

KAZAKH-BRITISH TECHNICAL UNIVERSITY



OBJECT-ORIENTED PROGRAMMING AND DESIGN

PAKIZAR SHAMOI

ALMATY 2013

Content

Введение	7
1 Объектно-ориентированный подход.....	9
1.1 Парадигма объектно-ориентированного программирования.....	9
1.2 Java как объектно-ориентированный язык	13
2 Введение в язык программирования Java.....	18
2.1 Почему Java? (функции Java)	19
2.2 Виртуальная машина Java (JVM).....	21
2.3 Компиляция и запуск Java-программы.....	22
2.4 Синтаксис Java.....	23
3 Основы объектов и классов.....	39
3.1 Что такое класс?	39
3.2 Что такое объект?	43
3.3 Модификаторы класса	46
3.4 Члены класса и члены экземпляра	46
3.5 Подробнее о методах.....	48
3.6 Инкапсуляция	51
3.7 Конструкторы	53
3.8 Это ключевое слово	55
3.9 Блоки инициализации.....	57
3.10 Импорт классов	58
3.11 Абстракция класса	60
3.12 Перечисления.....	60
3.13 Классы-оболочки.....	61

3.14 Базовые классы Java.....	62
3.15 Определение собственных классов	65
3.16 Дизайн класса.....	70
Глава 4. Наследование, полиморфизм и абстрактные классы.....	78
4.1 Концепция наследования. Подкласс и суперкласс.....	78
4.2 Иерархия классов.....	79
4.3 Специальное ключевое слово супер	80
4.4 Этапы создания объекта	81
4.5 Перегрузка и переопределение методов.....	83
4.6 Полиморфизм.....	84
4.7 Преобразование типов	89
4.8 Абстрактные классы и методы.....	93
4.9 Класс объекта.....	96
4.10 Советы по проектированию для наследования	98
4.11 Подробный пример.....	99
5 интерфейсов	110
5.1 Интерфейсы: основная концепция.....	111
5.2 Члены интерфейса.....	112
5.3 Порядок реализации	113
5.4 Расширение интерфейсов.....	114
5.5 Почему мы используем интерфейсы?.....	117
5.6 Маркерные интерфейсы	119
5.7 Клонирование объекта	120
5.8 Интерфейсы против абстрактных классов.....	123
5.9 Пример: Стек.....	125
5.10 Внутренние классы.....	126

Объектно-ориентированное программирование и дизайн

5.11 Интерфейсы и наследование. Когда что использовать?.....	127
5.12 Принципы объектно-ориентированного проектирования.....	129
6 UML-диаграммы.....	136
6.1 Диаграммы вариантов использования	137
6.2 Диаграммы последовательности.....	138
6.3 Диаграммы классов.....	140
6.4 Ассоциация.....	144
6.5 Обобщение.....	146
6.6 Зависимость.....	147
6.7 Реализация	149
6.8 Агрегация.....	149
6.9 Состав	151
6.10 Важные вопросы – множественность, ограничения и примечания.....	153
6.11 Пакеты UML.....	154
6.11 Советы.....	155
6.12 Краткая информация о классовых отношениях.....	156
7 Коллекции и структуры данных.....	165
7.1 Корневой интерфейс — Коллекция.....	165
7.2 Итератор и ListIterator.....	167
7.3 Интерфейс списка	169
7.4 Интерфейсы Set и SortedSet	170
7.5 Интерфейсы карты и SortedMap	173
7.6 Класс коллекций	174
8 Файлы и потоки.....	180
8.1 Иерархия потоков в Java.....	180
8.2 Потоки данных.....	181

8.3	Печать потоков.....	183
8.4	Буферизованные потоки.....	184
8.5	Сериализация.....	185
8.6	Файлы последовательного и произвольного доступа.....	188
8.7	Класс файла.....	189
8.8	Строковый токенизатор.....	190
9	исключений	196
9.1	Иерархия исключений	196
9.2	Проверенные и непроверенные исключения.....	197
9.3	Попробуй – поймай блок.....	198
9.4	Заявление об исключении.....	199
9.5	Выдача исключений.....	200
10	последних достижений в компонентном программном обеспечении: превосходит ли Scala Java?..	208
10.1	От дедуктивной системы к программам.....	209
10.2	Полиморфизм и повседневное программирование.....	211
10.3	Смешение подходов к инкапсуляции	217
10.4	Сблизились ли объектно-ориентированные и функциональные языки?	
	223	
	Заключение.....	224
	Благодарности.....	226
	Использованная литература	227

Introduction

Oтехнология *object* находится в разработке более сорока лет. В настоящее время он внедрен в такие разнообразные области, как разработка требований, архитектура программного обеспечения, анализ, проектирование, программирование, тестирование, развертывание и обслуживание. Наиболее широко используемые современные языки программирования C++, Java и C#.Net используют объектно-ориентированный подход. В этом руководстве рассматривается применение объектно-ориентированной парадигмы в программировании.

Фундаментальные концепции объектно-ориентированного программирования объясняются с использованием языка программирования Java. Итак, основная цель этой книги — развить понимание ключевых принципов, лежащих в основе объектно-ориентированного программирования, и применить объектно-ориентированные подходы с использованием языка программирования Java.

Изучив учебное пособие, студент получит знания фундаментальных принципов объектно-ориентированного программирования. В частности, он познакомится с концепциями классов, объектов, интерфейсов, наследования, вложенных и абстрактных классов, полиморфизма и инкапсуляции данных. Особое внимание в этой книге удалено UML (унифицированному языку моделирования). Более того, в заключительных разделах книги представлена информация, касающаяся основных конструкций Java – коллекций Java (наборов, списков, карт), исключений, ввода/вывода, исключений Java и т. д.

Учебное пособие в основном ориентировано на студентов, изучающих объектно-ориентированное программирование, а также на всех, кто хочет освоить принципы ООП. В этой книге предполагается, что читатель имеет представление об основных конструкциях программирования, таких как переменные, повторения, условия и т. д.

Не прибегайте к запоминанию. Понимание — единственный ключ к объектно-ориентированности. Потратите некоторое время на размышления о том, как можно взять уже разработанный код и преобразовать его в правильно структурированную систему, используя принципы ООП, такие как инкапсуляция, абстракция, наследование, полиморфизм. Ключевым показателем производительности здесь является способность идентифицировать объекты системы, их состояния и поведение, а также способность указывать поведение, которое можно инкапсулировать и унаследовать (идентификация иерархических отношений).

Еще один вопрос, который следует прояснить в самом начале, — это системные требования для большинства программ, с которыми вы встретитесь в этом руководстве. Итак, чтобы выполнить упражнения этого руководства, вам потребуется установить и настроить среду разработки, состоящую из:

- JDK 6 (или выше) от Sun/Oracle

Введение

- Eclipse IDE для разработчиков Java.

Инструкции по установке и настройке в этой книге опущены. Все перечисленное программное обеспечение бесплатное и его можно скачать с официальных сайтов вместе с инструкцией по установке.

Удачи!

1 Object-oriented Approach

«Объектно-ориентированная версия «кода спагетти» — это, конечно же, «код лазаньи» .

(Слишком много слоев) .

Роберто Уолтман

«Люди думают, что информатика — это искусство гениев, но на самом деле все наоборот:

многие люди делают вещи, которые опираются друг на друга, как стена из маленьких камней» .

Дональд Кнут

Т его раздел проведет вас через основные концепции объектно-ориентированного программирования на платформе Java. В частности, мы рассмотрим существующие парадигмы программирования и укажем роль объектно-ориентированного подхода среди них. Кроме того, мы рассмотрим, как эти объектно-ориентированные конструкции реализованы на языке программирования Java.

1.1 Парадигма объектно-ориентированного программирования

В мире программирования существует множество парадигм, и объектно-ориентированное программирование — одна из них. Другие парадигмы программирования включают парадигму императивного программирования (поддерживаемую такими языками, как Паскаль или С), парадигму логического программирования (примером которой является Пролог) и парадигму функционального программирования (представленную такими языками, как ML, Haskell, Lisp и недавно появившиеся F # и Скала). Например, в парадигме функционального программирования мы рассматриваем вычисления как оценку математических функций.

В целом языки программирования можно разделить на два типа: императивные языки и декларативные языки. Императивный стиль описывает знание «как сделать», тогда как декларативный стиль описывает «что такое знание» . Следовательно, программа является декларативной , если она описывает , что представляет собой объект, а не как его создать. Очевидно, что это совершенно другой подход, чем традиционное императивное программирование, которое требует от программиста указания пошагового рецепта или точного алгоритма для выполнения. В других

¹ Код спагетти представляет собой неорганизованный код без четкой структуры, начала и конца.

Глава 1. Объектно-ориентированный подход

Другими словами, императивные программы делают алгоритм явным, оставляя цель неявной, тогда как декларативные программы делают цель явной, но оставляют алгоритм неявным.

Объектно-ориентированное программирование представляет собой попытку заставить программы более точно моделировать то, как люди думают о мире и взаимодействуют с ним (1). Императивный программист, решающий какую-либо задачу, должен сначала определить вычислительную задачу, которую необходимо выполнить для ее решения. В таком случае программирование состоит из поиска последовательности инструкций, которые позволят выполнить эту задачу (1). Напротив, в основе объектно-ориентированного программирования вместо задач мы находим объекты — сущности, которые имеют поведение, содержат информацию и могут взаимодействовать друг с другом. Итак, программирование состоит из проектирования набора объектов, представляющих реальные или абстрактные сущности в проблемной области. Предполагается, что это сделает дизайн программы более естественным и, следовательно, более простым для понимания и разработки.

Объектная ориентация предоставляет набор инструментов и методов, которые позволяют создавать надежные, удобные для пользователя, поддерживаемые, хорошо документированные, многократно используемые программные системы, отвечающие требованиям своих пользователей (2). Работа с объектно-ориентированным мышлением открывает двери новому способу программирования, в котором программная система рассматривается как сообщество объектов, которые взаимодействуют друг с другом путем передачи сообщений при решении проблемы.

В настоящее время утверждается, что методы решения проблем, используемые в объектно-ориентированном программировании, более точно моделируют то, как люди решают повседневные проблемы. В качестве простой иллюстрации рассмотрим, как мы решаем бытовую задачу: предположим, вы захотели заказать маме на день рождения торт с надписью «С Днем Рождения, мама!». Для решения этой задачи вы просто находитите человека, который принимает заказы на приготовление торты, предположим, его зовут Азиз. Вы сообщаете Азизу, какой торт нужно приготовить (например, медовик), его вес (например, 4 кг) и все остальные необходимые детали (например, надпись «С Днем Рождения, мама!»). Вы можете быть уверены, что пирог будет готов. Теперь давайте рассмотрим механизмы, используемые для решения вашей проблемы.

1. Сначала вы нашли подходящего агента (в данном случае Азиза) и передали этому агенту сообщение , содержащее запрос.
2. Ответственность за удовлетворение запроса лежит на Азизе.
3. Есть какой-то метод (алгоритм или набор операций), который Азиз использует для этого. Однако вам не обязательно знать конкретные способы (технику приготовления), используемые для удовлетворения запроса — такая информация от вас скрыта .

Конечно, вам не обязательно знать точные детали, но при расследовании вы можете обнаружить, что Азиз (предположим, он шеф-повар), выполняя вашу просьбу, передал другому еще несколько сообщений.

Объектно-ориентированное программирование и дизайн

повара в его бригаде, являющиеся его подчиненными (скажем, по приготовлению крема и теста). Если мы подумаем еще немного, то сможем представить, что шеф-повар, помимо взаимодействия с клиентами, заказывающими торты, также взаимодействует с оптовыми продавцами муки, масла, шоколада и т. д., которые, в свою очередь, взаимодействуют с другими людьми.

Этот небольшой, но очень полезный пример иллюстрирует потенциальную мощь объектно-ориентированного подхода и подводит нас к нашей первой концептуальной картине объектно-ориентированного программирования: объектно-ориентированная программа структурирована как сообщество взаимодействующих агентов, называемых объектами . Каждый объект имеет свою роль. Каждый объект предоставляет услугу или выполняет действие, используемое другими членами сообщества (3). Каждый объект играет очевидную роль в системе, и прелест этого в том, что когда требуется изменение или возникает ошибка, объект, который нужно изменить, становится очевидным. Поскольку каждый объект сохраняет контроль над манипулированием своими атрибутами, результирующий код понятен. Более того, очень часто мы можем столкнуться с проблемой, когда какой-то объект ошибочно ссылается на переменную, которая к нему не относится. Скрытие данных может помочь нам избежать таких ошибок, обеспечив соответствующую видимость между объектами обмена сообщениями.

Из приведенного ранее примера мы узнали, что члены объектно-ориентированного сообщества делают запросы друг к другу путем передачи сообщения агенту (объекту) , ответственному за действия. Сообщение содержит запрос на действие и любую дополнительную информацию (параметры), необходимую для выполнения запроса. В ответ на сообщение получатель выполнит некоторый метод (или набор методов) для удовлетворения запроса.

Здесь следует выделить несколько важных вопросов:

- Клиенту, отправляющему сообщение с запросом, не обязательно знать, каким образом выполняется запрос (скрытие информации).
 - Важным принципом передачи сообщений является идея найти кого-то другого, кто выполнит эту работу, т. е. повторно использовать компоненты, которые могли быть написаны кем-то другим. Как и в нашем примере, шеф-повар делегирует ответственность за приготовление крема и теста другим поварам, хотя тоже может это сделать.
 - Просьбы клиента о действиях лишь указывают на желаемый результат. Получатели могут использовать любые методы и приемы для достижения этой цели. Этот принцип обеспечивает большую независимость между объектами.
 - У агентов есть обязанности, которые они могут выполнить по запросу.
- Совокупность обязанностей, связанных с объектом, часто называют протоколом .

Глава 1. Объектно-ориентированный подход

Очевидно, что разбиение кода на ряд объектов, которые могут обмениваться сообщениями, способствует развитию гибкой архитектуры, а использование ООП помогает избежать неуправляемого кода или спагетти. Гибкая архитектура дает преимущества в том смысле, что ее легче модифицировать благодаря тому, что объекты, составляющие приложение, имеют четкие границы, что значительно упрощает замену объектов, с которыми вы работаете. Архитектуру системы можно смоделировать с помощью UML², на котором изображены основные агенты, их обязанности и связи между ними. UML является предметом обсуждения главы 6.

Без сомнения, императивное программирование может работать хорошо, если вы единственный разработчик проекта, поскольку вы знакомы со своим кодом. Однако, когда в проекте задействовано больше программистов, им может быть обременительно знакомиться с кодом друг друга и просматривать тысячи строк кода, чтобы увидеть, где необходимо внести изменения. При использовании ООП каждое поведение в приложении содержится в уникальном классе, что обеспечивает более элегантный способ просмотра взаимодействия объектов. Поскольку каждый уникальный класс имеет имя, его легко отследить.

Идя глубже, давайте рассмотрим ключевые принципы ООП, которые жизненно важны для того, чтобы сделать код более организованным и гибким. А именно, это инкапсуляция, полиморфизм³, наследование и сокрытие данных (4). Давайте посмотрим на них глубже.

Важнейшая идея, реализуемая инкапсуляцией⁴, заключается в том, что то, что вам нужно знать для каждой проблемной области, должно быть четко отделено от того, что вам не нужно знать, чтобы вам не надоела ненужная информация. Таким образом, вы сможете сохранить концентрацию. Например, чтобы смотреть телевизор, нам не нужно точно (или даже вообще!) знать, как он работает. Вся информация, которой нам нужно владеть, — это то, как ею пользоваться — то есть как включать/выключать, переключать каналы, менять громкость и т. д. Между тем внутренние механизмы работы телевизора для нас — это черный ящик.

Более того, очевидно, что создание границ между объектами вашей системы обеспечивает взаимозаменяемость других вариантов поведения со схожими параметрами метода, именем метода и типом возвращаемого значения. Эти три компонента метода представляют собой контракты, необходимые для правильного обмена сообщениями, и вместе называются подписью. Таким образом, полиморфизм гарантирует, что пока сигнатуры между различными реализациями остаются одинаковыми,

² Унифицированный язык моделирования (UML) – стандартное представление, используемое для построения моделей для компьютерных приложений.

³ Латинское значение множества лиц или форм; процесс, который обеспечивает взаимозаменяемость объектов, имеющих одинаковый интерфейс.

⁴ Наследование — это принцип разделения и локализации поведения в объекте.

Объектно-ориентированное программирование и дизайн

поведение можно менять местами для достижения различных результатов без необходимости изменения большого количества кода.

Концепция наследования моделируется в объектно-ориентированных языках, что позволяет разработчикам писать код в форме иерархических отношений.

В результате программисты могут инкапсулировать набор поведений и атрибутов в суперкласс, который впоследствии можно будет использовать при создании дополнительных классов, что можно сделать, производя их от исходного класса. Подобно детям, получающим выгоду от имущества своих матери и отца, так и предметы могут получать выгоду от наследования. Дочерний объект в иерархии объектов называется подклассом или производным классом, а его родительский класс называется его суперклассом или базовым классом.

Скрытие данных — это скрытие информации от пользователя приложения и возможной проблемной области. Это помогает поддерживать правильную инкапсуляцию и обеспечивается использованием модификаторов доступа, таких как частный, общедоступный, защищенный, пакет, которые будут обсуждаться позже.

Теперь, когда мы рассмотрели принципы ООП, давайте сосредоточимся на их реализации в объектно-ориентированном языке.

1.2 Java как объектно-ориентированный язык

Объектно-ориентированные языки предназначены для облегчения структурирования кода на высоких уровнях абстракции. Одной из важнейших особенностей этих языков является возможность структурировать код на уровне классов. В языке Java весь код находится в методе, все методы принадлежат классу, а все классы, в свою очередь, принадлежат пакету .

Java, как объектно-ориентированный язык, обеспечивает поддержку всех объектно-ориентированных концепций, обсуждавшихся ранее, таких как объекты и классы, наследование, полиморфизм, инкапсуляция и т. д.

Проще говоря, инкапсуляцию можно определить как защитный барьер, который предотвращает произвольный доступ к данным со стороны другого кода, определенного вне класса. Он усилен различными модификаторами видимости (или доступа) в Java, такими как частный, защищенный, общедоступный и пакетный (без модификатора). В таблице 1, представленной ниже, представлена информация о них.

Как легко видеть из таблицы, `protected` является наиболее строгим модификатором, а `protected` — немного менее строгим, чем `Private`. Важно отметить, что по умолчанию поведения и атрибуты имеют модификатор пакета, если не указано иное.

Глава 1. Объектно-ориентированный подход

Модификатор доступа	Определение
частный	Доступ к атрибутам и поведениям возможен только в той области, в которой они объявлены.
защищенный	Поведения и атрибуты можно использовать только внутри класса, который их определил, и внутри его подклассов.
Упаковка (без модификатора)	Может быть просмотрен любым классом в одном пакете.
Общественный	Поля и методы могут быть просмотрены любым классом любого пакета.

Таблица 1. Модификаторы доступа в Java

Обратите внимание, что общедоступные методы являются точками доступа к полям частного класса из внешнего мира. Обычно эти методы называются геттерами (аксессорами) и сеттерами (мутаторами). Поэтому любой класс, который хочет получить доступ к переменным, должен получить к ним доступ через эти методы получения и установки.

Методы Accessor и Mutator будут обсуждаться позже в последующих разделах.

Наследование в Java определяет отношение Is-a между суперклассом и его подклассами. Это означает, что объект подкласса можно использовать везде, где можно использовать объект суперкласса. Это дает возможность создавать новые классы из существующих. В качестве иллюстрации рассмотрим класс Cat, который наследует некоторые свойства от общего класса Mammal. Здесь мы обнаруживаем, что базовым классом является класс Mammal, а подклассом является более конкретный класс Cat. Согласно синтаксису Java, подкласс должен использовать ключевое слово Extentions, чтобы быть производным от суперкласса.

```
класс Млекопитающих {
    ...
    класс Cat расширяет Mammal{
        ...
    }
}
```

Подкласс наследует члены суперкласса и, следовательно, способствует повторному использованию кода. Более того, сам подкласс может добавлять свои новые поля и методы. Класс java.lang.Object всегда находится вверху, он является корневым предком всех классов Java. Кстати, в Java нет прямой поддержки множественного наследования. Эта тема, а также другие темы, связанные с наследованием, подробно обсуждаются в главе 4.

Полиморфизм — важная объектно-ориентированная концепция, широко используемая в Java и других современных языках программирования. В Java есть

Объектно-ориентированное программирование и дизайн

отличная поддержка полиморфизма с точки зрения интерфейса, абстрактного класса, перегрузки и переопределения методов. Вы также не должны забывать, что наследование также дает вам возможность реализовать полиморфизм, предоставляя вам возможность замены, когда базовый класс может содержать ссылку на производный класс.

Переопределение метода позволяет программисту выполнять метод на основе определенного объекта во время выполнения вместо объявленного типа во время кодирования. Это называется динамической привязкой.

В качестве демонстрации рассмотрим простой пример, содержащий три иерархически связанных класса — суперкласс Person и два производных класса Student и Worker. Обратите внимание, что все классы имеют имена методов showDescription.

```
класс Человек {
    ...
    общественный недействительный showDescription () {
        вернуть "человек с именем " +имя;
    }
}
Студент класса {
    ...
    общественный недействительный showDescription () {
        вернуть «студент с идентификатором » +id;
    }
}
класс Сотрудник {
    ...
    общественный недействительный showDescription () {
        вернуть "сотрудник с зарплатой " +зарплата;
    }
}
```

Но как вы можете это использовать? Теперь эта структура позволяет вам использовать супертип в аргументе метода, что даст вам возможность передать любую реализацию при вызове метода. Например:

```
public void showDescription(Person StudentOrEmployee){
    StudentOrEmployee.showDescription();
}
```

Полиморфизм позволяет передавать в качестве параметра метод выше всех классов, расширяющих Person, поэтому вы можете передавать Student или Сотрудники. Как результат,

Глава 1. Объектно-ориентированный подход

у вас может быть код, который будет работать даже с еще не реализованными классами, единственное требование — они должны быть производными от Person.

Обратите внимание, что в этом разделе мы только что ввели понятие полиморфизма. Подробнее об этом читайте в разделе X.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Сообщение	Ответственный агент	Объект
Модификаторы доступа	Инкапсуляция	Полиморфизм
Наследование	Объектно-ориентированный UML public Private	
зашщищенный	Императивное программирование	Джава

ИСПЫТАНИЯ

1. Какие из этих модификаторов доступа недоступны в других пакетах?

- | | |
|---------------|-----------------------------|
| личное | б) общественный |
| в) защищенный | г) нет модификатора доступа |

2. ... позволяет создавать новые классы из существующих классов.

- | | |
|-----------------|---------------|
| а) наследование | б) абстракция |
| в) инкапсуляция | г) обобщение |

3. Основное понятие объектно-ориентированного программирования – это...

- | | | | |
|-------|---------|-----------------|-----------|
| метод | б) поле | в) наследование | г) объект |
|-------|---------|-----------------|-----------|

4. Термин, используемый для описания внутреннего представления объекта, скрытого от взгляда за пределами определения объекта?

- | | |
|-----------------|-----------------|
| а) полиморфизм | б) состав |
| в) инкапсуляция | г) наследование |

5. Когда у объекта много форм, это...

- | | |
|----------------------|-------------------|
| а) наследственный | б) масштабируемый |
| в) инкапсулированный | г) полиморфный |

6. Объекты разных классов общаются друг с другом посредством...

а) наследование б) полиморфизм

в) сообщения г) обязанности

7. Какое ключевое слово Java используется для указания наследования?

а) расширяется б) общественный в) реализует г) наследует

8. Какой класс является корнем всех классов Java?

а) Интерфейс б) Класс

в) Объект г) Сбор

9. Как можно вызвать класс, производный от другого класса?

а) суперкласс б) вложенный

в) подкласс г) клонируемый

10. Какие из представленных ниже языков являются обязательными?

а) Лисп

до нашей арки

в) Ява

г) Паскаль

ПРОБЛЕМЫ

1. Инкапсуляция в ООП используется для сокрытия данных от пользователя и обеспечения изменения изменяемых состояний. Представьте себе класс Person с частным полем возраста. Подумайте, как сделать так, чтобы невозможно было установить возраст меньше 0 или больше 90.

2. В изученном разделе мы утверждали, что ООП напоминает способ решения задач в реальной жизни. В доказательство мы привели простую иллюстрацию того, как мы решаем бытовую задачу – заказываем маме на день рождения торт с надписью «С Днем Рождения, мама!». Определите объекты (или агенты) в этой системе и их обязанности.

3. Когда мы говорили о наследовании, мы продемонстрировали это на примере Mammal и Cat (обратите внимание, что Mammal сам по себе может быть дочерним классом Animal). Укажите аналогичные иерархические отношения, но не в животном мире, а в любой понравившейся вам организации (хорошим выбором может стать КБТУ).

2 Introduction to Java programming language

«Большинство хороших программистов занимаются программированием не потому, что ожидают, что им заплатят или получат лесть от публики, а потому, что программировать — это удовольствие» .
Линус Торвальдс

«Java — это самое неприятное событие, произошедшее с компьютерами со времен MS-DOS» .

Алан Кей

«Если вы научитесь программировать на java, вы никогда не останетесь без работы!»
Патрисия Сейболд

Т его глава призвана представить некоторые основные понятия и методы, используемые при проектировании и разработке программ Java, включая фундаментальный синтаксис Java и его использование.

В настоящее время технология Java используется для разработки приложений для широкого спектра сред: от потребительских устройств до гетерогенных корпоративных систем (3). В предыдущем разделе мы уже описали парадигму программирования, используемую Java. В этом разделе вы изучите синтаксис Java, с которым вы, скорее всего, столкнетесь профессионально, а также стандартные идиомы программирования Java, которые можно использовать для создания надежных приложений.

Помните, что программирование — это не просто вопрос набора кода. Вместо этого он включает в себя значительный объем планирования и тщательного проектирования. Плохо спроектированная программа вряд ли когда-нибудь будет работать корректно. Поэтому помните, что тщательное проектирование программы должно предшествовать написанию кода. Это особенно верно для объектно-ориентированных программ.

Другая распространенная ошибка, которую вы можете допустить, — это ввести весь код программы, а затем скомпилировать и запустить его. Чаще всего это приводит к десяткам ошибок, исправление которых может оказаться трудным и отнять много времени. Решением этой проблемы является использование принципа поэтапного уточнения . Например, вы можете написать ⁵ код для одного метода или какой-то небольшой логической части и протестировать этот фрагмент кода, прежде чем переходить к другой части программы. Благодаря этому небольшие ошибки будут обнаружены, прежде чем перейти к

⁵ Поэтапная доработка означает, что программа пишется небольшими этапами, и после каждого этапа код компилируется и тестируется.

Объектно-ориентированное программирование и дизайн

следующий этап. Есть известная пословица: вовремя сделанный стежок спасает девять человек. Это означает, что своевременные усилия предотвратят дальнейшую работу в дальнейшем.

В связи с тем, что изначально Java играл роль языка программирования микропроцессоров, встроенных в бытовую технику (5), в него был разработан ряд интересных особенностей, описанных в подразделе 2.1.

2.1 Почему Java? (функции Java)

Java — относительно молодой объектно-ориентированный язык программирования общего назначения. Интересно отметить, что первоначально он был назван «Дуб» в честь дерева возле офиса его разработчика Джеймса Гослинга.

Впоследствии его переименовали в «Зеленый», а затем, наконец, в «Ява» от кофе «Ява», который, как говорят, в больших количествах потребляют создатели Явы (6).

Первоначально он был разработан компанией Sun Microsystems в 1991 году как язык для встраивания программ в электронные потребительские устройства, такие как микроволновые печи и домашние системы безопасности (4). Тем не менее, фантастическая популярность Всемирной паутины побудила Sun преобразовать Java в язык для написания встроенных программ в веб-приложения.

Помимо веб-приложений, Java также вызывала огромный интерес в бизнес-сообществах, где она оказалась весьма выгодной с коммерческой точки зрения. Более того, язык Java также является хорошим выбором для написания распределенного программного обеспечения и предоставления услуг сотрудникам и клиентам в частных корпоративных сетях (интранетах) (7).

Теперь давайте рассмотрим некоторые основные особенности Java, которые делают ее чрезвычайно полезный и конкурентоспособный язык:

- Java объектно-ориентирована. Как мы уже узнали в разделе 1, объектно-ориентированные языки делят программы на отдельные модули, называемые объектами, которые инкапсулируют данные и операции программы.
В отличие от языка C++, Java разрабатывался с нуля как объектно-ориентированный язык.
- Java надежен. Это означает, что ошибки в программах Java не вызывают сбоев системы так часто, как ошибки в других языках программирования. Определенные особенности языка позволяют обнаружить множество потенциальных ошибок еще до запуска программы.

Глава 2. Введение в язык программирования Java.

- Java является переносимой (независимой от платформы). Торговая марка Java — «Напиши один раз, запускай где угодно» .6 Это означает, что программу Java можно запускать без изменений практически на любой платформе (например, Windows и Linux). Это не относится к другим языкам программирования высокого уровня.
- Java надежен. Эта функция поддерживается расширенной проверкой ошибок во время компиляции и выполнения (например, проверка байт-кода). Кроме того, здесь нет указателей, а есть настоящие массивы. Поэтому повреждения памяти или несанкционированный доступ к памяти невозможны. Наконец, Java поддерживает динамическое управление памятью посредством автоматической сборки мусора, которая предназначена для отслеживания использования объектов с течением времени (8).
- Java интерпретируется. На этапе компиляции компилятор Java генерирует байт-коды, а не собственный машинный код. Очевидно, что скомпилированные байт-коды не зависят от платформы. Затем во время выполнения байт-коды Java преобразуются на лету в машиночитаемые инструкции (виртуальная машина Java, обсуждаемая далее в подразделе 2.2).
- Java динамичен. Java спроектирован так, чтобы адаптироваться к развивающейся среде. В частности, библиотеки могут свободно добавлять новые методы и переменные. Более того, интерфейсы обеспечивают гибкость и возможность повторного использования кода, определяя набор методов, которые может выполнять объект, без указания того, как эти методы должны быть реализованы.
- Java имеет богатый API. Java поставляется с обширной коллекцией богатых библиотек кода — программного обеспечения, которое было разработано для использования непосредственно для определенных типов приложений.
- Java безопасен. Java, предназначенная для использования в сетевых средах, содержит функции, защищающие от ненадежного кода — кода, который может внести вирус или каким-либо образом повредить вашу систему. В частности, устанавливаются принудительные ограничения доступа (частный, публичный) и проверяются границы доступа к массиву (в отличие от C++).
- Java является многопоточной. Обычно это означает, что несколько параллельных потоков выполнения могут выполняться одновременно.
- Java — это просто. Языковые конструкции интуитивно понятны и просты. Кроме того, библиотеки Java легко доступны и расширяемы.

Несмотря на список привлекательных функций Java, представленный выше, вероятно, лучшая причина выбора Java — это ее способность приносить удовольствие и

6

Программисты шутят по этому принципу: «Java — напиши один раз, беги» .

желание научиться программировать, особенно в объектно-ориентированном стиле.

Простота конструкции Java делает такие достижения доступными для большинства начинающих программистов (4).

2.2 Виртуальная машина Java (JVM)

JVM — это часть программного обеспечения, написанная специально для конкретной платформы. Среда выполнения Java (JRE) включает JVM, библиотеки кода и компоненты, необходимые для запуска программ, написанных на языке Java.

Во-первых, программный код Java компилируется и преобразуется в промежуточное представление, называемое байт-кодом Java (т. е. последовательность нулей и единиц), а не непосредственно в машинный код, специфичный для платформы.

Инструкции байт-кода предназначены для интерпретации JVM, написанной специально для аппаратного обеспечения хоста. Добавляя этот уровень абстракции, компилятор Java отличается от компиляторов других языков, которые записывают инструкции, подходящие для набора микросхем ЦП, на котором будет работать программа. Таким образом, во время выполнения JVM читает и интерпретирует файлы .class и выполняет инструкции программы на собственной аппаратной платформе, для которой была написана JVM (3).

JVM — это сердце принципа языка Java «напиши один раз, работай где угодно» . Таким образом, ваш код может работать на любом наборе микросхем, для которого доступна подходящая реализация JVM (7). JVM доступны для основных платформ, таких как Linux и Windows.

Основным преимуществом использования байт-кода является переносимость (обсуждалась ранее). Это гарантирует, что программы, написанные на Java, будут одинаково работать на любой поддерживаемой платформе ОС. Тем не менее, интерпретируемые программы почти всегда работают медленнее, чем программы, скомпилированные в собственные исполняемые файлы. Использование виртуальной машины происходит медленнее, чем компиляция в соответствии с собственными инструкциями. Практически, JIT-компиляторы (JIT-компиляторы) преобразуют байт-код Java в машинный язык во время выполнения. Кроме того, безопасность замедляет работу (все операции доступа к массиву требуют проверки границ, многие операции ввода-вывода требуют проверки безопасности). По этим причинам программы, написанные на Java, имеют репутацию более медленных, чем программы, написанные на C++.

Глава 2. Введение в язык программирования Java.

2.3 Компиляция и запуск программы Java

Теперь пришло время взглянуть на то, как скомпилировать и запустить программу Java (показано на рисунке 1). Для этих целей есть 2 команды — `javac` и `java`:

- `javac` — компилятор Java, он читает исходный код и генерирует байт-код, как обсуждалось выше. Обычно мы пишем исходный код в файлах `.java`, а затем компилируем их. Компилятор проверяет код на соответствие синтаксическим правилам языка, а затем записывает байт-коды в файлы `.class`.
- `java` — интерпретатор Java, он фактически выполняет байт-код. Помните, что вы указываете интерпретатору запустить класс, а не файл!

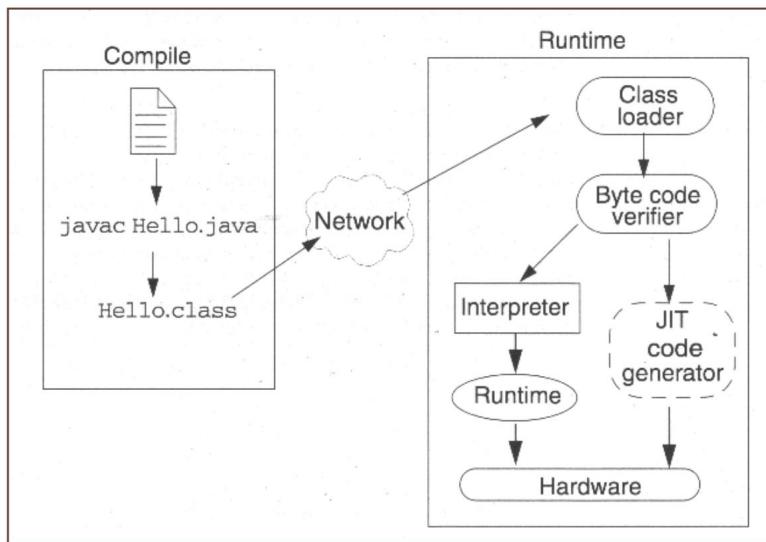


Рисунок 1. Операции компиляции и выполнения

Следует подчеркнуть, что Eclipse IDE, которую вам советуют использовать в этом курсе, выполняет компиляцию автоматически — каждый раз, когда вы сохраняете программу (по нажатию `Ctrl+S`) или добавляете какой-либо код. Напротив, запуск программы необходимо выполнять вручную.

Важно подчеркнуть, что когда ваша Java-программа создает экземпляр объекта во время выполнения, JVM автоматически выделяет динамическую память.

для этого объекта (4). Более того, сборщик мусора Java⁷, работающий в фоновом режиме, отслеживает объекты, которые больше не нужны приложению, и освобождает от них память. Этот подход к управлению памятью называется неявным или динамическим управлением памятью, поскольку он не требует от программиста написания кода обработки памяти. Сбор мусора — одна из важнейших функций производительности платформы Java (5).

2.4 Синтаксис Java

В этом разделе мы познакомим вас с некоторыми ключевыми элементами синтаксиса Java, описав детали небольшой программы. Мы рассмотрим, как организована программа и что делают ее различные части. Наша главная цель — представить основные элементы языка, многие из которых будут объяснены более подробно в последующих разделах.

Структурно язык Java начинается с пакетов. Пакет — это механизм пространства имен языка Java. Внутри пакетов находятся классы, а внутри классов — методы, переменные, константы и т. д.

В нашем примере мы рассмотрим Java-версию традиционной программы Hello World — «традиционной» , поскольку практически каждый вводный текст по программированию начинается с нее. При запуске программа Hello World просто отображает на консоли приветствие «Hello World!» .

```
public class HelloWorld { // заголовок класса
    частное строковое приветствие = «Привет, мир» ;
    public void Greeting(){ // определение метода
        /* оператор
         * вывода*/
        System.out.println(приветствие);
    }
    public static void main(String[] args) {
        // объявляем и создаем объект HelloWorld
        hw = new HelloWorld(); хв.приветствие(); //
        вызов метода
    }
}
```

⁷ Сбор мусора — это процесс автоматического поиска фрагментов памяти, которые больше не используются программой («мусор») и снова делаются доступными. В отличие от ручного освобождения, которое используется во многих языках (например, C и C++) Java автоматизирует этот подверженный ошибкам процесс (6).

Глава 2. Введение в язык программирования Java.

Обратите внимание: когда вы сохраняете объявление публичного класса в файле, имя файла должно быть именем класса, за которым следует расширение «.java» . Поэтому для нашего приложения имя файла — HelloWorld.java.

Кроме того, `System.out` известен как стандартный объект вывода. Этот метод просто печатает переданный текст, не перемещая курсор вывода на новую строку.

Теперь давайте обсудим некоторые базовые, но все же важные конструкции Java на основе приведенного выше примера (`HelloWorld.java`).

- **Комментарии**

Первое, что вы наверняка заметили в программе `HelloWorld`, — это использование комментариев. Как известно, комментарий — это неисполнимая часть программы, используемая в основном для ее документирования (3). Итак, поскольку комментарии не являются исполняемыми инструкциями, они просто игнорируются компилятором. Их единственная цель — облегчить разработчику чтение и понимание программы.

Программа `HelloWorld` содержит примеры двух типов комментариев Java:

1. Многострочный комментарий. Любой текст, содержащийся в `/*` и `*/`, считается комментарием. Как вы можете видеть в `HelloWorld`, комментарии такого типа могут занимать несколько строк.

2. Однострочный комментарий. Второй тип комментария — это любой текст, который следует за двойной косой чертой — `//` || в строке. Этот тип комментария не может быть расширен за пределы одной строки (4). Однострочный комментарий должен содержаться в одной строке, хотя для формирования блока можно использовать соседние однострочные комментарии.

Хорошим стилем является включение краткого описания в начало программы, чтобы объяснить, что делает программа, ее ключевые функции, поддерживающие структуры данных и любые уникальные методы, которые она использует (8).

- **Ключевые слова**

Как и любой язык программирования, язык Java содержит определенные слова — так называемые **ключевые слова** — которые компилятор распознает как специальные, и поэтому вам не разрешается использовать их для именования ваших собственных переменных.

Язык Java содержит 48 предопределенных ключевых слов, представленных в Таблице 2 ниже (5). Это слова, которые имеют особое значение в языке и использование которых ограничено. Например, ключевые слова, используемые в нашем `HelloWorld` программы: класс, частная, общедоступная, статическая и недействительная.

Объектно-ориентированное программирование и дизайн

abstract	default	goto	package	this
boolean	do	if	private	throw
break	double	implements	protected	throws
byte	enum	import	public	transient
case	elses	instanceof	return	try
catch	extend	int	short	void
char	final	interface	static	volatile
class	finally	long	super	while
const	float	native	switch	
continue	for	new	synchronized	

Таблица 2. Ключевые слова Java

Очевидно, что, поскольку их использование ограничено, ключевые слова не могут использоваться в качестве имена методов, переменных или классов.

- Соглашения об именах

В дополнение к правилам синтаксиса, регулирующим соглашения , У Java есть определенные об именах идентификаторов при создании имен классов, переменных и методов.

Это помогает унифицировать стиль программирования и легко определить, является ли объект методом, классом или переменной (2).

Имена в Java чувствительны к регистру. Таким образом, два разных идентификатора могут содержать одни и те же буквы в одном и том же порядке. В качестве иллюстрации: currentValue и CurrentValue — это два разных идентификатора.

Основные соглашения об именах Java обобщены и представлены в Таблице 3 ниже.

	Правило	Примеры
Имена классов	Начинайте с заглавной буквы каждого слова в имени	Привет, мир ArrayList Человек
Имена переменных и методов	Начните со строчной буквы, но также используйте заглавные буквы, чтобы различать слова в имени.	получитьИмя() УстановитьСообщение() привет() мойСчетчик радиус
Константы	Напишите все буквы с заглавной буквы	ПИ ШИРИНА МАКС

Таблица 3. Соглашения об именах в Java

Глава 2. Введение в язык программирования Java.

Глядя на пример программы `HelloWorld`, мы видим, что все переменные, методы и имена классов соответствуют соглашениям об именах.

Как уже упоминалось, существенным преимуществом этих соглашений является то, что различные элементы программы — классы, методы, переменные — легко отличить по тому, как они написаны. Соблюдение этих рекомендаций обеспечит более доступность вашего кода для других разработчиков, следующих тем же соглашениям (6).

Еще одно важное соглашение, о котором вам всегда нужно помнить, — это выбирать осмысленные и описательные имена. Даже если вы полагаетесь на свою память и уверены, что через месяц сможете вспомнить, что переменная, которую вы назвали `var1`, хранит количество книг в библиотеке, а `var2` хранит количество разделов в библиотеке, например, это крайне плохой стиль программирования. В реальной жизни разработчики работают в командах, и им постоянно необходимо сотрудничать друг с другом. Итак, ваш код становится общим и может потребоваться его модификация другому программисту, который наверняка не будет обладать столь необыкновенной интуицией, чтобы догадаться, что вы имели в виду под `var1` и `var2`.

- Переменные

Как известно, переменная — это место в памяти, где можно сохранить значение для дальнейшего использования в программе. Очевидно, что переменные должны быть объявлены с указанием имени и типа, прежде чем их можно будет использовать.

Оператор объявления — это оператор, который объявляет переменную определенного типа. В свою очередь, оператор присваивания (или инициализации) — это оператор, который сохраняет (присваивает) значение переменной. Оператор объявления переменной указывает имя и тип переменной, например:

Строковое приветствие;

число интервалов;

Оператор присваивания переменной инициализирует переменную специальным значением, например:

`приветствие = "Привет, мир";`

`количество = 883;`

В нашей программе `HelloWorld` мы объединили объявление и присваивание в один оператор, например:

`Строковое приветствие = «Привет, мир» ;`

Наконец, чтобы объявить постоянную, используйте ключевое слово `Final`:

итоговый двойной PI = 3,14159;

- Примитивные и ссылочные типы.

В Java используются два основных типа данных, а именно примитивные типы и ссылочные типы.

Примитивные типы — это логический тип и числовые типы. Числовые типы — это целочисленные типы — byte, short, int, long и char, а также типы с плавающей запятой — float и double. Ссылочные типы — это классы, типы интерфейсов и типы массивов. Существует также специальный нулевой тип.

```
класс Point { int[] metrics; }

интерфейс Переместить {
    void move(int deltax, int deltay);
}
```

Значения примитивных типов хранятся непосредственно в стеке, а не в куче, как это обычно бывает с объектами. Это было сознательное решение разработчиков Java из соображений производительности (1). Значение переменной примитивного типа можно изменить только с помощью операций присваивания этой переменной. ВСЕГДА используйте оператор двойного равенства (`==`) для сравнения на равенство.

Как уже упоминалось, ссылочные значения представляют собой указатели на объекты массива, интерфейса или класса, а также специальную нулевую ссылку, которая не ссылается ни на один объект. Поскольку переменная ссылочного типа хранит в стеке только указатель, а само значение хранится в куче, изменить его значение с помощью оператора невозможно. `==` Поэтому, чтобы сравнить два объекта на равенство, используйте метод `equals()` (обсуждаемый в следующем разделе).

В таблице 4 ниже обобщена ключевая информация о примитивах и ссылочных типах.

	Значение примитивных типов	Типы ссылок
Переменная содержит		ссылка (указатель)
Хранится на	стек	куча
Инициализация	0, false, <code>_0'</code>	нулевой
Назначение	копирует значение	копирует ссылку

Таблица 4. Примитивные и ссылочные типы

Глава 2. Введение в язык программирования Java.

