Динамические массивы**
  ArrayList (List) – позволяет хранить элементы списка в виде массива изменяемого размера
  Vector (List) – относится к числу устаревших, является прямым аналогом ArrayList
- **Двухсвязный список**
  LinkedList (List) – обеспечивает реализацию двусвязного списка
- **Упорядоченные множество и карта**
  TreeSet (Set) – обеспечивает хранение элементов в структуре данных сбалансированного бинарного дерева
  TreeMap (Map) – реализует интерфейс Map и обеспечивает хранение ключей в упорядоченном виде теми же способами, которые предусмотрены классом TreeSet
- **Ряд других классов**
  HashMap (Map) – реализует интерфейс Map на основе модели хеш-таблицы
  HashSet (Set) – представляет собой реализацию интерфейса Set, основанную на использовании хеш-таблицы

Проблемы однопоточного подхода. Особенности многопоточности. Использование класса Thread. Использование интерфейса Runnable. Приоритеты потоков.

Понятие многопоточности

Последовательность операций, выполняемых программой по одной в каждый момент времени, называют **потоком (или нитью – thread).**

Проблемы однопоточного подхода:

- Монопольный захват задачей процессорного времени
- Смешение логически несвязанных фрагментов кода
- Попытка их разделения приводит к возникновению в программе новых систем

Реализация многопоточности

Реализацию многопоточной архитектуры проще всего представить себе для системы, в которой есть несколько центральных вычислительных процессоров. В этом случае для каждого из них можно выделить задачу, которую он будет выполнять. В результате несколько задач будут обслуживаться одновременно. Однако возникает вопрос – каким же тогда образом обеспечивается многопоточность в системах с одним центральным процессором, который в принципе выполняет лишь одно вычисление в один момент времени?

В таких системах применяется процедура **квантования времени** – время разделяется на небольшие интервалы (кванты времени). Во время одного кванта обрабатывается один поток команд. Перед началом каждого интервала принимается решение, какой именно поток выполнения будет обрабатываться на протяжении этого кванта времени. За счет частого переключения между задачами эмулируется многопоточная архитектура, т.е. создается иллюзия одновременности выполнения.

На самом деле, как правило, и для многопроцессорных систем применяется процедура квантования времени, так как даже в мощных серверах приложений процессоров не так много, а потоков исполнения запускается, как правило, гораздо больше. Т.о., в многопроцессорной системе поток не занимает монопольно один процессор.

Особенности многопоточности

- Потоки выполняются условно независимо
- Потоки могут взаимодействовать друг с другом
- Простота выделения подзадач
- Более гибкое управление выполнением задач
- Более медленное выполнение
- Выигрыш в скорости выполнения при разделении задач по используемым ресурсам
- Недетерминизм при выполнении

Использование класса Thread

Поток выполнения в Java представляется экземпляром класса Thread. Для того, чтобы написать свой поток исполнения, необходимо наследоваться от этого класса и

переопределить метод run(). Стандартная реализация метода run() не предполагает выполнения каких бы то ни было действий.
Объявление:

```
public class MyThread extends Thread {
  public void run() {
    // Действия, выполняемые потоком
  }}
```

Метод run() содержит действия, которые должны исполняться в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника, и вызвать унаследованный метод start(), который сообщает виртуальной машине, что необходимо запустить новый поток исполнения и начать в нем исполнять метод run().
Запуск:

```
MyThread t = new MyThread();
t.start();
```

Вызов start() для каждого потока может быть осуществлен только один раз – повторное обращение приводит к выбрасыванию исключения типа *IllegalThreadsStateException.* Потоку разрешено присвоить имя – либо с помощью аргумента String, переданного конструктору, либо посредством вызова метода setName(), текущее значение имени потока можно получить с помощью метода getName(). Имена потоков служат только для удобства программиста (исполняющей системой они не используются), но поток должен обладать именем, и если оно не задано, исполняющая система генерирует имена в соответствии с неким простым правилом, например, thread_1, thread_2 и т.д. Ссылку на объект Thread текущего выполняемого потока можно получить с помощью статического метода Thread.currentThread(). Во время работы программы текущий поток существует всегда, даже если вы не создавали потоки явно – метод main() активизируется с помощью потока, создаваемого исполняющей системой.

Использование интерфейса Runnable

Описанный подход обладает одним недостатком: поскольку в Java отсутствует множественное наследование, требование наследоваться от Thread может привести к конфликту. Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать **интерфейс Runnable**, в котором объявлен только один метод – уже знакомый void run().
Объявление:

```
public class MyThread implements Runnable {
  public void run() {
    // Действия, выполняемые потоком
  }}
```

Запуск:

```
Runnable r = new MyThread();
Thread t = new Thread(r);
t.start();
```

Подчеркнем, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

Управление потоками. Нерекомендуемые действия над потоками. Прерывание потока. Группы потоков. Операции в группе потоков. Демон-потоки. Пример. Демон-группы потоков.

Управление потоками

- **void start()**
  Запускает выполнение потока
- **void stop()**
  Прекращает выполнение потока
- **void suspend()**
  Приостанавливает выполнение потока
- **void resume()**
  Возобновляет выполнение потока
- **static void sleep(long millis)**
  Останавливает выполнение потока как минимум на **millis** миллисекунд
- **static void yield()**
  Приостанавливает выполнение потока, предоставляет возможность выполнять другие потоки
- **void join()**
  Ожидает завершения указанного потока

Нерекомендуемые действия над потоками

- **Thread.suspend()**
  увеличивают количество взаимных блокировок
- **Thread.resume()**
  увеличивают количество взаимных блокировок
- **Thread.stop()**
  использование приводит к возникновению поврежденных объектов

Группы потоков

- Каждый поток находится в группе
- Группа указывается при создании потока
- Если группа не была указана, то поток помещается в ту же группу, где находится поток, породивший его
- Группы потоков образуют дерево, корнем служит начальная группа
- Поток не имеет доступа к информации о родительской группе
- Изменение параметров и состояния группы влияет на все входящие в нее потоки
- **Создание группы:**

```
//Без явного указания родительской группы
ThreadGroup group1 = new ThreadGroup("Group1");
//С явным указанием родительской группы
ThreadGroup group2 = new ThreadGroup(group1,  "Group2");
```

- **Создание потока:**

```
//Без явного указания группы
MyThread t = new MyThread("Thread1");
//С явным указанием группы
```

```
MyThread t = new MyThread(group2, "Thread2");
```

Операции в группе потоков

- **int activeCount()**
  Возвращает приблизительное количество действующих (активных) потоков группы, включая и те потоки, которые принадлежат вложенным группам. Количество нельзя считать точным, т.к. в момент выполнения метода оно может измениться – одни потоки «умирают», другие – создаются
- **int enumerate(Thread[] list)**
  Заполняет массив list ссылками на объекты активных потоков группы и возвращает количество сохраненных ссылок. Также учитываются потоки, относящиеся ко всем вложенным группам
- **int activeGroupCount()**
  Возвращает количество групп, включая вложенные
- **int enumerate(ThreadGroup[] list)**
  Заполняет массив list ссылками на объекты вложенных групп потоков
- **void interrupt()**
  Прерывает выполнение всех потоков в группе

Приоритеты потоков

- **Приоритет** – количественный показатель важности потока
- Недетерминированно воздействует на системную политику упорядочивания потоков
- Базовый алгоритм не должен зависеть от схемы расстановки приоритетов потоков
- В Java потокам можно назначать приоритеты
- Для этого в классе Thread существуют методы getPriority() и setPriority(), а также объявлены три **Константы:**
  ```
  static int MAX_PRIORITY
  static int MIN_PRIORITY
  static int NORM_PRIORITY
  ```
- Их значения описывают максимальное, минимальное и нормальное (по умолчанию) значения приоритета
- **Методы потока**
  final int getPriority() – возвращает ранее заданное значение приоритета выполнения для текущего потока
  final void setPriority(int newPriority) – устанавливает приоритет выполнения для текущего потока
- **Методы группы потоков**
  final int getMaxPriority() – возвращает ранее заданное значение верхней границы приоритетов выполнения для текущей группы потоков
  final void setMaxPriority(int maxPri) – устанавливает верхнюю границу приоритетов выполнения для текущей группы потоков

Демон-потоки

- **Демон-потоки** позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них
- Уничтожаются виртуальной машиной, если в группе не осталось не-демон потоков
- Пример демон-потока – сборщик мусора

- **void setDaemon(boolean on)** – устанавливает вид потока. Вызывается до запуска потока
- **boolean isDaemon()** – возвращает вид потока:
  true – демон, false – обычный

Демон-группы потоков

- Демон-группа автоматически уничтожаются при останове последнего ее потока или уничтожении последней подгруппы потоков
- **void setDaemon(boolean on)** – устанавливает вид группы
- **boolean isDaemon()** – возвращает вид группы:
  true – демон, false – обычная

Совместное использование ресурсов. Характерные ошибки. volatile. Специальные методы класса Object. Особенности использования методов класса Object.

Совместное использование ресурсов

При многопоточной архитектуре приложения возможны ситуации, когда несколько потоков будут одновременно работать с одними и теми же данными, используя их значения и присваивая новые. В таком случае результат работы программы становится невозможным предопределить, глядя только на исходный код. Отсюда следует возможность некорректной работы алгоритма, возникновения исключительных ситуаций. Решением проблемы является применение **механизма блокировки объекта**:

- Только один поток в один момент времени может установить блокировку на некоторый объект
- Попытка блокировки уже заблокированного объекта приводит к останову потока до момента разблокирования этого объекта
- Наличие блокировки не запрещает всех остальных действий с объектом

В Java-программе для того, чтобы воспользоваться механизмом блокировок, существует **ключевое слово synchronized.** Оно может быть применено в двух вариантах – для объявления synchronized-блока и как модификатор метода. В обоих случаях действие его примерно одинаковое.

Синхронизированный блок:

```
//Блокируется указанный объект
synchronized (ref )   // ref - ссылка на объект
{
          // Тело блока синхронизации
}
```

Прежде чем начать выполнять действия, описанные в этом блоке, поток обязан установить блокировку на объект, на который ссылается переменная ref (поэтому она не может быть null). Если другой поток уже установил блокировку на этот объект, то выполнение первого потока приостанавливается до тех пор, пока не удастся выполнить блокировку. После этого блок выполняется. В случае успешного либо не успешного завершения исполнения, производится снятие блокировки, чтобы освободить объект для других потоков.

Synchronized-методы работают аналогичным образом. Прежде чем начать выполнять их, поток пытается заблокировать объект, у которого вызывается метод. После выполнения блокировка снимается.

Синхронизированный метод:

```
//Блокируется объект-владелец метода
public synchronized void myMethod() {
          // Тело метода
}
```

Характерные ошибки

- Отсутствие синхронизации
- Необоснованная длительная блокировка объектов
- Взаимная блокировка
- Возникновение монопольных потоков

- Нерациональное назначение приоритетов

volatile

Пример 1

```
currentValue = 5;
for(;;){
    display.showValue(currentValue);
    Thread.sleep(1000);
}
```

Если метод showValue() сам по себе не обладает возможностью изменения значения curentValue, компилятор может предположить, что внутри цикла for это значение можно трактовать как неизменное, и использовать одну и ту же константу 5 на каждой итерации цикла при вызове showValue(). Но если содержимое поля currentValue в ходе выполнения цикла подвержено обновлению посредством других потоков, предположение компилятора окажется неверным. Преодолеть проблему может соответствующим способом оформленный синхронизированный код.
Однако существует альтернативный вариант решения проблемы – объявление поля можно снабдить модификатором volatile. Этот признак свидетельствует о том, что значение поля может быть изменено в любой непредсказуемый момент. Если объявить currentValue как volatile, компилятор будет вынужден, выполняя каждую итерацию цикла, заново перечитывать значение переменной.

Специальные методы класса Object

Каждый объект в Java имеет так называемый wait-set, набор потоков исполнения. Любой поток может вызвать метод **wait()** любого объекта и таким образом попасть в его wait-set. При этом выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у точно этого же объекта метод **notifyAll()**, который пробуждает все потоки из wait-set. Метод **notify()** пробуждает один, случайно выбранный поток из этого набора. Однако применение этих методов связано с некоторыми особенностями:

- Метод может быть вызван потоком у объекта только после установления блокировки на этот объект
- Потоки, прежде чем приостановить выполнение после вызова метода **wait()**, снимают все блокировки
- После вызова освобождающего метода потоки пытаются восстановить ранее снятые блокировки

Прерывание потока

- **public void interrupt()** – изменяет статус потока на прерванный
- **public static boolean interrupted()** – возвращает и очищает статус потока (прерван или нет)
- **public boolean isInterrupted()** – возвращает статус потока (прерван или нет)
- В том случае, если в текущий момент поток выполняет методы **wait()**, **sleep()**, **join()**, а его прерывают вызовом метода **interrupt()**, метод прерывает свое выполнение с выбросом исключения **InterruptedException**
- Потоку не сообщается, что его прервали

```
Пример 2
class VectorSynchronizer {
    private double [] v;
```

```
    private volatile int current = 0;
    private Object lock = new Object();
    private boolean set = false;
    public VectorSynchronizer(double [] v) {
        this.v = v;
    }
    public boolean canRead() {
        return current < v.length;
    }
    public boolean canWrite() {
        return (!set && current < v.length) || (set && current <
v.length - 1);
    }
public double read() throws InterruptedException {
        double val;
        synchronized (lock) {
            if (!canRead()) {
                throw new InterruptedException();
            }
            while (!set) {
                lock.wait();
            }
            val = v[current++];
            System.out.println("Read: " + val);
            set = false;
            lock.notifyAll();
        }
        return val;
    }

public void write(double val) throws InterruptedException {
        synchronized (lock) {
            if (!canWrite()) {
                throw new InterruptedException();
            }
            while (set) {
                lock.wait();
            }
            v[current]= val;
            System.out.println("Write: " + val);
            set = true;
            lock.notifyAll();
        }
    }
}
```