Важным вопросом, на который следует обратить внимание, является операция присваивания для ссылочных типов. Поскольку мы храним в стеке только указатель на ссылочную переменную, а не само значение, когда вы пытаетесь присвоить одну ссылочную переменную другой, вы фактически копируете этот адрес (указатель), а не значение. Это означает, что теперь у вас есть две переменные, указывающие на одно место в памяти. Следовательно, когда вы меняете одну из них, значение другой переменной тоже изменится. Давайте подумаем об этом на реальном примере. Предположим, у вас с другом есть общий телевизор и два пульта к нему. Итак, в этом примере указатель похож на пульт дистанционного управления, а значение — на телевизор. Очевидно, что когда ваш друг переключает канал, вы оба видите изменения, происходящие по телевизору, и наоборот.

На рисунке 2 ниже вы можете увидеть иллюстрацию операции присваивания для переменных примитивного (слева) и ссылочного (справа) типа.

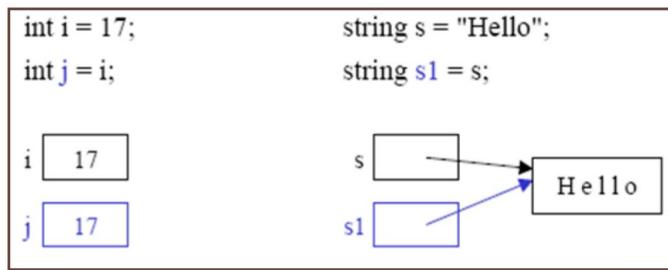


Рисунок 2. Операция присваивания для примитивных и ссылочных типов

- Методы

Мы знаем, что метод представляет собой набор операторов, сгруппированных для выполнения операции. Давайте посмотрим на конкретный пример реализации метода в Java:

```

public static int max(int num1, int num2)
{
    если (число1 > число2)
        вернуть число1;
    еще
        вернуть номер2;
}

```

Приведенный выше метод возвращает максимум два предоставленных значения.

Рисунок 3 ниже может помочь вам пересмотреть известную вам информацию о методах и определить основные части метода.

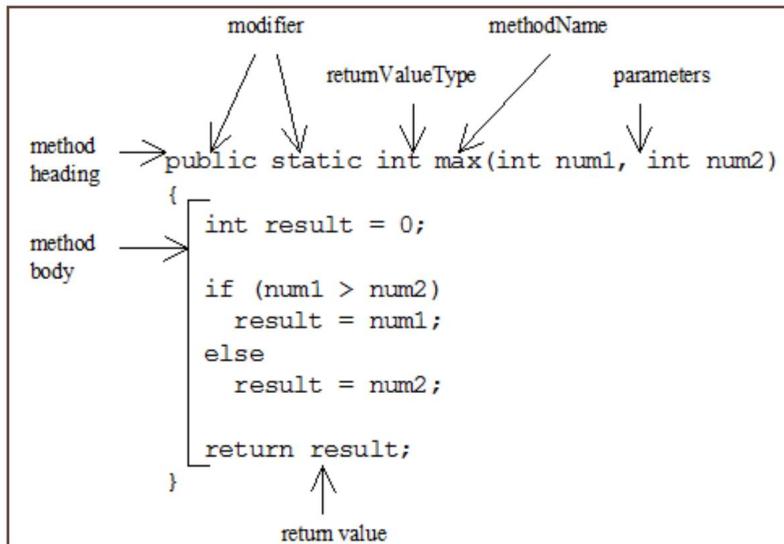


Рисунок 3. Структура метода

Java поддерживает перегрузку методов и может различать методы с разными сигнатурами. Следовательно, в одном классе можно определить два или более метода с одинаковым именем, если объявления их параметров различны. Тем не менее, возвращаемое значение не рассматривается как часть сигнатуры метода (6). Это означает, что вы не можете перегрузить метод, изменив только возвращаемое значение. Так, например, следующий код не сможет скомпилироваться, поскольку у нас есть дублирующийся метод Greeting (те же сигнатуры):

```

общественное недействительное приветствие () {
    System.out.println("Привет!");
}

```

```

публичная строка приветствие() {
    вернуть «Привет!» ;
}

```

Теперь давайте перегрузим метод max, определенный ранее. Определенный метод max принимает два целых числа и находит максимальное из них. Но что, если мы

Глава 2. Введение в язык программирования Java.

хотите найти максимум два дубля? Для этой цели начинающие программисты определяют новые методы с такими именами, как max2, maxx, _max и т. д.

Но если мы воспользуемся концепцией перегрузки, мы сможем написать напрямую: двойной максимум (двойной номер1, двойной номер2)

```
{
    если (число1 > число2)
        вернуть номер1;
    еще
        вернуть номер2;
}
```

Наконец, важно обратить внимание на концепцию абстракции метода. На самом деле вы можете думать о теле метода как о черном ящике, содержащем подробную реализацию метода (3). Схема для этого представлена на рисунке 4 ниже. В качестве простой иллюстрации: в нашей реальной жизни вам не нужно знать, как устроен телевизор, чтобы его смотреть. Вы просто знаете о входах (номера разных каналов) и выходе (канал на телевизоре), это все, что вам нужно иметь при себе. Все остальное для вас черный ящик. Точно так же любой метод из Java API имеет входные параметры и выходное значение, и для его использования вам не обязательно знать, как он реализован.

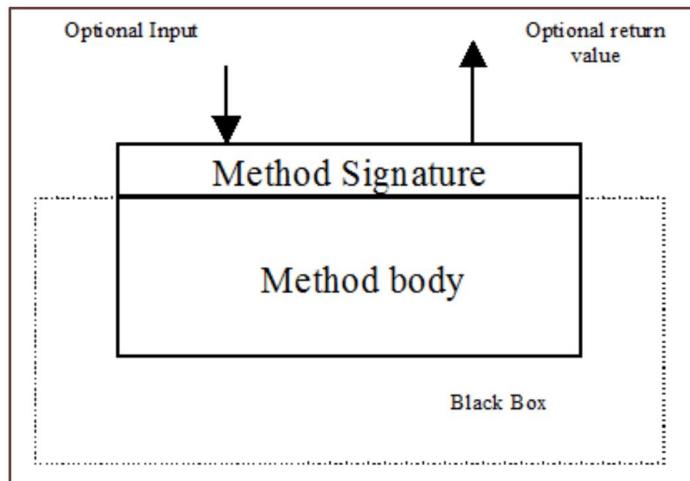


Рисунок 4. Абстракция метода

Наконец, прежде чем перейти к классам и объектам, давайте рассмотрим еще одну простую программу на Java.

Объектно-ориентированное программирование и дизайн

Представленная ниже программа считывает два числа из командной строки и выводит их сумму. Чтобы получить ввод от пользователя, нам нужно создать объект (т.е. экземпляр) класса Scanner, ссылающийся на System.in. Чтобы использовать этот класс, нам нужно его импортировать (первая строка). Вы также можете импортировать класс Scanner вместе с другими классами из пакета java.util, вам просто нужно заменить Scanner на «*», эта звездочка означает «все». Кстати, Scanner тоже может ссылаться на файл.

Кроме того, мы используем метод nextInt() для чтения введенных целых чисел. В конце концов мы суммируем целые числа и выводим результат на консоль. Вы можете проверить Java API на наличие других методов, которые есть в классе Scanner.

```
импортируйте java.util.Scanner;
общественный класс Дополнение {
    public static void main(String[] args) {
        // создаем сканер для получения ввода с консоли
        Сканер в = новый сканер(System.in);
        интервал число1;
        интервал число2;
        целая сумма;
        // запросить у
        пользователя System.out.println("Введите первое целое число");
        число1 = in.nextInt(); // читаем первый номер
        System.out.println("Введите второе целое число");
        число2 = in.nextInt(); // читаем второе число
        сумма = число1+номер2; // добавляем числа
        System.out.println("Сумма: "+sum);
    }
}
```

Эту программу можно написать в сильно сжатом виде. Давайте посмотрим на один из них:

```
общественный класс Дополнение {
    public static void main(String[] args) {
        Сканер в = новый сканер(System.in);
        System.out.println("Введите два целых числа");
        System.out.println("Сумма: "+
                           (in.nextInt()+in.nextInt()));
    }
}
```

- Java API

Глава 2. Введение в язык программирования Java.

Интерфейс программирования приложений Java (API) — (также называемый Javadoc) — официальная онлайн-документация, на которую постоянно ссылается большинство разработчиков Java. Большим преимуществом Java является богатый набор предопределенных классов, которые программисты могут использовать повторно, а не «изобретать велосипед». В нашем примере (дополнительная программа) мы используем предопределенный класс Java Scanner из пакета java.util.

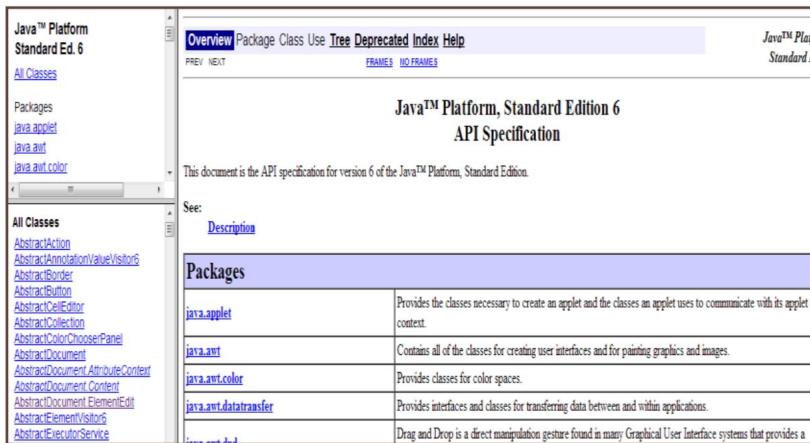


Рисунок 5. Сайт Java API

Теперь посмотрим на структуру сайта документации (см. рисунок 5).

(9). По умолчанию в Javadoc вы можете увидеть три кадра. В верхнем левом фрейме показаны все пакеты в документации, а под каждым пакетом указаны классы. Итак, когда вы выбираете какой-то пакет, там отображаются его классы.

В основном кадре (расположенном справа) отображаются сведения о выбранном в данный момент пакете или классе. Например, если вы выберете пакет java.util в верхнем левом фрейме, а затем выберите класс LinkedList, указанный под ним, в правом фрейме вы увидите подробную информацию о LinkedList, включая описание того, что он делает, как используйте его и его методы (3). Веб-версию этой документации можно найти по адресу

<http://docs.oracle.com/javase/6/docs/api/>

Кроме того, вы можете загрузить эту документацию на свой компьютер, чтобы иметь к ней доступ, когда вы не в сети.

Как мы узнали из этой главы, Java, как и любой язык программирования, имеет свою собственную структуру, правила синтаксиса и парадигму программирования.

Объектно-ориентированное программирование и дизайн

Как мы знаем, парадигма программирования языка Java основана на концепции объектно-ориентированного программирования (ООП).

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Постепенное усовершенствование	Надежность	Переносимость JVM
Байт-код соглашений об именах		Интерпретация
API	Затмение IDE	Примитивный тип
Перегрузка	Методы	Переменные
	Ссылки	Сканер

ИСПЫТАНИЯ

1. Какие из следующих слов являются зарезервированными в Java?

1. бегать
 2. импортировать
 3. функция
 4. реализует
- а) 1 и 2 б) 2 и 3 в) 3 и 4 г) 2 и 4

2. Каков будет результат?

```
String s1 = новая строка («Привет» );  
String s2 = новая строка («Привет» );  
if (s1 == s2) System.out.println("True");  
еще System.out.println("False");
```

- правда б) Ложь
- в) Верно Неверно г) Код не скомпилируется.
3. Каково будет значение переменной с?

Глава 2. Введение в язык программирования Java.

```
интервал a=8;  
интервал б=3;  
интервал с=0;  
в = ++б+а+б++;
```

а) 11

б) 15

в) 16

г) 0

4. Угадайте, каким будет результат работы программы.

```
для (int i=0; i<5; i++)  
    если (я == 2) сломать;  
    еще System.out.println(i);
```

а) 0 1 2 3 4 5

б) 0 1 3 4 5

в) 0 1 3 4

г) 0 1

5. Что будет результатом выполнения следующей программы?

```
Пример публичного класса {  
    статическая строка s;  
    public static void main(String[] args) {  
        System.out.println(">>" + s + "<<");  
    }  
}
```

а) >>ноль<<

б) >><<

в) >>строка s<< г) >>s<<

6. Приложения Java обычно компилируются в байт-код, который может выполняться на:

а) Среда выполнения Java

б) ЯДК

в) Виртуальная машина Java

г) любая программа.

7. Что из перечисленного относится к динамическому управлению памятью?

а) JVM

б) JIT-компилятор

в) Операции ввода/вывода

г) сбор мусора

8. Java-компилятор генерирует

а) машинный код

б) код компилятора

в) байт-код

г) код ошибки

9. Что такое переменная?

- а) компоненты, необходимые для запуска программ, написанных на языке Java
- б) место в памяти компьютера, где значение может быть сохранено для последующего использования в программе.
- в) указатели на некоторые объекты
- г) экземпляр класса или массив.

10. Как сравнить две переменные ссылочного типа на равенство?

- а) метод сравнения()
- б) метод равенства()
- в) с помощью оператора `==||`
- г) с использованием оператора `==||`

11. Выберите правильный вариант перегрузки методов:

- а) `int имя_метода(int)` , `float имя_метода (целое)`
- б) `int имя_метода (двойной x, двойной у), (int x)` `int имя_метода`
- в) Метод `Sring1` (строка а) , `int метод1 (целое число s)`

12. Каков результат следующего кода:

```
интервал x=0;  
переключатель(x)  
{  
случай 1: System.out.println("Один");  
случай 0: System.out.print («Ноль» );  
случай 2: System.out.println («Два» );  
}
```

- а) один б) ноль
- в) Два
- г) НольДва

13. В каком порядке увеличивается видимость модификаторов Java?

- а) частный, пакетный, защищенный и общедоступный.
- б) частный, защищенный, пакетный и общедоступный.
- в) пакет: частный, защищенный и общедоступный.
- г) пакет, защищенный, частный и общедоступный.

14. Классы можно сгруппировать в коллекцию под названием:

Глава 2. Введение в язык программирования Java.

группа

б) пакет

в) сбор

г) папка

15. Выберите правильный импорт всех классов из пакета java.io:

а) импортировать java.io.*;

б) импортировать java.io*;

в) импортировать java.io.all;

г) включить java.io.*;

ПРОБЛЕМЫ

1. Какие преимущества вы видите в независимости от платформы?

2. В нашей короткой версии программы Addition мы написали:

```
System.out.println("Сумма: "+(in.nextInt()+in.nextInt()));
```

Будет ли программа работать корректно, если вместо нее напишем:

```
System.out.println("Сумма: "+in.nextInt()+in.nextInt());
```

Если нет, то почему? Что будет, например, для чисел 7 и 8?

3. Угадайте, каким будет результат работы программы.

```
для (int i=0; i<5; i++)
    если (я == 2) продолжить;
    еще System.out.println(i);
```

Как насчет этого:

```
интервал я = 0;
в то время как (я <5) {
    если (я == 2) продолжить;
    еще System.out.println(i++);
}
```

4. Имеется массив из многих элементов, но собственно данные хранятся в первых n элементах, после которых идут только нули. Как найти

Объектно-ориентированное программирование и дизайн

последний элемент? Напишите специальный метод, выполняющий эту работу.
Тщательно продумайте входные и выходные параметры

5. Написать метод переворачивания элементов массива без использования каких-либо специальных методов из Java API и дополнительной памяти?

6. Рассмотрим 3 метода ниже.

Какие пары из трех методов представляют собой перегрузку метода?
Приведите свой правдоподобный пример.

A:

```
public void setPrice(double newPrice){  
    цена = новая цена;  
}
```

B:

```
public void setPrice (Карандаш p){  
    цена = p.getPrice();  
}
```

C:

```
общественный двойной setPrice (Карандаш p) {  
    цена = p.getPrice();  
}
```

7. Реализуйте метод, который возвращает строку, состоящую из заглавных букв, из исходной строки. Например, для строки «HeLlo, ALiCe» возвращается «HLLALC». Вам не разрешено использовать метод Character.isUpperCase() или регулярные выражения.

8. Разработайте метод, суммирующий все цифры многозначного целого числа.
Используйте только базовую арифметику Java. Не используйте строковые методы.

9. Напишите метод, который преобразует данное двоичное число (в строковой переменной) в целое число. Не используйте для этого какие-либо специальные методы преобразования.

10. Реализуйте свой собственный метод Split(), который разбивает введенную строку на массив строк по некоторым разделителям.

Глава 2. Введение в язык программирования Java.

11. Создайте метод `CalculDuulates`, который подсчитывает, сколько раз вы встречаете элемент в массиве, и возвращает массив пар с элементом и его количеством.

Пример:

[1,1,2,3,1,2] -> [[1,3], [2,2], [3,1]]

12. Реализуйте метод `NumberOfEvenNumbers`, который подсчитывает, сколько раз вы встречаете четный элемент в массиве, и возвращает массив пар с четным элементом и его счетчиком.

Пример:

[4,6,1,1,2,3,1,2,2,2,4,4] -> [[0,0], [2,4], [4,3], [6,1]]

13. Создайте программу, которая вычисляет площадь, периметр и длину диагонали квадрата со стороной a, которые ваша программа должна считывать из пользовательского ввода, используя класс `Scanner`.

14. Напишите программу для поиска корней квадратного уравнения. Используйте сканер, чтобы получить параметры a, b, c из пользовательского ввода. Не забудьте показать сообщение об ошибке, если D отрицательное. Более того, убедитесь, что некоторые вычисления не выполняются дважды, что приводит к потере времени выполнения.

15. Напишите программу, которая отображает оценку (A, A-, B+...) в соответствии с числом, которое пользователь вводит на экране консоли. Какие условные операторы использовать, решать вам. (Используйте систему оценок КБТУ!).

3 Fundamentals of Objects and Classes

Вон представил базовую терминологию и концепции объектно-ориентированного подхода. Программированию в главе 1. Позже в главе 2 мы рассмотрели основные принципы программирования на Java. Теперь вы хорошо подготовлены к подробному изучению того, что такое объект, как объекты группируются в классы, как классы связаны друг с другом и как объекты используют сообщения для взаимодействия и связи друг с другом. Более того, вы узнаете, как использовать предопределенные классы из Java API и как писать свои собственные классы, создавать объекты и добавлять к ним поведение.

3.1 Что такое класс?

Стандартная библиотека Java предоставляет несколько тысяч классов для таких разнообразных целей, как дизайн пользовательского интерфейса, даты и календари, а также сетевое программирование (2). Тем не менее, для описания объектов проблемных областей ваших приложений все равно придется создавать свои классы.

В объектно-ориентированном мире возможно иметь множество объектов одного типа, имеющих общие характеристики. Например, студенты, прямоугольники, сотрудники, книги и т. д. Класс — это программный проект таких объектов, который используется для создания объектов. Он определяет переменные и методы, общие для всех объектов определенного типа.

Итак, класс может иметь два типа членов: поля и методы. Поля — это переменные данных, которые определяют состояние класса или объекта. В свою очередь методы представляют собой исполняемый код класса, построенный из операторов. Они позволяют нам изменять состояние объекта или получать доступ к значению элемента данных. Итак, переменные отражают состояние объекта, а методы определяют поведение объекта.

После создания класса вы можете создать любое количество объектов из этого класса. Класс — это своего рода фабрика по конструированию объектов (1).

Когда вы создаете объект из класса, это означает, что вы создали экземпляр класса.

В качестве простой иллюстрации рассмотрим два примера — классы, представляющие запись телефонного справочника и карандаш. Это может быть определено так:

Глава 3. Основы объектов и классов

```
Запись публичного класса {
    частное имя строки; // имя в виде символов
    частный строковый номер; //номер телефона
}
```

Действительно, все телефонные записи содержат имя и соответствующий номер. Эти переменные были отражены в классе Entry. Более того, класс Pensil.java может выглядеть так:

```
класс Карандаш {
    public String color = «красный» ;
    публичная длина int;
    общественный двойной диаметр;
    public void setColor (String newColor){
        цвет = новыйЦвет;
    }
}
```

Используя приведенный выше пример «Карандаш» , давайте посмотрим, как необходимо объявлять поля, например, поле, представляющее цвет Карандаш:

```
public String color = «красный» ;
```

Итак, ясно видно, что для определения поля вы сначала указываете модификатор доступа, затем имя типа, а затем имя поля. Обратите внимание, что объявлениюм полей могут предшествовать разные модификаторы, а именно:

- модификаторы контроля доступа (публичный, частный, пакетный, защищенный)
- статический
-

финальные. Рассмотрим каждый из них подробно.

- Модификаторы доступа

Модификаторы доступа уже были представлены в предыдущих разделах. Тем не менее, давайте посмотрим на сводную таблицу (Y – «Да» , N – «Нет»):

Access Modifiers	Same Class	Same Package	Subclass	Other packages
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Таблица 5. Модификаторы доступа

Из таблицы выше можно сделать вывод, что, например, частные члены доступны только внутри самого класса, тогда как общедоступные видны везде, где доступен класс. В качестве примера модификатора частного доступа давайте расширим наш класс Pencil, определенный ранее, и добавим частную переменную Price:

```
общественный класс Pencil
{ public String color = "красный"; публичная
длина int; общественный
двойной диаметр; частная двойная
цена; общедоступный
статический длинный номерOfPencils = 0; общественный
недействительный setPrice (float newPrice) {цена = newPrice;

}
}
```

Возможно, вы заметили, что помимо цены мы добавили новую переменную NumberOfPencils (обсуждаемую позже). Теперь, имея частную переменную Price, мы не можем изменить ее значение напрямую, используя цену поля:

```
общественный класс CreatePencil {public
static void main (String args []) {Карандаш p1 = новый Карандаш
(); p1.цена = 0,5f;
}
}
```

Этот код приведет к следующей ошибке во время компиляции:

Глава 3. Основы объектов и классов

```
%> javac Карандаш.java
%> javac CreatePencil.java
CreatePencil.java:4: цена имеет частный доступ в Pencil.
    p1.цена = 0,5f;
```

Это происходит из-за уровня защиты, установленного на переменной цене, которая является частной. Проблему можно решить, изменив видимость переменной цены (например, сделав ее общедоступной), что не очень хорошая идея. Другое решение — использовать мутатор — метод setPrice.

```
Карандаш p1 = новый Карандаш();
p1.setPrice(0,5f);
```

Этот метод относится к инкапсуляции (подробнее обсуждается в этой главе).

- Статическое ключевое слово

Эмпирическое правило, которое вам нужно запомнить, заключается в том, что существует только одна копия статического поля, и она используется всеми объектами класса. Доступ к переменной, имеющей модификатор static, можно получить непосредственно в самом классе. Доступу извне класса должно предшествовать имя класса следующим образом:

```
System.out.println(Pencil.numberOfPencils);
```

Обратите внимание, что доступ к нестатическим полям вне класса должен осуществляться через ссылка на объект, а не имя класса.

Подробнее об этом читайте в следующем разделе Члены класса и члены экземпляра.

- Последнее ключевое слово.

Ключевое слово Final указывает на постоянную переменную. После инициализации она значение не может быть изменено. Итак, Final часто используется для определения именованных констант.

Важно понимать, что статические конечные поля должны быть инициализированы при инициализации класса, тогда как нестатические конечные поля должны быть инициализированы при создании объекта класса.

Обратите внимание: если переменная не была инициализирована, то ей присваивается начальное значение по умолчанию в зависимости от ее типа. В частности, переменные числовых типов (byte, short, int, long, double) имеют значение по умолчанию 0, логические — false, ссылка на объект — нулевое значение (5).

3.2 Что такое объект?

Как и в реальном мире, объектом является любая вещь. Объект может быть физическим (например, автобус, часы, банкомат) или ментальным (например, Событие, Идея). Это может быть и естественная вещь, например животное, студент, рабочий и т. д. Например, программа, управляющая банкоматом, будет включать объекты BankAccount и Customer . Шахматная программа, скорее всего, будет включать в себя объект Board и объекты Piece (4).

В мире программирования объект — это экземпляр класса. Любой объект имеет класс, который определяет его данные и поведение.

Возвращаясь к примеру из главы 1 (заказ торта на день рождения мамы), мы видим, что Азиз — это экземпляр категории или класса людей, т. е. Азиз — это экземпляр класса поваров. Термин «плита» обозначает класс или категорию всех плит. Азиз — это объект или экземпляр класса. Мы взаимодействуем с экземплярами класса, но класс определяет поведение экземпляров. Мы можем многое сказать о том, как поведет себя Азиз, поняв, как ведут себя плиты. Это ключевой момент в понимании концепции классов и объектов. Мы знаем, например, что Азиз, как и все кулинары, умеет готовить торты разных видов и размеров.

Теперь давайте установим соответствие между программными объектами и объектами реального мира. Как мы видим, объекты в нашей жизни имеют две общие характеристики: все они имеют состояние и поведение. Например, у собак есть состояние (имя, цвет, порода, голодны они или нет...) и поведение (лай, бег, влияние хвостом...). У студентов есть состояние (имя, студенческий билет, курсы, на которые они зарегистрированы, пол, способ оплаты и т. д.) и поведение (сдавать тесты, посещать курсы, регистрироваться на курсы, писать тесты, участвовать в вечеринках и т. д.).

В ООП мы создаем программные объекты, моделирующие реальные объекты. Объекты программирования моделируются по образцу объектов реального мира в том смысле, что они тоже имеют состояние и поведение. Программный объект сохраняет свое состояние в одной или нескольких переменных (полях) и реализует свое поведение с помощью методов. (некоторая функция, связанная с объектом). Поэтому мы можем сказать, что объект — это программный пакет переменных и связанных с ними методов.

В работающей программе может быть множество экземпляров объекта. Например, может быть много объектов Student . Каждый из этих объектов будет иметь свои собственные переменные экземпляра, и каждый объект может иметь разные значения, хранящиеся в переменных экземпляра. Например, каждый объект Student будет иметь разное число, хранящееся в переменной Id (1).

Думайте об объектах как о независимо плавающих в памяти компьютера. Фактически, существует особая часть памяти, называемая кучей, в которой живут объекты (1). Вместо хранения самого объекта переменная, хранящаяся в стеке, содержит информацию, необходимую для поиска объекта в

Глава 3. Основы объектов и классов

Память. Эта информация называется ссылкой или указателем на объект и представляет собой адрес ячейки памяти, в которой хранится объект. Программа использует ссылку в переменной для поиска фактического объекта.

Важным моментом, который вам следует иметь в виду, является то, что объявление переменной на самом деле не создает объект. Помните, что в Java ни одна переменная не может содержать объект. Переменная может содержать только ссылку на объект. Объекты фактически создаются с помощью оператора new, который создает объект и возвращает ссылку на этот объект.

В качестве простой иллюстрации к описанным выше понятиям предположим, что у нас есть класс Student с полями name (String), test1, test2, test3 (все типа int) и методом расчета среднего балла. Итак, заявление-декларация выглядит так:

```
Студент s1, s2;
```

А теперь давайте создадим сам объект, используя новое ключевое слово:

```
s1 = новый студент()
s2 = новый студент();
```

Вам нужно понимать, что s1 и s2 хранят только ссылку на объект. Приведенные выше инструкции создадут новые объекты, которые являются экземплярами класса Student, и сохранят ссылки на эти объекты в переменных s1 и s2 соответственно. Опять же, имейте в виду, что значение переменной — это ссылка на объект, а не сам объект! Следовательно, когда мы копируем значение одной переменной в другую, мы копируем указатели, а не сами объекты:

```
c2 = c1;
```

Приведенный выше оператор скопирует опорное значение, хранящееся в s1, в переменную c2. Вы также можете сохранить нулевую ссылку в переменной, например:

```
Студент s3 = ноль;
```

Давайте посмотрим, как установить значения некоторых переменных экземпляра:

```
s1.name = "Джон Смит";
s2.name = "Мэри Джонс";
```

Как уже упоминалось, другие переменные экземпляра будут иметь начальные значения по умолчанию, равные нулю, если они не будут установлены вручную. Рисунок 6

демонстрирует, какой будет ситуация в памяти компьютера после того, как компьютер выполнит эти инструкции:

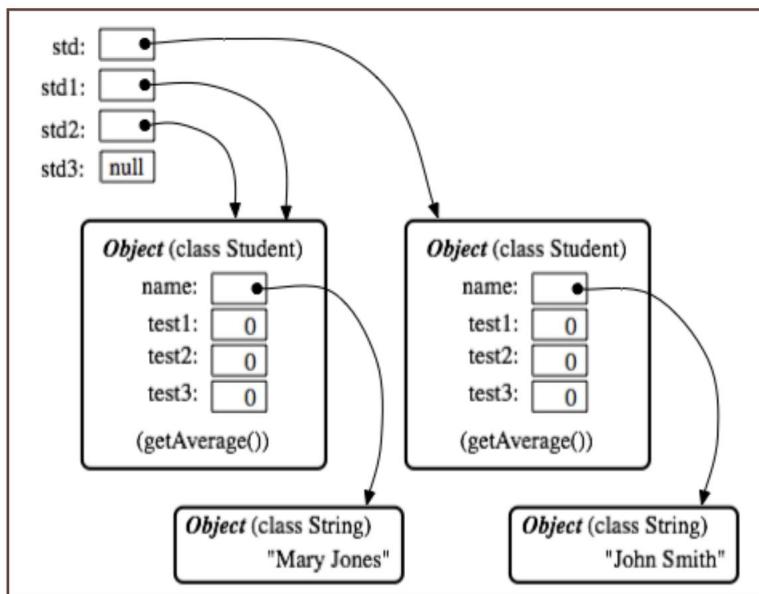


Рисунок 6. Ситуация в памяти компьютера

На рисунке выше мы видим, что когда одна объектная переменная присваивается другой, копируется только ссылка. Указанный объект не копируется. Когда присвоение `s2 = s1;` было выполнено, новый объект не был создан. Вместо этого `s2` был настроен для ссылки на тот же объект, на который ссылается `s1`.

Вы не можете проверять объекты на равенство и неравенство, используя операторы `==` и `!=`. Тест `if (s1 == s2)` проверяет, совпадают ли значения, хранящиеся в `s1` и `s2`. Однако значения являются ссылками на объекты, а не на объекты.

Итак, вы проверяете, являются ли `s1` и `s2` на один и тот же объект, то есть указывают ли они на одно и то же место в памяти. Но объекта нет в переменной, два одинаковых объекта могут храниться в разных местах. Поэтому этот метод не подходит для проверки равенства объектов. Чтобы сравнить реальные объекты, вам необходимо использовать метод `quals()` (описанный далее в этой главе) (1).

Глава 3. Основы объектов и классов

3.3 Модификаторы классов

Класс также может иметь модификаторы. Их описание приведено в таблице 6. ниже:

Модификатор	Описание
общественный	Класс общедоступен. Без этого модификатора класс доступен только внутри собственного пакета.
абстрактный	Никакие объекты абстрактного класса не могут быть созданы. Все его абстрактные методы должны быть реализованы его подклассом. В противном случае этот подкласс также должен быть объявлен абстрактным.
финальный	Не может быть подклассом (наследование будет обсуждаться в следующей главе).

Таблица 6. Модификаторы классов

Обычно файл может содержать несколько классов, но только один общедоступный. Имя файла и имя общедоступного класса должны быть одинаковыми.

3.4 Члены класса и члены экземпляра

Нестатические члены класса (поля и методы) также известны как переменные и методы экземпляра . Кроме того, существуют статические члены, также известные как переменные класса и методы (1).

Каждый объект (экземпляр) класса имеет собственную копию переменных экземпляра, определенных в классе. Итак, когда вы создаете экземпляр класса, система выделяет достаточно памяти для объекта и его переменных экземпляра.

Помимо переменных экземпляра, классы могут объявлять переменные класса. Переменная класса имеет статический модификатор и содержит информацию, которая является общей для всех объектов класса (2). Таким образом, если один объект изменяет переменную, он изменяет все остальные объекты этого типа.

Вы можете вызвать метод класса или получить доступ к полю класса непосредственно из класса, тогда как вы должны вызывать методы экземпляра и получать доступ к переменным экземпляра в конкретном экземпляре. Например, методы в Math классы статичны и могут быть вызваны без создания экземпляра Math . сорт. Итак, мы можем сказать:

```
двойной d = Math.sqrt(x);
```

Глядя на наш класс `Pencil`, мы видим, что есть переменные экземпляра цвета, длины, диаметра, цены и одна статическая переменная `NumberOfPencils`.

Итак, у каждого карандаша будет свой цвет, длина и т. д., а количество карандашей будет общим для всех карандашей. Когда создается новый объект `Pencil`, число `NumberOfPencils` увеличивается.

Как уже упоминалось, существует только одна копия статической переменной, поэтому инициализация этой переменной выполняется только один раз, при первой загрузке класса. Помните, что если вы не указали начальное значение для переменной экземпляра, начальное значение по умолчанию предоставляется автоматически (8) – Переменные экземпляра числового типа (`int`, `double` и т. д.) автоматически инициализируются нулем, если вы не указываете других значений, а логические переменные инициализируются значением `false`.

Наконец, давайте посмотрим на пример того, как можно использовать переменную совместно. Как вы помните, переменная `NumberOfPencils` имеет статический модификатор. Взгляните на следующий код:

```
Карандаш p1 = новый Карандаш();
Карандаш.КоличествоКарандашей++;
System.out.println(p1.numberOfPencils);
//Результат? 1
```

```
Карандаш p2 = новый Карандаш();
Карандаш.КоличествоКарандашей++;
System.out.println(p2.numberOfPencils);
//Результат? 2
```

```
System.out.println(p1.numberOfPencils);
//Результат?      Опять 2!
```

Сначала мы создаем экземпляр `Pencil` и называем его `p1`. Затем, используя ссылку на класс, мы увеличиваем количество карандашей. На данный момент есть 1 карандаш. Кроме того, мы создаем еще один экземпляр `Pencil` – `p2` и снова увеличиваем количество карандашей. На данный момент получение значения `NumberOfPencils` через экземпляр `p2` возвращает 2. Но даже если вы получите доступ к этому значению через `p1`, оно все равно будет 2, поскольку переменная является общей, существует только одна копия. Хотя это вряд ли имеет смысл, вы также можете изменить значение статической переменной, используя ссылку на объект, например:

Глава 3. Основы объектов и классов

```
Карандаш p1 = новый Карандаш();
p1.numberOfPencils++;
```

3.5 Еще о методах

Мы уже рассмотрели концепцию метода, который соответствует действию или поведению объекта. Другими словами, это именованный фрагмент кода, который можно вызвать или вызвать для выполнения определенного заранее определенного набора действий (3). Любой метод имеет заголовок (включает модификаторы доступа, абстрактные или нет, статические или нет, окончательные или нет) и тело (сам код).

Методы вызываются как операции над объектами/классами с помощью оператора точки (.):

ссылка.метод(аргументы)

Помните, что если метод статический, «ссылка» может быть либо именем класса, либо ссылкой на объект, принадлежащий классу. С другой стороны, если метод нестатический, «ссылка» должна быть ссылкой на объект.

Как мы уже знаем, класс может иметь более одного метода с одним и тем же именем, если они имеют разные списки параметров из-за перегрузки методов, которая поддерживается в Java. Итак, давайте перегрузим `setPrice` метод, который раньше:

мы	представил
----	------------

```
public void setPrice (double newPrice) {
    цена = новая цена;
}

public void setPrice (Другой карандаш) {
    цена = другое.getPrice();
}
```

Итак, как видите, у нас есть два метода с одинаковыми именами, но первый принимает переменную типа `double`, а второй — переменную типа `Pencil`. Оба они устанавливают значение переменной цены, но делают это немного по-разному. Первый метод делает это напрямую, тогда как второй устанавливает цену, равную цене другого карандаша. Для лучшего понимания давайте рассмотрим аналогичный пример из реальной жизни. Предположим, у вас есть небольшой магазин, где вы продаете офисные принадлежности. Например, когда вы устанавливаете цену на карандаш, вы можете подумать обо всех сопутствующих расходах и указать некоторую цифру, скажем, 100 тенге (1-й метод в списке кодов). Другой способ установить цену — это когда вы понимаете, что карандаш, которому вы хотите установить цену, им

Объектно-ориентированное программирование и дизайн

примерно такая же стоимость, как и у любого другого карандаша, затраты на который вы уже подсчитали (2-й способ).

Как вы думаете, откуда компилятор узнает, какой метод вы вызываете? Фактически, он делает это путем сравнения количества и типа параметров и использует соответствующий.

Наконец, давайте посмотрим, как передаются параметры метода. Параметры всегда передаются по значению, например:

```
общественный недействительный метод1 (int a) {
    a = 6;

} общественный недействительный метод2 () {
    интервал б = 3;
    метод1 (6); // теперь b = ?
}
```

В результате выполнения значение *b* будет равно 3. Значение *b* не изменяется методом, что эквивалентно:

```
a = 6;
a = 6;
```

Однако, если у вас есть переменная ссылочного типа вместо *b*, это значение будет изменено. Итак, что произойдет после вызова метода2 :

```
public void метод1 (Студент а) {
    а.setName("Арай");

} публичный недействительный метод2 () {
    Студент б = новый студент();
    б.setName("Адия");
    метод1 (б); // теперь s ?
}
```

Значение *b* не изменяется. Но имя Студента *b* изменено, поскольку оно эквивалентно:

```
a = 6;
a.setName("Арай");
```

Важно понимать, что объектная переменная на самом деле не содержит объекта. Оно относится только к объекту. Итак, в нашем случае у нас есть два

Глава 3. Основы объектов и классов

ссылки (a и b), указывающие на одного и того же Студента. Оба они могут одинаково изменять состояние объекта (помните пример с двумя пультами от одного телевизора?). Итак, a.getName() и b.getName() вернут «Array» .

Давайте посмотрим на другой пример, где параметр является ссылкой на объект. Как вы можете видеть на рисунке 7 ниже, у нас есть экземпляр класса Pencil с именем PlainPencil. Мы установили его цвет как простой. В правой части рисунка можно наблюдать, что происходит в памяти на каждом шагу. Итак, согласно картинке, у нас есть одна ссылка (plainPencil) на объект Pencil, цвет которого — обычный. Взгляните на метод PaintRed . Этот метод принимает карандаш и окрашивает его в красный цвет. После этого он делает ссылку p на нулевой (подумайте, почему?). Теперь вернемся к нашему пошаговому анализу кода. Мы вызываем PaintRed, передавая только что созданный простой карандаш.

Вы видите, что в методе пройденный карандаш обозначается как p. Это означает, что теперь у нас есть две переменные — p и PlainPencil , которые указывают на наш объект Pencil. Теперь внутри метода мы используем ссылку p , чтобы раскрасить карандаш. На третьем рисунке справа вы можете видеть, что для обеих ссылок (plainPencil и p) цвет был изменен. Итак, p сработало, оно нам больше не нужно. Итак, мы делаем ссылку p на нулевой.

На самом деле, эти интересные процессы действительно происходят, когда вы таким образом работаете с объектами.

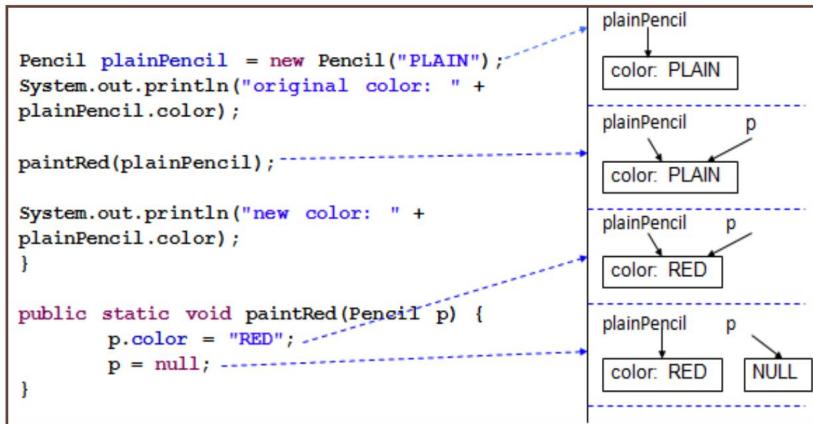


Рисунок 7. Передача объекта в качестве параметра

Следовательно, когда параметр является ссылкой на объект, передается именно ссылка на объект, а не сам объект. Из этого примера вам нужно

Объектно-ориентированное программирование и дизайн

чтобы понять и запомнить это простое правило: если вы измените какое-либо поле объекта, на который ссылается параметр, объект изменится для каждой переменной, содержащей ссылку на этот объект.

Наконец, несколько слов о основном методе. Система находит и запускает основной метод класса при запуске программы (он должен содержать общедоступный класс). Другие методы выполняются при явном или неявном вызове основным методом (3). Помните, что он должен быть публичным, статичным и недействительным.

3.6 Инкапсуляция

Мы уже затронули некоторые основы концепции инкапсуляции. Если вы воспользуетесь этой техникой, вы сможете добиться следующего:

- Поля класса можно сделать доступными только для чтения или только для записи (с помощью методов доступа и мутатора).
- Класс может иметь полный контроль над тем, что хранится в его полях.

Однако пользователи класса не знают, как класс хранит свои данные (6).

Давайте рассмотрим суть понятия инкапсуляция. На диаграмме объекта (см. рисунок 8 ниже) показано, что переменные объекта (круги, начала, треугольники и т. д.) составляют центр или ядро объекта.

Методы окружают и скрывают ядро объекта от других объектов программы. Итак, объекты общаются друг с другом, отправляя друг другу сообщения (с помощью методов). Объекты упаковывают свои переменные в рамках защитной защиты своих методов, и это в точности относится к явлению инкапсуляции (1).

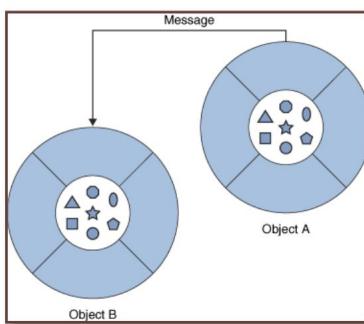


Рисунок 8. Диаграмма объекта

Глава 3. Основы объектов и классов

Таким образом, вы сохраните сокрытие информации. То есть объект имеет общедоступный интерфейс, который другие объекты могут использовать для взаимодействия с ним. Объект может хранить конфиденциальную информацию и методы, которые можно изменить в любое время, не затрагивая другие объекты, которые от него зависят (1). Это возможно, потому что вы меняете реализацию, а не публичный интерфейс.

Многие программисты утверждают, что почти все переменные-члены должны быть объявлены закрытыми. Это позволяет вам контролировать, что можно сделать с переменной. Даже если переменная является частной, вы можете позволить другим объектам узнать ее значение, предоставив общедоступный метод доступа, который возвращает значение переменной. Например, если ваш класс «Сотрудник» содержит частную переменную-член «зарплата» типа double, вы можете предоставить метод доступа get, метод которого возвращает значение зарплаты.

```
public double getSalary() { возврат  
зарплаты; }
```

Вы также можете предоставить метод-мутатор, чтобы сделать зарплату редактируемой. То есть вы можете захотеть предоставить другим классам возможность указывать новое значение для переменной:

```
public void setSalary(двойная зарплата) {  
    this.salary = зарплата;  
}
```

Обратите внимание, что метод set должен иметь параметр того же типа, что и переменная и метод get должны возвращать переменную этого типа.

Как правило, имя метода доступа/мутатора для переменной получается путем написания имени переменной с заглавной буквы и добавления «get» / «set» перед именем. Итак, для переменной зарплаты мы получаем метод доступа с именем «get» + «Salary» или getSalary(). Метод получения обеспечивает «доступ для чтения», тогда как метод установки обеспечивает «доступ для записи» к частной переменной.

По соглашению желательно предоставить как метод доступа, так и метод мутатора (получатель и установщик) для частной переменной-члена.

Возможно, вы захотите задать вопрос: поскольку предоставление методов get/set позволяет другим классам видеть и редактировать значение переменной, почему бы просто не сделать переменную общедоступной? Ответ заключается в том, что геттеры и сеттеры не ограничиваются простым чтением и записью значения переменной. Например, метод получения может отслеживать количество обращений к переменной:

```
общественный двойной getSalary ()
    {salaryAccessCount++;
    возврат зарплаты; }
```

Более того, метод установки может проверить правильность значения, присваиваемого переменной:

```
public void setSalary(двойная зарплата) {
    если(зарплата > 0)
        this.salary = зарплата;
    }
```

Существует концептуальная разница между методами `get` и `set`. Метод `get` только ищет состояние объекта и возвращает его. Метод `set`, напротив, изменяет состояние объекта (2). Итак, они называются методами доступа и мутатора. Давайте определим геттеры и сеттеры для `Entry`:

```
Запись публичного класса {
    частное имя строки; // имя в виде символов
    частный строковый номер; //номер телефона

    public void setName(String person) {
        имя = человек;
    }
    общественная строка getName ()
        {возвращение (имя);
    }
    общественный недействительный setNumber (String
        телефон) {номер = телефон;
    }
    публичная строка getNumber() {
        возврат (число);
    }
}
```

Помните, что инкапсуляция переменных-членов является хорошей практикой программирования.