Пакет java.util.concurrent. ReentrantLock, ReadWriteLock. Интерфейсы Callable и Future.

java.util.concurrent

- **Concurrent Collections** — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из **java.util** пакета, где вместо базовой обертки **Collections.synchronizedList()** с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных
- **Queues** — неблокирующие (направлены на скорость и работу без блокирования потоков) и блокирующие (используются, когда нужно «притормозить» потоки чтения или записи, если не выполнены какие-либо условия, например, очередь пуста или переполнена, или же нет свободного читателя) очереди с поддержкой многопоточности
- **Synchronizers** — вспомогательные утилиты для синхронизации потоков, которые представляют собой мощное оружие в параллельных вычислениях.
- **Executors** — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов
- **Locks** — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми **synchronized()**, **wait()**, **notify()**, **notifyAll()**
- **Atomics** — классы с поддержкой атомарных операций над примитивами и ссылками

java.util.concurrent.locks

- В пакете содержатся интерфейсы и классы, служащие основой для блокировки и ожидания условий, которая отличается от встроенной синхронизации и мониторов
- Структура дает гораздо большую гибкость в использовании блокировок и условий за счет более богатого синтаксиса.
- Интерфейс **Lock** поддерживает правила блокировки, которые отличаются по семантике (reentrant, fair и т.д.) и которые могут быть использованы в не блочно-структурированных контекстах, в том числе в алгоритмах передачи или сортировки блокировок (основная реализация **ReentrantLock)**
- Интерфейс **ReadWriteLock** аналогично определяет блокировки, которые могут разделяться между потребителями, но являются исключительными для создателей (представлена только одна реализации **ReentrantReadWriteLock**, поскольку она охватывает большинство стандартных условий использования, но программисты могут создавать свои собственные реализации для покрытия нестандартных требований)
- **Condition** интерфейс описывает переменные состояния, которые могут быть связаны с объектами блокировок:
    - Они похожи по использованию на неявные мониторы, доступные посредством **Object.wait()**, но предлагают расширенные возможности
    - В частности, несколько объектов **Condition** могут быть связаны с одной блокировкой
    - Чтобы избежать проблемы совместимости, имена методов **Condition** отличаются от соответствующих методов **Object**

Синхронизаторы

- Класс **AbstractQueuedSynchronizer** служит полезным суперклассом для определения блокировок и других синхронизаторов, которые полагаются на очереди заблокированных потоков
- Класс **AbstractQueuedLongSynchronizer** обеспечивает ту же функциональность, но расширяет поддержку 64-битное состояния синхронизации
- Оба расширяют класс **AbstractOwnableSynchronizer**, простой класс, который помогает записывать поток, который в настоящее время держит эксклюзивную синхронизацию
- Класс **LockSupport** обеспечивает поддержку блокировок и разблокировок более низкого уровня, что будет полезно для разработчиков, реализующих свои собственные индивидуальные классы блокировок

Lock Objects

- Синхронизированный код основывается на простейшего вида реентерабельной блокировке, она проста в использовании, но имеет много ограничений
- Более сложные блокирующие подходы поддерживаются пакетом **java.util.concurrent.locks**.
- Объекты **Lock** работают подобно неявным блокировкам, используемым синхронизированным кодом
- Как и в неявных блокировках, только один поток может владеть объектом **Lock** в одно и тоже время
- Объекты **Lock** также поддерживают механизм ожидания/уведомления, через связанные с ними состояния объектов.
- Самым большим преимуществом объектов **Lock** над неявной блокировкой является их способность вернуться из попытки получения блокировки:
    - Метод **tryLock()** откатывает попытку, если блокировка не доступна сразу или по истечении времени ожидания (если оно указано)
    - Метод **lockInterruptibly()** откатывается, если другой поток посылает прерывание перед наложением блокировки

ReentrantLock

Реентерабельная взаимоисключающая блокировка с тем же основным поведением и семантикой, что предоставляются неявными блокировками, получаемыми с помощью синхронизированных методов и блоков, но с расширенными возможностями. Объектом ReentrantLock владеет поток, последним наложивший успешно блокировку, но еще не разблокировавший объект блокировки. Поток, пытающийся наложить блокировку, успешно продолжит выполнение с наложением блокировки, если объект блокировки не принадлежит другому потоку. Попытка немедленно завершается, если текущий поток уже владеет блокировкой. Это можно проверить с помощью методов isHeldByCurrentThread() и getHoldCount().

Пример 3

```
public boolean impendingBow(Friend bower) {
  Boolean myLock = false;
  Boolean yourLock = false;
  try {
    myLock = lock.tryLock();
    yourLock = bower.lock.tryLock();
```

```
  } finally {
    if (!(myLock && yourLock)) {
      if (myLock) {
        lock.unlock();
      }
      if (yourLock) {
        bower.lock.unlock();
      } } }
  return myLock && yourLock;
}
```

ReadWriteLock

ReadWriteLock сохраняет пару связанных блокировок: одну для операции чтения, другую – для записи. Блокировки чтения могут проводиться одновременно из нескольких потоков чтения, пока нет писателей, а блокировка записи является эксклюзивной. Все **ReadWriteLock** реализации должны гарантировать, что синхронизация операций **writeLock()** (как указано в **Lock** интерфейсе) справедлива и по отношению к связанной **readLock()**, то есть поток, успешно наложивший блокировку на чтение, увидит все обновления, сделанные до последнего снятия блокировки на запись.

Пример 4

```
public class Dictionary {
  private final ReentrantReadWriteLock readWriteLock = new
ReentrantReadWriteLock();
  private final Lock read = readWriteLock.readLock();
  private final Lock write = readWriteLock.writeLock();
  private HashMap<String, String> dictionary = new HashMap<String,
String>();
  public void set(String key, String value) {
    write.lock();
    try { dictionary.put(key, value); } finally { write.unlock(); }
  }
  public String get(String key) {
    read.lock();
    try{ return dictionary.get(key); } finally { read.unlock(); }
  }
  public String[] getKeys() {
    read.lock();
    try {
      String keys[] = new String[dictionary.size()];
      return dictionary.keySet().toArray(keys);
    } finally { read.unlock(); }
  } }
```

Интерфейсы Executor, ExecutorService, ScheduledExecutorService. Пул потоков. Executors.

Исполнители (Executors)

Во всех предыдущих примерах, существует тесная связь между задачей, выполняемой новым потоком и определяемой ее **Runnable** объектом, и самим потоком **Thread.** Этот подход работает для небольших приложений, но в крупных приложениях имеет смысл отделить управление потоками и их создание от остальной части приложения. Объекты, внутри которых реализуются эти функции, называются исполнителями:

- **Executor Intefaces** определяют три типа исполнителей
- **Thread Pools** являются наиболее общим типом реализации исполнителей

Интерфейсы исполнителей

Пакет **java.util.concurrent** содержит три интерфейса исполнителей:

- **Executor** – простой интерфейс, поддерживающий запуск новой задачи
- **ExecutorService**, расширяющий **Executor**, добавляет возможности по управлению жизненным циклом отдельных задач и самого исполнителя
- **ScheduledExecutorService**, расширяющий **ExecutorService**, поддерживает периодическое выполнение задач или их запуск в будущем

Обычно переменные-ссылки на объекты исполнителей объявляются одного из этих трех интерфейсных типов без указания конкретного имени класса исполнителя.

Интерфейс Executor

Интерфейс **Executor** предоставляет единственный метод, **execute()**, разработанный для замены общего подхода создания потоков. Если **r** является **Runnable** объектом и **e** является **Executor** объектом, вы можете заменить

```
(new Thread(r)).start();
```

на

```
e.execute(r);
```

Однако определение **execute()** менее специфично:

- Низкоуровневые подходы создают новый поток и немедленно запускают его
- В зависимости от реализации **Executor**, **execute()** может сделать то же самое, но более вероятно использует существующий рабочий поток для запуска **r** или размещения **r** в очереди ожидания доступного рабочего потока

Реализации исполнителей в java.util.concurrent спроектированы так, чтобы в полной мере использовать более продвинутые **ExecutorService** и **ScheduledExecutorService** интерфейсы, хотя они также работают с базовым интерфейсом **Executor.**

Интерфейс ExecutorService

Интерфейс **ExecutorService** поддерживает выполнение с похожим, но более универсальным методом **submit().** Подобно **execute()**, **submit()** принимает **Runnable** объекты, а также **Callable** объекты, которые позволяют задаче возвращать значение. Метод **submit()** возвращает **Future** объект, который используется для получения возвращаемого значения **Callable** и имеет возможность отслеживать состояние **Callable** и **Runnable** задач.**ExecutorService** также содержит методы для запуска больших коллекций **Callable** объектов. Наконец, **ExecutorService** предоставляет ряд методов для управления

остановкой исполнителя. Для поддержки немедленной остановки задачи должны корректно обрабатывать прерывания.

Интерфейс ScheduledExecutorService

Интерфейс **ScheduledExecutorService** поддерживает методы своего родителя **ExecutorService** с возможностью составления графика, по которому выполняет **Runnable** или **Callable** задачу после заданной задержки. В дополнение, интерфейс объявляет методы **scheduleAtFixedRate()** и **scheduleWithFixedDelay()**, которые исполняют указанные задачи периодически с определенным интервалом.

ThreadPools

Большинство реализаций исполнителей в **java.util.concurrent** используют пулы потоков, которые состоят из рабочих потоков. Этот тип потоков существует отдельно от **Runnable** и **Callable** задач, которые он выполняет, и часто используется для выполнения нескольких задач. Использование рабочих потоков сводит к минимуму накладные расходы на создание потоков. Потоки используют значительный объем памяти, а в крупных приложениях выделение и освобождение ресурсов под множество объектов потоков создает значительные накладные расходы на управление памятью.
Один из распространенных типов пулов потоков – это фиксированный пул потоков:

- У пула этого типа всегда есть определенное количество выполняющихся потоков
- Если поток почему-то уничтожается, пока он еще используется, то он автоматически заменяется на новый поток
- Задачи представляются в пул через внутреннюю очередь, в которую помещаются дополнительные задачи всякий раз, когда активных задач больше, чем потоков

Важным преимуществом фиксированного пула потоков является то, что приложения, использующие этот подход, плавно снижают свою производительность. Чтобы понять это, рассмотрим веб-сервер, где каждый запрос HTTP обрабатывается в отдельном потоке. Если приложение просто создает новый поток для каждого нового запроса HTTP и система получает больше запросов, чем она может обработать немедленно, приложение может вдруг перестать отвечать на все запросы, когда накладные расходы на все эти потоки превысят возможности системы. С лимитом на количество потоков, которые могут быть созданы, приложение не будет обслуживать HTTP-запросы так быстро, как они приходят, но он будет обслуживать их настолько быстро, насколько система может выдержать.

Executors

Простой способ создания исполнителя, который использует фиксированный пул потоков, заключается в вызове метода **newFixedThreadPool()** фабрики **java.util.concurrent.Executors.**
Этот класс также предоставляет следующие методы фабрики:

- Метод **newCachedThreadPool()** создает исполнителя с расширяемым пулом потоков, этот исполнитель подходит для приложений, которые запускают множество коротких задач
- Метод **newSingleThreadExecutor()** создает исполнителя, который выполняет одну задачу за один раз
- Несколько фабричных методов версии **ScheduledExecutorService** для представленных выше исполнителей

Если ни один из исполнителей, представленных вышеупомянутыми фабричными методами, не отвечает вашим требованиям, создание экземпляров **java.util.concurrent.ThreadPoolExecutor** или **java.util.concurrent.ScheduledThreadPoolExecutor** через конструкторы даст вам дополнительные опции.

Пример 5

```
class NetworkService implements Runnable {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;
  public NetworkService(int port, int poolSize) throws
IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }
  public void run() {
    try {
      for (;;) {
        pool.execute(new Handler(serverSocket.accept()));
      }
    } catch (IOException ex) { pool.shutdown(); }
  }
}
class Handler implements Runnable {
  private final Socket socket;
  Handler(Socket socket) { this.socket = socket; }
  public void run() { … }
}
```

Пример 6

```
ExecutorService executorService =
Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new
HashSet<Callable<String>>();
callables.add(new Callable<String>() {
  public String call() throws Exception { return "Task 1"; }
});
callables.add(new Callable<String>() {
  public String call() throws Exception { return "Task 2"; }
});
callables.add(new Callable<String>() {
  public String call() throws Exception { return "Task 3"; }
});
List<Future<String>> futures =
executorService.invokeAll(callables);
for(Future<String> future : futures) {
  System.out.println("future.get = " + future.get());
}
executorService.shutdown();
```

InterruptedException

InterruptedException должен бросать любой метод, который может заблокировать поток и причиной блокировки является перевод потока в состояние WAITING или SLEEPING. Помимо методов **Object.wait()**, **Thread.sleep()** и **Thread.join()** есть ещё масса других, например, **Lock.lockInterruptibly()**, **Condition.await()**, **Semaphore.aquire().** Это исключение выбрасывается только в том случае, когда на объекте потока, который заблокирован вызовом такого метода, вызывается **interrupt().**