3.7 Конструкторы

Конструктор позволяет инициализировать объект. Это единственный способ создать экземпляр объекта какого-либо класса в вашей программе.

Каждый класс имеет хотя бы один конструктор. Даже если вы не предоставите любой конструктор в классе, то система предоставит значение по умолчанию без аргументов. ⁸ Конструктор для этого класса. Этот конструктор по умолчанию используется для выделения памяти и инициализации переменных экземпляра (3). Он устанавливает для всех полей, не имеющих инициализации, значения по умолчанию (помните, для числовых типов это 0, для логических – false).

Определение конструктора во многом похоже на определение любого другой метод, с некоторыми отличиями:

1. Конструкторы не имеют возвращаемого типа.
2. Имя конструктора должно совпадать с именем класса. 3. Конструктор может иметь только

модификаторы доступа public, Private и protected (конструктор не может быть объявлен статическим).

(1)

Класс может иметь несколько конструкторов при условии, что у них разные списки параметров (обсуждается позже).

Помните о вступительном классе. Давайте определим конструктор для этого класса, который устанавливает значения для полей имени и номера.

```
Запись публичного класса {
    частное имя строки ; // имя в виде символов
    частный строковый номер; //номер телефона public
    Entry(String person, String phone) { name = person; //
        инициализируем имя
        номер = телефон; // инициализируем номер
    }
}
```

Тогда Entry можно настроить следующим образом:

```
Запись newEntry = новая запись("Айнур", "+77013467222");
```

Помните порядок параметров конструктора при создании объекта с помощью этого конструктора.

⁸ Конструкторы без аргументов называются конструкторами без аргументов.

3.8 Это ключевое слово

Это ключевое слово можно использовать двумя способами: как ссылку на текущий объект и для вызова другого конструктора в том же классе. Давайте посмотрим на эти два способа глубже.

- Это ссылка на текущий объект

Внутри метода и конструктора экземпляра это служит ссылкой на текущий объект (то есть объект, метод или конструктор которого вызывается). Он позволяет ссылаться на любой член текущего объекта из метода или конструктора (9).

Обычно поля имеют те же имена, что и параметры метода или конструктора. В этом случае вы можете использовать это, чтобы различать их.

На самом деле, это наиболее распространенная причина использования этого ключевого слова. Например, класс Point можно записать так:

```
общественный класс Point {  
    интервал x = 0;  
    интервал y = 0;  
    //конструктор  
    public Point(int newX, int newY) {  
        x = новыйX;  
        y = новыйY;  
    }  
}
```

Однако иногда может быть полезно использовать одни и те же имена для параметров. Итак, используя это ключевое слово, мы можем написать класс немного по-другому:

```
общественный класс Point {  
    интервал x = 0;  
    интервал y = 0;  
    //конструктор  
    public Point(int x, int y) {  
        это.x = x;  
        это.y = y;  
    }  
}
```

Это также можно использовать для вызова конструктора внутри другого конструктора.

Глава 3. Основы объектов и классов

- Это как вызов конструктора

Внутри конструктора это служит для вызова другого конструктора в тот же класс. Это относится к явному вызову конструктора.

Рассмотрим следующий класс Circle, который можно использовать для представления двумерного круга. Обратите внимание, что он содержит переменную типа Point (определенную ранее), представляющую центр круга в системе координат:

```
класс Circle { двойной
    радиус;
    Центр точки;
    двойной findArea() {
        вернуть радиус*радиус*Math.PI;
    }
}
```

Настраивать поля по отдельности неудобно, поэтому явные конструкторы и методы полей также будут полезны:

```
публичный круг(){
    радиус = 1,0;
    центр = новая точка (0,0);
}
общественный круг (двойной r) { радиус
    = r;
}
общественный круг (центр точки, двойной r) {
    это (р);
    this.центр = центр;
}
```

Этот класс содержит два конструктора. Каждый конструктор инициализирует некоторые или все переменные-члены круга. Первый задает значение радиуса, второй - радиуса и центра. Чтобы не дублировать код, мы используем его здесь для вызова конструктора, который устанавливает радиус, а затем устанавливает центр.

Как вы могли догадаться, компилятор определяет, какой конструктор использовать. вызов, в зависимости от количества и типа аргументов.

Теперь вы можете создавать круги разными способами в зависимости от имеющейся у вас информации:

```
Круг по умолчанию Круг = новый Круг(); Круг
myCircle = новый Круг (5.0); Круг CoolCircle =
новый круг (новая точка (883,1729), 5.0);
```

По соглашению, если вы используете `this()`, это должен быть первый оператор в теле конструктора, если он существует.

Имейте в виду, что это нельзя использовать в статическом методе (см. вопрос в конце главы).

3.9 Блоки инициализации

Блок инициализации представляет собой блок операторов, используемых для инициализации полей объекта. Его необходимо разместить вне любого объявления члена или конструктора. Блоки инициализации всегда выполняются перед телом конструкторов (7).

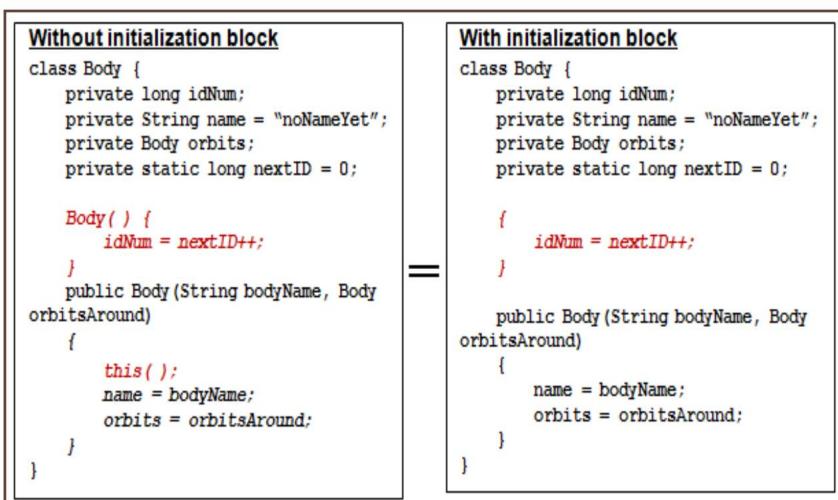


Рисунок 9. Использование блоков инициализации

На рисунке выше вы можете увидеть, как может выглядеть определение класса с блоком инициализации и без него. Когда мы не используем блок инициализации, нам необходимо создать пустой конструктор без аргументов, чтобы выполнить код, который должен выполняться для каждого объекта. Затем в каждом из других оставшихся конструкторах мы используем `this()` для выполнения no-arg.

Глава 3. Основы объектов и классов

конструктор. Недостаток этого подхода в том, что если у вас сто перегруженных конструкторов, вам придется вызывать `this()` для каждого из них.

Теперь взгляните на правую часть рисунка, где мы используем блок инициализации для увеличения идентификатора вновь созданного объекта. Все, что нам нужно сделать, это поместить код, инициализирующий переменные, в блок, окруженный фигурными скобками. Этот код будет выполнен перед телом конструктора для всех объектов.

Кроме того, существует статический блок инициализации. Он похож на нестатический блок инициализации (уже обсуждавшийся), за исключением того, что он объявлен статическим и может ссылаться только на статические члены (8). Он выполняется при первой загрузке класса.

3.10 Импорт классов

Вы уже видели пример того, как импортировать классы, которые вы используете в программе. Любой сложный объект использует другие объекты для выполнения некоторых функций. Оператор импорта сообщает компилятору, где найти классы, которые вы используете внутри своего кода. Оператор импорта должен быть размещен в верхней части исходных файлов.

Чтобы создать оператор импорта, вы указываете ключевое слово импорта, за которым следует класс, который вы хотите импортировать, а затем точка с запятой (5). Имя класса включает его пакет. Итак, оператор импорта обычно выглядит так:

`импортировать ИмяКлассаТоИмпорт;`

Например:

`импортировать java.util.Date;`

Если вы хотите импортировать все классы в пакете, вы можете поставить `.*`⁹ вместо имени класса.

На самом деле есть еще один способ получить доступ к общедоступным классам, расположенным в других пакетах. Помимо импорта одного класса или всех общедоступных классов (*), вы можете явно указать полное имя пакета перед именем класса, как в этом примере:

⁹ Обратите внимание, что * используется для импорта классов на текущем уровне пакета. Таким образом, он не будет импортировать классы в своих подпакетах.

```
java.util.Date сегодня = новый java.util.Date();
```

Есть несколько тонких вопросов, на которые нам необходимо обратить внимание.

Один из них связан с конфликтом имен. Рассмотрим следующий код:

```
импортировать java.util.*;
импортировать java.sql.*;
```

```
Дата сегодня = новая дата(); // ОШИБКА: java.util.Date // или java.sql.Date?
```

У нас возникла ошибка, поскольку оба пакета имеют класс Date. Итак, как решить эту проблему? Если вам нужно сослаться только на один из них, импортируйте этот класс явно, например:

```
импортировать java.util.*;
импортировать java.sql.*;
импортировать java.util.Date;
```

```
Сегодняшняя дата = новая дата(); // java.util.Date
```

Если вам нужны оба класса данных, вам необходимо использовать полное имя пакета перед именем класса, например:

```
импортировать java.util.*;
импортировать java.sql.*;
```

```
java.sql.Date сегодня = новый java.sql.Date();
java.util.Date someOtherDay = новый java.util.Date();
```

Ситуация со статическими членами немного другая, вам нужно обратиться к их как className.memberName, как в примере ниже:

```
импортировать java.lang.Math;

общественный класс importTest {
    двойной x = Math.sqrt(1.44);
}
```

Итак, вы не можете использовать sqrt без имени класса, поскольку это статический метод. Начиная с версии J2SE 5.0, импорт также можно применять к статическим полям и методам, а не только к классам (б). Вы можете напрямую обратиться к ним после статического импорта. В качестве иллюстрации давайте импортируем все статические поля и методы класса Math:

Глава 3. Основы объектов и классов

```
импортировать статический java.lang.Math.*;  
двойной x = ПИ;
```

Кроме того, вы можете импортировать определенное поле или метод:

```
импортировать статический java.lang.Math.abs;  
двойной x = abs (-1,0);
```

Это экономит ваше время, если вы часто используете методы и поля таких пакетов.

3.11 Абстракция классов

Абстракция класса означает изолировать реализацию класса от его использования. Другими словами, разработчик класса предоставляет описание класса, чтобы пользователь знал, как этот класс можно использовать. Следовательно, чтобы использовать класс, пользователю не обязательно знать, как этот класс реализован. Детали реализации инкапсулированы и скрыты от пользователя. Так, например, если вы хотите написать класс «Сотрудник» и протестировать его, настоятельно рекомендуется поместить их, например, в разные классы «Сотрудник.java» (реализация, а не «публичный класс») и «СотрудникТестер.java» (открытый класс).

3.12 Перечисления

Перечисление — это тип, имеющий фиксированный набор возможных значений, указанный при создании перечисления. Определение типов перечисления имеет следующую форму:

```
enum enum-name { значение1, значение2, ...};
```

Например, следующее перечисление может использоваться для представления времена года:

```
enum Season { ВЕЧА, ЛЕТО, ОСЕНЬ, ЗИМА };
```

Определение перечисления не может находиться внутри метода. Вы можете положить его снаружи метод main() программы.

Еще примеры, когда может быть полезно использовать перечисления:

```
enum Day { ПОНЕДЕЛЬНИК, ВТОРНИК, СРЕДА, ЧЕТВЕРГ,
    ПЯТНИЦА, СУББОТА };
перечисление Пол { МУЖЧИНА, ЖЕНЩИНА};
размер перечисления { XS, S, M, L, XL, XXL};
```

Вы должны понимать, что значения перечисления не являются переменными. На самом деле каждое значение перечисления — это константа (помните, что имена констант пишутся заглавными буквами!), которая всегда имеет одно и то же значение. Вы можете называть их Season.SPRING, Season.SUMMER и т. д.

Вы можете создавать объекты перечисления точно так же, как и для других типов. Например, вы можете создать следующие переменные из перечисленных выше перечислений:

```
Пол g = Пол.МУЖЧИНА;
День d = День.СУББОТА;
Сезон s = Season.SPRING;
```

Вы можете распечатать значение перечисления с помощью обычного System.out.println(). оператор, а выходное значение будет именем константы перечисления .

Наконец, давайте рассмотрим некоторые методы, которые можно использовать при работе с перечислениями. Одним из полезных методов является ordinal(). Он просто возвращает позицию значения в списке (начиная с 0). То есть Season.SUMMER.ordinal() — это целое значение 1. Кроме того, существует метод Values(). метод, возвращающий список, содержащий все константы, составляющие перечисление (1).

3.13 Классы-оболочки

Иногда возникает необходимость манипулировать примитивными типами, как если бы они были объектами. В этом случае вы можете использовать один из классов-оболочек, предоставляемых Java API, или определить свой собственный. В результате вы сможете создавать объекты, представляющие значения примитивного типа. Например, Java API содержит классы Double, Integer, Character, Boolean,... (которые оборачивают одиночный double, int, char, Boolean,...).

Эти классы содержат различные статические методы, включая Integer.parseInt(), который используется для преобразования строк в числовые значения. содержит такие значения, как Integer.MIN_VALUE и Integer.MAX_VALUE, которые равны наибольшему и наименьшему возможным значениям целого числа. Это 2147483648 и 2147483647.

Глава 3. Основы объектов и классов

соответственно. Подобные методы предоставляются другими классами-оболочками, представляющими числовые типы.

Для создания объекта классов-оболочек используется тот же синтаксис, что и для обычных классов:

Целое число = новое целое число (0);

Итак, значение count содержит ту же информацию, что и значение типа int, но является объектом. Если вы хотите получить значение int , заключенное в объект, вы можете вызвать метод count.intValue(). Аналогичным образом вы помещаете двойное значение в объект типа Double, логическое значение в объект типа Boolean и т. д.

3.14 Базовые классы Java

Вы познакомились с понятием класса. Классы могут быть сгруппированы в коллекцию под названием package. Стандартная библиотека Java состоит из иерархических пакетов, таких как java.lang и java.util (7). Давайте рассмотрим некоторые полезные пакеты, которые вы можете часто использовать при программировании на Java.

Пакет	Описание
java.lang	Содержит основные классы Java, такие как числовые классы, строки и объекты. Этот пакет по умолчанию импортируется в каждую программу Java.
java.io	Содержит классы для входных и выходных потоков и файлов.
java.util	Содержит множество утилит, таких как дата и время, токенизатор, случайные классы, устаревшие коллекции.
java.net	Классы для поддержки сетевых коммуникаций.

Таблица 7. Базовые пакеты

Теперь посмотрим, как можно использовать некоторые классы из этих пакетов. Однако здесь мы приводим лишь краткий обзор использования. Более подробную информацию см. в разделе Java API.

- Григорианский календарь

Этот класс предоставляет стандартный календарь, используемый в большинстве стран мира.

Объектно-ориентированное программирование и дизайн

(9). Он имеет несколько полезных конструкторов. Выражение new GregorianCalendar() создает новый объект, который представляет дату и время создания объекта.

Вы можете создать объект календаря для полуночи определенной даты, предоставив данные за соответствующий год, месяц и день, например:

```
День рождения GregorianCalendar = новый
GregorianCalendar(2011, 06, 15);
```

Интересно отметить, что месяцы отсчитываются с 0. Следовательно, 06 – это июль. Чтобы избежать путаницы, вы можете использовать константы, например Calendar.JULY. Использование класса Calendar позволяет вам установить время внутри конструктора:

```
День рождения по григорианскому календарю = новый
Григорианский календарь(2011, Календарь.ИЮЛЬ, 15, 14, 21, 59);
```

GregorianCalendar предоставляет очень полезные методы. Например, вы можете получить месяц, день недели и т. д. какой-либо даты. Код ниже делает это для даты. текущий

```
aGregorianCalendar now = новый GregorianCalendar();
int месяц = now.get(Calendar.MONTH);
int день недели = now.get(Calendar.DAY_OF_WEEK);
```

Проверьте API, чтобы найти список всех констант, которые вы можете использовать. Чтобы внести изменения в состояние даты, используйте метод set:

```
Birthday.set(Календарь.ГОД, 2011);
день рождения.set(Календарь.МЕСЯЦ, Календарь.СЕНТЯБРЬ);
Birthday.set(Calendar.DAY_OF_MONTH, 25);
// или за один вызов
Birthday.set(2011, Календарь.СЕНТЯБРЬ, 25);
```

При работе с датами вам может потребоваться добавить к ним определенное количество дней, недель, месяцев и т. д. Вы даже можете добавить отрицательное число. В этом случае календарь переместится назад (8):

Глава 3. Основы объектов и классов

```
myDate.add(Календарь.МЕСЯЦ, 4);
myDate.add(Календарь.МЕСЯЦ, 2);
```

Ознакомьтесь с API, чтобы изучить дополнительные функции GregorianCalendar.

- Стока

Класс String используется для представления строк символов. Строки в Java неизменяемы, что означает, что их значения не могут быть изменены после их создания. Если вам нужны динамические строки, вы можете использовать StringBuffer(9) или StringBuilder.

В класс String включены следующие основные методы:

- charAt() — для проверки отдельных символов строки.
- CompareTo() — для сравнения строк по алфавиту.
- substring() — для извлечения подстрок
- toLowerCase(), toUpperCase() — для создания копии строки, в которой все символы переведены в верхний или нижний регистр.
- length() — возвращает количество символов, содержащихся в строке.
- Split() — разделяет строку по некоторому разделителю.

Дополнительные методы можно найти на сайте Java API. Небольшой пример некоторых строковых методов представлены ниже.

```
for(int i=0; i<s.length(); i++){
    for(int j=i+1;j<=s.length();j++){
        System.out.println(s.substring(i,j).toUpperCase());
    }
}
```

В этом примере мы печатаем все подстроки заданной строки s, написанные заглавными буквами:

- Вектор

Класс Vector представляет собой расширяемый массив объектов (динамический массив). Подобно массиву, он содержит объекты, к которым можно получить доступ по индексу. Разница в том, что размер Вектора может меняться по мере необходимости в соответствии с добавлением и удалением элементов (9).

```
Студенты Vector<String> = новый Vector<String>();
студенты.add("Эльмира");
студенты.add("Гузаль");
for(String cur: студенты)
    System.out.println(cur);
```

Обратите внимание, что класс Vector может хранить любой объект — String, Integer или любой объект, определенный вами. Например, вы можете хранить объекты Student вместо String.

Vector имеет широкий спектр служебных методов, таких как очистка содержимого, удаление, добавление, вставка элементов, проверка наличия элемента и т. д.

3.15 Определение собственных классов

В этой части главы мы объединим все представленные концепции и создадим два класса: Body и Сотрудники.

- Тело

Рассмотрим версию класса Body, которую можно использовать для представления космический объект:

```
класс Тело {
    частный длинный idNum;
    имя частной строки = «пусто» ;
    частные орбиты тела ;
    частный статический длинный nextID = 0;
}
```

Как видно из приведенного выше листинга, любое космическое Тело имеет идентификатор, имя и ссылку на тело, вокруг которого оно вращается. Все экземпляры Body имеют общую переменную nextID, которая отражает количество тел и каждый раз используется для присвоения idNum .

Помните, что объект создается по ключевому слову new. Только после этого утверждения система выполнения выделит достаточно места для хранения нового объекта. Итак, давайте создадим его. Останавливаться! Чтобы создавать объекты, нам нужны конструкторы. Хотя конструктор без аргументов по умолчанию создается неявно, давайте определим другие конструкторы, которые инициализируют переменную Body:

Глава 3. Основы объектов и классов

```

Body( )
    { idNum = nextID++;
    }

Body(имя строки, орбиты тела) { this(); это.имя
    = имя;
    this.orbits = орбиты;
}

}

```

Теперь мы можем создать пару объектов Body:

```

Body sun = new Body("Сол", null);
Тело земля = новое Тело("Земля", солнце);

```

Значения переменных Солнца и Земли показаны на рисунке 8 ниже.



Рисунок 10. Ситуация в памяти

Возможно, вам захочется сохранить все объекты тела в какой-нибудь коллекции, например Vector. Для этого вам нужно создать статический объект Vector и добавлять каждое вновь созданное тело в этот вектор:

```
частные статические тела Vector<Body> = new Vector<Body>();
```

```
частная пустота addBody()
    { body.add(this);
}
```

Теперь давайте определим методы get/set для полей экземпляра, которые являются закрытыми. Обратите внимание: если бы мы определили их как общедоступные, они могли бы быть изменены любым пользователем. Но теперь их приватность не позволяет получить доступ к полям. Чтобы разрешить доступ к полям, давайте реализуем gettes:

```

public long getID() {return idNum;} public String
getName() {возвращенное имя;} public Body getOrbits()
{обратные орбиты;}

```

После этого куска кода поля idNum, name и орбиты стали

Объектно-ориентированное программирование и дизайн

только для чтения вне класса. Но мы по-прежнему не можем изменить значения полей. Итак, давайте сделаем это, предоставив методы набора мутаторов:

```
public void setName (String newName) {
    имя = новое имя;
}
public void setOrbits(Body OrbitsAround) {
    орбиты = орбиты вокруг;
}
```

Теперь пришло время взглянуть на другой класс — Сотрудник.

- Сотрудник

Теперь давайте рассмотрим следующий пример — упрощенную версию класса «Сотрудник» :

```
класс Сотрудник {
    частное имя строки ;
    частная двойная зарплата;
    частная дата HireDay;
    публичный сотрудник (имя строки) {
        это.имя = имя;
        GregorianCalendar now = новый
            GregorianCalendar();
        this.hireDay = now.getTime();
    }
    общественный сотрудник (строковое имя, двойная зарплата) {
        это имя);
        this.salary = зарплата;
    }
}
```

Из приведенного выше листинга кода мы видим, что есть 3 частных поля экземпляра: имя, зарплата и дата приема на работу сотрудника. Мы также определили два конструктора: один принимает один параметр типа String (представляющий имя) и инициализирует имя. Кроме того, он устанавливает дату найма как текущую дату. Действительно, создание объекта сотрудника должно произойти при его приеме на работу. Еще у нас есть второй конструктор, который помимо имени принимает переменную типа double для установки зарплаты. Помимо зарплаты, нам нужно, указать имя и дату приема на работу. Как видно, мы используем это ключевое слово для вызова первых конструкторов, в которых мы уже проделали эту работу.

Кроме того, нам нужно определить методы get/set для частных полей.

Надеюсь, вы уже можете это сделать, поэтому давайте сделаем это для одного поля — зарплаты:

Глава 3. Основы объектов и классов

```
общественный двойной getSalary () {
    возврат зарплаты;
}

public void setSalary(двойная зарплата){
    this.salary = зарплата;
}
```

Обратите внимание: нам не нужно определять метод `set` для `name` — это нелогично. Сделайте его доступным только для чтения, предоставив только метод `get`. Сделав это, вы получите гарантию, что это поле никогда не будет повреждено.

Если какой-то сотрудник какое-то время работает идеально, шеф-повар может захотеть повысить ему зарплату. Давайте определим метод для этого:

```
public void raiseSalary(двойной процент){
    двойное повышение = зарплата *    процент / 100;
    зарплата += повышение;
}
```

Теперь рассмотрим метод «равно» , который используется для сравнения двух сотрудников:

```
public boolean равно(Другой сотрудник){
    вернуть имя.равно(другое.имя);
}
```

Этот метод утверждает, что два сотрудника одинаковы, если у них одинаковое имя. Типичное использование выглядит так:

```
если (zilan.equals(валет)){}
```

Мы реализовали сырую версию нашего класса «Сотрудник» . Теперь пришло время определить для него тестовый класс. Прежде всего, давайте создадим хранилище (вектор) для наших сотрудников (назовем его рабочими). Затем создайте несколько экземпляров Сотрудника и сохраните их в векторе:

```
Vector<Employee> работники = новый Vector<Employee>();
Зилан Сотрудника = новый Сотрудник("Зилан", 2000);
Сотрудник Али = новый Сотрудник("Али"); //зарплата
для Али еще не определена
рабочие.add(зилан);
рабочие.add(али);
// мгновенно передаем вновь созданный объект
Workers.add(новый Сотрудник("Мадина", 1500));
```

Давайте будем добрыми и поднимем зарплату каждому сотруднику на 10%. Используйте `foreach` для этой цели:

```
для (Сотрудник e: персонал)
    e.raiseSalary(10);
```

Распечатать информацию о сотрудниках. Проверьте, что зарплаты обновлены:

```
для (Сотрудник e: персонал)
    System.out.println("имя — " + e.getName() +
        ", зарплата равна " + e.getSalary() +
        и HireDay равна " + e.getHireDay());
```

Немного неудобный способ распечатать информацию о сотруднике, не правда ли? В Java у нас есть специальный метод `toString()` класса `Object` (помните, что класс `Object` является родительским для любого класса в Java). Он используется для возврата строкового представления объекта. Если вы не предоставите никакой реализации этого метода, он вернет местоположение объекта в памяти. Давайте определим это:

```
публичная строка toString(){
    вернуть «имя» + имя +
        ", зарплата равна " + зарплата + ", а
        HireDay равна " + HireDay;
}
```

После добавления этого кода в «Сотрудник» мы можем распечатать информацию следующим образом:

```
для (Сотрудник e: персонал)
    System.out.println(e.toString());
```

Или, еще проще, так:

Глава 3. Основы объектов и классов

```
для (Сотрудник e: персонал)  
System.out.println(e);
```

Когда вы печатаете имя переменной, хранящей объект, метод `toString()` вызывается неявно.

3.16 Дизайн классов

Алгоритм эффективного проектирования классов прост. Во-первых, вам необходимо посмотреть на основных действующих лиц и объекты в системе и определить на их основе классы. Во-вторых, опишите атрибуты и методы для каждого из них. Затем, когда у вас уже есть схема классов, попробуйте установить отношения между ними (между классами могут быть различные типы отношений, такие как ассоциация, агрегация, обобщение (наследование), реализация и другие). Наконец, вам нужно реализовать классы.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Перечисление	равно()	Класс-оболочка	получить набор
методы	Вектор	Конструктор	Переменная класса
Переменная экземпляра	Блок инициализации	Импорт	
это ключевое слово	Инкапсуляция	Состояние и поведение объекта	

испытания

1. Компилятор создает конструктор по умолчанию только в том случае, если у класса нет других конструкторов. Правда или ложь?

правда б) Ложь

2. Этот процесс также известен как скрытие информации. Он скрывает функциональные детали класса от объектов, которые ему отправляют.

а) Полиморфизм б) Наследование

в) Инкапсуляция

г) Абстракция

3. Он содержит абстрактные характеристики объекта, включая его состояния и поведение. Также относится к шаблону для создания объектов.

а) Экземпляр

б) Класс

в) API

г) Запись

4. Член класса, определяющий поведение объекта:

а) поле

б) функция

в) интерфейс

г) метод

5. Каков тип возвращаемого значения конструктора?

а) ноль

б) значение по умолчанию 0

в) определяемый пользователем

г) нет возвращаемого типа

6. Предположим, у вас есть перечисление Season { WINTER, SPRING, SUMMER, AUTUMN }.

Как мы можем ссылаться на перечисление:

а) Сезон.ЗИМА

б) сезонный отпуск; отпуск.enum();

в) Сезонный отпуск; отпуск = Сезон.ЗИМА;

г) а) и в) верны

7. Что верно в отношении этого ключевого слова?

а) Может использоваться во всех методах

б) Должен быть 1 оператор в теле конструктора, если он существует.

в) Может использоваться как ссылка на текущий объект.

г) а) б) в) верны

д) б) в) верны

8. Что верно в этом утверждении:

Собака-животное = новое Животное(«Шурик»);

а) Собака — объект класса Animal.

б) Класс Animal имеет только один конструктор с параметром типа String;

в) Класс Animal может иметь множество конструкторов, один из них имеет один параметр типа String.

Глава 3. Основы объектов и классов

г) а) и в) верны

9. Выберите правильный метод:

а) public void setName(String newName) {возвращаемое имя;}

б) публичная строка getName() {возвращенное имя;}

в) публичная строка getName() {имя=новое имя;}

10. Какое утверждение о предметах верно?

а) Один объект используется для создания одного класса

б) Один класс используется для создания одного объекта

в) Один объект может создать множество классов

г) Один класс может создавать множество объектов

11. Предположим, у вас есть метод setValue , который присваивает значение CoolName полю экземпляра с именем name . Что вы могли бы написать внутри setValue ?

а) имя = крутое имя б) this.name = CoolName;

в) CoolName == имя г) а и б верны

12. Учитывая объявление Rectangle r = new Rectange(), какое из следующих утверждений является наиболее точным?

а) r содержит ссылку на объект Rectangle.

б) Вы можете присвоить int значение r.

в) r содержит целое число.

г) r содержит объект типа Rectangle.

13. Как можно сравнить два объекта класса Студент (s1 и s2)?

а) если (s1 == s2) {...}

б) если (s1.equals(s2)) {...}

в) если (равно(s1, s2)) {...}

14. Выберите класс оболочки:

двойной

б) Двойной

- | | |
|--|----------------------|
| в) Инт | г) Чар |
| 15. Какие из следующих полей могут быть статическими внутри класса «Сотрудник» ? | |
| имя | б) опыт |
| в) зарплата | г) Название компании |

ПРОБЛЕМЫ

1. Попробуйте провести объектно-ориентированный анализ взаимодействия между студентом, библиотекарем и базой данных книг на случай, если студент захочет взять книгу(ы) из библиотеки.
2. Статические конечные поля должны быть инициализированы при инициализации класса, тогда как нестатические конечные поля должны быть инициализированы при создании объекта класса. Объяснить, почему.
3. Подумайте о собаке и реализуйте соответствующий класс. Создайте несколько конструкторов для этого класса.
4. Конструкторы используются для инициализации класса. Если конструктор является приватным, это означает, что он не будет виден за пределами класса. Итак, зачем нам могут понадобиться частные конструкторы?
5. Почему это нельзя использовать в статическом методе?

6. Напишите программу, которая отображает календарь на текущий месяц. Отметьте текущий день звездочкой (*).

Подсказка: как вычислить продолжительность месяца и день недели данного дня?

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19*	20	21	22
23	24	25	26	27	28	29
30	31					

7. Реализуйте собственный класс для обработки дат — Date. Любая дата имеет год, месяц, день, час, минуты и секунды. Предоставьте конструктор без аргументов, который устанавливает текущую дату. Затем создайте

Глава 3. Основы объектов и классов

экземпляр вашего класса, назовите его днем рождения. Затем создайте еще один экземпляр — дедлайн и приравняйте дедлайн к дню рождения. Затем сдвиньте срок на 3 дня. Нарисуйте ситуацию в памяти (покажите ссылки, объекты, их значения и т. д.).

8. Реализуйте класс Student. У студента есть имя, удостоверение личности и год обучения. Предоставьте конструктор с двумя параметрами и создайте методы для доступа к имени, идентификатору и увеличению года обучения.
9. Напишите класс StarTriangle, который можно использовать для создания следующей треугольной фигуры (справа).

[*]

[*][*]

- Ваш класс должен иметь конструктор с параметром width, указывающим количество [*] в последней строке треугольника.
- Более того, должен существовать метод `toString()`, который вычисляет строку, представляющую треугольник, и возвращает строку, состоящую из символов [*] и новой строки.

[*][*][*]

...

Использование класса:

```
StarTriangle small = новый StarTriangle(3);
System.out.println(small.toString());
```

Результат такой же, как на иллюстрации.

10. Напишите класс Data, который вычисляет информацию о наборе данных. ценности. Он должен иметь:

- 3 приватных поля: 2 типа `double` и 1 типа `int` (вам нужно угадать, что хранить в этих полях)
 - Конструктор, создающий пустой набор данных.
 - Метод, добавляющий значение в набор данных.
- Метод, который возвращает среднее значение добавленных данных или 0, если данные не были добавлены.
- Метод, возвращающий наибольший из добавленных данных.

Примечание. Вам не разрешено хранить ВСЕ значения! Не используйте массивы или что-то в этом роде!

Объектно-ориентированное программирование и дизайн

Затем напишите анализатор класса, который использует класс данных, описанный выше, для вычисления среднего и максимального значения набора входных значений.

- Используйте сканер для получения данных от пользователя.
- Процесс получения входных данных от пользователя должен продолжаться до тех пор, пока пользователь вводит «Q» вместо числа.

Например:

Введите номер (Q, чтобы выйти): 10

Введите номер (Q для выхода): 0

Введите номер (Q, чтобы выйти): -1

Введите номер (Q для выхода): Q

Среднее = 3,0

Максимум = 10,0

11. Создайте класс `Interval`, представляющий интервал на оси X. Интервал — это набор точек в диапазоне [слева, справа]. Включите проверку того, что левая конечная точка не превышает правую конечную точку, и метод `intersects()`, чтобы `a.intersects(b)` возвращал `true`, если интервалы `a` и `b` пересекаются, и `false` в противном случае.

Использование класса:

Интервал `i` = новый интервал (3,5);

Помните, что точка также является интервалом.

12. Создайте перечисление `Days` для хранения дней недели. Также создайте класс `Time` с полями часы, минуты. Затем создайте урок класса наличие экземпляра вашего перечисления, экземпляра времени и имени поля (строка). Чтобы мы могли написать:

```
Lesson oop = new Lesson (<oop>, Days.MONDAY, new Time(14,30));
```

13. Спроектируйте и реализуйте класс `Polynomial`, который представляет многочлен с действительными коэффициентами. Коэффициенты полинома следует передавать как параметр массива с типом массива `double` в конструкторе вашего класса. Подсказка: результирующий полином определяется коэффициентами`[0] + коэффициенты[1] * x + коэффициенты[2] * x2 + ... + коэффициенты[n] * xn`

Реализуйте 3 метода:

Глава 3. Основы объектов и классов

- умножить двойное значение на многочлен
 - вернуть первую производную •
- вернуть значение $f(x)$ многочлена для заданного значения x .

14. Вам необходимо написать класс температуры, который имеет два поля: значение температуры (двойное число) и символ шкалы: «С» для Цельсия или «F» для Фаренгейта. Убедитесь, что доступ к этим двум полям возможен только через методы доступа вне класса.

Конструкторы:

Класс должен иметь четыре конструктора:

- по одному для каждого поля экземпляра (принимаем нулевую степень, если значение не указано).
указано и Цельсия, если шкала не указана)
- один с двумя параметрами для двух переменных экземпляра
- Конструктор по умолчанию (установлен на ноль градусов Цельсия).

Методы:

- Два метода возврата температуры: один для возврата градусов по Цельсию, другой для возврата градусов по Фаренгейту. Для преобразования используйте следующие формулы:
градусы Цельсия = $(\text{градусы Фаренгейта} - 32) / 9$
градусы Фаренгейта = $(9(\text{градусы Цельсия})/5) + 32$
- Три метода установки полей: один для установки значения, другой для установки масштаба («F» или «C») и один для установки обеих.
- Метод возврата масштаба.

15. Реализуйте класс Car. Автомобиль имеет определенную топливную экономичность, измеряемую в км/литрах, и определенное количество топлива в бензобаке. В конструкторе указан КПД, начальный уровень топлива равен 0.

- Предоставьте метод Drive(), который имитирует движение автомобиля на определенное расстояние, уменьшая количество бензина в топливном баке.
- Также создайте метод getGasInTank(), возвращающий текущее количество бензина в топливном баке, и метод addGas(), чтобы добавить бензин в топливный бак.

Объектно-ориентированное программирование и дизайн

Примечание. Можно предположить, что метод езды никогда не вызывается на расстоянии, которое потребляет больше, чем доступный газ. Также создайте класс CarTester, который проверяет все методы.

16. Напишите класс Time со следующими полями: час, минута, секунда. Кроме того, вам необходим соответствующий конструктор и метод, которые устанавливают время в соответствии с предоставленными параметрами часа, минуты и секунды (проверьте наличие недопустимых входных данных). Кроме того, вам необходимо иметь два метода преобразования формата времени (универсальный и стандартный) и метод для добавления двух объектов времени. Он может быть как статическим (в этом случае будет 2 параметра типа Time), так и нестатическим методом экземпляра, принимающим один параметр Time.

17. В городе Алматы недалеко от КБТУ живет очень страшный дракон.

Ежедневно ему нужно съесть несколько молодых студентов для запуска. Обычно он похищает их по одному утром и ест во время запуска, поставив их в очередь в камере своей тюрьмы. Но иногда у него возникают проблемы с запуском, потому что пропадают студенты! Он еще не знает, что пара мальчик и девочка (БГ) могут исчезнуть, если встанут вместе именно в таком порядке (БГ), благодаря магии любви. После этого линия становится меньше. Значит, есть вероятность, что на запуск дракона никого не останется!

Вам нужно смоделировать запуск дракона. Это должно иметь:

- Перечисление пола, которое используется для различения мальчиков и девочек.
- Класс Person, содержащий переменную экземпляра типа Gender, метод `toString()` и любые поля, которые вы хотите.
- Основной класс – DragonLaunch, с методами:
 - о `kidnap(Person p)`
 - о `willDragonEatOrNot()`

Например, для линии BBGG запуска не будет, так как сначала исчезнет средняя пара, после чего два угловых станут парой BG и исчезнут таким же образом. Однако линия GBGB оставляет на запуск 2 человека.

Примечание. В основном классе перегрузите метод `будетДраконЕстьИлиНет` принять вереницу мальчиков и девочек. Затем используйте класс Random, чтобы сгенерировать 10 случайных последовательностей, состоящих из мальчиков и девочек. Для каждой последовательности выведите ответ.

Глава 4 – Наследование, полиморфизм и абстрактные классы

4 Inheritance, Polymorphism and Abstract Classes

«Общие положения не решают конкретных случаев» .
Оливер Венделл Холмс

Как мы уже изучали в предыдущей главе, класс представляет собой набор А объектов, имеющие одинаковую структуру и поведение. Класс определяет структуру объектов путем указания переменных, содержащихся в каждом объекте класса, а поведение выражается с помощью методов экземпляра. Это мощная идея. Тем не менее нечто подобное можно сделать в большинстве языков программирования. Прорывная идея в объектно-ориентированном программировании, отличающая его от традиционного императивного программирования. Это способность выражать сходство между объектами, которые частично, но не полностью, разделяют их состояние и поведение. Такое сходство может быть выражено посредством наследования и полиморфизма. Несмотря на то, что фундаментальные идеи ООП достаточно просты и понятны, за ними, к сожалению, стоит множество деталей (1).

4.1. Понятие наследования.

Подкласс и

Суперкласс

Концепция наследования позволяет разработчикам писать код в форме иерархических отношений. Таким образом, совокупность поведений и атрибутов можно инкапсулировать в изолированное тело, известное как объект.

Следовательно, этот объект можно использовать при создании дополнительных объектов путем наследования их от исходного объекта. Точно так же, как лошади наследуют параметры и поведение, присущие млекопитающим и позвоночным животным или даже детям, которые получают выгоду от имущества своих матери и отца, так и объекты могут получать выгоду от наследования и наследуют атрибуты и поведение суперклассов (4).

Еще раз, основная идея наследования заключается в том, что вы можете создавать новые классы (подклассы), построенные на основе существующих классов (суперклассов). Благодаря способу наследования вы можете повторно использовать методы и поля существующего класса, а также добавлять новые методы и поля, чтобы адаптировать новые классы к новым ситуациям.

Подкласс и суперкласс также могут называться базовым классом и производным классом, родительским классом и дочерним классом. Родительский и дочерний классы имеют отношение **IsA**: объект подкласса **IsA(n)** — объект его суперкласса, например, **Student** (подкласс) — это **Person** (суперкласс).

Superclass	Subclass
<pre>public class Person { private String name; public Person () { ← name = "no_name_yet"; } public Person (String initialName) {← this.name = initialName; } public String getName () { return name; } public void setName (String newName) { name = newName; } }</pre>	<pre>public class Student extends Person { private int id; public Student () { super(); id = 0; } public Student (String initialName, int id) { super(initialName); this.id = id; } public int getId () { return id; } public void setId (int newId) { id = newId; } }</pre>

Рисунок 11. Студент (подкласс) — Человек (суперкласс)

Следует признать, что отношения наследования транзитивны: если класс В расширяет класс А, то класс С, расширяющий класс В, также будет наследовать от класса А, который является родительским для своего родителя В.

4.2 Иерархия классов

Каждый класс является расширенным (унаследованным) классом, независимо от того, объявлен он таковым или нет. Если класс не объявлен явно расширяющим какой-либо другой класс, то он неявно расширяет класс **Object** (подробно описанный далее в этом разделе). Итак, иерархия классов предыдущего примера:

Объект

Человек

Студент

Глава 4 – Наследование, полиморфизм и абстрактные классы

Очевидно, что объект расширенного класса содержит два набора полей и методов: те, которые определены локально в расширенном классе, и те, которые унаследованы от суперкласса.

Чтобы указать родительский класс, вам нужно использовать ключевое слово `Extensions`. Примеры приведены ниже:

```
класс Транспорт {...}
класс Автомобиль расширяет Транспорт{...}
класс Audi расширяет автомобиль {...}
класс AudiA6 расширяет возможности Audi {...}
```

```
класс Человек {...}
класс Student расширяет Person {...}
класс Сотрудник расширяет Person {...}
класс KBTUStudent расширяет Student{...}
```

4.3 Специальное ключевое слово `super`

Конструктор расширенного класса может вызвать один из конструкторов суперкласса, используя метод `super`. Если конструктор суперкласса не вызывается явно, то конструктор суперкласса `super()` вызывается автоматически в качестве первого оператора конструктора расширенного класса.

Имейте в виду, что конструкторы не являются методами и НЕ наследуются!

Итак, ключевое слово `super` относится к суперклассу класса, в котором появляется `super`. Это ключевое слово можно использовать двумя способами:

- Чтобы вызвать конструктор суперкласса
- Чтобы вызвать метод суперкласса

На рисунке 11 выше показано использование ключевого слова `super` для вызова конструктора суперкласса. Не путайте `super()` и `this()`. Он используется как ссылка на текущий объект и может использоваться для вызова конструктора внутри другого конструктора того же класса. Итак, это совершенно не связано с суперклассом и сильно отличается от супер.

Причина существования суперкласса заключается в том, что вы можете получить доступ к членам суперкласса, которые скрыты членами подкласса. Например, `super.x` всегда ссылается на переменную экземпляра с именем `x` в суперклассе. На практике это может быть чрезвычайно полезно, если класс содержит переменную экземпляра с тем же именем, что и переменная экземпляра в его суперклассе. В этом случае

Объект фактически будет содержать две переменные с одинаковым именем: одну, определенную как часть самого класса, и одну, определенную в суперклассе. Переменная подкласса не заменяет одноименную переменную в суперклассе, а просто скрывает ее. Таким образом, к переменной, определенной в суперклассе, все еще можно получить доступ, используя ключевое слово `super`. То же самое справедливо и для методов: когда вы пишете метод в подклассе, который имеет ту же сигнатуру, что и суперкласса, метод суперкласса скрыт почти таким же образом.

В общем, вам нужно понимать, что хотя мы и говорим, что метод в подклассе переопределяет метод суперкласса, к нему все равно можно получить доступ (1).

Например, предположим, что в классе `Student`, который расширяет базовый класс `Person`, вы хотите объединить результаты методов по умолчанию и нового метода `toString()`. Метод `toString()` по умолчанию можно вызвать с помощью ключевого слова `super`, как в примере ниже:

```
класс Человек {  
    Строковое имя;  
    . . .  
    публичная строка toString(){  
        вернуть "Я" +имя;  
    }  
}  
класс Student расширяет Person{  
    . . .  
    публичная строка toString(){  
        return super.toString()+"студент КБТУ";  
    }  
}
```

4.4 Этапы создания объекта

При создании объекта память выделяется для всех его полей, для которых изначально установлены значения по умолчанию. Затем следует трехэтапное строительство:

- Вызов конструктора суперкласса
- Инициализируйте поля, используя их инициализаторы и инициализацию блоки
- Выполнить тело конструктора

Конструктор вызванного суперкласса выполняется с использованием того же трехфазного конструктора. Этот процесс выполняется рекурсивно до тех пор, пока объект

Глава 4 – Наследование, полиморфизм и абстрактные классы

класс достигнут. На рисунке 12 вам предоставлен пошаговый журнал событий, связанных с созданием объекта.