Прерывание потока

Каждый поток имеет связанное с ним булево свойство, которое отображает его статус прерывания. Статус прерывания изначально имеет значение false; когда поток прерывается каким-либо другим потоком путем вызова **Thread.interrupt()**, то происходит одно из двух:

- Если другой поток выполняет прерываемый метод блокирования низкого уровня, то он разблокируется и выдает **InterruptedException**
- Иначе **interrupt()** просто устанавливает статус прерывания потока

Код, действующий в прерванном потоке, может позднее обратиться к статусу прерывания, чтобы посмотреть, был ли запрос на прекращение выполняемого действия; статус прерывания может быть прочитан с помощью **Thread.isInterrupted()**, и может быть прочитан и сброшен за одну операцию при помощи неудачно названного **Thread.interrupted().**

Когда один поток прерывает другой, прерванный поток не обязательно немедленно прекратит делать то, что он делал, напротив, прерывание является вежливым способом, чтобы попросить другой поток прекратить то, что он делает, если он не против, когда ему это будет удобно. Некоторые методы, например, **Thread.sleep()**, серьезно относятся к такой просьбе, но методы не обязаны обращать внимание на прерывание. Неблокирующие методы, выполнение которых занимает много времени, могут положительно отнестись к просьбам о прерывании, путем опроса статуса прерывания, и в случае прерывания вернуться к более раннему этапу. Вы можете проигнорировать запрос о прерывании, но это может нанести ущерб ответной реакции.

Пример 7

```
public class PlayerMatcher {
  private PlayerSource players;
  public PlayerMatcher(PlayerSource players) { this.players =
players; }
  public void matchPlayers() throws InterruptedException {
    try {
      Player playerOne, playerTwo;
      while (true) {
        playerOne = playerTwo = null;
        playerOne = players.waitForPlayer();
        playerTwo = players.waitForPlayer();
        startNewGame(playerOne, playerTwo);
      }
    } catch (InterruptedException e) {
      if (playerOne != null)
        players.addPlayer(playerOne);
      throw e;
    }
  }
```

```
}
```

Пример 8

```
public class TaskRunner implements Runnable {
  private BlockingQueue<Task> queue;
  public TaskRunner(BlockingQueue<Task> queue) {
    this.queue = queue;
  }
  public void run() {
    try {
      while (true) {
        Task task = queue.take(10, TimeUnit.SECONDS);
        task.execute();
      }
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Пример 9

```
public class PrimeProducer extends Thread {
  private final BlockingQueue<BigInteger> queue;
  PrimeProducer(BlockingQueue<BigInteger> queue) {
    this.queue = queue;
  }
  public void run() {
    try {
      BigInteger p = BigInteger.ONE;
      while (!Thread.currentThread().isInterrupted())
        queue.put(p = p.nextProbablePrime());
    } catch (InterruptedException consumed) {
      /* Allow thread to exit */
    }
  }
  public void cancel() {
    interrupt();
  }
}
```

Модель OSI. Модель «Клиент-сервер». Понятие порта. Абстракция сокета. Пакет java.net. Класс Socket. Порядок работы с сокетом клиента. Класс ServerSocket. Сервер параллельной обработки запросов. Дейтаграммы. Uniform Resource Locator.

Модель OSI

В 1984 г. была разработана **модель OSI** – модель взаимодействия открытых систем (Open Systems Interconnected). Эта модель состоит из семи уровней:

- **Физический уровень**
- **Уровень соединения (канальный уровень)**
- **Сетевой уровень**
- **Транспортный уровень**
- **Сеансовый уровень**
- **Уровень представления**
- **Прикладной уровень**

Физический уровень. Этот уровень описывает среду передачи данных. Стандартизируются физические устройства, отвечающие за передачу электрических сигналов (разъемы, кабеля и т.д.) и правила формирования этих сигналов. Физический уровень также отвечает за преобразование сигналов между различными средами передачи данных. Например, при необходимости соединить сегмент сети, построенной на оптоволокне и витой паре применяют т.н. конверторы (в данном случае они преобразуют световой импульс в электрический).
Уровень соединения (канальный уровень). На физическом уровне пересылаются просто набор сигналов – битов. При этом не проверяется, что несколько компьютеров могут в одну среду передачи данных одновременно передавать информацию в виде битов. Поэтому одной из задач канального уровня является проверка доступности среды передачи. Также этот уровень отвечает за доставку между источником и адресатом в пределах сети с одной топологией.
Сетевой уровень. Одним из ограничений канального уровня является использование “плоской” модель адресации. Протокол, который поддерживается сетевым уровнем, использует иерархическую структуру для уникальной идентификации компьютеров. Для примера представим себе телефонную сеть. Благодаря такой иерархической структуре мы можем определить расположение требуемого абонента с наименьшими затратами. Иерархическая адресация для сети также должна позволять передавать данные между разрозненными и удаленными сетями.
Транспортный уровень. Рассмотрим TCP/IP протокол транспортного уровня модели OSI. TCP/IP имеет два протокола – TCP и UDP. **Transmission Control Protocol TCP** – основанный на соединениях протокол, обеспечивающий надежную передачу данных между двумя компьютерами с сохранением порядка данных. Используется в HTTP, FTP.
User Datagram Protocol UDP – не основанный на соединениях протокол, реализующий пересылку независимых пакетов данных, называемых дейтаграммами, от одного компьютера к другому без гарантии их доставки.
Сеансовый уровень. Обеспечивает установку, контроль и окончание сессии между приложениями. Уровень сессий координирует приложения, когда они взаимодействуют между двумя хостами.

Уровень представления. Уровень представлений отвечает за представление данных в форме, понятной получателю. Например, для представления данных могут быть использованы такие кодировки, как Extended Binary Coded Decimal Interchange Code (EBCDIC) или American Standard Code for Information Interchange (ASCII). Если компьютеры, между которыми установлено сетевое соединение, используют различные протоколы, то presentation layer обеспечивает их корректное взаимодействие. Уровень представлений также отвечает за шифрацию и сжатие данных.
Прикладной уровень. Уровень приложений определяет, какие ресурсы существуют для связи между хостамиэ
Каждый уровень взаимодействует только с соседними уровнями, протокол взаимодействия стандартизован. Это позволяет использовать реализации сетевого и программного обеспечения от разных производителей.

Понятие порта

- Обычно компьютер имеет только одно физическое соединение с сетью
- Соединение описывается IP-адресом
- Сокет привязывается к порту
- Порт описывается 16-битным числом
- Порты 0-1023 зарезервированы

Модель «клиент-сервер»

Большинство сетевых приложений можно классифицировать как клиент-серверные приложения. Каждая из сторон виртуального соединения называется **«сокет» (socket).** Приложение-сервер инициализируется при запуске и далее бездействует, ожидая поступления запроса от клиента.
Типы приложений-серверов:

- сервер последовательной обработки запросов
- сервер параллельной обработки запросов

Процесс-клиент посылает запрос на установление соединения с сервером, требуя выполнить для него определенную функцию.
Сетевое соединение – это процесс передачи данных по сети между двумя компьютерами или процессами.
Сокет – конечный пункт передачи данных. Для программ **сокет** – одно из окончаний сетевого соединения. Для установления соединения каждая из сетевых программ должна иметь свой собственный сокет. **Связь между двумя сокетами** может быть **ориентированной** или **не ориентированной на соединение.**
Сокет связан с номером порта. В Java сокеты инкапсулированы в экземпляры специальных классов. Все низкоуровневое взаимодействие скрыто от пользователя. Существует семейство классов, обеспечивающих настройку сокетов и работу с ними. Классы для работы с сетью в Java располагаются в пакете java.net.

Пакет java.net

- Адресация
- Установление TCP-соединения
- Передача/прием дейтаграмм через UDP
- Обнаружение/идентификация сетевых ресурсов
- Безопасность: авторизация/права доступа

Адресация

Класс **InetAddress** является интернет-адресом, или IP. Экземпляры этого класса создаются не с помощью конструкторов, а с помощью статических методов:

- InetAddress getLocalHost() – возвращает IP-адрес машины, на которой исполняется Java-программа
- InetAddress getByName(String name) – возвращает адрес сервера, чье имя передается в качестве параметра. Это может быть как DNS-имя, так и числовой IP, записанный в виде текста, например, "67.11.12.101"
- InetAddress[] getAllByName(String name) – определяет все IP-адреса указанного сервера

Класс **Inet4Address** служит для описания адреса, состоящего из 4 байтов, с помощью класса **Inet6Address** описывается адрес, состоящий из 16 байтов. Начиная с JDK 1.4. существует класс **InetSocketAddress**(String hostname, int port), который создаёт объект адреса для указанного узла и порта.

Класс Socket

Реализует клиентский сокет и его функции

- **Конструкторы**
  - Socket()
  - Socket(InetAddress address, int port)
  - Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
  - Socket(String host, int port)
  - Socket(String host, int port, InetAddress localAddr, int localPort)
- **Методы**
  - void close() – закрывает используемый сокет
  - InputStream getInputStream() – возвращает поток, позволяющий читать данные, переданные по сети
  - OutputStream getOutputStream() – возвращает поток, позволяющий передавать данные по сети
  - И т.д.

Порядок работы с сокетом клиента

- Открытие сокета
- Открытие потока ввода и/или потока вывода для сокета
- Чтение и запись в потоки согласно установленному протоколу общения с сервером
- Закрытие потоков ввода-вывода
- Закрытие сокета

Пример 1