<pre>class X { protected int xOri = 1; protected int whichOri; public X() { whichOri = xOri; } }</pre>	<pre>class Y extends X { protected int yOri = 2; public Y() { whichOri = yOri; } }</pre>			
Y objectY = new Y();				
Step	What happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	Object constructor invoked	0	0	0
4	X field initialization	1	0	0
5	X constructor executed	1	0	1
6	Y field initialization	1	2	1
7	Y constructor executed	1	2	2

Рисунок 12. Пример, иллюстрирующий порядок построения

Как мы уже упоминали, подкласс расширяет свойства и методы суперкласса. Вы также можете добавлять новые свойства, добавлять новые методы и переопределять методы суперкласса. Итак, предположим, что у вас есть класс Circle, имеющий свои собственные поля (радиус) и методы (например, findArea()). Класс Цилиндр расширение круга имеет все поля и методы, унаследованные от Circle. Кроме того, у него также есть поле длины, и, помимо методов получения/установки для него, в Cylinder есть метод для определения объема. Поскольку объем является результатом умножения длины и площади круга, мы можем вызвать метод суперкласса (вы можете сделать это с ключевым словом super или без него, если ваш дочерний класс имеет метод с таким же именем), чтобы найти площадь и умножьте его на длину. Рисунок 13 дает более подробное объяснение этому.

Чтобы гарантировать, что все определения всех суперклассов связаны, метод конструктора инициирует и вызывает конструктор своего суперкласса и так далее, пока не будет инициализирован конструктор класса Object. Это называется цепочкой наследования (3).

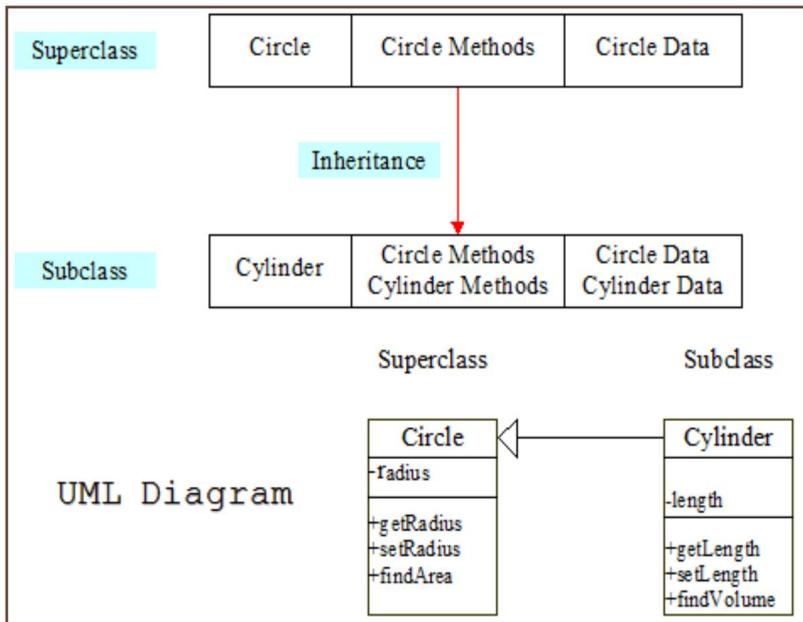


Рисунок 13. Суперкласс (круг) и подкласс (цилиндр)

4.5 Перегрузка и переопределение методов

Как мы уже знаем, под перегрузкой подразумевается предоставление более одного метода с одинаковым именем, но с другим списком параметров. Таким образом, перегрузка унаследованного метода означает простое добавление нового метода с тем же именем и другой сигнатурой.

Напротив, переопределяя , вы заменяете реализацию метода суперкласса своей собственной разработкой. Например, в приведенном выше листинге кода внутри класса `Student` мы переопределили метод `toString()` , определенный в суперклассе. Несколько проблем, которые важны для переопределения:

1. И списки параметров, и типы возвращаемых значений должны быть точно такими же.
2. Если для объекта подкласса вызывается переопределяющий метод, то реализуется версия этого метода подкласса.

Глава 4 – Наследование, полиморфизм и абстрактные классы

3. Переопределяющий метод может иметь спецификатор доступа, отличный от версии его суперкласса, но допускается только более широкий доступ.

Кстати, метод суперкласса можно переопределить, только если он доступен в подклассе. Следовательно, частные методы суперкласса не могут быть переопределены. Таким образом, если подкласс содержит метод, который имеет ту же сигнатуру, что и метод его суперкласса, эти методы совершенно не связаны между собой.

Более того, методы пакета в суперклассе могут быть переопределены, если подкласс находится в том же пакете, что и суперкласс. Защищенный и общедоступный методы всегда можно переопределить.

4.6 Полиморфизм

Понятие полиморфизма используется во многих областях науки, а также в повседневной жизни (физика, экономика, программирование и т. д.). Слово полиморфизм происходит от греческого слова, означающего «много форм». Итак, в общих чертах, полиморфизм означает способность проявляться во многих формах.

Существует два типа полиморфизма: статический и динамический (3). Перегрузка является примером статического полиморфизма и переопределения. Является примером динамического полиморфизма. Перегрузка метода относится к статическому полиморфизму, поскольку она разрешается во время компиляции, а переопределение метода разрешается во время выполнения. Объект данного класса может иметь несколько форм: либо как объявленный тип класса, либо как любой его подкласс. Очевидно, что объект расширенного класса можно использовать везде, где используется исходный класс.

Итак, полиморфизм означает способность во время выполнения определять, какой код запускать, учитывая несколько методов с одинаковым именем, но разными операциями в одном и том же классе или в разных классах. Эта способность также известна как динамическое связывание.

Помните, что когда вы вызываете метод через ссылку на объект, фактический класс объекта решает, какая реализация будет использоваться. Напротив, при доступе к полю объявленный тип ссылки решает, какая реализация будет использоваться во время выполнения. Давайте посмотрим на следующий пример, чтобы прояснить эти утверждения:

```

класс СуперШоу {
    общественная строка ул = «СуперСтрока» ;

    публичное недействительное шоу( ) {
        System.out.println("Super.show:" + str);
    }
}

класс ExtendShow расширяет SuperShow {
    public String str = "ExtendedStr";

    публичное пустотное шоу( ) {
        System.out.println("Extend.show:" + str);}

    public static void main (String[] args) {
        ExtendShow доб = новый ExtendShow ();
        СуперШоу суп = доб.;
        sup.show(); //1
        ext.show(); //2 метода, вызываемые через //ссылку на
        объект
        System.out.println("sup.str =" + sup.str); //3
        System.out.println("ext.str = // доступ к      " + ext.str); //4
        полям 3, 4
    }
}

```

Каков будет вывод приведенного выше кода? Принимая во внимание два правила, упомянутые выше, результат должен быть:

```

Extend.show: ExtendStr
Extend.show: ExtendStr
суп.стр = СуперСила
ext.str = РасширитьСтроку

```

Как видите, когда вы вызываете методы, выполняются те, которые принадлежат реальному классу (для обоих экземпляров, ext и sup, ExtendShow является реальным классом). Однако, когда вы хотите получить доступ к полю, на самом деле осуществляется доступ к полям объявленного класса (для ext объявленный класс — ExtendShow, тогда как для sup — объявленный класс SuperShow).

Хотя существует множество способов определения полиморфизма, и это сильно зависит от контекста использования, есть одна вещь, которую мы можем с уверенностью сказать о полиморфизме: один метод, объект может принимать несколько форм. Так, talk() может принимать форму «Гав» , «Кря» , «Мяу» и т. д.

Глава 4 – Наследование, полиморфизм и абстрактные классы

Поскольку мы знаем, что почти все животные издают тот или иной звук, мы можем определить метод talk() в классе Animal. Затем внутри всех классов, расширяющих Animal (Dog, Cat, Crocodile и т. д.), мы переопределяем определение talk(), чтобы оно соответствовало конкретному животному. Обратите внимание, что в случае, если какое-то животное не издает никаких звуков и вы не предоставляете для него реализацию, вызов talk() все равно будет действительным и приведет к строковому результату «Нет голоса». Это происходит потому, что реализация метода по умолчанию была унаследована от родительского класса.



```
класс Животное {
    публичный недействительный разговор () {
        System.out.println("Нет голоса");
    }
}
класс Dog расширяет Animal{
    публичный недействительный разговор () {
        System.out.println("Гав");
    }
}
```

В приведенном выше примере мы использовали переопределение метода, поскольку подкласс заменяет реализацию родительского метода.

```
тест публичного класса {
    public static void main(String[] args) {
        м (новый A()); // Б
        м (новый B()); //Б
        м (новый C()); //C
        м (новый объект());           //java.lang.Object@192d342
    }
    public static void м(Object o) {
        System.out.println(o.toString());
    }
}
```

```
класс А расширяет В {}
класс В расширяет С {
    публичная строка toString() {           вернуть «Б» ;}
}
класс С расширяет Object {
    публичная строка toString() {           вернуть «С» ;}
}
```

Метод `м` принимает параметр типа `Object`. Итак, вы можете вызвать `м` с любыми объектами (например, `new A()`, `new B()`, `new C()` и `new Object()`). Объект подкласса может использоваться любым кодом, предназначенным для работы с объектом его суперкласса.

Когда выполняется метод `м`, вызывается метод `toString` объекта `o`. Этот `o` может быть экземпляром `A`, `B`, `C` или `Object`. Классы `A`, `B`, `C` и `Object` имеют собственную реализацию метода `toString`. Какая реализация будет использоваться, будет определяться виртуальной машиной Java динамически во время выполнения. Эта возможность известна как динамическая привязка. Рассмотрим другой пример:

Глава 4 – Наследование, полиморфизм и абстрактные классы

```

класс Человек {
    Строковое имя;
    публичный человек (строковое имя) {
        это.имя = имя;
    }
    публичная строка toString(){
        return "Я "+имя+", человек из КБТУ";
    }
}
класс Student расширяет Person{
    общественный студент (имя строки) {
        супер(имя);
    }
    публичная строка toString(){
        return "Я "+имя+", студент КБТУ";
    }
}
класс Сотрудник расширяет Person{
    публичный сотрудник (имя строки) {
        супер(имя);
    }
    публичная строка toString(){
        return "Я "+имя+", сотрудник КБТУ";
    }
}

```

Как можно видеть, и Student, и Worker переопределяют метод `toString()`, определенный в суперклассе Person. Может быть чрезвычайно полезно определить некоторый метод в родительском классе и переопределить этот метод в дочерних классах, предоставив более конкретную реализацию. Это позволяет нам, например, хранить все объекты дочерних классов Person (Студент, Сотрудник, Менеджер (подкласс Сотрудника), Охранник, Декан и т.д.) в одном хранилище (например, списке или массиве) и вызывать этот метод даже не зная фактического типа объекта:

```

Vector<Person> люди = новый Vector<Person>();
люди.add(новый студент("Алия"));
люди.add(новый человек("Айжан"));
люди.add(новый сотрудник("Марал"));
for(Человек p: люди)
    System.out.println(p);

```

Видите ли, векторные люди имеют различные типы объектов, и вы можете использовать оператор `foreach` для перемещения по списку, не вызывая `toString()` отдельно для каждого типа. Обратите внимание, что в этом контексте `p.toString()` эквивалентен `p.toString()`, поскольку

Метод `toString()` используется для описания объекта и обеспечения его строкового представления. Вывод приведенного выше кода будет следующим:

Я Алия, студентка КБТУ.
Я Айжан, человек из КБТУ.
Я Марал, сотрудница КБТУ.

В этом подразделе мы представили идею полиморфизма. Ключевой вопрос, который вы должны усвоить из этого подраздела, заключается в том, что в силу полиморфизма вызов метода, такого как `obj.toString()`, может иметь различное поведение в зависимости от типа объекта `obj`, для которого он вызывается. Итак, основное преимущество полиморфизма заключается в том, что он позволяет обрабатывать несколько объектов разных подклассов.

как объекты одного родительского класса, автоматически выбирая соответствующую реализацию метода для применения к конкретному объекту на основе подкласса, к которому он принадлежит. Это упрощает написание кода и облегчает его понимание другими.

4.7 Преобразование типов

Если вы подумаете о каком-либо подклассе/суперклассе, например, о `Person` или о `работнике` , то легко увидеть, что всегда можно преобразовать подкласс в суперкласс — `сотрудник` , конечно же , тоже является человеком . По этой причине явное приведение типов можно опустить. Например,

Круг `myCircle = myCylinder;`

эквивалентно:

Круг `myCircle = (Circle) myCylinder;`

Тем не менее, при преобразовании объекта из суперкласса в подкласс необходимо использовать явное приведение. Нужно понимать, что этот тип кастинга не всегда может увенчаться успехом.

Цилиндр `myCylinder =(Цилиндр)myCircle;`

Вы уже знаете об иерархии типов. Классы, находящиеся выше в иерархии типов, считаются более широкими или менее конкретными , чем типы, расположенные ниже по иерархии. Точно также о низших типах говорят, что они более узкие или более специфичные.

Глава 4 – Наследование, полиморфизм и абстрактные классы

Расширяющее преобразование означает присвоение подтипа супертипу. Этот тип преобразования можно проверить во время компиляции. Напротив, сужающее преобразование означает преобразование ссылки супертипа в ссылку подтипа. При этом типе преобразования программист должен явно преобразовать объект, используя явный оператор приведения — имя типа в круглых скобках перед выражением. Давайте рассмотрим несколько примеров расширяющей и сужающей конверсии, чтобы сделать различие между ними более ясным.

- Расширяющее преобразование (нет необходимости предоставлять оператор приведения, и это безопасное приведение):

```
Строка str = «тест» ;
Объект obj1 = (Объект)str; //хорошо
Объект obj2 = ул; // хорошо
```

- Сужающее преобразование (должен быть указан оператор приведения, и это небезопасный каст)

```
Строка str1 = «тест» ;
Объект объект = стр1;
Строка str2 = (String)obj; // хорошо
Двойное число = (Double)obj; // нет
```

Вы можете спросить — зачем нам создавать гибридные экземпляры, почему бы просто не создать, например, обычный объект Человека или Студента. Ответ станет ясен после следующего наглядного примера-аналогии из нашей реальной жизни. Какой-то студент просыпается утром, идет на станцию метро и платит за проезд 80 тенге, как и любой человек. В данном контексте нас не волнует, кто этот человек – студент, менеджер или художник. При поступлении в КБТУ ему необходимо предъявить удостоверение личности, начиная с этого момента его заявленный класс становится Студентом. Позже он изучает математику в университете как студент. Это означает, что поведение ученика напрямую связано с ситуацией.

Следующий пример более сложен. Прежде чем двигаться дальше, убедитесь, что вы понимаете, почему в строках 17, 19, 20, 21 возникли определенные ошибки, а в других строках ошибок не возникло. В приведенном примере Student является подклассом Person. Очевидно, что Student, помимо имени поля, унаследованного от родительского Person, имеет поле StudentNumber и соответствующие ему методы get/set.

```

общественный класс Test
{ static Person [] p = новый Person [10]; статический {

    for (int i = 0; i <10; i++) { if(i<5) p[i] = new
        Student();
        еще p[i] = новый Person();

    }

}

public static void main (String args[]) { Person o1 = (Person)p[0];
    Человек o2 = p[0]; Студент o3 = p[0]; //
    строка 17, ошибка комп-
    он Student o4 = (Student)p[0]; Студент o5 = p[9]; // строка 19,
    ошибка комп-он Student o6 =
    (Student)p[9]; // строка 20, ошибка выполнения int x =
    p[0].getStudentNumber(); //ошибка комп-он

}
}

```

Итак, первые 5 объектов в массиве являются гибридными: их заявленный класс — Person, а реальный — Student. Остальные 5 объектов — чистые личности. Теперь давайте посмотрим, что произойдет, если вы попытаетесь скомпилировать приведенный выше код:

```
%> javac typeTest.java
typeTest.java:Найдено 17 несовместимых типов:
Требуется лицо: Студент
Студент o3 = p[0];
```

^

```
typeTest.java:Найдено 19 несовместимых типов:
Требуется лицо: Студент
Студент o5 = p[9];
```

^

```
typeTest.java:21: невозможно разрешить символ символ:
метод getStudentNumber () местоположение: класс
Person
```

```
int x = p[0].getStudentNumber()
```

^

3 ошибки

Глава 4 – Наследование, полиморфизм и абстрактные классы

Почему мы встречаем такие ошибки? По поводу ошибки в строке 17.

Помните, что вы не можете преобразовать суперкласс в подкласс. Объявленный экземпляр `p[0]` — `Person`. Таким образом, вы не можете преобразовать его в экземпляр `Student`. Почему? Потому что фактический класс `p[0]` может быть, например, «Сотрудник». Другими словами, не каждый человек является студентом. Та же история и со строкой 19, поскольку объявленный класс `p[9]` — `Person`, вы не можете преобразовать его в `Student`. Что касается строки 21, вы не можете получить доступ к полю `StudentNumber`, поскольку объявленный тип `p[0]` — `Person`, тогда как `StudentNumber` — это поле класса `Student`. Даже если вы закомментируете эти проблемные три строки, во время компиляции у вас не возникнет проблем, но во время выполнения вы получите следующее: ошибки:

```
%> Исключение java  
typeTest в потоке «main»  
java.lang.ClassCastException: Person  
      в typeTest.main(typeTest.java:20)
```

Чтобы лучше понять, давайте рассмотрим еще один простой пример.

Предположим, есть класс `Food`. Какие виды еды вы знаете? Банан, Курица, Хлеб. Вы абсолютно правы. Все это еда. Можем ли мы сказать, что банан — это еда? Очевидный ответ — да! Можно ли сказать, что любая еда в мире — это банан? Нет конечно! Пример кода ниже продемонстрировал это на Java:

```
Еда еда = новая еда();  
Еда банан = новый банан();  
Еда банан1 = новый банан();  
Банан банан = новая еда();//НЕПРАВИЛЬНО! Ошибка комп-включения  
еда = банан;  
банан1=еда; //НЕПРАВИЛЬНЫЙ! Ошибка комп-включения
```

На самом деле вы можете проверить фактический класс объекта, используя метод `instanceof` оператор:

```

if (объект экземпляра строки)
{
    Стока str2 = (String)obj;
}

Круг myCircle = новый Круг();

if (myCircle экземпляр цилиндра) {

    Цилиндр myCylinder = (Цилиндр) myCircle;
    ...
}

```

4.8 Абстрактные классы и методы

В абстрактных классах могут быть методы, которые только объявлены, но конкретные реализации не предусмотрены. Они должны быть реализованы расширяющими классами. Самый простой пример — класс Person и его подклассы — Сотрудник, Студент, Ребенок, Пенсионер и т. д. В этой ситуации Person является абстрактным классом. Абстрактный метод — это объявление метода без тела; например:

```

абстрактный класс Person {
    Строковое имя;
    //...
    публичная абстрактная строка getDescription();
    //...
}

класс Student расширяет Person {
    частный струнный мажор;
    //...
    публичная строка getDescription() {
        вернуть "студента специальности" + майор;
    }
    //...
}

```

Глава 4 – Наследование, полиморфизм и абстрактные классы

```

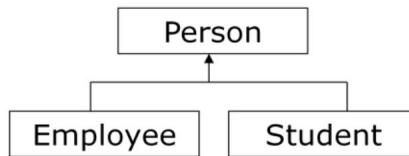
класс Сотрудник расширяет Person {
    частная плавающая зарплата;
    //.

    публичная строка getDescription() {
        вернуть «сотрудник с окладом $» + оклад;

    }
    //.
}

}

```



Основные правила, которые вам следует запомнить:

- Каждый метод, не имеющий реализации в абстрактном классе, должен быть объявлен абстрактным.
- Кроме того, класс, имеющий хотя бы один абстрактный метод, должен быть объявлен абстрактным.
- Если подкласс абстрактного суперкласса не реализует все абстрактные методы своего родителя, то подкласс должен быть объявлен абстрактным. Другими словами, в неабстрактном подклассе, расширенном из абстрактного класса, все абстрактные методы должны иметь реализацию.

Итак, когда вы расширяете абстрактный класс, возможны две ситуации:

- Вы оставляете некоторые или все абстрактные методы неопределенными. Тогда подкласс также должен быть объявлен как абстрактный
- Определить конкретную реализацию всех унаследованных абстрактных методов. Просто в этом случае подкласс уже не является абстрактным.

Главное правило, которое вам нужно усвоить: объект абстрактного класса НЕЛЬЗЯ создать. Тем не менее, по-прежнему разрешено объявлять объектные переменные абстрактного класса, но такая переменная может ссылаться только на объект неабстрактного подкласса, например:

```
Человек p = новый студент();
```

Теперь давайте посмотрим на пример еды. Какой может быть Еда? Оно могло быть вкусным или безвкусным, сладким или горьким, соленым или кислым. Каждый прием пищи имеет

вкус, но у каждого свой вкус. Давайте попробуем создать метод `getTaste()` в абстрактном суперклассе и указать его в подклассе.

```
абстрактный класс Еда {
    публичная абстрактная пустота getTaste();
}
класс Lemon расширяет Food {
    общественный недействительный getTaste () {
        System.out.println ("Кислый");
    }
}
```

Как вы уже знаете, этот метод называется переопределением.

Ключевые проблемы, которые вам следует иметь в виду, заключаются в том, что абстрактный класс не может быть создан и его следует расширять и реализовывать в подклассах. Абстрактный метод представляет собой просто сигнатуру метода без реализации.

Экземпляр абстрактного класса нельзя создать с помощью оператора `new`, но вы все равно можете определить его конструкторы, которые вызываются в конструкторах его подклассов.

Подкласс может переопределить метод своего суперкласса, чтобы объявить его абстрактным. Это случается редко, но полезно, когда реализация метода в суперклассе становится недействительной в подклассе (2). В этом случае подкласс должен быть объявлен абстрактным. Интересно отметить, что хотя вы и не можете создавать экземпляры абстрактных классов, вы все равно можете определять конструкторы для абстрактных классов! Звучит странно. Действительно, мы используем конструкторы для создания объектов, но если мы не умеем создавать экземпляры абстрактных классов, то зачем нам там конструкторы? Ответ довольно простой — вызывать их из подклассов через `super`. Иллюстрацией этого является пример ниже:

```
абстрактный класс Point {
    частный int x, y;
    public Point(int x, int y) { this.x = x; это.y = y;

    }
    общественный недействительный ход (int dx, int
        dy) { x += dx; y += dy;
        сюжет();
    }
    публичный абстрактный недействительный сюжет();
}
```

Итак, у нас есть абстрактный класс `Point` с позицией, заданной `x` и `y`. Это

Глава 4 – Наследование, полиморфизм и абстрактные классы

имеет методы для перемещения и построения точки (абстрактный, без реализации). Метод `plot()` является абстрактным, так как мы еще не знаем необходимых для рисования деталей — стиля, цвета и т. д. Определим его в подклассе `ColoredPoint`:

```
класс ColoredPoint расширяет точку {
    частный цвет int;
    public ColoredPoint (int x, int y, int color) {
        супер(x, y);
        this.color = цвет;
    }
    public void plot () { // код для
        построения SimpleColoredPoint
    }
}
```

Класс `ColoredPoint` демонстрирует вызов конструктора абстрактной точки. Это очень полезно, поскольку позволяет избежать дублирования кода и обеспечить согласованность общего поведения подклассов.

Иногда вам может потребоваться запретить возможность наследования определенного вами класса. В этом случае вам нужно использовать ключевое слово `Final`. Последний класс не может быть расширен, последняя переменная является константой:

итоговый статический двойной PI = 3,14159;

Последний метод не может быть изменен (переопределен) его подклассами.

4.9 Класс объекта

Точно так же, как людей можно классифицировать как млекопитающих, любой подкласс в ООП можно обобщить как конкретный набор характеристик и поведения любого из его предков. Чтобы обеспечить полиморфное поведение различных объектов, нам нужно какое-то обобщение среди различных реализаций.

В предыдущей главе вы уже познакомились с концепцией класса `Object`, который является корнем всех классов Java: абсолютно каждый класс в Java расширяет `Object`. Этот класс предоставляет ряд служебных методов, перечисленных ниже.

- `Equals()` — возвращает, имеют ли две ссылки на объект одинаковые значения.
- `hashCode()` — ценить

```

общедоступное логическое значение равно (Другой объект) {
    if (other == null) вернуть false;
    if (other == this) возвращает true;
    if (!другой экземпляр Person) возвращает false;
    Человек другойЧеловек = (Человек)другой;
    if(this.name.equals(otherPerson.name)) возвращает true;

    вернуть ложь;
}

```

Итак, внутри метода равенства вам просто нужно проверить параметры, используемые для проверки объектов на равенство, например, два человека равны, если их имена равны. Обратите внимание, что этот метод должен принимать параметр типа Object, поскольку мы хотим переопределить метод, а не перегрузить его. Использование может быть следующим:

```

если (somePerson.equals (anotherPerson)) {
}

```

Обратите внимание, что somePerson и otherPerson являются экземплярами класса Person.

- clone() – возвращает клон объекта. Пример реализации:
представлено ниже.

```

public Object clone() выдает
CloneNotSupportedException{
    Клонированный сотрудник = (Сотрудник)super.clone();
    вернуть клонированный;
}

```

Клонирование — очень сложная и горячая тема. Подробнее о клонировании мы поговорим в следующей главе «Интерфейсы», в части, посвященной Cloneable интерфейсу.

- getClass() — возвращает выражение запуска класса объекта, который является объектом класса. Используя это, вы можете проверить фактический тип объекта:

```
if(p.getClass().getName().equals("Человек")) {...}
```

На первый взгляд может показаться, что этот метод идентичен использованию

Глава 4 – Наследование, полиморфизм и абстрактные классы

оператор экземпляра. Однако между ними существует важное различие: instanceof проверяет, является ли объект экземпляром типа или какого-то подтипа. Напротив, getClass() проверяет, идентичны ли типы.

- `toString()` — возвращает строковое представление объекта.
 - о Метод `toString()` возвращает строковое представление объекта.
 - о Реализация по умолчанию возвращает строку, состоящую из имени класса, экземпляром которого является объект, знака (@) и адреса (abc100), по которому объект хранится в памяти, например `Student@abc100`.

Мы уже рассмотрели реализацию `toString()`:

```
публичная строка toString(){
    return "Я "+имя+", сотрудник КБТУ";
}
```

4.10 Советы по дизайну для наследования

Существует ряд простых правил, которым необходимо следовать, чтобы сделать архитектуру вашего приложения гибкой, четко определенной, безопасной и расширяемой. Запомните следующие рекомендации:

1. Скройте личные данные и частные методы.
2. Свойство, общее для всех экземпляров класса, должно быть объявлено как свойство класса. То есть у него должен быть модификатор `static`.
3. Если вам нужно, ваш класс может иметь различные конструкторы, но старайтесь всегда предоставлять общедоступный конструктор по умолчанию. Более того, вам следует по возможности переопределить методы `equals()` и `toString()`, определенные в классе `Object`.
4. Страйтесь выбирать информативные имена для переменных и методов и придерживайтесь единообразного стиля. Это сделает ваш код более читабельным.
5. Помните, что класс должен описывать одну сущность или набор подобных операций.
6. Необходимо разместить общие операции и поля в суперкласс.

7. Используйте наследование для моделирования отношений IsA , не копируйте код!
8. Хотя наследование дает много преимуществ, не используйте его, если все унаследованные методы не имеют смысла.
9. Не меняйте ожидаемое поведение, если вы переопределяете метод.
10. Используйте полиморфизм, а не тип информации. Например, рассмотрим код, представленный ниже:

```
если (x имеет тип 1)
    действие1(x);
иначе, если (x имеет тип 2)
    действие2(x);
```

Прежде чем писать такой код, спросите себя: представляют ли действия action1 и action2 общую концепцию? Если да, то сделайте концепцию методом общего суперкласса или интерфейса обоих типов, а затем просто вызовите x.action().

4.11 Подробный пример

Рассмотрим приложение, которое позволяет пользователю рисовать на холсте различные фигуры, такие как круги, овалы, треугольники, квадраты, прямоугольники и т. д. На этапе проектирования мы можем выделить в системе следующие классы:

- Холст — представляет «область рисования» .
- Цвет — для обозначения разных цветов наших фигур.
- Shape — абстрактный класс, используемый для представления нарисованных фигур, со следующими подклассами:
 - о Круг
 - о Прямоугольник и т. д.

Абстрактная форма имеет следующие поля и методы:

о Цвет Цвет

о Точка привязки — некоторое место, которое представляет положение фигуры на холсте для рисования, например, центр круга или верхний левый угол квадрата и т. д. Итак, мы используем его для определения местоположения фигуры.

о move(int dx, int dy) — меняет положение фигуры на холсте на dx, dy.

о draw(Canvas c) — рисует фигуру на холсте.

Приложению необходимо каким-то образом хранить все нарисованные фигуры в массиве или векторе:

Глава 4 – Наследование, полиморфизм и абстрактные классы

Векторные фигуры<Shape>;

Когда вы обновите холст, вам нужно будет нарисовать все фигуры.

Полиморфизм позволяет нам рисовать фигуры общим способом, это фантастика:

```
for (int i=0; i < shape.length; i++) {
    shape.get(i).draw(theCanvas);
}
```

Тем не менее, на самом деле рисование фигур не является универсальным, например, код рисования круга будет сильно отличаться от кода, используемого для рисования квадрата, и так далее. Что нам нужно сделать, так это предоставить абстрактный метод в Shape, который будет переопределен конкретно в его подклассах Circle, Triangle, Square и т. д.

```
абстрактный класс Shape {
    частный цвет цвета;
    частный точечный якорь;
    public void move(int dx, int dy) {
        якорь.x += dx;
        якорь.y += dy;
    }
    публичная абстрактная пустота (Canvas c);
}
```

```
класс Circle расширяет форму {
    частный внутренний радиус;
    public void draw(Canvas c) {
        // код рисования круга
    }
}
```

Прелесть этого в том, что вам не важно, какую именно фигуру вы рисуете — круг, квадрат или прямоугольник! Вы просто предоставляете метод draw() для каждого из них и вызываете его, используя ссылку Shape суперкласса (но фактический класс — Square, Circle и т. д.).

Обратите внимание, что метод move() не обязательно должен быть специально реализован для всех подклассов Shape — для всех них перемещение означает просто изменение точки привязки. Вот почему мы реализуем move() в суперклассе Shape. Итак, наш абстрактный класс Shape также имеет абстрактные и неабстрактные методы.

Метод Circle.draw() переопределяет метод Shape.draw(). В случае shape[i] оказывается экземпляром Circle, тогда

```
shape.get(i).draw(theCanvas);
```

выполнит код в Circle.draw(). Аналогично, если i-я фигура является экземпляром Square, то выполняется код Square.draw(). Как мы уже знаем, такое определение во время выполнения того, какой код выполнять, называется динамическим связыванием.

Полиморфизм и динамическое связывание позволяют нам избежать ужасного кода, подобного приведенному ниже:

```
if (shapes.get(i) instanceof Circle) {
    Круг с = (Круг)(shapes.get(i));
    // код рисования круга
} else if (shapes.get(i) instanceof Square) {
    Square s = (Square)(shapes.get(i));
    // код рисования квадрата
} еще // и т. д. и т. п.
```

Это полезный пример того, как наследование, абстрактные классы и динамическое связывание позволяют писать полиморфный код — код, работающий общим образом для ряда классов.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Цепочка наследования Полиморфизм Динамическое связывание

Суперкласс Переопределение подкласса Расширение

Преобразование Сужающее преобразование Объявлено Классом Фактическим

Класс Абстрактный класс и метод `toString()` равно()

ИСПЫТАНИЯ

1. Какой из следующих классов может расширить класс Order?

письмо б) Транзакция в) Онлайн-порядок г) Хаос

2. Полиморфизм позволяет:

а) Скрыть информацию

Глава 4 – Наследование, полиморфизм и абстрактные классы

б) Использовать объекты подклассов в любом коде, предназначенном для работы с объектами их суперкласса

в) Реализуйте только одну идею, чтобы исключить дублирование данных

г) Использовать объекты подклассов в любом коде

3. Когда объект имеет множество форм, его можно назвать:

а) Полиморфный

б) Инкапсулированный

в) Масштабируемость

г) повторно используемый

4. Предположим, вы хотите, чтобы подклассы в любом пакете имели доступ к членам суперкласса.

Какой доступ является наиболее ограничительным для достижения этой цели?

а) Общественный

б) Защищенный

в) Пакет

г) Частный

5. По какому принципу ООП следует использовать замену конструкции if-then-else в этом коде:

```
if (animal.IsCat()) { /* код */ } else if
(animal.IsDog()) { /* код */ } else if (animal.IsKoala()) { /
* код */
}
.
.
.
else if (animal.isMouse()) { /* код */ }
```

а) Полиморфизм

б) Наследование

в) Инкапсуляция

г) Используйте оператор экземпляра

6. Хоть мы и не обсуждали это, подумайте, почему Java отказывается от поддержки множественного наследования?

а) Множественное наследование практически не используется

б) Поддержка множественного наследования приводит к большим потерям производительности

в) Множественное наследование требует гораздо более сложных алгоритмов.

г) Из-за неоднозначности выбора поведения в случае, если суперклассы некоторого класса содержат методы с одинаковыми сигнатурами

7. Класс, производный от другого класса, называется:

а) Суперкласс

б) вложенные

в) Подкласс

г) Родитель

8. Животное a = новый Кот(). Животное это _____ класс, а Кот _____ класс

а) Фактический, заявленный

б) Заявленный, фактический

9. Для какого типа преобразования не обязательно предоставлять оператор приведения и это безопасное приведение?

а) Расширение конверсии

б) для них обоих

в) ни для одного из них

г) Сужение конверсии

10. Какие из приведенных ниже классов, скорее всего, будут абстрактными?

1) Книга

2) Форма

3) Транспорт

4) Учитель

а) 1, 2

б) 2, 3

в) 3, 4

г) 2, 4

ПРОБЛЕМЫ

1. Рассмотрите следующие определения классов и подумайте, что будет выведено из сегмента кода:

```

класс A {
    публичный недействительный метод() {
        System.out.println("A");
    }
}
класс B расширяет A {
    публичный недействительный метод() {
        System.out.println("B");
    }
}
A a = новый A();
метод();
a = новый B();
метод();
B b = новый B();
б.метод();
```

Глава 4 – Наследование, полиморфизм и абстрактные классы

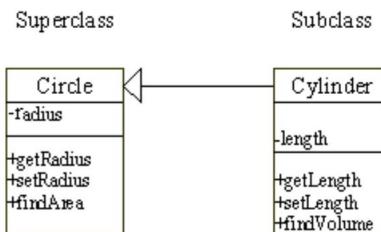
2. Для дочернего класса B, определенного в предыдущем упражнении, измените его метод() так, чтобы он вызывал версию метода() класса A перед выводом «B» .
3. Предположим, у вас есть класс Person и Student, расширяющий его (как в примерах из этой главы).

```
Человек p = новый Человек ();
Студент s = новый Студент();
Человек ps = новый студент();
```

Какие из следующих операций являются законными и почему? Какие из них незаконны и почему?

```
p = ps;
p = c;
c = p;
c = ps;
ps = c;
ps = p;
ps.getStudentNumber();
```

4. Создайте классы в соответствии со следующей диаграммой UML. Обратите внимание, что «+» означает «публичный» . _ означает личное.



5. Каков результат следующего сегмента кода?

```

класс T {
    Т() {
        System.out.println("T");
    }
}
класс X расширяет T {
    ИКС() {
        System.out.println("X");
    }
}
class OrderOfConstruction { public static
    void main(String args[]) {
        X-тест = новый X();
    }
}

```

6. Создайте класс Animal и производный класс Animal по вашему выбору (Cat, Dog, Crocodile и т. д.). В подклассе (или производном классе) продемонстрируйте:

- Методы, переопределяющие и перегружающие базовый класс.
- Использование ключевого слова super() с параметрами и без них.

7. Создайте абстрактный класс для трехмерных фигур, например Shape3D, имея Volume(), SurfaceArea() (добавьте другие методы по вашему выбору!). Затем создайте типы данных Cylinder, Sphere, Cube, расширяющие этот класс.

8. Создайте класс «Сотрудник» , объекты которого являются записями сотрудника. Этот класс будет производным классом класса Person (ДОЛЖЕН содержать методы Equals и toString).

Запись о сотруднике содержит имя сотрудника (наследованное от класса Person), годовую зарплату, представленную как одиночное значение типа double, год, когда сотрудник начал работу, как одиночное значение типа int, и номер национального страхования, который является значением введите Стока. Внутри этого класса вам нужно переопределить toString и равно методы класса. из Человек

Ваш класс должен иметь разумное количество конструкторов и методов доступа. Затем создайте класс «Менеджер» , расширяющий «Сотрудник» , каждый менеджер имеет команду сотрудников и может получить бонус. Вам необходимо переопределить методы toString и Equals. Писать

Глава 4 – Наследование, полиморфизм и абстрактные классы

другой класс, содержащий основной метод для полной проверки определения вашего класса.

Совет: по возможности используйте ключевое слово `super()`.

9. Создайте тип данных для шахматных фигур. Наследуйте базовый абстрактный класс `Piece` и создайте подклассы `Rock`, `King` и так далее. Включите метод `isLegalMove(Position a, Position b)`, который определяет, может ли данная фигура переместиться из `a` в `b`.

Затем создайте классную доску и тестовый класс, чтобы полностью имитировать игру в шахматы. Подумайте, как вы будете хранить текущее состояние игры, принимать ходы от пользователя, рисовать доску на консоли, проверять недопустимые ходы и т. д.

10. Когда электричество движется по проводу, оно подвергается электрическому трению или сопротивлению. Когда резистор с сопротивлением R подключается к разности потенциалов V , закон Ома утверждает, что он потребляет ток $I = V/R$ и рассеивает мощность V^2/R . Сеть резисторов, соединенных через разность потенциалов, ведет себя как один резистор, которое мы называем эквивалентным сопротивлением.

У вас должен быть: абстрактный суперкласс `Circuit`, который инкапсулирует основные свойства резисторной сети. Например, в каждой сети есть метод `getResistance`, который возвращает эквивалентное сопротивление цепи.

```

общественный абстрактный класс Circuit
{ общественный абстрактный двойной getResistance ();
  публичный абстрактный двойной getPotentialDiff();
  public Abstract void applyPotentialDiff(double V);

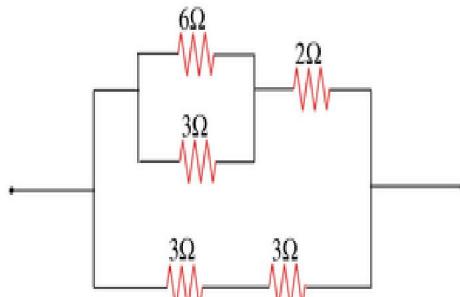
  общественный двойной getPower() {
    //ваш код
  }

  общественный двойной getCurrent() {
    //ваш код
  }
}

```

Сеть последовательно-параллельных резисторов представляет собой либо (i) один резистор, либо (ii) создается путем соединения двух цепей резисторов последовательно или параллельно.

Поэтому создайте три подкласса класса Circuit : Резистор, Последовательный и Параллельный. Ваша цель — уметь составлять схемы, как показано в следующем фрагменте кода, который представляет схему, изображенную ниже.



```

Цепь a = новый резистор (3.0);
Цепь b = новый резистор (3.0);
Цепь c = новый резистор (6.0);
Цепь d = новый резистор (3.0);
Цепь e = новый резистор (2.0);
Схема f = новая серия(a, b);
Схема g = новая параллель(c, d);
Схема h = новая серия(g, e);
Схема цепи = новая параллель(h, f);
двойной R = Circuit.getResistance();

```

Класс Resistor содержит конструктор, который устанавливает сопротивление в Омах, и метод доступа для его возврата. Он также имеет частное поле potentialDifference и методы получения/установки для него.

Глава 4 – Наследование, полиморфизм и абстрактные классы

Класс `Series` содержит конструктор, который принимает в качестве входных данных два объекта схемы резисторов и представляет схему с двумя последовательно соединенными компонентами.

Класс `Parallel` почти идентичен классу `Series`, за исключением того, что для вычисления эквивалентного сопротивления он использует правило взаимности вместо аддитивного правила.

Примечание. Разность потенциалов на каждой секции зависит от того, является ли цепь последовательной или параллельной. Для параллельной цепи разность потенциалов на каждой ветви равна разности потенциалов во всей параллельной цепи. Для последовательной цепи сначала найдите ток (I) по закону Ома $I = V/R$, где V — разность потенциалов в последовательной цепи, а R — ее общее сопротивление. Разность потенциалов на каждом резисторе равна силе тока, умноженной на сопротивление этого резистора.

11. Предположим, вам дан следующий класс `Account`:

```
класс Аккаунт {
    личный двойной баланс; //Текущий баланс
    частный int accNumber; //Номер счета
    публичный аккаунт(int a){
        баланс=0,0;
        номер аккаунта = a;
    }
    общественный недействительный депозит (двойная сумма) { ... }
    public void вывода(двойная сумма) { ... }
    общественный двойной getBalance() { ... }
    публичный двойной getAccountNumber(){...}
    публичный недействительный перевод (двойная сумма, другой счет) {
    }

    публичная строка toString() {
        ""
    }
    публичная окончательная недействительная печать()
    {
        //Не переопределяйте это, переопределите
        // метод toString
        System.out.println(toString());
    }
}
```

Завершите класс `Account` и, используя его в качестве базового, напишите два производные классы `SavingsAccount` и `CheckingAccount`.

Объектно-ориентированное программирование и дизайн

Объект `SavingsAccount`, в дополнение к атрибутам объекта `Account`, должен иметь переменную процентной ставки и метод, который добавляет проценты к счету. Объект `CheckingAccount` (здесь за каждую транзакцию взимается плата), помимо атрибутов объекта `Account`, должен иметь переменную-счетчик, в которой будет храниться количество транзакций, совершенных пользователем, и переменную `FREE_TRANSACTIONS` – количество бесплатных транзакций. Здесь же у вас будет метод `deductFee()`, который снимает со счета деньги за совершенные транзакции (предположим, на каждую транзакцию приходится \$0,02 -

снять или внести депозит). Убедитесь, что у вас есть переопределенные методы класса `Account`, если это необходимо, в обоих производных классах.

- После этого создайте класс `Bank`, объект которого содержит объекты `Vector of Account`. Счета в векторе могут быть экземплярами класса `Account`, класса `SavingsAccount` или класса `CheckingAccount`. Создайте несколько тестовых учетных записей (по несколько каждого типа).
- Напишите метод обновления в банковском классе. Он перебирает каждый счет и вносит/снимает деньги со счетов. После этого к сберегательным счетам добавляются проценты (с помощью метода, который вы уже написали), а с `CheckingAccounts` вычитаются комиссии. • Класс `Bank` требует методов для открытия и закрытия счетов.

Подумайте, какие изменения мы можем внести в учетную запись класса?

12. Создайте абстрактный класс `Shape` с полями цвета, `locationX`, `locationY` и абстрактные методы рисования фигуры a) в определенном месте б) в определенном месте с использованием определенного цвета. Затем создайте класс `Circle` и реализуйте эти методы.

Подсказка: b должен быть перегруженным методом a.

5 Interfaces

«Ходить по воде и разрабатывать программное обеспечение на основе спецификации легко,
если и то и другое заморожено» .
Эдвард Берард

Интерфейсы представляют собой чрезвычайно полезный инструмент в объектно-ориентированном программировании. Я программирование, которое недооценивают многие неопытные программисты. Чтобы повысить продуктивность программирования, разработчики хотят иметь возможность повторно использовать программные компоненты в нескольких системах. Интерфейсы дают нам возможность отделить повторно используемую часть вычислений от частей реализации, которые различаются в каждом сценарии повторного использования. Многоразовая часть просто вызывает методы интерфейса, реализованные определенным классом. Чтобы создать другое приложение, вы просто подключаете другой класс, реализующий те же методы. Таким образом, поведение программы варьируется в зависимости от подключенного класса – это относится к полиморфизму или динамическому связыванию (уже обсуждалось в главе 4).