```
import java.io.*;
import java.net.*;
 public class EchoClient {
  public static void main(String[] args) throws IOException {
    Socket echoSocket = null;
    PrintWriter out = null;
    BufferedReader in = null;
    try {
      echoSocket = new Socket("taranis", 7);
      out = new PrintWriter(echoSocket.getOutputStream(), true);
```

```
      in = new BufferedReader(new
InputStreamReader(echoSocket.getInputStream()));
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host: taranis.");
      System.exit(1);
    }
catch (IOException e) {
      System.err.println("Couldn't get I/O for the connection
to: taranis.");
      System.exit(1);
    }
   BufferedReader stdIn = new BufferedReader(
                                new
InputStreamReader(System.in));
   String userInput;
   while ((userInput = stdIn.readLine()) != null) {
      out.println(userInput);
      System.out.println("echo: " + in.readLine());
   }
   out.close();
   in.close();
   stdIn.close();
   echoSocket.close();
   }
}
```

Класс ServerSocket

Реализует серверный сокет и его функции

- **Конструкторы**
  - ServerSocket()
  - ServerSocket(int port)
  - ServerSocket(int port, int backlog)
- **Методы**
  - void close() – закрывает используемый сокет
  - **Socket accept()** – приостагавливает выполнение программы и ожидает обращения клиента
  - И т.д.

Пример 2

```
try {
   serverSocket = new ServerSocket(4444);
} catch (IOException e) {
   System.out.println(
        "Could not listen on port: 4444");
   System.exit(-1);
}
Socket clientSocket = null;
try {
   clientSocket = serverSocket.accept();
} catch (IOException e) {
   System.out.println("Accept failed: 4444");
   System.exit(-1);
}
```

Сервер параллельной обработки запросов

- Установить соединение клиент-сервер
- Сервер параллельной обработки передает управление дочернему процессу
- Если в момент обработки запроса от первого клиента поступает запрос от второго, сервер параллельной обработки передает управление второму дочернему процессу

Дейтаграммы

Дейтаграмма – независимое, самодостаточное сообщение, посылаемое по сети, чья доставка, время (порядок) доставки и содержимое не гарантируются. Могут использоваться как для адресной, так и для широковещательной рассылки. Экземпляры класса **DatagramPacket** являются прототипами дейтаграмм-сообщений. Экземпляры класса **DatagramSocket** являются не ориентированными на соединение сокетами.

Uniform Resource Locator

URL – адрес ресурса в Интернет

- Имя протокола
  Протокол, используемый для связи
- Имя хоста
  Имя компьютера, на котором расположен ресурс
- Имя файла
  Путь к файлу на компьютере
- Номер порта
  Номер порта для соединения (необязателен, если порт не указан, то используется значение по умолчанию для указанного протокола)
- Ссылка
  Ссылка на именованный якорь (необязательна)

- Может быть абсолютным и относительным
- URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");
- Для простого извлечения содержимого заданного ресурса достаточно использовать метод openStream() класса URL
- Этот метод возвращает объект InputStream

Пример 3

```
import java.net.*;
import java.io.*;
public class URLReader {
  public static void main(String[] args) throws Exception
  {
        URL oracle = new URL("http://www.oracle.com/");
        BufferedReader in = new BufferedReader( new
InputStreamReader(
                              oracle.openStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
              System.out.println(inputLine);
        in.close();
  }
}
```

Для получения дополнительной информации о ресурсе потребуется использовать класс URLConnection, который предоставляет гораздо больше средств управления доступом к Web-ресурсам.
Для получения объекта URLConnection нужно вызвать метод openConnection() класса URL.

Пример 4

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
  public static void main(String[] args) throws Exception {
       URL oracle = new URL("http://www.oracle.com/");
       URLConnection yc = oracle.openConnection();
       BufferedReader in = new BufferedReader(new
InputStreamReader(yc.getInputStream()));
       String inputLine;
       while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
       in.close();
   }}
```

Существует несколько методов класса URLConnection, предназаначеннных для указания свойств соединения ещё до подключения к серверу. По умолчанию соединение получает входной поток для чтения, но не получает выходной поток для записи. Для получения выходного потока нужно вызвать метод setDoOutput(true).

Пример 5

```
import java.io.*;
import java.net.*;
public class Reverse {
public static void main(String[] args) throws Exception {
if (args.length != 2) {
  System.err.println("Usage: java Reverse " + "
string_to_reverse");
  System.exit(1);
 }
String stringToReverse = URLEncoder.encode(args[1], "UTF-8");
 URL url = new URL(args[0]);
 URLConnection connection = url.openConnection();
 connection.setDoOutput(true);
 OutputStreamWriter out = new OutputStreamWriter(
connection.getOutputStream());
 out.write("string=" + stringToReverse);
 out.close();
 BufferedReader in = new BufferedReader( new
InputStreamReader(connection.getInputStream()));
 String decodedString;
 while ((decodedString = in.readLine()) != null) {
       System.out.println(decodedString);
}
in.close();
}
}
```

Апплет. Тег <applet>. Передача параметров. Класс Applet. Скелетная структура апплета. Методы отрисовки. Класс Graphics. Работа с цветом. Работа со шрифтами.

Апплет. Тег <applet>

Апплеты – небольшие приложения, доступные на интернет-сервере и транспортируемые по сети Интернет. Автоматически устанавливаются и выполняются как часть HTML-документа.
После доставки клиенту апплет имеет ограниченный доступ к ресурсам.
Апплет имеет тег <APPLET>, используемый для запуска апплета как приложения из HTML-документа и из программы appletviewer.
Программа appletviewer выполняет апплет в отдельном окне, в то время как браузер может расположить несколько апплетов на одной странице.
Атрибут CODE – обязательный атрибут задания имени файла, в котором содержится оттранслированный код объекта.
Имя файла задано относительно либо текущего каталога, либо относительно каталога, указанного в атрибуте CODEBASE.
Вместо атрибута CODE может использоваться OBJECT: указывается имя файла, содержащего сериализованный объект, из которого последний будет восстановлен.
При запуске определенный таким образом апплет должен вызываться не методом init(), а методом start().
Употреблять одновременно CODE и OBJECT нельзя.
WIDTH и HEIGHT задают начальный размер видимой области апплета (обязательны).
ARCHIVE – необязательный атрибут, задает список jar-файлов, которые предварительно загружены в браузер.
BASE – необязательный атрибут, задает базовый URL кода апплета. Является каталогом, в котором будет выполнен поиск исполняемого файла апплета, задаваемое в признаке CODE (если не задан каталог по умолчанию).
ALT – необязательный атрибут; задает короткий текст сообщения, выводимое в виде всплывающей подсказки при нахождении курсора над областью апплета. В том случае, если используемый браузер распознает синтаксис тега <APPLET>, но выполнять апплеты не умеет.
Это не то же, что HTML-текст, который можно вставлять между открывающим и закрывающим тегом апплета для браузеров, которые вообще не поддерживают апплеты.
NAME – необязательный атрибут, используется для задания имени апплета. Имя апплета необходимо, чтобы другие апплеты на этой же странице могли их находить и обмениваться информацией.
ALIGN – необязательный, используется для задания стиля выравнивания апплета
VSPACE и HSPACE – задают поля
PARAM NAME = appletAttribute VALUE = value – задает возможность передать из HTML-страницы необходимые аргументы

Передача параметров

Метод getParameter(String) – возвращает значение, соответствующее указанному имени параметра. Преобразовывать значение самостоятельно.

Пакет java.applet. Класс Applet

Applet – класс-предок любого апплета. При наследовании переопределяют ряд методов:

1) init() – вызывается при инициализации
2) start() – вызывается каждый раз при выборе документа, содержащего апплет
3) stop() – вызывается каждый раз, когда браузер покидает документ с апплетом
4) destroy() – вызывается, когда требуется закончить выполнение апплета.

Методы отрисовки:

1) paint(Graphics) – вызывается каждый раз при повреждении апплета
   AWT следит за состоянием окон в системе и замечает такие случаи, как, например, перекрытие окна апплета другим окном. В этих случаях, когда апплет оказывается видимым, он перерисовывается методом paint();
2) update(Graphics) – используется по умолчанию. Класс Applet закрашивает фон цветом по умолчанию, после чего вызывает paint(). Если в paint() задан другой цвет, то пользователь будет видеть вспышку цветом по умолчанию. Чтобы избежать этого, все операции рисования выполняются в update(), а в paint() вызывается update().
3) repaint() используется для принудительной перерисовки апплета, в свою очередь вызывает update(). Близкие по времени запросы на перерисовку объединяет AWT. Имеются вариации для перерисовки линий и прямоугольников.

Класс Graphics

Класс Graphics в пакете java.awt.graphics. Нужен для рисования в контекстах компонентов, изображений в памяти. Например, имеет метод drawstring(String, int, int).

Работа с цветом – класс jawa.awt.Color. Может использовать константы для задания цвета. Используются конструкторы Color (int, int, int), Color (int), Color (float, float, float). Метод getRed() – возвращает количество красного в младших битах значения базового цвета

Работа со шрифтами – java.awt.Font. Содержит константы и конструкторы. Пример – Font(String, int, int) //имя, стиль, размер

Содержит методы модификации и выяснения размера и стиля шрифта.

Класс java.awt.FontMetrics хранит геометрические характеристики шрифтов.

Особенности AWT. Менеджеры компоновки. Проблемы AWT. Особенности Swing. Look And Feel. Апплеты в Swing. Создание оконных приложений. Отрисовка компонентов.

AWT (Abstract Window Toolkit).
Особенности AWT – компоненты принадлежат ОС, что вызывает нестабильный код, отображение меняется при смене ОС. Применяется в апплетах и оконных приложениях. Абстрактный класс Component определяет базовую функциональность всех компонентов.

Containter – абстрактное подключение класса Component, определяет дополнительные методы, которые позволяют помещать в него другие компоненты, что дает возможность построения иерархической системы визуальных объектов. Отвечает за расположение содержащихся компонентов с помощью интерфейса LayoutManager.

Panel – простая специализация класса Container (с возможностью создания объектов). Обычно представляют в виде экранного компонента, допускающего рекурсивную вложенность. С помощью метода add() в Panel добавляются другие компоненты, а после этого можно задать расположение, размер, границы: setLocation(), setSize(), setBounds().

Canvas – семантический пустой компонент, позволяющий отрисовывать на себе произвольного вида компоненты.

Label – отображает строку, можно управлять параметрами текста.

Button – реализует кнопку, создает событие.

Checkbox – реализует флажок, имеет методы установки значения, создает событие.

CheckboxGroup – реализует выбор одного из вариантов, имеет методы установки значений, создает событие.

Choice – реализует выпадающий список, имеют методы управления состоянием, создает событие.

List – реализует список с возможностью выбора нескольких вариантов и прокруткой, имеет методы управления состоянием, создает событие.