Основные функциональные возможности, предоставляемые интерфейсами, включают в себя:

- Раскрыть программный интерфейс объекта (т.е. функциональность объекта), не раскрывая его реализацию. Как мы уже знаем, это представляет собой концепцию инкапсуляции.
- Возможность иметь несвязанные классы, реализующие схожие методы (поведения). Это особенно полезно, когда один класс не является подклассом другого.
- Моделировать множественное наследование. Как мы знаем, класс может расширять только один класс, но может реализовывать несколько интерфейсов.

Интерфейсы также демонстрируют полиморфизм, поскольку программа может вызвать метод интерфейса, и будет выполнена правильная версия этого метода. В зависимости от типа экземпляра объекта, передаваемого при вызове метода интерфейса. В качестве иллюстрации рассмотрим следующий общий метод тах:

```

класс Макс {
    // Возвращаем максимум между двумя объектами
    общедоступный статический сопоставимый максимум (сопоставимый o1,
        Сопоставимое o2){
        если (o1.compareTo(o2) > 0) вернуть o1;
        иначе верните o2;
    }
}

```

В этом примере Comparable — это интерфейс, поэтому экземпляры любого класса, реализующего этот интерфейс, можно передать методу max, чтобы найти среди них максимум.

5.1 Интерфейсы: основная концепция

Интерфейсы — это спецификации для множества различных реализаций. Интерфейсы используются для определения контракта о том, как вы взаимодействуете с объектом, который не зависит от базовой реализации. Задача профессионального объектно-ориентированного разработчика — отделить интерфейс от скрытых и очень часто (в определенных ситуациях) ненужных деталей реализации.

Как мы знаем, абстрактный класс может иметь данные в виде переменных экземпляра, неабстрактных и абстрактных методов. Интерфейс можно назвать абстрактным классом, в котором есть только статические конечные переменные экземпляра и все методы, которые являются абстрактными. Итак, интерфейс — это своего рода спецификация списка методов, которым класс, реализующий интерфейс, должен предоставить реализацию. В этом случае обязательно реализовать все методы или в противном случае сделать класс абстрактным. Короче говоря, вы можете думать об интерфейсе как об абстрактном классе, имеющем только абстрактные методы.

Важно отметить, что все методы в интерфейсе по умолчанию являются абстрактными и общедоступными. Как правило, не требуется, да и фактически избыточно объявлять метод в интерфейсе абстрактным или публичным, поскольку неявно они публичны и абстрактны, что бы вы ни делали.

Реже определяются некоторые данные в интерфейсе. Если в интерфейсе определены поля данных, то по умолчанию они определяются как общедоступные, статические и окончательные (8).

Проще говоря, интерфейс — это способ описать, что должны делать классы, без указания того, как они должны это делать. Например, в Java у нас есть интерфейс Comparable с одним методом CompareTo(). Этот интерфейс должен быть реализован классами, объекты которых должны быть

Глава 5. Интерфейсы

сопоставимы, то есть для объектов *a* и *b* мы должны иметь возможность сказать, выполняется ли *a>b*, *a<b* или *a=b*. Этот интерфейс выглядит следующим образом:

```
общедоступный интерфейс Сопоставимый
{
    int CompareTo (Объект другойОбъект);
}
```

Интерфейс Comparable требует, чтобы любой класс, реализующий его, предоставлял реализацию метода CompareTo(), и этот метод должен принимать параметр Object и возвращать целое число.

Как видно из приведенного выше примера, объявление интерфейса состоит из ключевого слова интерфейс, его имени и членов.

Что касается модификаторов интерфейсов, то по умолчанию они имеют модификатор package и все они неявно абстрактны (он опускается в соглашение). Вы также можете назначить ему модификатор public.

5.2 Члены интерфейса

Как и классы, интерфейсы могут иметь внутренние поля и методы. Поля интерфейса могут быть только постоянными переменными.

- Константы

Интерфейс может автоматически определять именованные константы, которые являются общедоступными, статическими и окончательными (эти модификаторы по соглашению опущены). Имейте в виду, что интерфейсы никогда не содержат мгновенных полей. Другое правило заключается в том, что все именованные константы интерфейса ДОЛЖНЫ быть инициализированы. Рассмотрим пример интерфейса Educated:

```
интерфейс Образованный {
    интервал УЧЕНИК = 0;
    int БАКАЛАВР = 1;
    интервал МАСТЕР = 2;
    инт ПХД = 3;
    void setEducationLevel (уровень int);
    int getEducationLevel();
}
```

Используя этот интерфейс, вы можете назначить уровень образования любому объекту, какой класс его реализует, и быть уверенным в согласованности.

- Методы

Методы интерфейса неявно абстрактны (отпущены по соглашению). Таким образом, каждое объявление метода состоит из заголовка метода и точки с запятой. Кроме того, они неявно публичны. Никакие другие типы модификаторов доступа для методов не допускаются. Кроме того, члены интерфейса не могут быть ни окончательными, ни статическими. Простой пример приведен ниже:

```
void setEducationLevel (уровень int);
```

5.3 Процедура реализации

Чтобы заставить класс реализовать интерфейс, нужно выполнить два шага:

1) Объявите, что класс намерен реализовать данный интерфейс с помощью используя ключевое слово «`implements`» , например:

```
класс Сотрудник implements Comparable {  
    . . .  
}
```

Вы можете встретить все реализации класса интерфейсов, используя запятую.

2) Предоставьте определения для всех методов в интерфейсе:

```
public int CompareTo(Object otherObject) {  
    Другой сотрудник = (Сотрудник) другойОбъект;  
    if (зарплата < другое.зарплата) вернуть -1;  
    if (зарплата > другое.зарплата) вернуть 1;  
    вернуть 0;  
}
```

Позже вы можете использовать API Java для сортировки массива, списка, вектора (любого коллекции) объектов, класс которых реализует этот интерфейс, так же просто:

Глава 5. Интерфейсы

```
Vector<Сотрудник> персонал = новый Vector<Сотрудник>();
Staff.add(новый Сотрудник("Вася", "Пупкин", 3000));
Staff.add(new Сотрудник("Иван", "Иванов", 2000));
Collections.sort(персонал);
```

Collections — это класс, предоставляющий базовые и расширенные служебные методы для всех коллекций, реализующих интерфейс Collection. Подробнее о коллекциях мы поговорим в следующих главах.

Помните, что если класс оставляет какой-либо метод интерфейса неопределенным, класс становится абстрактным классом и должен быть объявлен абстрактным. Еще одна важная вещь, которую следует подчеркнуть, это то, что один класс может реализовывать несколько интерфейсов. Просто разделите имена интерфейсов запятой, например:

```
класс Сотрудник реализует Comparable, Cloneable {
    .
    .
}
```

Имейте в виду, что интерфейсы не являются классами. Следовательно, вы никогда не сможете использовать оператор new для создания экземпляра интерфейса:

```
общедоступный интерфейс
    .
    .
}
Comparable x = new Comparable(); //ошибка!
```

Тем не менее, вы все равно можете объявить переменные интерфейса:

```
Сопоставимый x;
```

Эти переменные должны ссылаться на объект класса, реализующего интерфейс:

```
класс Сотрудник реализует Comparable {
    .
    .
}
x = новый сотрудник();
```

5.4 Расширение интерфейсов

Как мы уже упоминали, особенностью интерфейсов, отличающей их от абстрактных классов, является поддержка нескольких

реализация интерфейсов по 1 классу. Кроме того, интерфейс может расширять более одного интерфейса, поэтому для одного подинтерфейса может быть несколько суперинтерфейсов. Например, интерфейс ниже объединяет методы, определенные в `Serializable` (позже вы увидите, что интерфейс `Serializable` не имеет методов) и интерфейсы `Runnable` (один из основных интерфейсов в Java):

```
общедоступный интерфейс SerializableRunnable расширяет
Serializable, Runnable {
    .
    .
    .
}
```

Расширенный интерфейс наследует все константы своих суперинтерфейсов.

Обратите внимание на случаи, когда субинтерфейс наследует более одной константы с одинаковым именем или субинтерфейс и суперинтерфейс содержат константы с одинаковым именем — всегда используйте достаточно информации для ссылки на целевые константы. Рассмотрим эти случаи отдельно:

- Когда интерфейс наследует две или более константы с одинаковым именем

В этом случае в подинтерфейсе нужно явно использовать имя суперинтерфейса для ссылки на константу этого суперинтерфейса:

```
интерфейс А {
    интервал значения = 1;
}
интерфейс Б {
    интервал значения = 2;
}
интерфейс С расширяет А, В {
    System.out.println("A.val = " + A.val); System.out.println("B.val
        = " + B.val);
}
```

- Если суперинтерфейс и субинтерфейс содержат две константы с одинаковым именем, то константа, принадлежащая суперинтерфейсу, скрыта.

Доступ к константам версии подинтерфейса можно получить напрямую, используя его имя. Однако для доступа к константам версии суперинтерфейса вы

Глава 5. Интерфейсы

необходимо указать имя суперинтерфейса, за которым следует точка, а затем имя константы, например, X.val:

```
интерфейс Икс {
    интервал значения = 1;
}
интерфейс Y расширяет X{
    интервал значения = 2;
    int sum = значение + X.val;
}
```

- Если суперинтерфейс и субинтерфейс содержат две константы с одинаковым именем, а класс реализует субинтерфейс

В этом случае класс наследует константы версии подинтерфейса в качестве своих статических полей.

```
класс Z реализует Y { // внутри }
класса
System.out.println("Z.val:"+val); //Z.val = 2
//вне класса
System.out.println("Z.val:"+Z.val); //Z.val = 2
```

Вы можете использовать ссылку на объект для доступа к константам:

- Доступ к константам версии подинтерфейса осуществляется с помощью ссылки на объект, за которой следует точка, за которой следует имя константы.
- Доступ к константам версии суперинтерфейса осуществляется путем явного приведения

Пример ниже демонстрирует это:

```
Z v = новый Z();
System.out.print( "v.val = "+, ((Y)v).val = "+", " + v.val
                ((X)v).val =
                                " + ((Y)v).val +
                                " ((X)v).val );
```

вывод: v.val = 2, ((Y)v).val = 2, ((X)v).val = 1

Что касается методов, то существует ряд вопросов, на которые следует обратить внимание:

- Если объявленный метод в подинтерфейсе имеет ту же сигнатуру, что и унаследованный метод, и тот же тип возвращаемого значения, то новый

объявление переопределяет унаследованный метод в его суперинтерфейсе (3).

Как мы знаем, это работает одинаково для суперкласса и подкласса.

- Если разница только в типе возвращаемого значения, то быть ошибкой времени компиляции
- Интерфейс может наследовать несколько методов с одной и той же сигнатурой и типом возвращаемого значения. Класс может реализовывать разные интерфейсы, содержащие методы с одинаковой сигнатурой и типом возвращаемого значения.
- Методы с одинаковым именем, но с разными списками параметров, как вы знаете, называются перегруженными .

5.5 Почему мы используем интерфейсы?

Полезность интерфейсов выходит далеко за рамки простой публикации протоколов для других программистов:

- Любая функция может иметь параметры типа интерфейса.
- Любой объект класса, реализующего интерфейс, может быть передан в качестве аргумента.

Одна из основных причин, по которой в Java интерфейсам уделяется особое внимание, заключается в том, что часто требуется определить некоторую операцию для объектов, которые все соответствуют одному и тому же контракту. Таким образом, благодаря интерфейсу вы можете определить спецификацию очень общим способом с гарантией, что все объекты, соответствующие этому интерфейсу, будут иметь определенные реализации для всех методов.

Опять же, основная цель интерфейсов — определить контракт. Например, в Java у нас есть интерфейс `Iterable<E>`, который определяет контракт между оператором `foreach` и ЛЮБОЙ вещью, которая может быть итерируемой, то есть мы можем ее перемещать (`Vector`, `HashSet`, `ArrayList` и т. д.). Итак, `Iterable<E>` говорит: «Кем бы вы ни были, пока вы соответствуете контракту (реализуете `Iterable<E>`), я обещаю, вы сможете перебирать элементы» .

Еще одним прекрасным примером использования интерфейсов в Java является платформа `Collections`. Предположим, вы создаете функцию, которая принимает список (интерфейс) в качестве параметра, тогда абсолютно не имеет значения, что передавать в функцию: вектор , `ArrayList` или `LinkedList`. Более того, вы можете передать этот список (`Vector`, `ArrayList` или `LinkedList`) в любую функцию, требующую экземпляр интерфейса `Collection` или `Iterable` в качестве параметра. Это существенное преимущество интерфейсов делает возможным использование таких функций, как `Collections.sort(List list)`, независимо от того, какая реализация `List` передается.

Глава 5. Интерфейсы

В качестве еще одного простого примера предположим, что мы определяем интерфейс Shape с помощью метода area() и поэтому можем быть уверены, что любой класс, реализующий интерфейс Shape, будет определять метод area(). Практически это очень полезно, поскольку в случае множественных ссылок на объекты, реализующие интерфейс Shape, вы можете вызвать метод area() для каждого из этих объектов и рассчитывать получить в результате значение, представляющее площадь некоторого форм.

Для лучшего понимания давайте абстрагируемся от мира программирования и посмотрим, как интерфейсы используются в реальном мире. Посмотрите на картинку ниже. Что общего у всех предметов, изображенных на картинке? Подумав некоторое время, вы можете догадаться, что все они могут быть использованы с объектами розеток для получения электричества. Для этого им необходимо следовать определенным стандартным правилам, например, им необходимо реализовать интерфейс IPowerPlug .

Прелесть этого в том, что PowerSocket не нужно ничего знать о других объектах, за исключением того факта, что они реализуют IPowerPlug. Всем объектам требуется питание, предоставляемое PowerSocket , поэтому они просто реализуют IPowerPlug, и это позволяет объектам подключаться к нему.

Это определенно делает интерфейсы чрезвычайно полезными, поскольку они предоставляют контракты, которые объекты могут использовать для «разговора» друг с другом без необходимости знать что-либо еще друг о друге.

These objects implement the interface IPowerPlug



So they can be used with PowerSocket objects



Рисунок 14. Реальный пример использования интерфейса.

Подводя итог, выделим основные преимущества интерфейсов:

- Обеспечить легкое взаимодействие различных объектов.
- Скрыть детали реализации классов друг от друга.
- Поддержка проектирования по спецификации
- Разрешить использование подключаемых компонентов (7)

• Разрешить повторное использование программного обеспечения.

В целом, интерфейсы помогают обеспечить не только стандартизацию вашей системы, но также расширяемость, гибкость, удобство обслуживания, масштабируемость и возможность повторного использования.

5.6 Интерфейсы маркеров

Интерфейс маркера (также называемый иногда интерфейсом тега) не имеет ни методов, ни констант. Проще говоря, пустой интерфейс в Java называется интерфейсом маркера. Его основная цель — разрешить использование экземпляра в запросе типа. Клонируемый интерфейс является таким примером.

Примеры интерфейсов маркеров включают Serializable и Clonnable. Интерфейс маркера не содержит констант или методов, но имеет особое значение для системы Java. Например, система Java требует, чтобы класс реализовывал интерфейс Cloneable, чтобы его можно было клонировать.

Клонируемый

Интерфейс не имеет ни методов, ни констант, но помечает класс как участвующий в механизме клонирования. Таким образом, классам, реализующим эти интерфейсы, не нужно переопределять какой-либо из методов.

Возникает резонный вопрос: если в таком интерфейсе нет полей и методов, то зачем он нам нужен? Ответ довольно прост: они используются для указания или сигнала чего-либо компилятору или JVM. В частности, если JVM видит, что класс является сериализуемым, она позволяет выполнить над ним какую-то специальную операцию. Аналогично, если JVM видит, что один класс реализует интерфейс Cloneable, она выполняет некоторую операцию для поддержки клонирования. Итак, интерфейс маркера используется для обозначения или сигнала чего-либо.

JVM. Так, например, перед сериализацией объекта и его отправкой по сети Java проверяет, реализует ли класс интерфейс Serializable. Если нет, генерируется исключение.

С учетом этой цели возникает еще один вопрос: «Почему этот сигнал нельзя реализовать с помощью простого флага внутри класса?». Конечно, это имеет смысл. Но использование интерфейсов тегов делает его более читабельным и, что еще более важно, это также позволяет воспользоваться преимуществами полиморфизма в Java.

В последующих главах мы уделим больше внимания интерфейсам Cloneable и Serializable.

5.7 Клонирование объектов

Прежде чем углубляться в технику клонирования, давайте выясним, в чем разница между копией объекта и клоном объекта. Для этого вам необходимо пересмотреть, как работает оператор присваивания для ссылочных и примитивных типов (глава 2). Когда вы используете оператор присваивания для создания копии объекта, на самом деле у вас все еще есть один объект и два указателя, ссылающиеся на него. Таким образом, любые изменения, выполненные через одну ссылку, влияют на состояние объекта для всех ссылок. Напротив, клон объекта гарантирует, что последующие изменения нового объекта-клона не повлияют на состояние исходного объекта.

Клонирование может быть выполнено методом клонирования . Этот метод возвращает новый объект, начальное состояние которого является копией текущего состояния объекта, для которого было вызвано клонирование .

При написании метода клонирования необходимо учитывать три фактора:

- Пустой интерфейс Cloneable . Вы должны реализовать его, чтобы предоставить метод клонирования , который можно использовать для клонирования объекта. Клонируемый Интерфейс не имеет ни методов, ни констант, но помечает класс как участвующий в механизме клонирования. На самом деле это выглядит так:

```
публичный интерфейс Cloneable { //  
    здесь нет кода  
}
```

Здесь нам не нужен абстрактный метод clone() , поскольку у нас уже есть внутренняя часть объекта с частичной реализацией.

- Метод клонирования , реализованный классом Object . Он выполняет простое клонирование, копируя все поля исходного объекта в новый объект.
- Исключение CloneNotSupportedException , которое можно использовать для обозначения того, что метод клонирования класса не должен вызываться.

Итак, теперь вы знаете, что класс Object предоставляет метод clone , который выполняет простое клонирование путем копирования всех полей исходного объекта в новый объект. Этот тип клонирования обычно называют поверхностным.

клонирование. Хотя во многих случаях он работает нормально, иногда вам может потребоваться переопределить его для особых целей, когда вы хотите выполнить глубокое клонирование.

Объектно-ориентированное программирование и дизайн

Давайте сравним поверхностное и глубокое клонирование. Мелкое клонирование, как уже было сказано, представляет собой простое копирование поля за полем. Например, предположим, что вы реализуете класс «Сотрудник» и хотите предоставить метод для клонирования:

```
класс Сотрудник расширяет Person реализует Cloneable{
    двойная зарплата;
    Дата арендыДата;
    ...
    public сотрудник clone() выдает
        CloneNotSupportedException{
            вернуть (Сотрудник) super.clone();
        }
}
```

Это пример обычного поверхностного клонирования. Вы просто вызываете метод `clone()`, определенный в корневом классе объекта. Причина, по которой нам нужно переопределить его, чтобы просто вызвать, заключается в том, что он невидим для сублаксов `Object`. В противном случае мы даже не определили интерфейс `Cloneable`, и все объекты были клонируемы по умолчанию.

Однако иногда мы хотим запретить клонирование объектов, в этом случае мы просто не реализуем интерфейс `Cloneable`.

Проблема с поверхностным клонированием заключается в том, что это может быть неправильно, если оно дублирует ссылку на объект, который не должен использоваться совместно. Рассмотрим следующий пример:

```
класс IntegerStack реализует Cloneable {
    частный буфер int[]; // стек целых чисел
    частный int top; // // максимальный индекс в стеке
    (начиная с 0)
    ...
}
```

Ниже вы можете увидеть, как исходные и скопированные объекты хранятся в памяти.

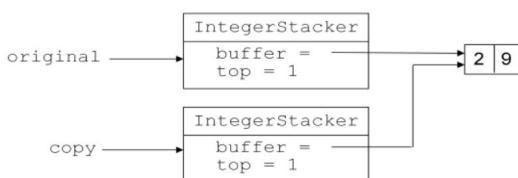


Рисунок 15. Мелкое клонирование

На рисунке 15 легко увидеть, что как исходные, так и клонированные объекты имеют свои собственные копии переменных примитивного типа (вверху), но у них есть только один общий экземпляр переменной ссылочного типа — буфер массива. Хотя у них есть отдельные указатели на этот массив, он общий для обоих стеков, поэтому, если один стек меняет массив, он будет изменен на другой (помните пример с телевизором и парой пультов из главы 2?). Это показывает непригодность поверхностного клонирования для определенных случаев — случаев, когда объект, который мы хотим клонировать, содержит объекты внутри.

Подходит ли поверхностное клонирование для класса сотрудников, определенного выше? Конечно, нет, поскольку у сотрудника есть поле типа reference, HireDate типа Date.

Для таких составных объектов, содержащих объекты внутри, необходимо реализовать глубокое клонирование: клонирование всех объектов из объекта, для которого вызывается клонирование. При реализации глубокого клонирования исходные и клонированные объекты будут храниться следующим образом:

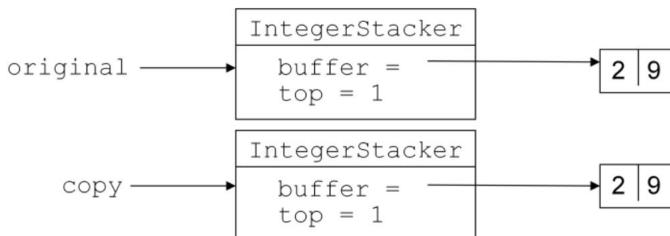


Рисунок 16. Глубокое клонирование

Давайте изменим метод клонирования для сотрудника, чтобы он выполнял глубокое клонирование:

```

public сотрудник clone() выдает
    CloneNotSupportedException{
        Клонированный сотрудник =(Сотрудник)super.clone();
        cloned.hireDate = (Дата)hireDate.clone();
        вернуть клонированный;
    }
  
```

При такой реализации и клон, и оригинал будут иметь собственную копию данных о найме.

Подводя итог, еще раз выделим, что нужно сделать, чтобы объект стал клонируемым:

1. Класс должен реализовывать интерфейс Cloneable. Чтобы обеспечить возможность клонирования, системе Java требуется класс для реализации интерфейса Cloneable.
2. Класс должен переопределить метод клонирования с открытым доступом. модификатор (1).
3. Шагов 1,2 достаточно для поверхностного клонирования. Если вам нужна глубокая клонирование, внутри метода clone() вам нужно вызвать clone() для каждого объекта внутри объекта, который вы хотите клонировать.

5.8 Интерфейсы и абстрактные классы

Наверное, вы спросите, зачем нам вводить два понятия: абстрактный класс и интерфейс? Почему бы просто не использовать абстрактные классы, например, как в коде ниже:

```
абстрактный класс Comparable {
    public Abstract int CompareTo (Другой объект);
}
класс Сотрудник расширяет Comparable {
    pulibc int CompareTo(Другой объект) { . . . }
}
общедоступный интерфейс
    int CompareTo (Другой объект)
}
класс Сотрудник реализует Comparable {
    public int CompareTo (Другой объект) { . . . }
}
```

Чтобы ответить на этот вопрос, нужно вспомнить ключевые различия между абстрактными классами и интерфейсами:

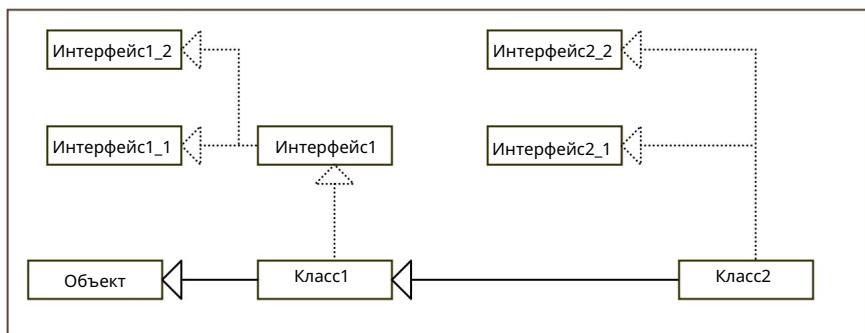
- Класс может расширять только один абстрактный класс, но может реализовать неограниченное количество интерфейсов
- Абстрактный класс, помимо абстрактных методов, может иметь методы с реализацией по умолчанию (неабстрактные методы). Напротив, интерфейсы ограничены общедоступными абстрактными методами, не имеющими реализаций.
- В интерфейсе данные должны быть константами, тогда как абстрактный класс может иметь все типы данных — статические методы, статические данные, частные и защищенные методы и т. д.

- Абстрактный класс должен иметь хотя бы один абстрактный метод, тогда как интерфейсы могут быть пустыми вообще (такие интерфейсы называются теговыми или маркерными интерфейсами, как мы изучали ранее).

Поскольку вы можете наследовать несколько интерфейсов, они часто служат очень полезным механизмом, позволяющим классу вести себя по-разному в разных ситуациях. Обычно реализация интерфейса является хорошей идеей, когда вам нужно определить методы, которые должны быть явно переопределены каким-либо подклассом. С другой стороны, если вы хотите, чтобы некоторые методы, реализованные с помощью реализаций по умолчанию, наследовались и использовались подклассом, используйте абстрактный базовый класс вместо интерфейса.

Опять же, поскольку все методы, определенные в интерфейсе, являются абстрактными методами, Java не требует, чтобы вы помещали абстрактный модификатор в методы интерфейса, но вы должны помещать абстрактный модификатор перед абстрактным методом в абстрактном классе.

Помните о ключевой разнице между абстрактным классом и интерфейсом — каждый метод в интерфейсе имеет только сигнатуру без реализации, но абстрактный класс может иметь конкретные методы.



Судя по рисунку выше, подумайте о следующем:

- Может ли класс расширять несколько классов?
 Может ли класс реализовать несколько интерфейсов?
 Может ли интерфейс расширять несколько интерфейсов?

5.9 Пример: стек

Рассмотрим спецификацию общего стека на базе LIFO¹⁰.

политика:

```
общедоступный интерфейс StackInterface {
    логическое значение пустой();
    void push (Объект x);
    Object pop() генерирует исключение EmptyStackException;
    Object peek() выдает EmptyStackException;
}
```

Обратите внимание, что приведенные выше методы работают с параметром типа Object. Это неудивительно, поскольку стек является «типом контейнера», поэтому для всех объектов очень часто используется базовый класс. Поскольку все объекты в Java расширяют класс Object, мы всегда можем сослаться на любой объект в системе, используя Object.

ссылка. Тем не менее, помните, что когда вы извлекаете из стека, необходимо явно указать регистр от типа объекта до фактического типа, чтобы мы могли соответствующим образом манипулировать им.

```
класс Stack реализует StackInterface {
    частный вектор v = новый вектор(); public boolean
    IsEmpty() { return v.size() == 0;

    }
    public void push (элемент объекта) {
        v.addElement(пункт);
    }
    общественный объект поп () {
        Объект obj = заглянуть();
        v.removeElementAt(v.size() - 1);
        вернуть объект;
    }
    public Object peek() выдает EmptyStackException {
        если (v.size() == 0)
            выбросить новое EmptyStackException();
        return v.elementAt(v.size() - 1);
    }
}
```

¹⁰ LIFO – «последним пришел — первым вышел», политика хранения элементов в структуре данных.

Глава 5. Интерфейсы

Выше вы можете увидеть реализацию примера класса, реализующего интерфейс `Stack`. Имейте в виду, что вам необходимо обеспечить реализацию всех методов интерфейса и соответствовать их сигнатурам.

5.10 Внутренние классы

Иногда у вас есть класс, который служит очень ограниченной цели. В этом случае вы можете объявить класс внутри метода или класса, которому он нужен. Такой класс относится к внутреннему классу (иногда его также называют вложенным классом). Итак, внутренний класс — это любой класс, определенный внутри другого класса. Будьте осторожны при использовании внутреннего класса — в случае, если вам может понадобиться использовать его за пределами класса, лучше сделать его общедоступным.

Как уже упоминалось, вы можете объявить внутренний класс внутри метода. и внутри самого класса. Примеры для обоих этих случаев приведены ниже:

- Объявление внутреннего класса внутри метода.

Шаблон:

```
класс Внешний {
    сигнатура метода {
        класс Внутренний {
            методы
            поля
        }
    }
}
```

Пример:

```
public static void main(String[] args) {
    класс Пара {
        общественная пара (int x, int y) {
            это.x = x;
            это.y = y;
        }
        интервал x;
        интервал y;
    }
    Пара p = новая пара(2,4);
}
```

- Объявление внутреннего класса внутри класса

Шаблон:

```
class Outer { поля
    методов

    assesModifier class Inner { поля методов

    }

    ...
}

}
```

Пример:

```
класс LinkedList { //...

Узел [] узлы; класс
Node{ Узел
    следующий;
    внутренний
    ключ; //...
}
}
```

Позже вы можете использовать их следующим образом:

```
Список LinkedList = новый LinkedList(); Узел n =
list.new Node();
```

Важно отметить, что методы внутреннего класса могут напрямую обращаться к членам внешнего класса.

В общем, попробуйте использовать внутренние классы для небольших тактических классов, которые должны не быть видимым в другом месте программы.

5.11 Интерфейсы и наследование. Когда что использовать?

Из этой главы вы получили представление о том, как интерфейсы могут быть полезны для добавления уровня абстракции в систему. Обычно это делается при проектировании.

Глава 5. Интерфейсы

этап. Однако надо признать, что интерфейсы полезны для больших и средних проектов, для небольших проектов это скорее всего «излишество» .

Другая задача — уметь тщательно определять, когда использовать наследование, а когда интерфейсы. Придерживайтесь следующих рекомендаций, правильный выбор обязательно повысит масштабируемость и возможность повторного использования вашего кода.

- A Is a B – используйте наследование.

Если можно сказать, что A есть B (где A и B могут быть некоторыми логическими элементами или понятиями), тогда очевидно использование наследования. В данном случае ваш класс является подклассом более обобщенного класса, например HouseCat (A) наследует от Feline (B), поскольку домашний кот определенно является кошачьим.

- A Имеет B – используйте поля-члены

Например, у LittleGirl есть машина, поэтому очевидно, что она не должна быть подклассом HouseCat, поскольку она не кошка. Обычно в таких ситуациях LittleGirl лучше всего иметь поле экземпляра:

```
класс LittleGirl
{
    средний возраст;
    Строковое имя;
    Домашний кот-питомец
}
```

- A выполняет B – использовать интерфейс

Интерфейсы следует использовать в том случае, если у нас есть класс или набор классов, предоставляющих схожую функциональность, но нет четкой линии наследования. Итак, интерфейс — это просто сертификат, в котором говорится: «Объекты А выполняют эту функциональность (Б)». Возвращаясь к нашим коту и девочке, хотя HouseCat наследует от Feline, он может реализовать интерфейс ICanHavePizza. Очевидно, что LittleGirl также может реализовать интерфейс ICanHavePizza. Итак, у нас может быть метод, которому мы можем передать LittleGirl, HouseCat и многие другие объекты, реализующие интерфейс ICanHavePizza, чтобы накормить их пиццей.

```
общественная подача пиццы (клиент ICanHavePizza) {
    ...
}
```

Это полезно, поскольку HouseCat и LittleGirl не имеют общего

подкласс (например, Person). Однако оба они выполняют действия, связанные с пиццей.

5.12 Принципы объектно-ориентированного проектирования

Последние несколько глав познакомили нас с основными концепциями, лежащими в основе объектно-ориентированного программирования. Было бы здорово завершить эту главу кратко остановимся на различных принципах объектно-ориентированного проектирования. Большинство из них должны быть вам знакомы:

- Принцип «разделяй и властвуй» .

Это означает, что задачи решаются путем разделения их на несколько классов, при этом каждый из классов делится на отдельные методы. Сама идея иерархии классов является применением этого принципа. Другое название этого принципа — модульность: разбиение на части. Модуль можно определить по-разному, но

обычно должен быть компонентом или подсистемой более крупной системы и работать внутри этой системы независимо от операций других компонентов (1).

- Принцип инкапсуляции. Этот принцип гласит, что суперклассы должны инкапсулировать те особенности иерархии классов, которые являются общими для всех объектов в иерархии. Подклассы, в свою очередь, инкапсируют особенности, которые делают их отличительными.
среди других классов иерархии (4).
- Принцип интерфейса. Программисты интерфейсов предоставляют спецификацию того, как различные типы связанных объектов взаимодействуют друг с другом через сигнатуры методов, содержащиеся в интерфейсах.
- Принцип сокрытия информации. Это включало последовательное использование квалификаторов Private, Protected и Public . • Принцип общности. Этот принцип гарантирует, что по мере продвижения вниз по хорошо продуманной иерархии классов вы переходите от более общих к более конкретным характеристикам объектов, участвующих в системе (4).
- Принцип абстракции. Проектирование иерархии классов — это упражнение в абстракции, поскольку более общие функции задействованных объектов переносятся в суперклассы. Аналогично, проектирование интерфейса Java или абстрактного метода суперкласса является формой абстракции, благодаря которой сигнатура метода отличается от его различных реализаций (4). Чтобы соответствовать этому принципу, вы

Глава 5. Интерфейсы

необходимо уметь определять ключевые особенности и игнорировать детали. Абстрактное программирование — это механизм и практика, предназначенные для сокращения и исключения деталей, чтобы можно было сосредоточиться на нескольких концепциях одновременно (1).

- **Принцип расширяемости.** Это означает переопределение унаследованных методов и реализацию абстрактных методов либо из абстрактного/не абстрактного суперкласса, либо из интерфейса. Это можно использовать для расширения функциональности существующей иерархии классов.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Интерфейс	Спецификация Сопоставимая Сериализуемая	
Клонируемый	Маркерный интерфейс	Клонирование объектов
Мелкое клонирование	Глубокое клонирование	Модульность
Принцип «разделяй и властвуй»		Расширяемость

ИСПЫТАНИЯ

1. Все методы интерфейса... ?

- | | |
|----------------|------------------|
| а) финальный | б) абстрактный |
| в) статический | г) экономический |

2. Какой из следующих интерфейсов является маркерным?

- | | |
|--------------------------------|-------------------------------|
| а) сериализуемый, сравнимый | б) Клонируемый, Равный |
| в) Сравнимый, Сравнительный<T> | г) Сериализуемый, Клонируемый |

3. Вы не можете создавать объекты интерфейса, но можете объявлять переменные интерфейса. Исходя из этого, выберите правильный ответ:

- | |
|---|
| а) x = новый сопоставимый(...); x = новый сотрудник(...); |
| б) Сопоставимый x = новый сотрудник(...); |
| в) Comparable x = новый Comparable(...); |

4. Какой термин используется для описания внутреннего представления объекта, скрытого от взгляда за пределами определения объекта?

- а) Инкапсуляция
- б) Полиморфизм
- в) Абстракция
- г) Наследование

5. Каковы основные принципы объектно-ориентированного проектирования?

- а) Модульность, инкапсуляция и абстракция
- б) Иерархия, параллелизм и абстракция
- в) Удобство использования, инкапсуляция и иерархия
- г) Типизация, параллелизм и абстракция

6. Что верно об интерфейсе маркера?

- а) Это пустой интерфейс
- б) Имеет только абстрактные методы
- в) Имеет логический флаг
- г) Имеет только финальные методы

7. Что неверно?

- а) Интерфейсы поддерживают множественное наследование
- б) Методы интерфейса могут быть статическими
- в) Интерфейс может иметь модификаторы public и package.
- г) Мы можем объявить переменную интерфейса

8. Какой метод используется для сравнения двух объектов и сколько типов логического вывода он может выдать?

- а) сравнение(), 3 ответа
- б) экземпляр(), 2 ответа
- в) равно(), 2 ответа
- г) CompareTo(), 3 ответа

9. Какой модификатор интерфейса используется по умолчанию?

- а) общественный
- б) пакет
- в) абстрактный
- г) финальный

ПРОБЛЕМЫ

1. Рассмотрим разработанный вами интерфейс под названием DoIt:

```
общедоступный интерфейс DoIt {
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

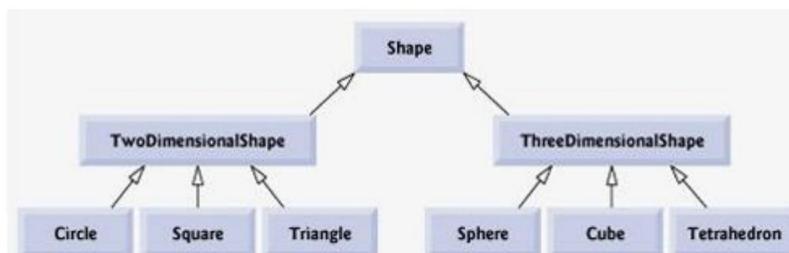
Давайте представим, что позже вы решили добавить в свой интерфейс еще один метод, чтобы он выглядел так:

```
общедоступный интерфейс DoIt {
    void doSomething (int i, double x);
    int doSomethingElse(String s);
    логическое значениеisOk(int i, double x, String s);
}
```

Теперь, если вы внесете это изменение, все классы, реализующие старый интерфейс DoIt , имеющий 2 метода, сломаются, поскольку они больше не реализуют все методы интерфейса — в частности, isOk ими не реализован. Какое решение этой проблемы вы можете предложить?

2. Создайте форму интерфейса , а затем реализуйте иерархию формы, как показано на рисунке ниже.

- Каждая двумерная фигура должна содержать: Метод getArea.
для того, чтобы вычислить площадь двумерной фигуры.
- Каждая трехмерная фигура должна иметь методы getArea.
и getVolume для расчета площади поверхности и объема трехмерной фигуры соответственно.



После этого напишите программу, которая использует массив ссылок Shape на объекты каждого конкретного класса в иерархии. Программа должна

вывести текстовое описание объекта, на который ссылается каждый элемент массива. Кроме того, в цикле обработки всех фигур в массиве определите, является ли каждая фигура двумерной или трехмерной фигурой. Если фигура является двумерной фигурой, отобразите ее площадь. Если фигура является трехмерной фигурой, осмысленно отобразите ее площадь и объем.

3. Когда использовать интерфейс, а когда использовать абстрактный класс. Для каждого «когда» приведите расширенные примеры (с кодами классов/интерфейсов).

4. Расширьте классы «Сотрудник» и «Менеджер», созданные в задаче № 8 предыдущей главы.

- Замените поле «год» полем HireDate типа java.util.Date.
- Ваши классы должны реализовывать интерфейс Comparable .
(Сотрудник1 > Сотрудник2, если его зарплата больше, чем зарплата Сотрудника2, то же самое для менеджеров, но если их зарплаты равны, сравнивайте по бонусам).

- Реализуйте интерфейс Cloneable , чтобы иметь возможность клонировать объекты.
Используйте поверхностное или глубокое клонирование по своему усмотрению.

5. Предположим, у вас есть интерфейс Moveable. Придумайте какой-нибудь интерфейс это может продлить его. Реализуйте оба интерфейса.

6. Вам нужно написать класс MinMax с методом minmax , который принимает в качестве параметра массив целых чисел и возвращает min и max одновременно (используя один метод и один вызов).

Подсказка: используйте внутренний класс

```
общественный класс MinMax {
    статический класс ??? {
        }
        статика ??? minmax(int значения[]){
            вернуть ???;
        }
    }
// Тестовый класс:
int a[] = {0, 8, -3, 20};
МинМакс м = новый МинМакс();
// Делаем что-нибудь, чтобы найти минимум и максимум //
используя экземпляр т класса MinMax.
```

7. Предположим, у вас есть два интерфейса, I и J, и класс C, реализующий интерфейс I:

```
интерфейс я {
    ...
}

интерфейс J{
    ...
}

класс C реализует I{
    ...
}
```

Предположим, что я объявлен следующим образом:

Я я = новый C();

Какое из следующих утверждений ниже вызовет исключение и почему?