Scrollbar – реализует полоску для прокрутки, МУС, СС

TextField – однострочная область для ввода текста, МУС, СС

TextArray – многострочный редактор, МУС, СС

Менеджеры компоновки управляют размещением компонентов в контейнере, учитывая параметры этих компонентов, например, предпочтительный размер
Реализует интерфейс java.awt.LayoutManager.
Устанавливается с помощью метода setLayout() в контейнере.

1) Простая поточная компоновка FlowLayout

2) Граничная компоновка BorderLayout
3) Сетка GridLayout – матрица с одинаковыми ячейками
4) Сетка с настраиваемыми параметрами GridBagLayout
5) CardLayout – колода карт

Проблемы AWT:

1) исходные элементы в различных ОС могут иметь различия
2) часто имеются элементы GUI, отсутствующие в ОС
3) Использование нативных методов может приводить к проблемам на различных платформах.

Особенности Swing:

1) Элементы GUI рисуются в пустых окнах
2) Нативные функции – только для вывода окна отрисовки и получения информации о действиях пользователя
3) Обладает свойством легковесности
4) Набор элемент шире, чем в AWT, и может быть расширен
5) Нет сильной привязки к нативным методам
6) Отображение на различных платформах единообразно

look & feel. При использовании Swing можно придать программе заданный стиль. Есть стандартные стили. Можно создать свои. Внешний вид можно изменить при выполнении. Перерисовка принудительная.

Апплеты в Swing (класс JApplet). Для добавления элемента в апплет не используют add(). Содержимое апплета – на панели, ссылку на которую получают, вызвав getContentPanel(), у которой можно вызвать add().

Для оконных приложений используют JFrame. Содержимое окна – на панели. Ссылку можно получить через getContentPanel(). Параметрами окна можно управлять.

Отрисовка компонентов в методе paintComponent(). Для перерисовки требуется repaint(). В целях экономии времени логично запоминать нарисованный статический объект как рисунок в памяти. Дополнительных действий для этого не нужно. Используется механизм буферизации – реализуется хранение информации на уровне механизмов отрисовки. Для одного участка видимой области используется не более одного изображения буфера.

Методы JComponent:

1) setDoubleBuffered() – двойная буферизация;
2) isDoubleBuffered().

Замечания:

1) Swing представляет более широкие и надежные возможности, чем AWT;
2) Эти возможности касаются в основном отрисовки
3) Модель обработки событий принадлежит AWT
4) Некорректно говорить, что Swing заменил AWT
5) Современные средства разработки имеют визуальные редакторы.

Нерассмотренные возможности:
Работа с графикой: java.awt.Graphics2D
С геометрическими примитивами: java.awt.geom
Меню: javax.swing.JMenuBar
javax.swing.JMenu

Модель делегирования обработки событий. Событие. Источник. Слушатель. Пример. Классы-адаптеры.

Событие – объект, описывающий состояние источника. В модели обработки событий Java разным типам событий соответствуют разные классы в java, являющиеся наследником EventObject. События пакета AWT – подклассы java.awt.AWTEvent.

Для удобства события различных типов пакета awt помещены в новый пакет java.awt.event.
Для любого события существует порождающий его объект, который может подключить с помощью метода public Object getSource() и для любого события есть идентификатор, получаемый методом getID(). Используется для различия разных типов событий.

Источник – объект, генерирующий событие. Может генерировать несколько типов событий. Регистрации нет.
addTypeListener(TypeListener) и removeTypeListener(TypeListener) – добавляют и удаляют слушателей

Слушатель – объект получает уведомление о событии. Может быть зарегистрирован несколькими источниками. Должен иметь методы приема и обработки уведомления. Существует набор интерфейсов TypeListener, описывающий обработку событий. Объекты ActionEvent реализуют ActionListener.

Класс Adapter. Для каждого интерфейса слушателя содержатся методы в пакете java.awt.event, определяющий просто класс-адаптер, который обеспечивает пустое тело для каждого из методов соответствующего интерфейса. Когда таких методов мало, проще получить подкласс адаптера. При получении подкласса адаптера требуется лишь переопределить те методы, которые нужны, а при прямой реализации интерфейса необходимо определить все методы. Заранее определяют классы-адаптеры, называемые как интерфейсы, которые они реализуют, но в их названии Listener заменяется на Adapter.

Статические вложенные классы. Вложенные интерфейсы. Нестатические вложенные классы. Локальные классы. Анонимные классы.

Вложенные классы и интерфейсы. Возможность определить вложенные типы, предусмотренные в Java, служат нескольким целям:

1) Вложенные типы позволяют представлять тип в виде логически связанных групп и контекстов;
2) Вложенные типы – простой эффективный инструмент объединения семантики соотносимых объектов.

Вложенные типы считаются частью внешнего типа, и оба типа находятся во взаимодоверительных отношениях, то есть один из них обладает правом обращаться ко всем членам другого.
Основные различия между вложенными типами обусловлены тем, является ли вложенный тип классом или интерфейсом и тем, к какой категории относится внешний тип – класс или интерфейс.

Вложенные типы могут быть объявлены как статические.

Статические вложенные классы – простейшая форма вложенного класса. Если класс вложен в интерфейс, он получает статус статического по умолчанию. static опускается. Статический вложенный класс может наследовать другие классы. Реализует любые интерфейсы и сам служит объектом расширения для любого класса с необходимыми правами доступа.
В объявлении статических вложенных классов, как и в обычных, можно применять final и abstract. Статические классы, вложенные в другие, являются членами последнего и допускают применение любых модификаторов доступа.

Вложенные интерфейсы
Всегда статические. static опускается. Доступ определяется доступом внешнего класса или интерфейса.

Нестатические вложенные классы.
Называют внутренними классами. Объект класса всегда ассоциируется с внешним объектом. Элементы внутреннего класса имеют доступ к полям и методам внешнего объекта. Внутренний класс не может содержать статические члены, кроме полей вида static final.
Внутренние классы, как и обычные, способны расширять любой класс, реализовывать интерфейсы и выступать в роли объекта наследования. Возможно использование в объявлении внутреннего класса final и abstract.

Локальные классы. Разрешается объявление вложенных классов внутри блоков кода – тел методов, конструкторов, блоков инициализации. Подобный локальный класс не является членом класса, к которому относится блок, а принадлежит этому блок, как локальная переменная. Такие классы недоступны за пределами внешнего класса – нет способов обращаться к ним. Единственный возможный модификатор класса –final. Экземпляры имеют доступ к полям и методам внешнего класса и к локальным переменным, помеченным final.

Анонимные классы описываются непосредственно в выражении и служат его частью. Тип, указанный после new, - базовый для анонимного класса. Метод расширяет один класс или реализует один интерфейс. Явно implements и extend не пишут. Нет конструкторов (у класса нет имени). Параметры, необходимые для создания объекта, передаются через super базовому конструктору.