```
Cc = (C)i;
J j = (J)i;
я = (я) ноль;
```

Что вы можете предложить для определения подобных ошибок во время компиляции?

8. Предположим, у вас есть следующий интерфейс:

```
фильтр публичного интерфейса {
    логическое принятие (Объект x);
}
```

Подумайте о паре классов, которые могли бы это реализовать, и предоставьте соответствующий код.

9. Создайте класс Date с полями год, месяц, день. Реализуйте интерфейс Comparable и предоставьте метод `toString()`. Затем создайте несколько объектов Date, сохраните их в векторе и отсортируйте.

10. Создайте класс Mark, имеющий поле точек (например, 95) и метод `getLetter()` (например, A-). Внедрить интерфейс Comparable и предоставить

Объектно-ориентированное программирование и дизайн

Методы `toString()` и `Equals()`. Затем создайте несколько объектов `Mark`, сохраните их в векторе и отсортируйте.

11. Методы, объявленные в интерфейсе, не могут быть объявлены окончательными. Почему?

6 UML diagrams

«Совершенство [в дизайне] достигается не тогда, когда больше нечего добавить, а когда нечего отнять» .

Антуан де Сент-Экзюпери

«Есть два способа создания программного обеспечения. Один из способов — сделать его настолько простым, чтобы не было очевидных недостатков. А другой способ — сделать его настолько сложным, чтобы не было явных недостатков» .

АВТОМОБИЛЬ Хоар

Б Прежде чем писать код для сложного приложения, вам необходимо спроектировать свой решение. Хотя для этого существует множество методов, в большинстве случаях этап проектирования состоит из следующих задач:

- Определение классов в системе.

Выделите существительные в описании задачи и подготовьте список существительных. Исключите те, которые кажутся нецелесообразными для занятий.

- Определение обязанностей каждого класса.

Во-первых, вы можете составить список основных функций, которые должна иметь ваша система. Затем для каждой задачи нужно найти класс (из списка, полученного на предыдущем шаге), отвечающий за выполнение этой задачи.

- Выявление связей между классами.

На этом этапе вам необходимо подготовить диаграмму классов, отражающую связи между всеми классами, которые вы определили ранее.

Как правило, существует три основные части модели системы, а именно:

- Функциональная модель, которая представляет функциональность системы. с точки зрения пользователя. Пример: диаграммы вариантов использования.
- Объектная модель, которая используется для представления структуры системы с использованием классов, атрибутов, операций и различных типов ассоциаций между ними. Пример: диаграммы классов.
- Динамическая модель, описывающая внутреннее поведение системы. Пример: диаграммы последовательности.

В этой главе вы познакомитесь с типами отношений, существующих между классами, и способами, которые вы можете использовать для отражения конструкции вашей системы.

Мы специально сосредоточимся на диаграмме классов UML , кратко обсудив

Диаграммы последовательности и вариантов использования . Мы надеемся, что после прочтения этой главы вы оцените полезность подхода к моделированию UML и сможете легко определять и реализовывать различные отношения диаграмм классов, которые используются в объектно-ориентированном моделировании.

6.1 Диаграммы вариантов использования

Диаграмма вариантов использования — это тип функциональной диаграммы, которая используется для представления графического обзора функциональных возможностей системы с точки зрения участников (обычно пользователей системы), их действий — вариантов использования и любых зависимостей между этими вариантами использования. Итак, основная цель диаграммы вариантов использования — показать, какие системные функции выполняются каждым действующим лицом.

Следует подчеркнуть, что диаграммы вариантов использования не подходят для представления детального проектирования системы и не могут описывать внутренние детали работы системы (4). Вместо этого они могут быть полезны для облегчения общения с будущими бизнес-пользователями системы и особенно полезны для определения необходимых функций, которыми должна обладать система. Следовательно, диаграммы вариантов использования указывают, что должна делать система, но не уточняют, как этого следует достичь.

Диаграммы вариантов использования содержат следующие элементы: действующие лица и роли. Эти элементы могут быть связаны различными ассоциациями. Пример простейшей возможной диаграммы вариантов использования представлен ниже:

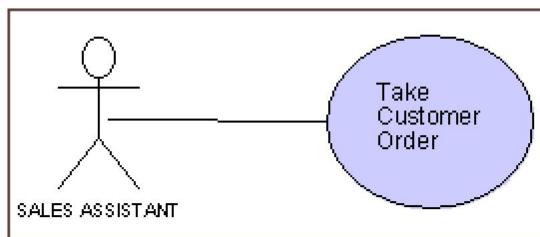
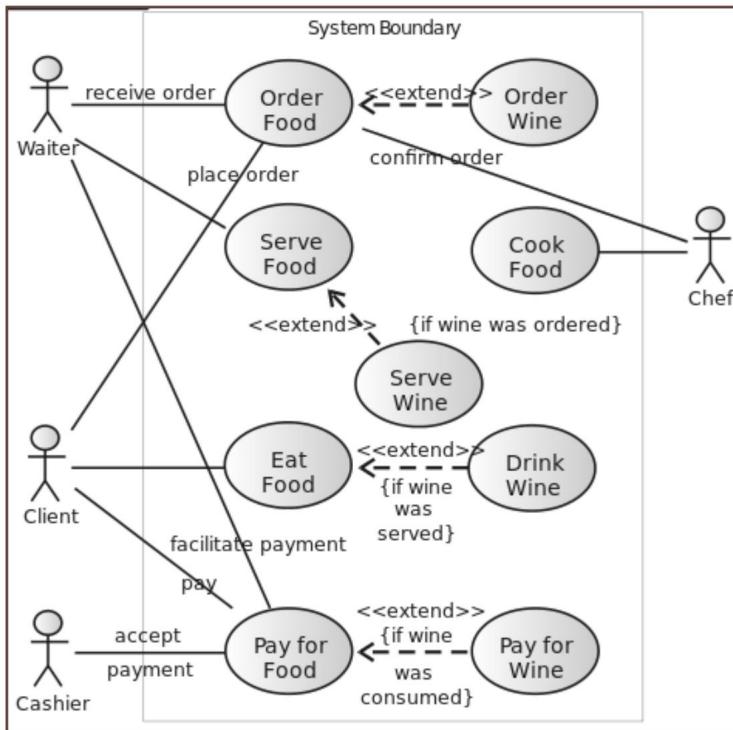


Рисунок 17. Пример использования диаграммы классов

Кроме того, вы можете использовать такие соединения, как include и Extend. Продлевать соответствует обычному наследованию (переход от общего к более конкретному). Отношение включения позволяет нам включать один вариант использования в другой. Рассмотрим пример ниже:

Рисунок 18. Диаграмма классов использования — ресторанный систем¹¹

На рисунке выше мы видим, что, например, в сценарии использования «Заказ еды» участвуют два актера – Официант (получает заказ) и Клиент (размещает заказ). Кроме того, существует расширенный вариант использования «Заказать еду», который называется «Заказать вино» .

6.2 Диаграммы последовательности

Диаграмма последовательности — это своего рода динамическая диаграмма, иллюстрирующая, как процессы взаимодействуют друг с другом. Он содержит объекты, участвующие в системном сценарии, и последовательность активаций, содержащую сообщения, которыми обмениваются объекты, участвующие в функционировании системы.

¹¹ Источник: Википедия.org.

сценарий. Основное удобство диаграммы последовательности состоит в том, что она позволяет визуально определять простые динамические сценарии.

Диаграммы последовательности состоят из объектов, активаций и сообщений.

(7). Вы уже знаете о предметах. Активации (или линии жизни – параллельные вертикальные прямоугольники) – это разные процессы, живущие одновременно. Сообщения элемент – (показан горизонтальными стрелками) – это функциональные сообщения, которыми обмениваются активации (с именем, написанным над ними), в том порядке, в котором они происходят.

Например, на рисунке 19 ниже показана последовательность событий, происходящих в системе онлайн-покупок.

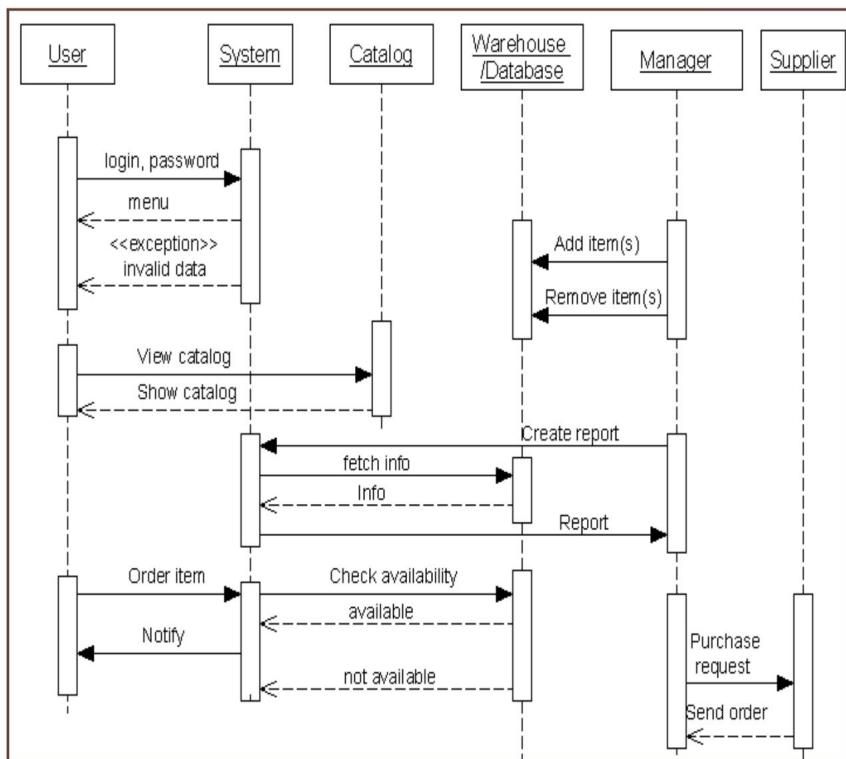


Рисунок 19. Диаграмма последовательности действий – ресторанный система

На диаграмме, представленной на рисунке X, у нас есть 6 объектов: Пользователь, Менеджер, Система, Поставщик, База данных и Каталог. Первый процесс — проверка пользователя и включает в себя два объекта — пользователя и систему. В частности, Пользователь фиксирует активацию авторизации, предоставляя свой логин и пароль.

Глава 6. UML-диаграммы

Система, в свою очередь, обрабатывает эту информацию и в случае правильного ввода логина и пароля пользователю предлагается меню опций. В противном случае возникает исключение из-за неверных данных.

Следующий процесс включает объекты «Пользователь» и «Каталог». Он простой: когда Пользователь желает просмотреть Каталог, ему предоставляется список товаров, выставленных на продажу.

Процедура заказа товара имеет следующий сценарий: пользователь запрашивает конкретный товар через систему, после чего в базе данных проверяется наличие выбранного товара. Здесь возможны 2 случая: товар либо имеется, либо отсутствует. В случае отсутствия такого товара на складе, возможно, Менеджер направит запрос на закупку Поставщику.

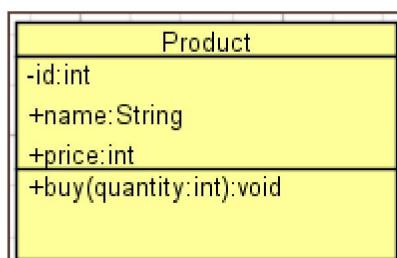
В любом случае Пользователь информируется о наличии товара.

Кроме того, Менеджер может формировать отчеты (по частоте заказов), в этом процессе участвуют 3 объекта: Система (стоящая между базой данных и менеджером), база данных – для получения информации и сам менеджер. Кроме того, Менеджер может сделать запрос на закупку Поставщику (например, после анализа сформированного отчета). Поставщик в ответ отправляет заказанные товары. Последние операции довольно просты: менеджер может управлять элементами и добавлять/удалять их.

6.3 Диаграммы классов

Диаграммы классов являются наиболее часто используемым методом моделирования и представляют собой самую богатую нотацию в UML. Основное назначение диаграммы классов довольно простое — описывать типы объектов в системе и различные виды отношений, которые существуют между ними. Это делается посредством графического представления. Очевидно, что элементы и отношения на диаграмме должны быть статичными.

Как известно, класс — это описание набора объектов со схожими атрибутами (полями) и операциями (методами). Визуально значок класса выглядит так прямоугольник, разделенный на три части: имя класса вверху, атрибуты посередине и операции внизу, как на рисунке 20. ниже:



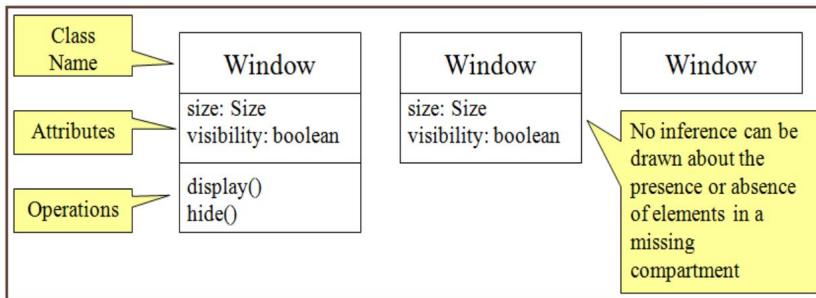


Рисунок 20. Простые значки классов

Важно помнить, что вам не нужно включать все атрибуты и методы класса в конкретную диаграмму (рис. X выше, пример окна). Вместо этого вам просто нужно перечислить те, которые помогут понять концепцию, которую вы хотите объяснить с помощью диаграммы.

Теперь давайте более подробно рассмотрим особенности атрибутов и операций. Проще говоря, атрибут — это свойство, которым обладает объект класса. Они используются для описания состояния объекта. Например, идентификатор, имя и цена могут быть типичными атрибутами класса Product. В большинстве случаев атрибуты соответствуют переменным экземпляра, поэтому они представляют собой атомарные сущности, не имеющие никаких связей. Синтаксис атрибута на значке класса следующий:

[видимость] имя [: тип] [= значение по умолчанию]

Кстати, атрибуты области действия класса (т.е. наличие модификатора static) необходимо подчеркнуть. Давайте возьмем, к примеру, класс Note, имеющий атрибуты автора, текста и атрибут статического типа, отражающий общее количество заметок.

Соответствующий значок класса и код на Java представлены ниже:

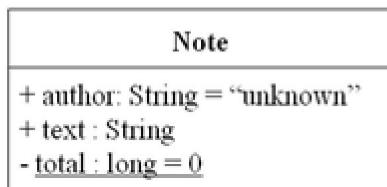


Рисунок 21. Значок класса для примечания (пример для атрибутов)

```
публичный класс Примечание
{
    публичная строка автора = «неизвестно» ;
    общедоступный строковый
    текст; частный статический длинный итог = 0;
    ...
}
```

Теперь перейдем к операциям. Операции с диаграммой классов UML представляют процессы, которые выполняют объекты класса. Как правило, операции соответствуют методам класса. Операции имеют следующий синтаксис:

[видимость] имя ([список параметров]) [: тип возвращаемого значения]

В приведенном выше синтаксисе список параметров представляет собой список формальных параметров, разделенных запятыми, которые имеет класс, каждый из которых должен быть указан с использованием синтаксиса: имя : тип [= значение по умолчанию]. Операции области действия как класса и атрибуты должны быть подчеркнуты.

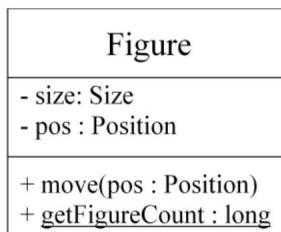


Рисунок 22. Значок класса для рисунка (пример для атрибутов)

```
общественный класс Рисунок
{ частный Размер
размера ; частная позиция
pos; частная статическая длинная фигураCount = 0;

public void move(Position pos) { ... } public static long
getFigureCount()
{ return figCount ; }
...
}
```

- Видимость

Обозначение видимости используется для указания того, является ли атрибут или операция видимым и разрешено ли на него ссылаться из других классов. UML предоставляет четыре типа видимости, соответствующие четырем модификаторам доступа (таблица X):

Тип видимости	Общественный	Частный	Защищено	Упаковка
Обозначения	+	-	#	-

Таблица 8. Обозначения кратности

- Множественность

Иногда может быть полезно добавить логическое обозначение того, сколько объектов можно агрегировать внутри другого объекта. Это называется кратностью и имеет следующие обозначения:

Ровно один	1
Ноль или более (без ограничений)	*(0..*)
Один или больше	1..*
Ноль или единица (необязательная ассоциация)	0..1
Указанный диапазон	2..4
Несколько непересекающихся диапазонов	2,4..6,8

Таблица 9. Обозначения кратности

Обратите внимание, что множественность обеспечивает нижнюю и верхнюю границу количества экземпляров. В качестве простого примера: один парк может включать несколько самолетов, тогда как один коммерческий самолет может содержать от нуля до многих пассажиров.

Очевидно, классы не могут просто перемещаться независимо друг от друга — они должны быть как-то взаимосвязаны. Помимо значков классов, еще одним важным элементом диаграмм классов UML являются отношения. Согласно стандарту UML, классы могут быть связаны друг с другом посредством ассоциации, обобщения, реализации, зависимости, агрегации и композиции. В представленной ниже таблице дано краткое описание типов отношений в диаграммах классов UML:

В целом диаграммы классов очень полезны на этапе проектирования для описания всех классов, пакетов и интерфейсов, составляющих систему, а также того, как эти компоненты связаны друг с другом. Кроме того, четко определенные подробные диаграммы классов могут использоваться в качестве источника для перевода спроектированной системы в

программный код (2). Почти все современные инструменты UML предоставляют эту возможность.

В следующих подразделах мы подробно рассмотрим каждое из этих отношений:

Отношение	Символ	Описание стиля линии
Ассоциация		Твердый Очень общее обозначение, указывающее, что два класса имеют логическую связь.
Обобщение		Твердый Представляет собой «наследование» (между дочерним классом и родительским классом). Подобное к
Зависимость		Пунктирный Изображает отношения «использования». Отношения между двумя классами, в которых изменение одного влияет на объект другого класса
Реализация		Пунктирный Связь между интерфейсом, определяющим набор функций, и классом, реализующим эту функциональность.
Агрегация		Твердый Обозначается «имеет» отношение. Объекты одного класса содержат объекты другого класса.
Состав		Твердый Сильная форма агрегации. Частичное лицо не может жить независимо от своего Владельца .

Таблица 10. Отношения между классами в UML

6.4 Ассоциация

На ранних стадиях проектирования полезно использовать более общий тип отношений между классами, называемый ассоциацией. Один класс связан с другим, если можно переходить от объектов одного класса к объектам другого.

другой класс (т. е. путем доступа к полю экземпляра или выполнения поиска в базе данных).

Ассоциация — это семантическая связь между классами, которая определяет связи между их экземплярами. Основная цель id — указать, что объекты одного класса связаны с объектами второго (это может быть один и тот же класс).

класс) класс.

Типичным примером являются отношения: «Сотрудник работает на 12:

Компания||



Прямая линия между классами указывает на то, что объект на одном конце может «распознавать» объекты на другом конце и может отправлять им сообщения.

Иногда может оказаться полезным уточнить значение ассоциации, дав ей имя и назначив соответствующие роли, которые класс играет на обоих концах пути ассоциации. Имя представлено в виде метки, расположенной в центре линии ассоциации. Имя обычно представляет собой глагол или глагольную группу, тогда как роль обычно представлена существительным или именной группой. Важно отметить, что указание роли обязательно для рефлексивных ассоциаций во избежание путаницы. В качестве иллюстрации рассмотрим следующую диаграмму, изображающую отношения между сотрудниками университета, студентами и курсами.

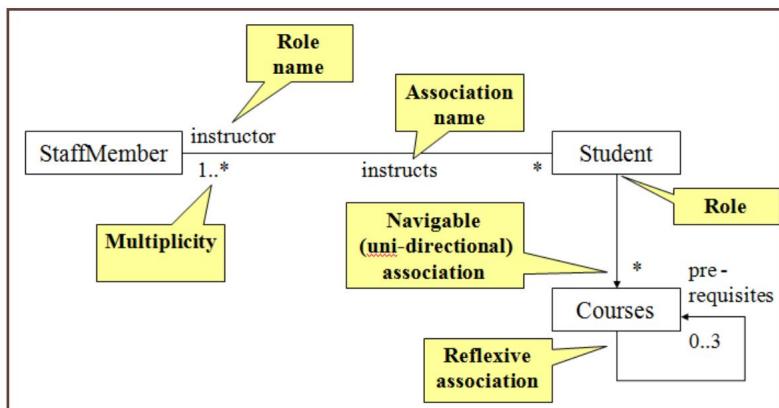


Рисунок 23. Пример связи ассоциации

¹²

Обратите внимание, что здесь и в некоторых последующих диаграммах для простоты мы не включаем поля и методы внутри значка класса, когда в этом нет необходимости

Рефлексивная ассоциация возникает, когда класс может иметь несколько функций или обязанностей.

Мы уже упоминали, что обозначение ассоциации в UML представляет собой сплошную линию. Кроме того, при желании можно добавить стрелки, показывающие, в каком направлении можно перемещаться по взаимосвязи. Например, на диаграмме ниже показано, что объект Bank может перейти к своему клиенту, но не наоборот. Таким образом, в этом конкретном проекте класс Customer не имеет механизма определения, в каком банке он хранит свои деньги.

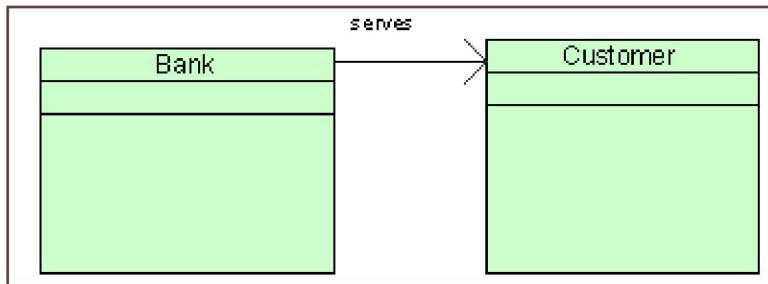


Рисунок 24. Связь между банком и сотрудником

6.5 Обобщение

Отношения обобщения используются для представления наследования. Так, это указывает на то, что объекты специализированного класса (подкласса) являются объектами обобщенного класса (суперкласса).

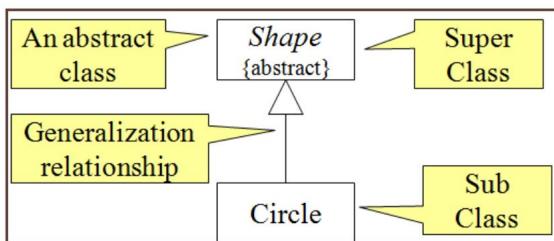


Рисунок 25. Пример отношения обобщения

Обратите внимание: если класс является абстрактным, вы должны указать значение тега `{abstract}` и выделить имя абстрактного класса курсивом.

Кроме того, давайте добавим некоторые детали к нашим классам. Кажется разумным добавить абстрактный метод `draw()` в класс `Shape` и неабстрактный метод `draw()` в круг:

Чтобы отобразить наследование на диаграмме UML, сплошная линия от дочернего элемента класс родительского класса рисуется с помощью незаполненной стрелки.

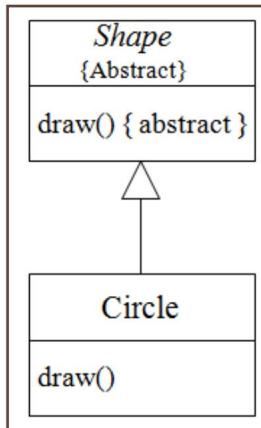


Рисунок 26. Пример отношения обобщения (с методами)

Пример кода для этой диаграммы:

```

публичный абстрактный класс Shape
{
    публичная абстрактная недействительная ничья();
    ...
}

общественный класс Circle расширяет форму {
    общественная недействительная ничья() { ... }
    ...
}
  
```

6.6 Зависимость

Зависимость — это отношение между двумя классами, в котором изменение одного может повлиять на другой класс и вызвать изменения, хотя между ними нет явной связи. Стереотип может использоваться для обозначения

Глава 6. UML-диаграммы

типа зависимости. Пример: класс вызывает операцию области действия другого класса.

Как показано на следующем рисунке, зависимость отображается в редакторе диаграмм в виде пунктирной линии с открытой стрелкой, указывающей от элемента модели клиента к элементу модели поставщика. По соглашению отношения зависимости не имеют имен.

В качестве примера рассмотрим приложение электронной коммерции, где элемент «Корзина» class зависит от класса Product, поскольку Basket использует Product в качестве параметра для операции addProduct (рис. 27). Таким образом, на диаграмме классов отношение зависимости должно указывать от класса Basket к классу Product. сорт. Эта связь указывает на то, что изменение класса «Продукт» может потребовать изменения класса «Корзина» .

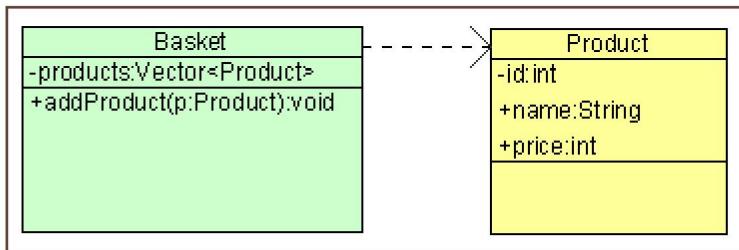


Рисунок 27. Пример отношений зависимости

В связи с тем, что отношение зависимости может указывать на несколько различных типов отношений, для отображения точного значения зависимости используются стереотипы (своего рода ключевое слово). Обычно они пишутся в центре линии отношений. Наиболее часто используется стереотипы представлены ниже (7):

- <абстракция> — используется, когда элементы представляют одну и ту же концепцию на разных уровнях абстракции или с разных точек зрения.
- <substitute> — указывает, что один элемент может заменить другой элемент
- <use>, <call> — указывает, что один элемент требует другого элемента для полноценной работы.
- <экземпляр> — один элемент может быть создателем другого элемента.

6.7 Реализация

Отношение реализации указывает, что один класс реализует поведение, заданное другим классом или интерфейсом. Для последовательного использования отношений реализации давайте восстановим два основных принципа отношений между классами и интерфейсами:

- Интерфейс может быть реализован множеством классов.
- Класс может реализовывать множество интерфейсов.

Например, класс `LinkedList` из пакета `java.util` реализует интерфейс `List` из того же пакета. Вместе они представляют отношения реализации (рис. X ниже):

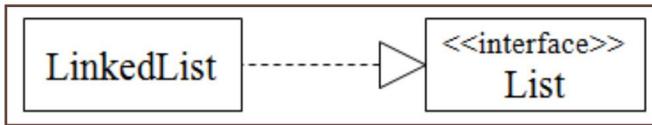


Рисунок 28. Пример отношения реализации

Исходный код, соответствующий этому UML:

```

Список общедоступных интерфейсов
{
    логическое добавление (Объект o);
    ...
}
публичный класс LinkedList реализует List {

    public boolean add(Object o) { ... }
    ...
}
  
```

6.8 Агрегация

Отношения агрегации — это особая форма ассоциации, которая моделирует отношения целое-часть между агрегатом (целым) и его частями. Таким образом, оно соответствует отношениям «имеет» или «является частью» отношений.

Например, класс «Отдел» может иметь отношение агрегации с классом «Компания», что указывает на то, что отдел является частью компании.

В этом типе отношений данные передаются от всего агрегата к его части. Часть может принадлежать более чем одному агрегату. Хотя агрегация тесно связана с композицией, между ними есть важное различие — в отношениях агрегации сущность-часть может существовать независимо от агрегата, в отличие от композиции (обсуждаемой далее в этой главе).

Следует признать, что отношения агрегации не обязательно должны быть односторонними. Они также могут быть двунаправленными (без стрелок на конце). Ассоциация агрегации выглядит как сплошная линия с незаполненным ромбом на конце связи, которая связана с сущностью, представляющей агрегат. Пример приведен ниже:

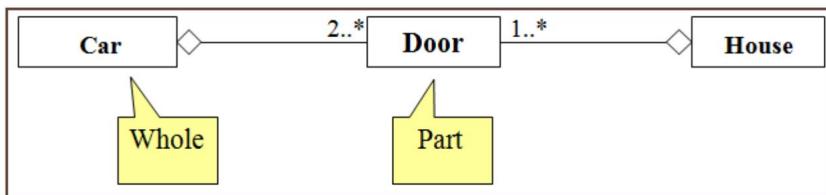


Рисунок 29. Пример отношения агрегации

Существует ряд вопросов, которые вы можете использовать, чтобы проверить, правильно ли вы выбрали тип отношений. Для агрегирования они следующие (для пояснения вопросов воспользуемся примером с рис. 29):

- Подходит ли словосочетание «часть» для описания отношений?
Да, поскольку дверь является частью автомобиля
- Отражаются ли некоторые операции в целом автоматически на своем части?
Да. Например, когда машина движется, дверь тоже движется.
- Распространяются ли некоторые значения атрибутов целого на все или некоторые его части?
Да. Например, если машина красная, дверь тоже красная.
- Есть ли асимметрия в отношениях?
Для этого вам нужно проверить, является ли это частью отношений. после обмена целого и части. Это не должно выполняться для агрегации. Например: дверь — часть автомобиля, но машина — не часть двери.

Код Java для этого отношения агрегации представлен ниже:

```
Автомобиль общественного класса {
    частные двери Vector<Door> = новый Vector<Door>();
    public void addDoor(Дверь) { ... }
    ...
}
public static void main(String[] args) {

    Дверь дверь = новая дверь();
    Дом дом = новый дом(дверь);
    Автомобиль автомобиль = новый автомобиль();
    car.addDoor(дверь);
    ...
}
```

Другим примером агрегирования являются отношения «Компания – отдел» :

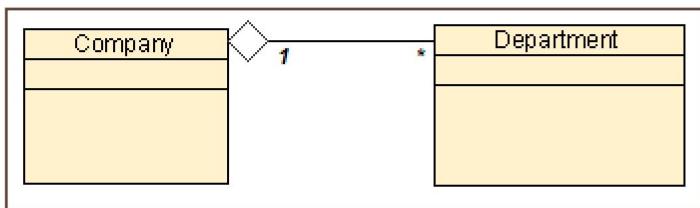


Рисунок 30. Пример отношения реализации

Опять же, самое запутанное в агрегации то, что ею, как и почти любыми отношениями, можно управлять. Помните, что направленная ассоциация указывает на то, что управление передается от одного классификатора к другому. Другими словами, вы можете перемещаться от одного конца ассоциации к другому (т.е. если вы знаете компанию, вы можете получить доступ к отделам), но не наоборот.

6.9 Состав

Отношения композиции тесно связаны и очень похожи на агрегацию. Проще говоря, композиция — это сильная форма агрегации, в которой целое является единственным владельцем своей части. Таким образом, объект детали может только

Глава 6. UML-диаграммы

принадлежат только одному целому. Это означает, что кратность на всей стороне должна быть равна нулю или единице.

Композит (владелец) несет ответственность за управление формированием и разрушением его частей. Таким образом, время жизни частичного объекта зависит от целого.

Как правило, в отношениях композиции данные передаются только в одном направлении — от целого классификатора к его части. Например, в кредитной системе образования отношение композиции может связывать Класс Student с классом Schedule, что подтверждается тем фактом, что если вы удалите студента, расписание также будет удалено.

Отношения композиции иллюстрируются сплошной линией, соединяющей два класса с заполненным ромбом, расположенным на конце ассоциации, который соединен с составным (целым) объектом.

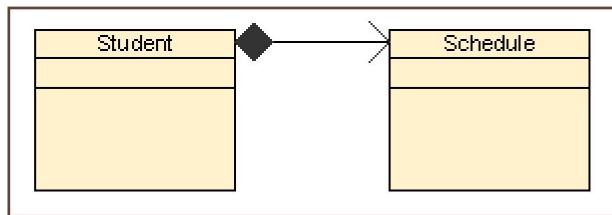


Рисунок 31. Пример отношения композиции

Важно понимать, что в случае уничтожения класса-контейнера (владельца), будет убит и класс-часть. Например, в случае порчи сумки это сделает и ее боковой карман.

Ниже представлены еще несколько примеров композиции: между классом Circle и его центром — объектом класса Point (рис. 32), а также между стандартным компьютерным окном и его частями — слайдером, заголовком и панелью (рис. 33).

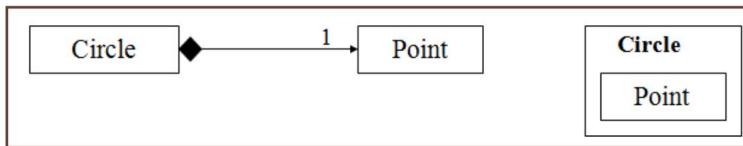


Рисунок 32. Пример отношения композиции — круг и точка

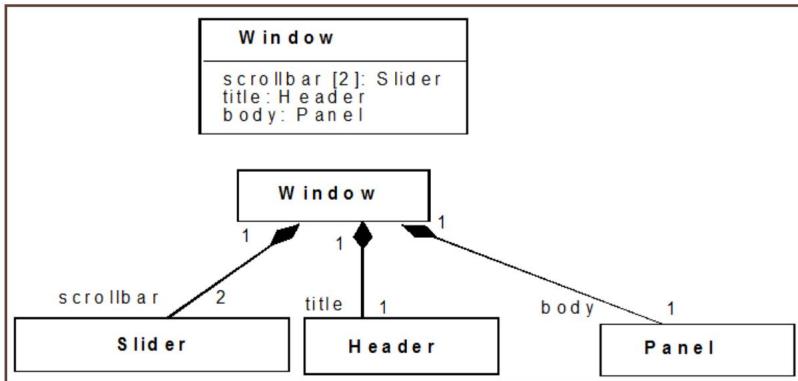


Рисунок 33. Пример композиционных отношений – Окно и его части

6.10 Важные вопросы – множественность, ограничения и примечания

Ограничения и примечания — важные инструменты моделирования UML. Их можно использовать для аннотирования ассоциаций, атрибутов, операций и классов. Следует признать, что ограничения — это семантические ограничения, выраженные в виде выражений.

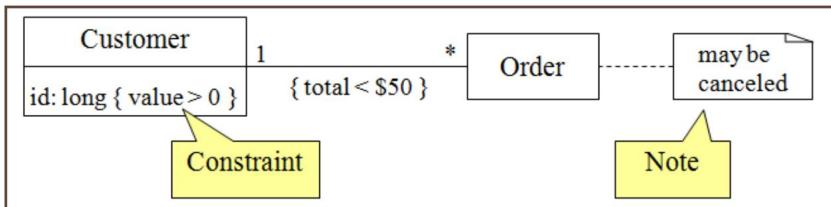


Рисунок 34. Пример ограничений и примечаний

Ограничение, показанное на рисунке 34 (значение $id > 0$), может быть реализовано, например, в конструкторе или методе установки поля **id**. Примечание на рисунке X представляет собой примечание, связанное с элементом модели. Он также может стоять независимо и комментировать определенную часть диаграммы.

6.11 UML-пакеты

Как вы уже знаете из предыдущих глав, пакет представляет собой механизм группировки общего назначения. Его функции при UML-моделировании очень похожи на те, которые вы используете при кодировании системы — они обычно используются для указания логического распределения классов и других элементов в системе. Итак, вы используете Package , когда хотите сгруппировать вместе любой элемент UML (например, вариант использования, актеры, классы, компоненты и другие пакеты).

В UML пакеты можно представить так просто:

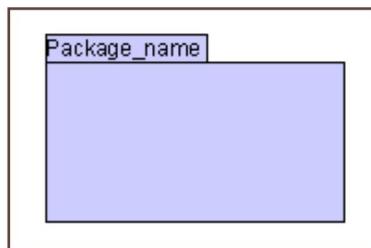


Рисунок 35. Представление пакета UML

Разработчики обычно подчеркивают логическую структуру системы, обеспечивая различные связи (например, зависимости, ассоциации) между пакетами. Это можно рассматривать как общий вид системы (рис. 36):

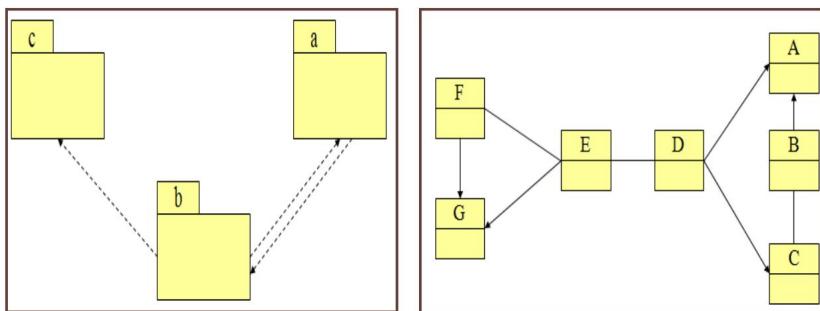


Рисунок 36. Указание связей между пакетами

Очевидно, что вы можете создавать отношения и зависимости между открытыми классами из разных пакетов аналогичным образом (рис. 37). Это полезно, поскольку помогает подчеркнуть интерфейс между пакетами.

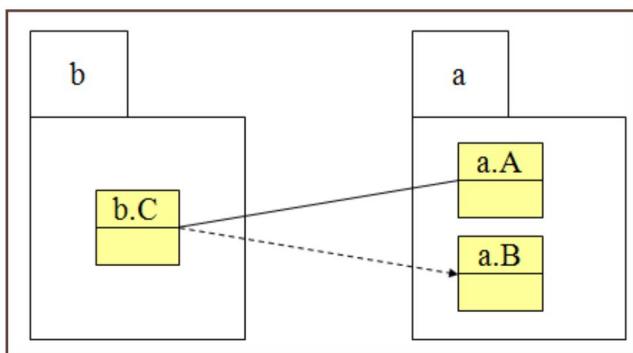


Рисунок 37. Указание связей между классами из разных пакетов

6.11 Советы

Полезные советы, представленные ниже, помогут вам не запутаться в различные методы моделирования UML и последовательно использовать их:

- Никогда не пытайтесь использовать все возможные соотношения и обозначения.
- Не рисуйте модели для каждой детали, лучше сосредоточьтесь на важнейших аспектах системы, которые важны для ее правильного функционирования.
- Страйтесь рисовать модели реализации только в том случае, если вы хотите проиллюстрировать конкретную технику реализации.

Наконец, давайте рассмотрим шаги, которые необходимо выполнить при моделировании UML:

- Создайте символ класса в используемом вами редакторе (например, TopCoder UML Tool) и назовите его.
- Укажите атрибуты класса
- Укажите операции класса
- Укажите связи между классами
- При необходимости предоставьте комментарии, обозначения и ограничения.

6.12. Краткое описание отношений классов

Вы изучили различные типы отношений классов и варианты их использования.

Однако вам нужно помнить, что это сильно зависит от контекста, поэтому для одних и тех же понятий в разных контекстах могут существовать разные отношения. Следовательно, все примеры должны быть специфичными для предметной области, поскольку с другой точки зрения ассоциация может стать более конкретной.

Для окончательного закрепления такой важной темы, как отношения классов, давайте рассмотрим несколько примеров отношений:

- Ассоциация – «использует»

Означает, что два класса имеют какие-то отношения, могут быть отношениями.

любой Пример: класс Human использует класс Pen

- Агрегация: «имеет

а»

Пример: класс Human имеет класс Car. Обратите внимание, что в случае Human умирает Автомобиль продолжит существовать.

- Состав: «содержит
- Пример: класс Человек владеет классом Сердце. Сердце не может существовать отдельно без Человека.

- Обобщение: —есть

а»

Пример: ученик класса является человеком класса.

Очень часто новички путают композицию и агрегацию. Самый простой способ их отличить — подумать о том, насколько крепка связь. В частности, подумайте о том, что произойдет, если вы удалите объект-владелец. В случае агрегации объект-часть продолжает жить. Например, после отмены объекта «Заказ» объект «Продукт» продолжает существовать. Что касается состава, то часть

объект умирает вместе со своим владельцем. Например, объект «Абзац» умирает вместе с соответствующим объектом «Документ» .

Надо признать, что композиция, агрегация и ассоциация — это семантические, а не программные концепции. В Java вы можете реализовать их все одинаково. Это концептуальная разница.

Как правило, старайтесь делать диаграммы классов как можно более простыми. Это значительно облегчит понимание диаграмм. Кроме того, не забудьте пометить свои классы и отношения, присвоив им описательные значения. имена.

Хотя классовые отношения кажутся очень простыми, будьте уверены, это только на первый взгляд. Когда работаешь над сложным проектом, как на рисунке X, очень легко заблудиться в таком обилии классов.

Объектно-ориентированное программирование и дизайн

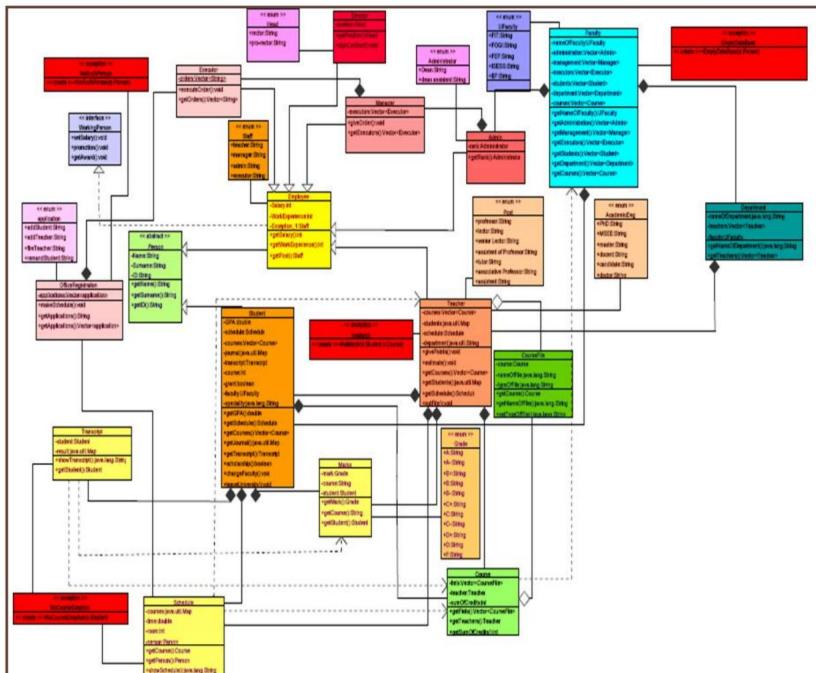


Рисунок 38. Сложная UML-диаграмма

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Диаграмма вариантов использования		Роль актера в диаграмме последовательности	
Активация	Сообщение	Диаграмма классов	Ограничения
Обобщение	Ассоциация	Зависимость реализации	
Агрегация	Состав	Множественность	Видимость

испытания

1. Какое соединение на схеме... ?



- а) Ассоциация б) Зависимость

- в) Агрегация г) Состав

2. В чем разница между композицией и агрегацией??

- а) Вся композиция должна иметь кратность 0..1 или 1.

- б) Часть должна принадлежать только одному целому

- в) Деталь может иметь любую кратность.

- г) а и б верны

3. Какое из приведенных ниже отношений соответствует отношениям Человек-Студент, Человек-Профессор?

- а) Ассоциация б) Обобщение

- в) Реализация г) Состав

4. Сильная форма агрегации?

- а) Ассоциация б) Направленное объединение

- в) Обобщение г) Состав

5. Связь, указывающая, что один класс реализует поведение, заданное интерфейсом:

- а) Реализация б) Направленное объединение

- в) Обобщение г) Состав

6. Отношение агрегации наиболее точно описывает отношения между...

- а) Ты и твои руки

- б) Ты и твои друзья

- в) Ваша комната и комната ваших соседей

7. Особая форма ассоциации, моделирующая отношения «целое-часть» между агрегатом и его частями?

- | | |
|--------------|---------------|
| а) Агрегация | б) Ассоциация |
| в) Обобщение | г) Состав |

8. Выберите отношения на диаграмме классов UML, которые указывают на наследование?

- | | |
|---------------|---------------|
| а) Обобщение | б) Ассоциация |
| в) Реализация | г) Состав |

9. Каковы отношения между Отделом и Сотрудником согласно приведенному ниже коду?

```
Отдел общественного класса {
    частный внутренний идентификатор;
    частные сотрудники Set<Employee>;
}
```

```
общественный класс Сотрудник {
    частный внутренний идентификатор;
    частное ИМЯ строки;
}
```

- | | |
|---------------|--------------|
| а) Ассоциация | б) Обобщение |
| в) Реализация | г) Агрегация |

10. В чем разница между диаграммами вариантов использования и диаграммами последовательности?

- | | |
|--|---|
| а) Диаграмма вариантов использования представляет собой функциональную модель, тогда как диаграмма последовательности
Диаграмма – динамическая модель | б) Диаграмма вариантов использования более подробная. |
| в) Диаграмма последовательности более понятна бизнес-пользователям. | |
| г) Диаграмма вариантов использования представляет собой объектную модель, тогда как последовательность
Диаграмма – динамическая модель | |

ПРОБЛЕМЫ

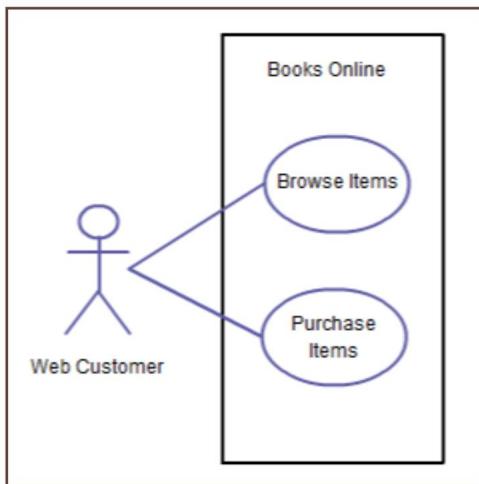
- Чтобы изобразить треугольник, нам нужны координаты каждого из трех его сторон (поэтому вам нужен класс Point) и методы построения треугольника, вычисления площади и периметра. Нарисуйте диаграмму классов UML, чтобы представить этот треугольник и любые другие задействованные классы.
 - Вам нужно написать программу, которая распечатывает счет. Счет-фактура используется для описания расходов на набор продуктов в определенных количествах. Для простоты мы опускаем некоторые сложности, такие как даты, налоги, номера клиентов и т. д. Программе просто нужно распечатать адрес для выставления счета, все позиции и сумму к оплате. Каждый товар имеет описание (название товара) и цену за единицу товара, заказанное количество и общую стоимость.
- Пример представлен ниже:

<i>INVOICE</i>			
Description	Price, \$	Q-ty	Total, \$
Toaster	29.95	3	89.95
Hair dryer	24.95	1	24.95
Microwave oven	19.99	2	39.98

AMOUNT DUE : \$ 154.78

Создайте диаграмму классов UML, а затем реализуйте и протестируйте систему.

- Проанализируйте простую диаграмму вариантов использования, представленную ниже, и создайте аналогичные диаграммы последовательностей и классов.



4. Проанализируйте диаграмму вариантов использования, представленную на рисунке 39, и внедрить соответствующую систему.
5. Проанализируйте диаграмму вариантов использования, представленную на рисунке 40, и внедрить соответствующую систему.
6. Разработайте систему для предприятия здравоохранения.

В частности, вам необходимо разработать внутреннее приложение, которое врачи и другие сотрудники будут использовать для управления информацией о клиентах, их текущих планах, графиках, лекарствах, назначенных врачами и т. д. Учитывайте следующие детали:

1. Доктор может уйти на пенсию
2. В случае, если какой-то врач уйдет на пенсию, система должна помочь пользователю найти нового врача.
3. К врачам необходимо обращаться ежегодно для продления контрактов. Как сохранить в приложении группу клиентов или врачей?
4. Есть ли способ написать единый метод, который будет обрабатывать адреса как клиентов, так и врачей? Как вы это реализуете?

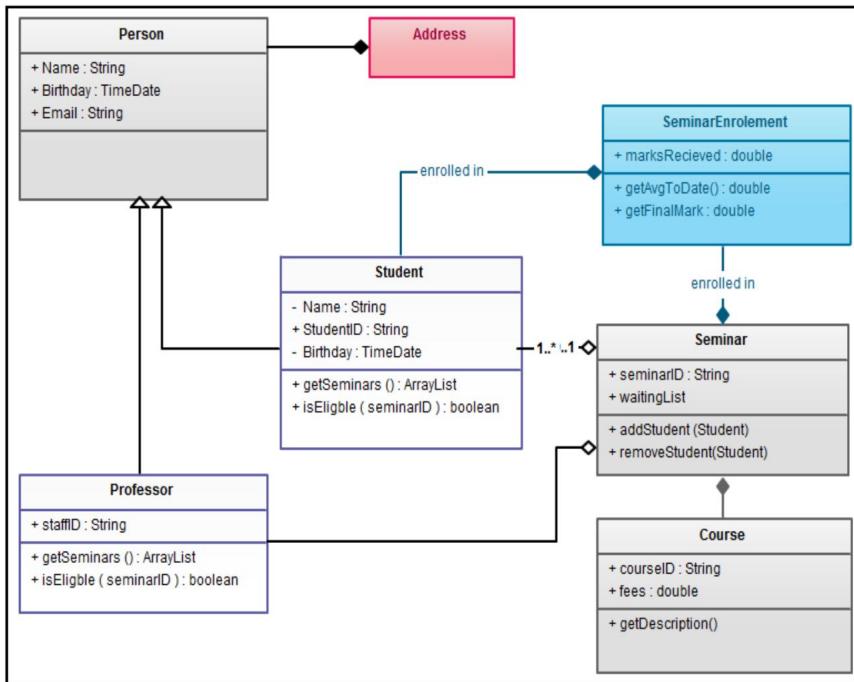


Рисунок 39. Диаграмма классов для задачи 313.

7. Как известно, существует два типа языков: естественный язык и язык программирования. Языки программирования можно разделить на объектно-ориентированные, императивные, логические и функциональные. Естественные языки также можно разделить на тюркские, европейские, фарси и др. Подумайте, как вы можете представить иерархию языка и нарисовать эту иерархию на диаграмме классов UML.
8. Создайте диаграмму классов UML, которую можно использовать для реализации электронной адресной книги. Подумайте хорошенъко, какие классы требуются такому приложению и какую функциональность оно должно предоставлять. Затем реализуйте это приложение.

¹³Схемы с рисунков X, XX взяты с сайта www.creately.com.

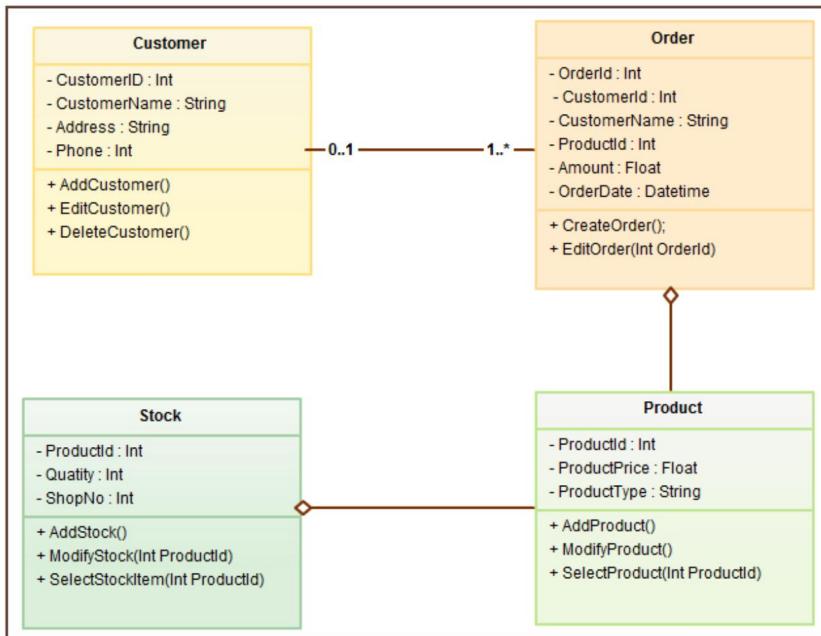


Рисунок 40. Сложная UML-диаграмма

9. Очередь — это абстрактный тип данных для добавления и удаления элементов.

Первый элемент, добавляемый в очередь, является первым элементом, который удаляется (первым пришел — первым обслужен, FIFO).

Разработайте интерфейс `AbstractQueue` с методами добавления и удаления элементов (целых чисел). Вам также необходим метод проверки того, пуста ли очередь. Затем создайте класс `Queue`, реализующий `AbstractQueue`. Прежде чем писать код, создайте соответствующую диаграмму классов UML. Затем протестируйте свою реализацию с маленькими и большими очередями.

Примечание. Реализуйте очередь с помощью массива или вектора. Если вы выбрали массив: если массив становится слишком маленьким, чтобы вместить все добавленные элементы, создайте новый массив большего размера (вдвое больше размера экземпляра) и скопируйте все элементы маленького массива в новый.

10. Напишите абстрактный класс `MathExpression`, представляющий некоторое математическое выражение неопределенного типа. Определите абстрактный метод `findDerivative()` для возврата производной выражения.

Глава 6. UML-диаграммы

Затем создайте класс `Polynomial`, который расширяет `MathExpression` и представляет полином с действительными коэффициентами.

Коэффициенты полинома следует передавать как параметр массива с типом массива `double` в конструкторе вашего класса. Кроме того, предоставьте методы `Equals()` и `CompareTo()`, которые сравнивают полиномы по степени.

Подсказка: результирующий полином определяется
коэффициентами[0] + коэффициенты[1] * x + ... +
коэффициенты[2] * x² + коэффициенты[n] * xⁿ

7 Collections and Data Structures

Джava Collection Framework — это унифицированная архитектура, используемая для представления и манипулировать различными коллекциями. В этой главе представлены основы Java Collection Framework, предоставляя обзор интерфейсов и конкретных классов в этой среде. Помимо простых массивов, ученые-компьютерщики обнаружили множество структур данных, имеющих разные компромиссы в производительности.

В общем, коллекция представляет собой структуру данных (сам объект), которая используется для хранения других объектов, для хранения, манипулирования и организации объектов полезными способами для эффективного доступа (8). Как правило, элементы коллекции представляют собой элементы данных, образующие естественную группу, например набор карточек, телефонный справочник, почтовая папка и т. д. Если вы проверите пакет java.util, вы найдете множество интерфейсов и классов, предоставляющих общую информацию. каркас коллекции. Они позволяют эффективно группировать несколько элементов в один блок.

Ниже вы можете увидеть классы и интерфейсы, составляющие фреймворк.
Как видно, существует один корневой интерфейс — Collection. Он имеет два дочерних интерфейса. Set (неупорядоченная коллекция уникальных элементов) и List (упорядоченная коллекция, которая может иметь дубликаты). Карта — это еще один корневой интерфейс для всех коллекций, хранящих пары ключ-значение. Мы рассмотрим их подробно позже в этой главе.

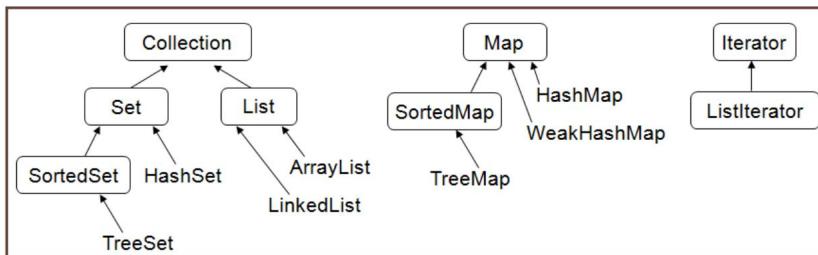


Рисунок 41. Иерархия коллекций Java

7.1 Корневой интерфейс — Коллекция

На самом деле Коллекция — это наименьший общий знаменатель, который объединяет все коллекции. осуществлять. Следовательно, каждый объект коллекции (кроме карт) является типом интерфейса коллекции . Это чрезвычайно полезно, поскольку мы можем использовать Collection

Глава 7 – Коллекции и структуры данных

интерфейс как тип и передавать объекты коллекции в качестве параметров и манипулировать ими, когда требуется максимальная общность (помните полиморфизм?).

Java не предоставляет никаких встроенных прямых реализаций Collection.

интерфейс, но он предоставляет ряд реализаций для более конкретных интерфейсов, расширенных из Collection, таких как Set и List. В таблице 11 ниже представлены основные абстрактные методы интерфейса Collection:

Подпись	Описание
public int size()	Возвращает размер коллекции
публичное логическое значение isEmpty()	Возвращает логическое значение, указывающее, находится ли элемент в коллекции или нет.
общедоступное логическое значение содержит (элемент, объекта)	Возвращает логическое значение, указывающее, находится ли элемент в коллекции или нет.
public boolean add(Object elem)	Зависит от того, допускает ли коллекция дубликаты
общедоступное логическое удаление (элемент объекта)	Удаляет указанный элемент
публичный итератор iterator()	Возвращает итератор для этой коллекции
публичная недействительная очистка()	Удалить все элементы из коллекции
общедоступный объект [] toArray()	Возвращает новый массив, содержащий ссылки на все элементы коллекции.

Таблица 11. Методы интерфейса коллекции

Помимо методов из таблицы 11, интерфейс коллекции также содержит методы, работающие совместно с другим объектом коллекции (таблица 12):

Подпись	Описание
public boolean containsAll(Collection col)	Возвращает размер коллекции
общедоступное логическое значение addAll (коллекция коллекций)	Возвращает true, если какое-либо добавление выполнено успешно. Что это за логическая операция?
общедоступное логическое значение removeAll (коллекция коллекций)	Возвращает true, если какое-либо удаление выполнено успешно. Что это за логическая операция?
общедоступное логическое значение saveAll (коллекция коллекций)	Удаляет из коллекции все элементы, не являющиеся элементами Col. Что это за логическая операция?

Таблица 12. Методы интерфейса коллекции

7.2 Итератор и ListIterator

Как вы уже знаете, интерфейс Collection определяет итератор().

метод для возврата объекта, реализующего интерфейс Iterator. Итераторы могут получать доступ к элементам коллекции, не раскрывая ее внутреннюю структуру (9). Однако он не дает никаких гарантий относительно порядка возврата элементов.

В интерфейсе Iterator есть три определенных метода:

- public boolean hasNext() – возвращает true, если итерация имеет больше элементов
- public Object next() – возвращает следующий элемент в итерации.
Если следующего элемента нет, будет выдано исключение. Обратите особое внимание на тот факт, что next() возвращает объект, поэтому вам может потребоваться специальное приведение типов.
- public void Remove() – удалить из коллекции последний элемент возвращается итерацией

можно вызывать только один раз за вызов next, в противном случае выдается исключение.

Листинг кода ниже демонстрирует классический пример использования итератора. Обратите внимание, что этот метод принимает любой объект, реализующий коллекцию. Интерфейс.

```
public void removeBigStrings (Collection col, int max) {
    Итератор it = col.iterator();
    в то время как (it.hasNext())
        Страна str = (String)it.next();
        если (str.length() > макс)
            это.удалить()
    }
}
```

ListIterator — это подинтерфейс интерфейса Iterator. Он расширяет своего родителя, добавляя методы для управления упорядоченным объектом List (например, LinkedList или Vector) во время итерации. В частности, вы можете удалить последний возвращенный элемент (метод Remove()), вставить объект в список после последнего возвращенного элемента (метод add()), установить последний возвращенный элемент другим объектом (метод set()).

Кроме того, помимо наследования от Iterator методов hasNext(), next(), ListIterator также имеет методы hasPrevious(), previous(), nextIndex(), previousIndex(). Важно знать, что когда итератор находится в

Глава 7 – Коллекции и структуры данных

В конце списка nextIndex() вернет list.size(), а когда он находится в начале списка, previousIndex() вернет -1.

С Iterator/ListIterator существует одна проблема : они не гарантируют снимок¹⁴ коллекции. Это означает, что если содержимое коллекции будет изменено во время использования итератора, это повлияет на значения, возвращаемые методами. Листинг кода ниже демонстрирует такую ситуацию:

```
ArrayList a = новый ArrayList();
    a.добавить("1");
    a.добавить("2");
    a.добавить("3");
Итератор it = a.iterator();
    в то время как (it.hasNext ()) {
        Страна s = (String)(it.next());
        если(s.equals("1")) {
            a.set(2,"изменено");
        }
        System.out.print(s+ " ");
    }
```

Результатом будет: «1 2 Changes» , хотя в случае снимка у нас должно быть «1 2 3» . Поэтому, если вам действительно нужен снимок коллекции, вам необходимо сделать копию коллекции.

Следует признать, что после возврата объекта Iterator невозможно изменить коллекцию до полного обхода коллекции. Если вы это сделаете, будет создано исключение ConcurrentModificationException , как в примере ниже:

```
ArrayList a = новый ArrayList();
    a.добавить("1");
    a.добавить("2");
    a.добавить("3");
Итератор it = a.iterator();
    a.добавить("4"); // сейчас нельзя изменить!
    в то время как (it.hasNext ()) {
        Страна s = (String)(it.next());
        System.out.println(s);
    }
```

Исключение в потоке «основной» java.util.ConcurrentModificationException

¹⁴ Шапшот возвращает элементы коллекции в том состоянии, в котором они были, когда Итератор объект был создан.

Что касается обхода коллекции, то для этого существуют две общие схемы:

- Итератор (уже показан)
- для каждого

Конструкция `for-each` позволяет кратко перемещаться по коллекции или массиву с помощью цикла `for`. Общий синтаксис для этого следующий:

```
for (Объект o: коллекция)
    System.out.println(o);
```

Например, если у вас есть `HashSet` объектов `Student` (`HashSet<Student>`), вы можете пройти его следующим образом:

```
для (Студент: myHashSet)
    System.out.println(s);
```

Используйте итераторы вместо `foreach`, если вам нужно удалить текущий элемент.

7.3 Интерфейс списка

Мы уже упоминали `Vector` в предыдущих главах, поэтому основы `List` Интерфейс должен быть вам знаком. Список — это упорядоченная коллекция, которая позволяет хранить повторяющиеся элементы вместе. Индексы списка варьируются от 0 в `list.size()-1`. Список позволяет точно контролировать, куда вставляется каждый элемент, и позволяет получать доступ к элементам по их положению.

Благодаря своим особенностям `List` имеет несколько новых методов для упорядоченной коллекции, которые позволяют:

- Позиционный доступ

`get(int index)`, `set(int index, элемент E)`, `add(int index, элемент E)`.

- Поиск (возвращает индекс)

`indexOf(Object o)`, `LastIndexOf(Object o)`.

Интерфейс `List` реализован следующими классами:

- `ArrayList` и вектор

Наиболее часто используемая реализация на практике. Проще говоря, это реализация интерфейса `List` с изменяемым размером массива. Предпочитайте `Vector`, если вам нужен быстрый позиционный доступ в постоянное время. Однако, если вам нужно часто добавлять или удалять элементы из середины, отдайте предпочтение `LinkedList`, поскольку `Vector` для этого требуется $O(n)$. Еще одно удобство `Vector` (или `ArrayList`)

Глава 7 – Коллекции и структуры данных

заключается в том, что его можно эффективно сканировать без необходимости создания итератора. объект.

Кстати, `ArrayList` и `Vector` идентичны друг другу с той лишь разницей, что методы `ArrayList` не синхронизированы, в отличие от `Vector`.

Углубляясь в иерархию, можно найти класс `Stack`, расширяющий `Vector`. Он имеет следующие типичные методы:

`peek()` — возвращает объект наверху этого стека, не удаляя его.

`pop()` — удаляет объект сверху этого стека и возвращает

это

`push(E item)` — помещает элемент на вершину этой стопки.

- Связанный список

Отдавайте предпочтение `LinkedList`, если вы знаете, что будете часто добавлять элементы в начало списка или перебирать список, чтобы удалить элементы из его внутренней части. Но получение элемента по определенному индексу для `LinkedList` обходится дороже — $O(i)$. Кстати, `LinkedList` в Java представляет собой двусвязный список. `LinkedList` также реализует интерфейс `Queue`, предоставляя операции очереди «первым пришел — первым обслужен» (FIFO) для добавления, опроса, просмотра, удаления¹⁵ и т. д. Наконец, имейте в виду, что `LinkedList` должен выделять объект узла для каждого своего элемента, поэтому вы платите большую цену за производительность. Так что используйте его при необходимости.

Помните, что позиционный доступ требует линейного времени в `LinkedList` и постоянного времени в `Vector` или `ArrayList`.

7.4 Интерфейсы `Set` и `SortedSet`

Интерфейс `Set` предоставляет те же методы, что и интерфейсы `Collection`, но предоставляет для своих методов более конкретный контракт (9). В частности, в набор нельзя добавить повторяющийся элемент, поскольку он содержит УНИКАЛЬНЫЕ элементы.

Интерфейс `Set` имеет дочерний интерфейс `SortedSet`, который расширяет `Set` и предоставляет дополнительный контракт — итераторы, возвращаемые из этого набора, возвращают элементы набора в указанном порядке. По умолчанию это будет указанный порядок элементов. за счет реализации интерфейса `Comparable` (для предопределенных типов, таких как `String` и `Date`, элементы хранятся в типичном алфавитном или хронологическом порядке, поскольку `String` и `Date` оба реализуют интерфейс `Comparable`).

15

Разница между опросом и удалением заключается в том, что в случае, если стек пуст, опрос выдаст исключение, а удаление вернет ноль.

интерфейс, который позволяет автоматически сортировать объекты этого класса). Кроме того, вы можете указать объект Comparator с целью упорядочивания элементов вместо естественного порядка, заданного методом CompareTo().

Существует две реализации интерфейса Set, которые широко используются в повседневном программировании:

- HashSet — реализует Set, реализованный с использованием хеш-таблицы.
- TreeSet — реализует SortedSet. • , использует сбалансированную древовидную структуру
- LinkedHashSet — реализуется как хеш-таблица со связанным списком, проходящим через нее.

Как правило, HashSet работает быстрее, чем TreeSet, обеспечивая для большинства операций постоянное время, а не время журнала. Тем не менее, HashSet не дает никаких гарантий упорядочивания, а TreeSet сохраняет элементы отсортированными. Итак, недостаток HashSet хаотичный порядок, у TreeSet — худшая производительность.

LinkedHashSet считается промежуточным между HashSet и TreeSet, поскольку он предлагает итерацию в порядке вставки и почти так же быстр, как HashSet.

Ниже приведен простой пример HashSet. У нас есть мешочек с фруктами (массив), в котором фрукты могут повторяться. Что, если мы хотим узнать только виды фруктов в сумке? Чтобы добиться этого, нам просто нужно перебрать массив и попытаться добавить каждый фрукт в набор.

```
String [] BagOfFruits = {"яблоко", "апельсин",
                        "слива", "слива", "груша", "яблоко"};
HashSet<String> fruits = new HashSet<String>(); for(Строка фруктов:
BagOfFruits)
    System.out.println(fruit+ " " (fruits.add(fruit)?      +
        "добавлено!" : "не добавлено");
System.out.println("Типы фруктов в сумке: "+ фрукты);
```

Метод Add() возвращает логическое значение, указывающее на успешность сложения. Итак, вывод программы будет:

```
добавлено яблоко!
добавлен оранжевый!
добавлена слива!
слива не добавлена
груша добавлена!
яблоко не добавлено
Виды фруктов в пакете: [апельсин, яблоко, груша, слива]
```

Глава 7 – Коллекции и структуры данных

Обратите внимание: если бы в этом примере мы использовали TreeSet вместо HashSet, не было бы никакой разницы, за исключением того факта, что фрукты будут печататься в алфавитном порядке.

В этом примере в качестве ключей мы использовали значения предопределенного типа String, поэтому HashSet отказался от добавления дубликатов (класс String имеет методы hashCode() и equals()). А как насчет пользовательских типов? Давайте обернем фрукты в класс Fruit:

```
класс Фрукты {
    Строковое имя;
    public Fruit (имя строки) {
        это.имя = имя;
    }
    публичная строка toString(){
        вернуть имя+"";
    }
}
```

Теперь посмотрим, что произойдет, если мы поместим в набор дубликаты фруктов:

```
HashSet<Fruit>fruits = новый HashSet<Fruit>();
Fruits.add(новый Fruit("яблоко"));
Fruits.add(new Fruit("банан"));
Fruits.add(новый Fruit("яблоко"));
System.out.println(фрукты);
```

Результат: [яблоко, яблоко, банан].

Почему так происходит, почему у нас в наборе дублируются фрукты? Ответ прост — в нашем наборе хранятся объекты типа Fruit, которые мы определили самостоятельно. Поскольку мы определили, что такое фрукт, нам также необходимо определить, когда два фрукта равны. Помните метод equals() класса объекта? Нам нужно использовать его, чтобы указать условия равенства. Более того, нам нужно определить метод hashCode(). Хэш-код — это целое число, которое формируется для каждого объекта по определенному правилу. Он используется при поиске в хеш-таблицах и может даже сократить время поиска. Важно понимать, что хэш-код не всегда уникален для разных ключей. Ключи равны, если их хэш-коды равны и метод равенства() возвращает true.

Итак, укажем эти методы (код написан внутри класса Fruit):

```

общедоступное логическое значение равно (Объект o {
    Другие фрукты = (Фрукты)o;
    if(other.name.equals(this.name)) возвращает true;

    вернуть ложь;
}

общественный int hashCode() {
    вернуть имя.hashCode();
}

```

После добавления этого кода у нас в наборе осталось всего два фрукта: яблоко и банан.

7.5 Интерфейсы карты и SortedMap

Интерфейс Map не расширяет интерфейс Collection , поскольку Map содержит пары ключ-значение, а не только ключи. Как вы знаете, карты не могут содержать повторяющиеся ключи, и каждый ключ должен соответствовать только одному значению.

Интерфейс SortedMap расширяет Map и сохраняет его ключи в отсортированном порядке. Сортированные карты широко используются для упорядоченных коллекций пар ключ-значение, например в словарях, адресных книгах и телефонных справочниках.

Есть три карты реализации: HashMap, ДеревоКарта и LinkedHashMap. Если вам нужно хранить пары в порядке возрастания ключей, используйте TreeMap, который является реализацией интерфейса SortedMap . Если вы хотите получить максимальную производительность и не заботитесь о порядке итерации, используйте HashMap. Наконец, если вам нужно что-то промежуточное, компромиссной альтернативой является LinkedHashMap, который обеспечивает хорошую производительность и итерацию в порядке вставки. Но все же, HashMap

реализация карты считается наиболее эффективной, обеспечивая постоянную производительность основных операций получения и размещения . Это положение дел похоже на то, которое мы имеем с реализациями 3 Set .

Давайте рассмотрим простой пример использования карт для хранения отображений программы к расширению их файлов:

Глава 7 – Коллекции и структуры данных

```
Расширения HashMap<String, String> = new
    HashMap<String, String>();
Extensions.put("Word", "docx");
Extensions.put("Excel", "xlsx");
Extensions.put("Power Point", "pptx");
System.out.println(extensions.containsKey("Word"));
System.out.println(extensions.get("Word"));
```

Существуют методы, которые возвращают только набор ключей или коллекцию значений, keySet() и Values() соответственно. Чтобы получить значение, сопоставленное с определенным ключом, используйте метод get:

```
for (Строковая программа: Extensions.keySet())
    System.out.println(program+ " имеет" +
        Extensions.get(program)+ "расширение");
```

Следует признать, что на коллекции, возвращаемые этими методами, ссылается Map, поэтому в случае удаления элемента из этих коллекций соответствующая пара ключ-значение также будет удалена с карты.

7.6 Класс коллекций

Collections — это служебный класс, который предоставляет исключительно статические методы, работающие с коллекциями. В таблице 13 представлены часто используемые методы класса Collections. Обратите внимание, что первый аргумент — это коллекция, над которой необходимо выполнить операцию — c, второй (если он есть) параметр типа Object — obj..

Метод	Описание
Collections.sort(c)	Сортирует список с в порядке возрастания. Если вы попытаетесь отсортировать список, элементы которого не реализуют интерфейс Comparable, будет выдано исключение ClassCastException.
Коллекции.shuffle(c)	Случайным образом переставляет элементы в списке c.
Коллекции.reverse(c)	Меняет порядок элементов в c
Коллекции.max(c)	Возвращает максимальный элемент в c. Существует аналогичный метод min.
Collections.fill(c,obj)	Заменяет все элементы в c указанным элементом.

Объектно-ориентированное программирование и дизайн

Collections.binarySearch(c, obj)	Ищет в списке с объектом obj , используя алгоритм двоичного поиска.
Коллекции.частота(c, obj);	Подсчитывает количество раз, когда указанный элемент встречается в коллекции c.
Collections.disjoint(c1, c2);	Возвращает true , если c1 и c2 не содержат общих элементов, и false в противном случае.

Табл. 13. Основные статические методы класса Collections

Помните, что все вышеперечисленные методы являются статическими, поэтому вам нужно их вызывать, используя ссылку на класс, как в таблице 13.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Коллекция	Набор	Список SortedSet HashSet TreeSet
Карта	LinkedHashSet HashMap	LinkedHashMap Vector
Класс коллекций	Stack Queue	LinkedList ArrayList
Итератор	ListIterator Comparable	равен хэш-коду

ИСПЫТАНИЯ

1. Что можно использовать для создания стеков, очередей, деревьев и деков (двусторонних очередей)...?

- а) Список массивов б) Связанный список

в) Сбор г) Хэшсет

2. Коллекция — это корневой интерфейс в структуре коллекции, интерфейсы которой _____ и _____ получены.

- а) Набор, Карта б) Список, Итератор

- в) Карта, Набор, Список г) Установить, Список

Глава 7 – Коллекции и структуры данных

3. Какая коллекция позволяет связать ее элементы с парами «ключ-значение» и объектами в соответствии с политикой FIFO (первым пришел — первым обслужен)?

- а) Вектор
- б) ХэшМап
- в) LinkedHashMap
- г) Древовидная карта

4. Если вы хотите хранить элементы в коллекции, которая гарантирует, что нет дубликатов и все элементы доступны в естественном порядке. Какую коллекцию вы выберете?

- а) Любой набор
- б) Набор деревьев
- в) Хэшсет
- г) Древовидная карта

5. Упорядоченная коллекция, допускающая наличие повторяющихся элементов?

- а) Любой набор
- б) ListIterator
- в) Любой список
- г) Вектор

6. Выберите интерфейс, который не расширяет интерфейс коллекции ?

- карта
- б) Установить
- в) СортированныйНабор
- г) Список

7. Что из следующего будет сортировать элементы в списке ?

- а) items.sort() с
- б) новый LinkedList (элементы, сортировка)
- Collections.sort(элементы)
- г) Arrays.sort(элементы)

8. Какое утверждение верно для класса HashSet?

- а) Элементы в коллекции упорядочены
- б) Элементы в коллекции гарантированно уникальны.
- в) Доступ к элементам коллекции осуществляется с использованием уникального ключа.

9. Какие методы не определены в интерфейсе Итератора ?

- а) размер публичной пустоты()
- б) public void удалить()
- в) публичный объект next()
- г) общедоступное логическое значение hasNext()

10. Выберите Коллекция, которая запоминает порядок добавления элементов?

- а) связанный список
- б) Хэшсет

в) LinkedHashMap

г) Древовидная карта

11. Выберите метод добавления элемента в любую коллекцию?

а) добавить()

б) поставить()

в) вставить()

г) установить()

12. Выберите метод, который можно использовать для добавления пары ключ-значение на карту?

а) добавить()

б) поставить()

в) вставить()

г) установить()

13. Какой из этих методов определяет объединение множеств?

а) сохранитьВсе()

б) сохранить()

в) добавитьВсе()

г) содержитВсе()

14. Каков родительский интерфейс TreeMap?

а) СортированнаяКарта

б) Карта

в) ХэшМап

г) сопоставимые

15. Что неверно?

а) Set, List, Map — это интерфейсы интерфейса Collection.

б) Карта содержит элементы с уникальными ключами

в) Список интерфейсов реализован Vector

ПРОБЛЕМЫ

1. Ознакомьтесь с изученными классами и интерфейсами в Java API и сделайте примеры с использованием некоторых распространенных методов.

2. Докажите, что служебные методы класса Collections ведут себя соответственно (привести примеры).

3. Создайте расширение перечисления. Затем создайте класс Document, имеющий экземпляр перечисления Extension, title(String), size (long). Переопределить метод равенства(). Затем создайте папку класса, имеющую HashSet документов и имя (String). Чтобы мы могли написать:

Глава 7 – Коллекции и структуры данных

```
Документ d = новый документ («lab7» , 200, Extension.DOC);
Папка f = новая папка («лабы» );
f.add(d); // добавляет в HashSet
```

Добавьте несколько документов в папку и распечатайте их с помощью foreach. оператор.

- 4.** Докажите, что служебные методы класса Collections ведут себя соответственно (привести примеры).
- 5.** Создайте класс Mark , имеющий поле точек (например, 95) и метод getLetter() (например, A-). Реализуйте интерфейсы Comparable и Cloneable и обеспечьте методы toString() и Equals() . Затем создайте класс Student с именем полей , идентификатором и отметкой (типа Mark). Затем создайте несколько объектов Student, сохраните их в векторе и отсортируйте по имеющимся у них меткам.
- 6.** Напишите класс Student с оценкой в поле типа int. Затем создайте несколько экземпляров своего класса и добавьте их в TreeSet. Затем выполните итерацию по вашему набору (используя foreach или итератор) и покажите, каким будет результат.
- 7.** Создайте абстрактный класс MyCollection со всеми приведенными ниже методами. Затем реализуйте класс MyVector (реализует интерфейс MyCollection) для представления расширяемого массива объектов. Как и массив, он содержит компоненты, доступ к которым можно получить с помощью целочисленного индекса. Ваш класс будет хранить целые числа.

Конструкторы:

MyVector() – создает пустой вектор.

MyVector(int [] a) – создает вектор со всеми целыми числами из массива a.

Методы:

add(int element) — добавляет указанный элемент в конец этого вектора.

add(int index, int element) — вставляет указанный элемент в указанную позицию в этом векторе. Не забудьте выдать исключение, если индекс > размер.

clear() — удаляет все элементы из этого вектора.

contains(int o) — возвращает true, если этот вектор содержит указанный элемент.

get(int index) — возвращает элемент в указанной позиции в этом векторе.

indexOf(int o) — возвращает индекс первого вхождения указанного элемента в этот вектор или -1, если этот вектор не содержит элемент.

InsertElementAt(int element, int index) — вставляет указанный целочисленный вектор по указанному индексу.

isEmpty() — проверяет, не имеет ли этот вектор компонентов.

RemoveAt(int index) — удаляет элемент в указанной позиции в этом векторе.

Remove(int element) — удаляет первое вхождение элемента в этот Вектор.

RemoveAll(int element) — удаляет все вхождения элемента в этом векторе.

reverse() — меняет местами элементы массива.

set(int index, int element) — заменяет элемент в указанной позиции в этом векторе указанным элементом.

size() — возвращает количество компонентов в этом векторе.

sort() — выполняет сортировку элементов массива.

toArray() — возвращает массив, содержащий все элементы этого массива.
Вектор
в правильном порядке.

toString() — возвращает строковое представление этого вектора.

8. Напишите консольное приложение, представляющее собой простую телефонную книгу, содержащую сопоставления имен с телефонными номерами. Тщательно подумайте, какую структуру данных можно использовать для такого типа приложения. В главном меню пользователь имеет возможность добавить новый контакт, просмотреть все контакты, обновить информацию о конкретном контакте и выполнить поиск контактов по имени.
Подсказка: используйте Scanner, while(true), labels.

8 Files and Streams

В общем, поток — это абстракция непрерывного одностороннего потока. Может Я данные. быть полезно думать о потоке как о каким-то образом упорядоченной последовательности данных, которая имеет источник (входные потоки) или пункт назначения (выходные потоки), см. рисунок 42 ниже.

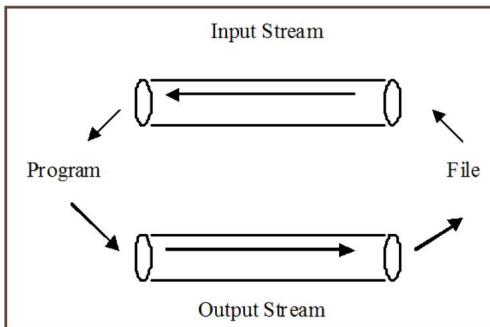


Рисунок 42. Иллюстрация потоков

8.1 Иерархия потоков в Java

Классы потоков обычно делятся на два типа: потоки байтов и потоки символов. Потоки символов представляют собой концепцию более высокого уровня по сравнению с потоками байтов. На самом деле, поток символов — это поток байтов, обернутый некоторой логикой, позволяющей отображать определенные символы кодировки, а не читать байты и затем декодировать символы, которые они представляют (5).

Все классы потоков байтов в Java имеют родительский корневой класс `InputStream` или `OutputStream`, а все классы потоков символов в Java имеют корневой класс `Reader` или `Writer`. Важно отметить, что подклассы этих двух групп корневых классов аналогичны друг другу.

На рисунке 43 ниже представлена иерархия байтовых потоков в Java.

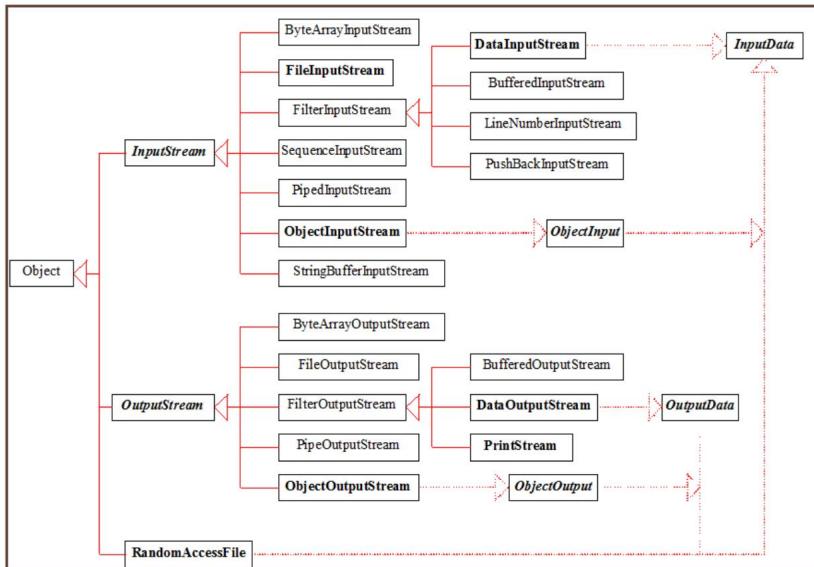


Рисунок 43. Потоки байтов в Java

На рисунке 44 представлена иерархия потоков символов в Java.

Чтобы прочитать форму или записать файл на диск, вам необходимо использовать файловые потоки.

Для потоков байтов используйте FileInputStream и FileOutputStream, а для потоков символов используйте FileReader и FileWriter для чтения и записи соответственно.

Чтобы создать объекты этих классов, вам необходимо передать имя файла соответствующему конструктору, например:

```

//поток ввода байтов
FileInputStream infile = новый FileInputStream("in.txt");
//поток вывода символов
FileWriter outfile = новый FileWriter("out.txt");
  
```

8.2 Потоки данных

Потоки данных используются для чтения и записи примитивных типов Java, таких как int, long, Boolean и т. д., машинно-независимым способом. Следовательно, вы можете записать файл данных на одном компьютере и прочитать его на другом компьютере, например, с другой операционной системой или файловой системой. В Java есть два соответствующих класса потоков данных: для чтения — DataInputStream, для записи — Выходной поток данных.

Глава 8. Файлы и потоки

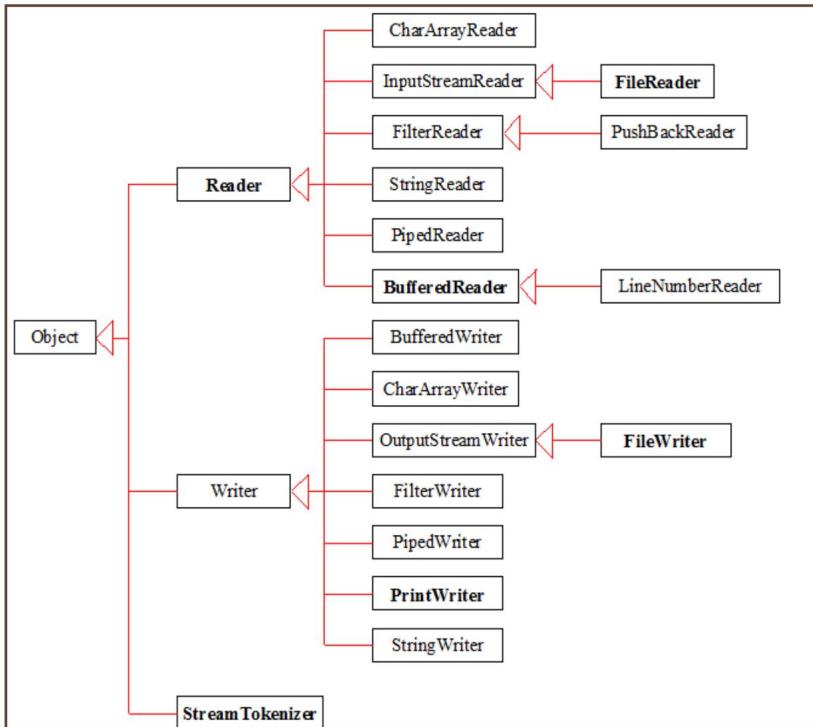


Рисунок 44. Символьные потоки в Java

Чтобы создать экземпляры этих классов, вам необходимо передать соответствующему конструктору `FileInputStream` (для `DataInputStream`) или Экземпляр `FileOutputStream` (для `DataOutputStream`), привязанный к файлу, который вы планируете читать или писать в:

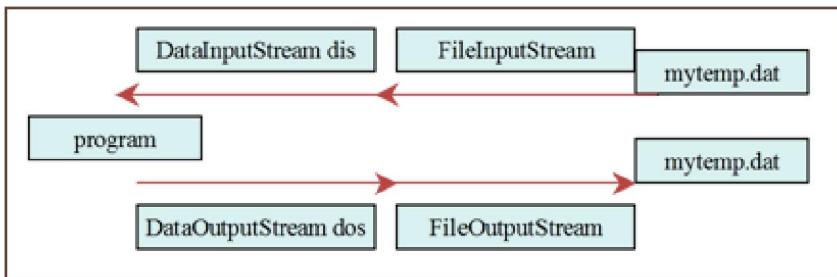


Рисунок 45. Обертывание файловых потоков внутри потоков данных

```
Выходной файл DataOutputStream = новый  
DataOutputStream(новый FileOutputStream("out.txt"));
```

Если у вас есть такой экземпляр, вы можете записывать операции (или операции чтения, если вы создаете объект DataInputStream). Собственно, с помощью потоков данных можно читать/записывать любую переменную примитивного типа, имя метода для этого следующее: [чтение/запись]Тип, например, для DataInputStream:

```
int readShort() выдает IOException int readInt()  
выдает IOException char readChar() выдает  
IOException
```

И для DataOutputStream:

```
void writeByte(byte b) выдает IOException void writeInt(int i)  
выдает IOException void writeBoolean(boolean b) выдает  
IOException
```

8.3 Потоки печати

Ограничением потоков вывода данных является то, что они выводят данные в двоичном формате. Следовательно, если вы откроете выходной файл, вы не сможете просмотреть его содержимое в виде текста — оно нечитабельно. Если вам нужна такая опция, вы можете использовать потоки печати (например, класс PrintWriter обеспечивает эту функциональность) для вывода данных в файлы. В этом случае вы сможете увидеть файл как читаемый текст.

Чтобы использовать PrintWriter, вам, как обычно, необходимо создать его экземпляр. Как параметром вы можете передать либо поток для записи в консоль, либо в файл:

```
PrintWriter (Выход записи)  
PrintWriter (Выходной поток)
```

Например:

```
//читать из консоли  
PrintWriter pw = новый  
PrintWriter (новый OutputStreamWriter (System.out));  
//читаем из файла  
PrintWriter pwFile = новый  
PrintWriter(новый FileWriter("a.out"));
```

Основными методами, предоставляемыми этим классом, являются методы print и println. Помните перегрузку? Из-за перегрузки вы можете передать переменную любого типа.

Глава 8. Файлы и потоки

примитивный тип для этих методов, включая int, double, long, boolean и т. д. Эти методы также могут принимать второй необязательный параметр логического типа, указывающий на автоматическую очистку. По умолчанию он имеет ложное значение. Autoflush означает автоматическую очистку содержимого буфера при получении некоторых данных (9). Если autoflush имеет значение false, данные будут напечатаны сразу после закрытия потока. Итак, практическое правило — всегда закрывать потоки, поскольку в итоге вы можете получить пустой файл.

8.4 Буферизованные потоки

Буферизованные потоки чрезвычайно полезны, поскольку они значительно ускоряют операции ввода и вывода за счет уменьшения количества операций чтения и записи. В частности, при вводе группы данных считывается сразу, а не по одному байту за раз. Аналогично, в случае вывода данные сначала кэшируются в буфер, а затем все вместе записываются в указанный файл. Настоятельно рекомендуется использовать буферизованные потоки. Вы можете обернуть потоки в BufferedReader/BufferedWriter для чтения/записи из/в консоль и файл.

Например:

```
BufferedReader in  
= новый BufferedReader(новый InputStreamReader(System.in)); BufferedReader  
in  
= новый BufferedReader(новый FileReader("имя_файла"));
```

Сразу после этого вы можете вызвать in.readLine() для чтения из файла построчно. Теперь давайте рассмотрим простой пример использования BufferedReader для чтения информации с консоли и PrintWriter для записи ее в файл:

```

попытаться {
    BufferedReader br = новый
        BufferedReader (новый InputStreamReader (System.in));
    PrintWriter pwFile = новый
        PrintWriter(новый FileWriter("a.out"));
    Строковая строка = br.readLine();
    while (!line.equals("q")) { pwFile.println(line);

        строка = br.readLine();
    }
    br.close();
    pwFile.close();
} catch (IOException ioe) {
    System.out.println("Не могу прочитать!");
}

```

Как видно, мы читаем информацию построчно, пока пользователь не введет «q», и печатаем ее в файл a.out. Обратите внимание: если вы хотите прочитать файл формы, а не из консоли, все, что вам нужно изменить, это поместить объект FileReader вместо InputStreamReader и указать имя файла в его конструкторе.

Имейте в виду, что использование буферизованных потоков очень полезно, поскольку они обеспечивают буферизацию символов, чтобы обеспечить эффективное чтение символов и строк.

8.5 СерIALIZАЦИЯ

Как правило, сериализация — это процесс преобразования объекта в последовательность битов. На самом деле, под объектом здесь мы подразумеваем любой объект. Это может быть тот, который предопределен в Java (Date, HashMap, String, массив TreeSet и т. д.) или тот, который вы определили самостоятельно. Десериализация — это обратный процесс распаковки последовательности нулей и единиц в исходный объект. Следовательно, механизмы сериализации и десериализации позволяют передавать объекты в поток и из него.

Чтобы разрешить чтение или запись объекта, определяющий его класс должен реализовать интерфейс Serializable (пакет java.io). Этот интерфейс является интерфейсом маркера, поэтому у него нет методов, поэтому вам не нужно добавлять дополнительный код в свой класс, чтобы сделать его сериализуемым. Реализация этого интерфейса позволит механизму сериализации Java автоматизировать процессы хранения и извлечения объектов из файла.

Чтобы выполнить ввод и вывод на уровне объекта, вам необходимо использовать объектные потоки. В частности, вам необходимо использовать следующие классы:

- `ObjectOutputStream` – для хранения объектов (записывания их в файл)
- `ObjectInputStream` – для восстановления объектов (их чтения)

Следует признать, что сериализация занимается только записью объектов и содержащихся в них полей в поток, следовательно, это исключает статические члены класса. (переменные статического типа будут иметь значения по умолчанию).

Выделим требования к классу, который должен быть сериализуемым:

1) Класс должен быть общедоступным. 2) Класс должен реализовывать сериализуемый интерфейс. 3) Все базовые классы класса также должны реализовывать сериализуемый интерфейс. Однако в противном случае такому базовому классу достаточно иметь конструктор по умолчанию (конструктор без аргументов).

Итак, предположим, мы хотим создать несколько объектов `Book`, а затем получить их. Прежде всего, создадим сам класс:

```
импортировать java.io.Serializable;
импортировать java.util.*;
общедоступный класс Book реализует Serializable { int
    NumberOfPages = 0; Private
    String title = "Нет заголовка"; Дата публикацииДата
    = новая дата(); общественная книга (int num,
    String title) {numberOfPages = num; this.title =
        заголовок;

    }

    } Public void setTitle(String title){
        this.title = заголовок;

    } Public String getTitle () {вернуть
        заголовок;

    } public String toString(){ return
        title+" "+numberOfPages+
        страницы, опубликованные: "+publishDate;
    }
}
```

Пример кода ниже показывает, что гораздо проще сериализовать объект, чем кажется:

```
FileOutputStream fos = новый FileOutputStream("book.out");
ObjectOutputStream oos = новый ObjectOutputStream (fos);
Book b = новая Книга(220,"Анна Каренина");
oos.writeObject(b);
oos.flush();
oos.close();
```

Опять же, здесь мы можем наблюдать обёртывание — мы оборачиваем объект fos внутри. ооо. Процедура сериализации полностью реализуется ObjectOutputStream, FileOutputStream просто представляет файл в этом контексте. Обратите внимание, что вам необходимо поместить этот код в блок try-catch (для перехвата исключений типа IOException). Мы изучим исключения позже в следующей главе. А пока просто подумайте, что при сериализации могут быть различные исключения: файл не найден, класс не реализует интерфейс Serializable или не является общедоступным и т. д.

Если вы откроете book.out, вы увидите что-то вроде этого:

```
java -jar serialization.Book.jar
numberOfPages=100
publishDate=2013-05-12T00:00:00
title=Anna Karenina
author=Лев Николаевич Толстой
```

Почему? Потому что в целях экономии места объекты сохраняются в таком формате. Этот файл не предназначен для чтения программистом. Он предназначен для чтения ObjectInputStream, который выполняет десериализацию, после чего мы можем получить информацию:

Теперь посмотрим, как восстановить сериализованный файл из book.out.

```
FileInputStream fis = новый FileInputStream("book.out");
ObjectInputStream oin = новый ObjectInputStream (fis);
Книга b = (Книга) oin.readObject();
System.out.println(b);
```

Обратите внимание, что метод readObject() возвращает объект, поэтому нам нужно применить приведение типов для преобразования обратно в Book. Вывод этой программы следующий:

Анна Каренина, 220 страниц, опубликовано: Вс, 12 мая 09:56:51
БДТ 2013

Как уже упоминалось, вы можете применять сериализацию даже к более сложным объектам (например, HashMap, у которого есть объекты в качестве ключа и значения). Только не забудьте правильно привести тип вашего объекта.

8.6 Файлы последовательного и произвольного доступа

Все потоки, которые мы рассмотрели до сих пор (`FileInputStream` и `FileOutputStream`, `FileReader` и `FileWriter`, `BufferedReader` и `BufferedWriter`), основаны на последовательном доступе. Другими словами, они позволяют вы можете рассматривать файл как поток для последовательного ввода или вывода. Напротив, класс `RandomAccessFile` позволяет вам читать и записывать данные, начиная с указанного места. Все, что вам для этого нужно сделать, — это установить указатель файла , указывающий начальную позицию, с помощью метода `Seek()` .

Более того, `RandomAccessFile` позволяет одновременно читать и записывать файл. Он включает в себя уже знакомые вам типовые методы, такие как `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()` и `writeLong()`.

Другие полезные методы представлены в Таблице 14 ниже:

Подпись метода	Описание метода
<code>void search(long pos)</code>	Устанавливает указатель на то место, где должно произойти следующее чтение или запись.
<code>длинный getFilePointer()</code>	Возвращает текущее смещение указателя (в байтах) от начала файла.
<code>Большая длина()</code>	Возвращает длину файла.
<code>Final void writeBytes(String s)</code>	Записывает строку в файл.

Табл. 14. Методы класса `RandomAccessFile`

В приведенном ниже примере мы используем `RandomAccessFile` , чтобы открыть файл как для чтения, так и для записи. Затем мы устанавливаем указатель на начало файла и работаем до достижения длины файла, считывая и печатая текущий символ. По окончании чтения пишем соответствующее сообщение в конец файла.

```

пытаешься{
    RandomAccessFile rand =
        новый RandomAccessFile(файл, «rw» );
    int я = (int) rand.length();
    rand.seek(0); //Ищем начальную точку файла
    for(int ct = 0; ct < i; ct++){
        байт b = rand.readByte();
        System.out.print((char)b)
    }
    rand.writeBytes("Сканирование завершено");
    randом.закрыть();
}
поймать (IOException e) {
    System.out.println(e.getMessage());
}

```

8.7 Класс файла

Класс `File` — очень полезный служебный класс, который используется для получения информации о файле или каталоге с диска.

Надо признать, что объект этого класса представляет собой всего лишь путь, а не базовый файл. Следовательно, его нельзя использовать для открытия файлов, чтения или записи или предоставления каких-либо других возможностей обработки (5).

Вы можете создать экземпляр этого файла, указав полный путь в конструкторе или указав два аргумента: каталог (типа `File`) и файл. имя:

общедоступный файл (строковое имя)
общедоступный файл (каталог файла, строковое имя)

Используя методы этого класса, вы можете проверить, существует файл или нет (метод `exists()`), является ли это файлом или каталогом (методы `isFile()`, `isDirectory()`), получить родительский каталог (`getParent()`), определить длину файл (длина()) и многие другие методы. Проверьте Java Api для получения дополнительной информации о методах класса `File`.

Кстати, каждый тип файлового потока имеет два типа конструкторов: один принимает одну строку, представляющую имя файла, а другой принимает объект типа `File`, ссылающийся на файл.

Пример кода представлен ниже. Эта программа считывает имя файла с консоли, а затем, если файл не существует, выходит из программы.

```

BufferedReader в = новый
    BufferedReader (новый InputStreamReader (System.in));
System.out.print("Введите имя файла: ");
Строка str = in.readLine();
Файл файл = новый файл (str);
если(!file.exists())
{
    System.out.println("Файл не существует.");
    Система.выход(0);
} еще {...}

```

8.8 Строковый токенизатор

Класс StringTokenizer позволяет разбивать строку на токены с помощью определенный разделитель или набор разделителей (9). Он часто используется для анализа текстовых файлов определенных форматов. Объект StringTokenizer может вести себя двумя способами, в зависимости от логического значения (третий параметр в конструкторе, называемый returnDelims). Если этот флаг имеет значение false, указанный разделитель используется для разделения строки на токены. Если это правда, разделитель считается самим токеном. Небольшой пример ниже поможет вам прояснить этот вопрос:

```

StringTokenizer st = new StringTokenizer("A *B* Cccc", " ");
в то время как (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}

```

Первый параметр конструктора StringTokenizer указывает строку, сам, второй – разделитель. Вывод этого кода следующий:

```

A
*B*
cccc

```

Теперь давайте посмотрим на следующий пример, в котором мы используем разделитель в качестве токена, установив для returnDelims значение true. Кстати, если вы не укажете третий параметр, по умолчанию он будет false, как и в предыдущем примере.

```
 StringTokenizer st2 = новый
    StringTokenizer("A*B*Cc|cc", "*|", true);
в то время как (st2.hasMoreTokens()) {
    System.out.println(st2.nextToken());
}
```

Обратите внимание на разделитель в этом примере «*|». Этот сложный разделитель указывает, что строка будет разделена как *, так и | персонажи. Пример кода выше имеет следующий вывод:

```
A
*
B
*
Копия
|
копия
```

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Поток Потоки байтов Потоки символов StringTokenizer

БуферизованныйЧтение	Сериализация	Файл
Десериализация	PrintWriter	ОбъектИнпутСтрим

Последовательный доступ к ObjectOutputStream RandomAccessFile

ИСПЫТАНИЯ

1. Как можно классифицировать классы Stream?

- а) Потоки байтов и потоки символов
- б) Потоки записи и потоки чтения
- в) Потоки данных и потоки объектов

2. BufferedReader используется для чтения:

- а) текст из потока вывода символов

б) текст из потока ввода символов

в) массив байтов

г) один байт из файла

3. Какие из этих классов позволяют одновременно читать и обновлять файл?

а) Читатель/Писатель

б) Объектный поток

в) Файл случайного доступа

г) Строковый токенизатор

4. Как десериализовать `HashMap<String, Integer>?`

а) `hm = (HashMap<String, Integer>) oin.read();`

б) `hm = (HashMap) oin.readObject();`

в) `hm = (HashMap<String, Integer>) oin.writeObject();`

г) `hm = (HashMap<String, Integer>) oin.writeObject();`

5. Как создать объект `BufferedWriter` для записи в файл `a.out`?

а) `BufferedWriter bw = новый BufferedWriter(новый FileWriter("a.out"));`

б) `BufferedWriter bw = новый BufferedWriter("a.out");`

в) `BufferedWriter bw = новый BufferedWriter(новый
FileOutputWriter("a.out"));`

г) `BufferedWriter bw = новый BufferedWriter(новый файл("a.out"));`

ПРОБЛЕМЫ

1. Создайте класс `Contact` с именем полей, номером телефона. Предоставьте метод равенства(). Покажите, как сериализовать объекты `Vector` of `Contact`.

2. Создайте класс `Mark`, имеющий поле точек (например, 95) и метод `getLetter()` (например, А-). Реализуйте интерфейс `Comparable` и предоставьте методы `toString()` и `Equals()`. Затем создайте несколько объектов `Mark`, сохраните их в векторе и сериализуйте.

3. Эта проблема состоит из 2-х частей.

a) Напишите программу, которая считывает оценки учащихся из файла «scores.txt» , сохраняет их в некоторой подходящей коллекции (угадайте, в какой), находит лучший результат, а затем выставляет оценки всем учащимся в соответствии с этим планом:

- Оценка A, если результат \geq лучший – 10;
- Оценка B, если результат \geq лучший – 20;
- Оценка C, если результат \geq лучший – 30;
- Оценка D, если результат \geq лучший – 40;
- В противном случае оценка F.

Результат необходимо сохранить в файле «grades.txt» именно в следующем формате:

1) Иванов Иван – «A»

Формат файла Scores.txt:

Иванов Иван 100

Алиев Али 22

Меньшиков Сергей 65

...

б) Теперь напишите вторую программу. Предположим, вам не нужно хранить имена учеников, достаточно лишь максимальной, минимальной и средней оценки. Подумайте хорошо, какая сборка должна использоваться в новых условиях.

Распечатайте ответ в тот же файл «grades.txt» . Учтите тот факт, что вам необходимо ИЗМЕНИТЬ (обновить) файл, а не переопределить его, поэтому ранее напечатанная информация не должна возникнуть. Итак, «grades.txt» может выглядеть так:

1) Иванов Иван – «A»

...

...

Средний – 60

Максимум – 100

Минимум – 25

4. Разработать заявку на предложение университетского курса. Должны быть классы для учебника, инструктора и курса. Этот

Приложение также должно иметь класс Driver для проверки созданных классов.

Классы:

1. Учебник: в этом классе должны быть переменные (поля) для isbn, названия и автора(ов).

И он должен включать методы конструктора, доступа и мутатора для полей, а также методы `toString` и `Equals`.

2. Инструктор: необходимо включить переменные `firstName`, `LastName`, `Department` и `email`. Определите конструктор и методы, как это было сделано для `Textbook`.

3. Курс: этот класс представляет собой совокупность данных, которая включает переменную для названия курса и переменные для учебника и преподавателя.

Конструктор курса инициализирует `CourseTitle` и ссылается на конструкторы `Textbook` и инструктора для инициализации полей для этих классов. Включите средства доступа и мутаторы, а также методы `toString()`, `quals()`.

Драйвер или тестовая программа:

Приложение Драйвер должно запускаться в двух режимах: режиме пользователя и режиме администратора.

Пользовательский режим должен предлагать пользователю следующие возможности:

а) Просмотр списка доступных курсов б)

Отображение информации о курсе

Для второго варианта (б) программа должна быть способна извлекать значения, связанные с объектами курса, а затем распечатывать эти сведения.

Учебники, курсы и инструкторы должны вводиться в систему администратором (режим администратора). Имя пользователя и хешированный пароль администратора должны храниться в файле «`admin.txt`». Каждый раз, когда администратор входит в систему, он должен регистрироваться в «`admin.txt`» (вам необходимо изменить файл, а не переопределить). Итак, формат «`admin.txt`» следующий:

Имя пользователя: `root`

Пароль: `z53h` /// это хэш!

27.10.12 13:44 админ залогинился в систему

27.10.12 13:47 админ добавил новый курс «Обработка естественного языка»

27.10.12 14:05 админ добавил новый учебник «Интеллектуальный анализ данных – инструменты и приложения»

...

Для ввода с консоли включите подсказки для пользователя. Включите описательные заголовки или метки для идентификации выходных данных. Данные для нескольких объектов учебника, инструктора и курса, введенные администратором, необходимо сохранить с помощью сериализации. Итак, для отображения информации пользователям необходимо ее десериализовать.

9 Exceptions

«Основная задача обработчика исключений — доставить ошибку из рук программиста в удивленное лицо пользователя. Если вы будете помнить об этом главном правиле, вы не ошибетесь» .

Верити Стоб

A Как вы, возможно, уже заметили, почти все в Java представляет собой объект. И Исключения не являются исключением. Поскольку исключения являются объектами, как и другие объекты, различные типы исключений могут быть подклассами друг друга. Например, `FileNotFoundException` — это класс, производный от `IOException`. Нужно четко понимать, что если мы поймаем `IOException`, то мы также перехватываем все его дочерние классы, включая `FileNotFoundException`.

9.1 Иерархия исключений

На рисунке 46 изображена упрощенная версия иерархии исключений в Java. Как видно, все исключения в Java расширяют класс `Throwable`, который напрямую распадается на две ветви: `Error` и `Exception` (9). `Error` и его подклассы представляют собой внутренние ошибки и исчерпание ресурсов внутри системы выполнения Java (например, `OutOfMemoryError`). Согласно документации Java, ошибка может возникнуть из-за некоторых ненормальных условий, которые никогда не должны возникать (9). Вы мало что можете сделать, чтобы справиться с этим. Еще один дочерний класс `Throwable`, который нас больше интересует и на котором нужно сосредоточиться, — это `Exception`. Подклассы `Exception` соответствуют ошибкам, после которых программа может восстановиться.

Класс исключений разделяется на две ветви. А именно, `IOException`¹⁶ и `RuntimeException`. Все исключения, полученные из `RuntimeException`, в основном возникают из-за наличия ошибок в вашем коде, следовательно, это ваша вина. Например, некорректное приведение типов (`ClassCastException`), выход за пределы массива (`IndexOutOfBoundsException`), доступ к нулевому объекту (`NullPointerException`) и т.д. Во вторую ветку классов входят исключения, которые на самом деле произошли не по вашей вине — твоя программа

¹⁶ Это основной подкласс `Exception`, не производный от `RuntimeException`. Там есть и другие, помимо `RuntimeException` и `IOException`, которые являются прямыми дочерними элементами `Exception`, например `SQLException`, `MalformedURLException` и другие.

это хорошо, но произошли и другие плохие вещи. Например, когда вы пытаетесь открыть неверный URL-адрес (MalformedURLException).

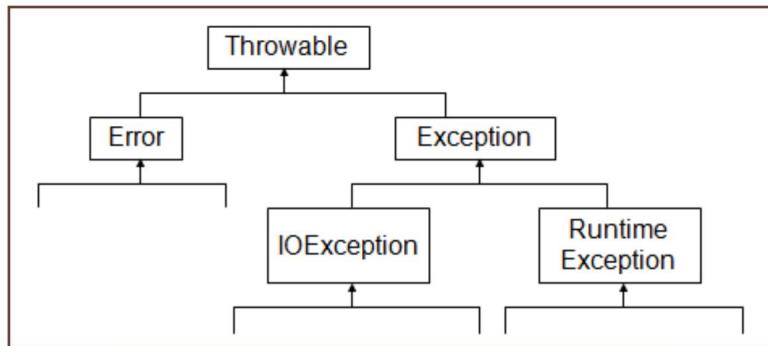


Рисунок 46. Иерархия исключений в Java

9.2 Проверенные и непроверенные исключения

Исключения, производные от класса Error или RuntimeException, считаются непроверенными . Как правило, они представляют собой ошибки программирования, а не ошибки, возникшие, например, из-за неверных данных, введенных пользователем. Таким образом, непроверенные исключения могут быть выброшены «в любое время» , и их не обязательно обрабатывать явно, методом не обязательно объявлять, что они могут выдавать такое исключение (3). Поскольку их может генерировать любой метод – на самом деле, некоторые инструкции (например, доступ к массиву, целочисленное деление и т. д.) могут их генерировать.

Примерами таких исключений являются NullPointerException, ClassCastException, OutOfMemoryError.

Когда происходит непроверенное исключение, Java обрабатывает его автоматически. Пример ниже демонстрирует это:

```

Имя строки = ноль;
char c = name.charAt(8);

```

После запуска этой программы во время выполнения вы получите следующее ошибку:

Исключение в потоке «основной»

java.lang.NullPointerException

в Tutorial.teststts.main(teststts.java:51) _____

Все остальные исключения представляют собой проверенные исключения. В основном их называют «проверенными», поскольку компилятор проверяет, являются ли программисты предоставить обработчики исключений для тех исключений, которые могут произойти. Это сделано для того, чтобы гарантировать, что в случае возникновения исключения оно каким-либо образом будет обработано. Следовательно, поток этих исключений явно контролируется.

9.3 Попытка – блокировка ловли

Стандартный синтаксис блока try/catch следующий:

```
попробуй {
    //заявления
} catch(ExceptionType1 идентификатор1) {
    //обработчик для типа1
} catch (идентификаторExceptionType2) {
    //обработчик для типа2
} . . .
```

Вы можете предоставить блоки перехвата для более чем одного исключения, но имейте в виду, что порядок должен быть от самого конкретного к самому общему, поскольку мы хотим обрабатывать исключение наиболее конкретным возможным способом. Существует множество примеров из реальной жизни, основанных на той же идее. Например, вы набиваете головную боль, а если у вас есть общеукрепляющее или анальгетик (например, «Ношпа»), и какое-то лекарство, которое конкретно помогает справиться с головной болью (например, «Темпалгин»), вы обязательно выберете «Темпалгин».

Как правило, в блок try помещается потенциально опасный код, который может генерировать исключения. В свою очередь, обработка этого исключения (действия, которые необходимо выполнить в случае возникновения исключения) помещаются в блок catch. При выполнении операторов в блоке try возможны две ситуации: исключение либо происходит, либо нет.

В случае возникновения исключения выполнение программы прерывается, а оставшийся код в блоке try пропускается. После этого предоставленные предложения catch проверяются на совместимость с типом выброшенного Исключения. В случае обнаружения соответствующего блока catch выполняется код внутри его тела. Излишне говорить, что после этого все оставшиеся блоки catch игнорируются. Если не найдено подходящих предложений catch, то выброшенное исключение далее выбрасывается во внешнюю попытку, которая может иметь перехват.

пункт, чтобы справиться с этим.

Если во время выполнения кода в блоке try не возникает исключений, все предложения catch просто игнорируются (не выполняются).

Давайте рассмотрим конкретный пример перехвата исключений. Рассмотрим код ниже:

```
public void read (String fileName) {
    попробуй {
        InputStream in = новый FileInputStream (имя_файла);
        интервал
        б; while ((b = in.read ()) != -1) {
            //обработка ввода

        } catch (IOException e) {
            e.printStackTrace ();
        }
    }
}
```

Метод in.read() в приведенном выше коде выдает

IOИсключение. В этом случае будет напечатана трассировка стека исключения.

Иногда вам может потребоваться выполнить некоторые действия независимо от того, выдано исключение или нет. Для этой цели используется предложение Final: Правило заключается в том, что код внутри блока finally всегда будет выполняться, даже если из блока try или catch выдается исключение. Более того, даже если в вашем блоке try catch есть оператор return, код внутри, наконец, будет запущен перед возвратом из метода. Давайте посмотрим пример:

```
Графика g = image.getGraphics ();
попробуйте
{...} catch (IOException e) {...} наконец {
    r.dispose ();
}
```

В приведенном выше примере мы размещаем объект Graphics в блоке finally, поскольку в любом случае (независимо от того, произошло исключение или нет) вам нужно сделать это, чтобы освободить память. Часто блок «finally» используется для закрытия потоков, подключений к базам данных, удаления некоторых объектов и т. д.

9.4. Требование исключений

Когда мы обсуждали блок try-catch, мы рассматривали метод read(). В блоке catch мы помещаем код, который нам нужно выполнить в случае исключения.

Глава 9. Исключения

имеет место. Другой вариант в этой ситуации — ничего не делать, а просто передать исключение вызывающему методу. Это относится к заявлению об исключении.

В Java метод может запросить исключение с помощью ключевого слова `throws`. Это необходимо поместить в конец прототипа метода (перед определением), как в примере ниже:

```
public void myMethod() выдает IOException
```

Вы также можете заявить несколько исключений, просто разделив их знаком запятая:

```
public void myMethod() выдает исключение IOException,  
другое исключение
```

Итак, давайте повторим пример, упомянутый выше:

```
public void read (String fileName) выдает IOException {  
    InputStream in = новый FileInputStream (имя_файла);  
    интервал  
    б; while ((b = in.read ()) != -1) {  
  
        //обработка ввода  
    }  
}
```

9.5 Выдача исключений

Программисты обычно выдают исключение в некоторых неприятных ситуациях. Рассмотрим пример ниже для пояснения.

Предположим, у вас есть метод с именем `readData()`, который читает файл, заголовок которого говорит, что он содержит 550 символов, но обнаруживает конец файла после 500 символов. Вы решаете создать исключение, когда возникает такая неприятная ситуация, с помощью оператора `throw`. `EOFException` используется для сигнализации о том, что во время ввода неожиданно был достигнут конец файла. Кстати, выбросить исключение можно двумя способами:

```
бросить (новое EOFException()); // 1-й способ  
EOFException e = новое EOFException(); // 2-й способ  
бросить e;
```

Использование (случай обсуждается выше):

Строка readData (входящий сканер) **выдает** EOFException {

```

Строка с = "";
пока(... ) {
    если (!in.hasNext()) {
        //Обнаружен EndOfFile {
            если (n < len)
                бросить (новое EOFException());
            .
            .
        }
    }
    вернуть с;
}

```

Не пытайтесь с утверждением, выдачей и перехватом исключений.

Рисунок X ниже поможет вам окончательно закрепить эти понятия:

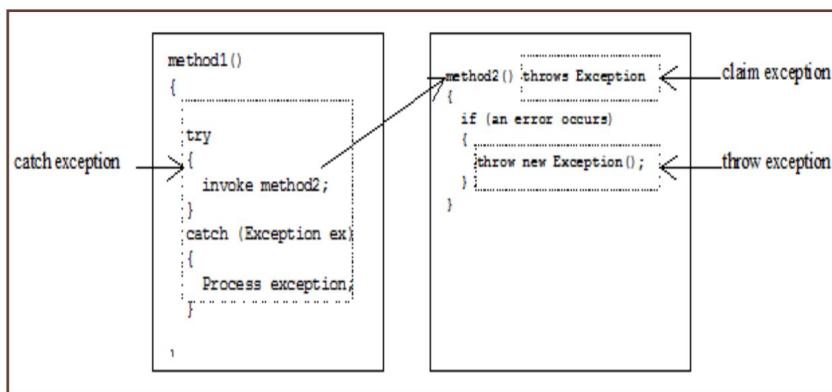


Рисунок 47. Ловля, выбрасывание, требование исключений

Взвесив все это, мы можем прийти к выводу, что обработка исключений очень полезна в том смысле, что она отделяет код обработки ошибок от обычных задач программирования, что значительно упрощает чтение и модификацию программ.

Однако в большинстве случаев обработка исключений требует выделения времени и ресурсов, поскольку необходимо создать экземпляр нового объекта исключения, откатить стек вызовов и т. д.

КЛЮЧЕВЫЕ ПОНЯТИЯ, КОТОРЫЕ НУЖНО ПОНЯТЬ ИЗ РАЗДЕЛА

Поэтапное усовершенствование Надежность

Переносимость JVM

Байт-код соглашений об именах

Интерпретация

API

Затмение IDE

Примитивный тип

точно в срок

Ссылочный тип

Перегрузка

Методы Переменные

Сканер

ИСПЫТАНИЯ

1. Какое из приведенных ниже утверждений демонстрирует, как генерировать исключение?

а) метод() выдает исключение;

б) попробовать {...}(Исключение e){...};

в) попробовать {...}поймать(Исключение e){}
Исключение();выбросить e;

г) Исключение а = новый

2. Какое из приведенных ниже утверждений демонстрирует, как перехватить исключение?

а) метод() выдает исключение;

б) попробовать {...}(Исключение e){...};

в) попробовать {...}поймать(Исключение e){}
Исключение();выбросить e;

г) Исключение а = новый

3. Какое из приведенных ниже утверждений демонстрирует, как требовать исключения?

а) void run() выдает исключение;

б) попробовать {...}(Исключение e){...};

в) попробовать {...}поймать(Исключение e){}
Исключение();выбросить e;

г) Исключение а = новый

4. Какой класс является корневым в иерархии исключений Java?

а) Ошибка б) Исключение

в) Ошибка г) Объект

5. Выбрать непроверенное исключение?

а) исключение NullPointerException

б) Исключение IO

в) FileNotFoundException

г) CloneNotSupportedException

ПРОБЛЕМЫ

1. В этой части лабораторной работы вы рассмотрите простой пример
Обработка исключений.

```
публичный класс Пример1{
    public static void main(String[] args){
        int знаменатель, числитель, отношение;
        числитель = 5;
        знаменатель = 2;
        соотношение = числитель/знаменатель;
        System.out.println(" Ответ: "+ratio);
        System.out.println("Готово."); // Не перемещайте его
    }
}
```

- Создайте класс Пример1.java в своей среде разработки.
- Скомпилируйте и запустите приложение Пример 1.
Что вывело приложение при его запуске?
- Измените значение знаменателя на 0.
- Перекомпилируйте и повторно выполните Пример 1.
Какая «ошибка» была сгенерирована приложением при его запуске?
Почему эта «ошибка» возникла во время выполнения (а не во время компиляции)?
- Добавьте оператор try-catch. В частности, поместите в блок try только оператор, который генерировал исключение, и не помещайте операторы в блок catch. (Подсказка: вы сможете определить, какое исключение перехватывать и какая строка генерировала исключение, из сообщения об ошибке, которое вы получили на предыдущем шаге.) Перекомпилируйте пример 1.
Какая ошибка выдается и почему?
- Переместите «выводной оператор» в блок try (также).

7. Добавьте оператор System.out.println("Делить на 0."); к блоку улова.

Перекомпилируйте и повторно выполните Пример 1.

Какой результат был получен?

8. Добавьте вызов метода printStackTrace() ArithmeticException в конец блока catch. Перекомпилируйте и повторно выполните Пример 1.

Какой результат был получен?

Приложение выполнилось правильно или нет?

2. В этой части лабораторной работы рассматривается пример обработки исключений. внутри и снаружи операторов блока.

публичный класс Пример2{

```
public static void main(String[] args){
    int я, соотношение;
    int[] числа = {100,10,0,5,2,8,0,30};
    попробовать{
        for (i=0; я < numbers.length-1; я++){
            соотношение = числа[i] / числа[i+1];
            System.out.println(numbers[i]+"/"+
                числа[i+1]+"="+ratio);
        }
    } catch (ArithmeticeException ae){
        System.out.println("Не удалось вычислить " +
            числа[i]+"/"+числа[i+1]);
    }
}
```

1. Напишите код для файла example2.java и скомпилируйте его.

Какая ошибка возникла?

2. Инициализируйте i значением 0 внутри блока try (но перед циклом for).

3. Скомпилируйте Пример 2.

Какая ошибка возникла?

4. Невозможно использовать i до его инициализации. Почему вообще возникает эта ошибка? (Подсказка: подумайте об операторах блоков.)

5. Переместите инициализацию і перед блоком try.
6. Скомпилируйте и выполните Пример 2.
Какой вывод генерируется?
Почему даже не предпринимаются все попытки разделения?
7. Исправьте Пример 2, чтобы он работал правильно. (Подсказка: переместите блок try catch внутрь блока for.) Что вы изменили?
Что произошло?
3. В этой части лабораторной работы рассматривается ненадлежащее использование обработки исключений и способы «исправления» .

```
публичный класс Пример3{
    public static void main(String[] args){
        интервал я;
        int[] данные = {50, 320, 97, 12, 2000};
        попробуй {
            для (я=0; я <10; я++){
                System.out.println(данные [я]);
            }
        } catch (ArrayIndexOutOfBoundsException aibe){
            System.out.println("Готово");
        }
    }
}
```

1. Скомпилируйте и выполните Пример 3 и убедитесь, что он выводит все значения, за которыми следует слово «Готово» .
2. Измените Пример 3, чтобы он выполнялся «правильно» и не требовал использования оператора try-catch. (Примечание: результат не должен измениться.) Что вы изменили?
4. В этой части лабораторной работы вы познакомитесь с некоторыми другими исключениями, узнаете, где они возникают и как их можно использовать.

```

публичный класс Пример 4
{ public static void main(String[] args){
    //двойной левый операнд, результат, правый операнд;
    Стока leftString, оператор, rightString; токенизатор
    StringTokenizer; Сканер в = новый
    сканер(System.in); токенизатор = новый
    StringTokenizer(in.nextLine(),
        <+> , правда);

    попробуйте { leftString = tokenizer.nextToken(); оператор
        = tokenizer.nextToken(); rightString = tokenizer.nextToken();
        leftOperand = Double.parseDouble(leftString);
        rightOperand = Double.parseDouble(rightString); if (operator.equals("+"))
        result = leftOperand + rightOperand; иначе результат = 0,0;
        System.out.println("Результат: " +
            + результат);
    }
    поймать (NoSuchElementException nsee) {
        System.out.println("Неверный синтаксис");
    }
    поймать (NumberFormatException nfe) {
        System.out.println("Операнд(ы) не являются числами"); }

    }
}

```

- 1.Какие функции предоставляет объект StringTokenizer? Давать пример.
- 2.Каковы три формальных параметра явного конструктора значений в классе StringTokenizer? Приведите пример.
- 3.Создайте и запустите файл example4.java.
- 4.После запуска программы пропишите в команду следующее строка: 5.3+9.2
Какой вывод генерируется?
- 5.Теперь запустите его еще раз и введите следующее: 5.3+
Какой вывод будет генерирован?

Почему? В частности, какое исключение выдается и почему?

6. Запустите еще раз и введите 5.3+a.

Какой вывод генерируется?

Почему? В частности, какое исключение выдается и почему?

5. Измените файл example4.java, чтобы он поддерживал сложение (+), вычитание (-),

умножение (*) и деление (/). Кроме того, измените файл example4.java, чтобы он обрабатывал для оценки более одного выражения. Так, например, должна быть возможность выполнить следующий ввод: 45.0+4.1 3.2*9.1.

Измените файл example4.java, чтобы он сообщал вам, какой операнд не является числом. (Подсказка: возможно, вам придется использовать вложенные блоки try-catch.)

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

10 Recent Advances in Component Software – Does Scala Beat Java?

«Чисто функциональные языки повредят ваш мозг. Люди, которым они нравятся, относятся к тому типу людей, которым нравился математический анализ в школе» .

Джеймс Гослинг

Hв настоящее время признано, что отсутствие прогресса в компонентном программном обеспечении возникает из-за недостатков языков программирования, используемых для определения и интеграции компонентов. В частности, большинство существующих языков предоставляют лишь ограниченную поддержку абстракции и композиции компонентов (10). Это особенно справедливо для статически типизированных языков, таких как Java и C#, на которых написана большая часть современного компонентного программного обеспечения. Тем не менее необходимость создания больших проектов требовала абстракций программирования, которые позволяли бы повторно использовать компоненты проекта. Ключевым фактором для этого является инкапсуляция, позволяющая скрыть детали реализации, которые могут быть использованы абстрактными классами и абстрактными типами данных.

Экзистенциальная квантификация второго порядка может использоваться для моделирования абстрактных типов данных. В свою очередь, переписка Карри Ховарда (обсуждаемая далее в 1 ул. раздел) можно расширить, чтобы охватить инкапсуляцию. Более того, открытие Митчелла и Плотки позволило принять концепцию абстракции в функциональных языках программирования. Полиморфизм (предмет обсуждения подраздела 2) и абстрактные типы данных (подраздел 3) вместе обеспечивают мощные абстракции данных.

В этой главе предпринята попытка детально рассмотреть рассмотренные выше вопросы на примере Scala PL. Scala была разработана примерно в 2004 году. Она возникла в результате исследований по разработке лучшей языковой поддержки для компонентного программного обеспечения (11). Он спроектирован так, чтобы быть МАСШТАБИРУЕМЫМ в том смысле, что одни и те же концепции могут описывать как маленькие, так и большие части. Более того, чтобы облегчить его внедрение, новый язык должен хорошо интегрироваться с существующими платформами, поскольку большинство компонентов, которые можно использовать повторно, основаны на этих платформах. Scala отвечает этим требованиям, поскольку она была разработана для хорошей работы с Java и C#. Он заимствует большую часть синтаксиса и систем типов этих языков. Наконец, важно отметить, что масштабируемую поддержку компонентов может обеспечить язык, сочетающий в себе объектно-ориентированное и функциональное программирование. Для статически типизированных языков (например, Scala, Java) эти две парадигмы до сих пор были сильно разделены.

10.1. От дедуктивной системы к программам

Существует переписка Карри Ховарда , которая влияет на большинство современных языков программирования. Основная идея этого принципа заключается в существовании точного соответствия между дедуктивными доказательствами в логических системах и хорошо типизированными программами. Таким образом, каждая формула минимальной логики высказываний может быть отображена в соответствующую функцию функционального языка программирования. Это сопоставление представлено в таблице ниже.

Минимальная пропозициональная логика	Функциональное программирование
Импликация ($A \rightarrow B$)	Тип функции
Соединение ($A \times B$)	Тип кортежа
Дизъюнкция $A + B$	Меченый союз

Таблица 15. Определение связей между пакетами

При этом для каждой из этих формул и для каждого типа существует правило введения и исключения . Это соответствие служит основой соответствия Карри-Ховарда между минимальной логикой высказываний и просто типизированным λ-исчислением. Лямбда-абстракция, которую чаще называют «функцией» , представляет собой значение, которое абстрагируется над другим значением. Кроме того, для каждого из представленных ниже правил мы можем найти соответствующую функцию на функциональном языке (в нашем случае Scala):

Минимальное предложение	Функциональное программирование	Скала
Логика		
Значение Введение	Лямбда-абстракция – $\lambda x: A t$	Определение тот функции
Устранение последствий	Применение функции – $t u$	Вызов функции
Введение союза	Конструктор пары – (t, u)	Кортеж конструктор
Устранение союза	Левая и правая проекция – левая t , правая t	Получение элемента Tuple
Введение в дизъюнкцию	Левый и правый впрыск – $inl t, inr t$	Любой тип
Устранение дизъюнкции	Оператор случая – случай $(t, \lambda x: A u, \lambda y: B v)$	Заявление по делу

Таблица 16. Сопоставление правил пропозициональной логики с функциями Scala

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

Очевидно, что лямбда-абстракция соответствует определению функции в любом языке программирования, и Scala не является исключением. В качестве иллюстрации давайте создадим функцию, которая увеличивает переданное ей целое число и, в свою очередь, возвращает его:

```
// введение в суть
def inc(num: Int): Int = число+1
```

Что касается устранения импликации, то ему соответствует функция вызов, как мы уже говорили выше:

```
//устраним последствий
вкл(8)
```

Что касается союза, то его введение соответствует кортежу конструктор в Scala, а исключение отражается на доступе к левому или правому элементу кортежа . Обратите внимание, что Tuple группирует простые логические коллекции элементов без использования класса:

```
//введение союза
def hostPort = (<localhost>, 80) // или «localhost» -> 80
Def Point = (50,5)
//удаление союза
хостПорт._1 // «локальный хост»
Точка._2 // 5
```

Что касается дизъюнкции, было немного неясно, какая функция в Scala ей соответствует. Тем не менее, после небольшого исследования я обнаружил «Или» тип. Либо представляет значение одного из двух возможных типов (непересекающееся объединение). Экземпляры либо являются экземплярами Left или Правый . Очевидно, что либо представляет собой введение дизъюнкции:

```
// введение дизъюнкции
введите myType = Либо[String, Int]
val слева: myType = Right(8)
```

Например, вы можете использовать определенный myType (Either[String, Int]), чтобы определить, является ли полученный ввод строкой или Int. Это соответствует устраниению дизъюнкции:

```
//устранили дизъюнкцию
def PrintRightOrLeft(l:myType):Unit = l match {
    case Right(x) => println("Вправо, целое число: " + x)
    case Left(x) => println("Left, String: " + x)
    PrintRightOrLeft(слева)
}
```

Проведя это небольшое исследование, я в конце концов понял связь между минимальной пропозициональной логикой и функциональным языком программирования. Более того, я узнал о некоторых специфических типах данных и функциях, которые предоставляет Scala. Я чувствую, что это может пригодиться мне в будущем.

10.2 Полиморфизм и повседневное программирование

Подавляющее большинство современных языков программирования в настоящее время являются полиморфными. Более того, полиморфизм является основной особенностью ряда этих языков, таких как Java и C#. Полиморфизм был обнаружен с помощью расширения λ -исчисления под названием System F, которое уточняет λ -исчисление с помощью переменных для типов и кванторов над переменными типа. Примечательно, что выразительность Системы F позволяет использовать эту систему для программирования любых бытовых функций.

Как известно, существует два основных подхода к полиморфизму — проверка типов и вывод типов. На самом деле Scala использует сочетание как проверки типов, так и вывода типов, при этом особое внимание уделяется проверке типов и обеспечению только локального вывода типов. По сути, проверка типов означает, что алгоритмы проверяют, правильно ли типизирована программа, используя аннотации типов, которые есть в программе.

Система полиморфных типов Scala опирается на расширение системы F, которое позволяет использовать подтипы, называемые System F<. Система F или λ -исчисление с полиморфными типами — это яркая реальная иллюстрация изоморфизма Карри-Говарда, который связывает логику и ЯП (12). В нескольких словах он утверждает, что тип — это предложение, а доказательство этого предложения соответствует термину, который классифицируется по этому типу. Таким образом, проверка типов и проверка доказательств тесно связаны. Фрагменты системы F послужили основой для многих полиморфных ФЛ. Самым ярким примером, помимо Scala, является Haskell PL. Его расширение F< сочетает в себе параметрический полиморфизм с подтипованием.

В F< подтипованием отражается в синтаксисе типов новой константой типа Top (супертип всех типов, максимальный тип) и подтипов, связанных с кванторами второго порядка:

$V(X <: A)A'$ (ограниченный квантификатор).

Обычные кванторы второго порядка восстанавливаются установкой квантора

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

привязан к Топу. Мы используем $V(X)A$ вместо $V(X <: \text{Top})A$. Очевидно, что существует ограниченный подтип полиморфных функций λ ($X <: A$) a . Мы используем $\lambda(X)a$ вместо $\lambda(X <: \text{Top})a$ (12). Основная идея дополнительных терминов заключается в том, что мы можем изменить тип любого аргумента, не используемого в теле термина, на Top, и при этом иметь термин того же типа.

Поскольку полиморфные функции работают независимо от своего параметра типа, их можно считать эквивалентными во всех экземплярах своего типа. В $F <$: мы утверждаем, что всякий раз, когда два экземпляра типа имеют общий супертипа, они будут равны, если рассматривать их как элементы этого супертипа.

Некоторые правила синтаксиса системы $F <$ вместе с их значением и эквивалентом в Scala представлены в таблице ниже.

Обозначения	Значение	Пример (в Scala)
$A, B ::= ; a, b ::=$	Типы; Ценности	<code>введите IntRow = Список[Int]</code>
$\text{Икс} ; \text{Икс}$	Типы переменных; Переменные значений	<code>def b : IntRow = Список(1,2,3)</code>
Вершина	Супертипы всех типов	Скала.Любой класс
$A \dashv B$	Функциональные пространства	<code>Целое=>Единица измерения</code>
$\neg B(X <: A)B$	Ограниченнные количественные оценки	тип $T <:$ заказано
$(X:A)b ; b(a)$	Функции; Приложения	<code>Защиту суб(число: Int): Int = число-1; суб(7)</code>
$(X <: A)b ; b(a)$	Функции ограниченного типа	<code>def do[T](t: T):String = t.toString()</code>

Таблица 17. Синтаксические правила системы $F <$

Следовательно, теперь мы можем определить некоторые основные суждения на основе определенного синтаксиса. В частности, тип $E \dashv A$ означает, что A является типом в среде E , а $E \dashv A <: B$ определяет, что A является подтипов B . Эквивалент этого суждения в Scala приведен ниже (Frog — это подтип Animal):

```
класс Frog расширяет Animal {
    переопределить def toString = "зеленый"
    защита Numlegs: Int = 4
}
```

Кроме того, часть приведенного выше кода, определяющая переменную Numlegs типа Int, представляет следующее суждение $E \dashv a:A$, что означает, что a имеет тип A . В нашем случае Frog имеет тип Int.

Другими полезными обозначениями в системе F< являются $B[X \leftarrow C]$ для замены X на C в B и $FV(-)$ для наборов свободных переменных. Эти правила используются для определения следующих суждений о равенстве:

$$\begin{array}{lll} \forall(X \leftarrow :A)B & \equiv \forall(Y \leftarrow :A) B\{X \leftarrow Y\} \text{ where } Y \notin FV(B) \\ \lambda(x:A)b & \equiv \lambda(y:A) b\{x \leftarrow y\} & \text{where } y \notin FV(b) \\ \lambda(X \leftarrow :A)b & \equiv \lambda(Y \leftarrow :A) b\{X \leftarrow Y\} & \text{where } Y \notin FV(b) \end{array}$$

Итак, мы видим, что к суждениям F добавляется подтиpicкое суждение.

Более того, суждение о равенстве значений выносится относительно типа, и это очень важно, поскольку значения в F< могут иметь много типов, и два значения могут быть или не быть эквивалентными в зависимости от типа, которым обладают эти значения. Как видно, суть суждения о подтипе состоит в том, что член типа также является членом любого супертипа этого типа. Правила определения типа $E |- a : A$ такие же, как и соответствующие правила в F, за исключением расширения до ограниченных кванторов (12). Однако дополнительные возможности типизации заключаются в правиле включения, которое позволяет функции принимать аргумент подтипа своего входного типа.

Большинство правил эквивалентности обеспечивают симметрию, транзитивность, конгруэнтность синтаксиса, а также правило Top-collapse, которое гласит, что любые два термина эквивалентны, если их «видеть» в типе Top; поскольку с членами Top не доступны никакие операции, все значения этого типа одинаковы. Эти правила можно увидеть ниже:

$$\begin{array}{ccc} (\text{Sub refl}) & (\text{Sub trans}) & (\text{Sub Top}) \\ \underline{E \vdash A \text{ type}} & \underline{E \vdash A \leftarrow :B \quad E \vdash B \leftarrow :C} & \underline{E \vdash A \text{ type}} \\ E \vdash A \leftarrow :A & E \vdash A \leftarrow :C & E \vdash A \leftarrow :Top \end{array}$$

Как мы уже упоминали, первое правило, представленное выше, — это Рефлексивное правило (каждый тип является подтипов своего типа), второе — транзитивное правило, последнее — Правило верхнего свертывания.

Другие важные стандартные правила — о предположениях, функциях.

и приложения приведены ниже:

$$\begin{array}{ccc} (\text{Subsumption}) & (\text{Val fun}) & (\text{Val appl}) \\ \underline{E \vdash a : A \quad E \vdash A \leftarrow :B} & \underline{E, x:A \vdash b : B} & \underline{E \vdash b : A \rightarrow B \quad E \vdash a : A} \\ E \vdash a : B & E \vdash \lambda(x:A)b : A \rightarrow B & E \vdash b(a) : B \end{array}$$

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

Стандартная кодировка пар в F показана ниже:

$$A \times B \triangleq \forall(C)(A \rightarrow B \rightarrow C) \rightarrow C$$

Кроме того, обычные операции над парой определяются следующим образом:

<i>pair:</i>	$\forall(A) \forall(B) A \rightarrow B \rightarrow A \times B$,
<i>fst:</i>	$\forall(A) \forall(B) A \times B \rightarrow A$, <i>snd:</i> $\forall(A) \forall(B) A \times B \rightarrow B$

Что касается кортежа, он представляет собой повторяющийся тип продукта, где n больше или равно 1 (n — количество элементов в кортеже). Основная полезная операция, которую предоставляет кортеж, — это *a_i* — выбор i-го элемента а. Интересно отметить, что эти соглашения соблюдаются именно в Scala. В качестве иллюстрации рассмотрим создание кортежа для хранения координат x,y,z в трехмерном пространстве.

Защищая точка = (883, 8, 3)

Точка._1 // 883

Точка._3 // 3

Что касается разрешимости проверки типов в Scala, в настоящее время слишком сложно дать однозначный ответ, поскольку полная Scala слишком сложна, чтобы допустить формализацию ее системы типов, которая была бы полной, но все же достаточно управляемой, чтобы допустить доказательство разрешимости.

Одна из целей разработки Scala заключается в обеспечении совместимости с такими языками, как Java. Очевидно, для этого требуется совместимость системы типов Scala. В частности, это означает, что Scala должна поддерживать подтиповование и перегрузку (12), например:

```
def add (x: Int, y: Int) : Int = //
def add (x: String, y: String) : String = //
```

Однако это затрудняет вывод типа. Тем не менее, Scala поддерживает вывод локального типа. В частности, в большинстве случаев можно определить возвращаемый тип определения и тип лямбда-переменной. Например, альтернативным определением power может быть следующее: def power (x : Int) = дважды (y => y * y, x), в котором выводятся как тип возвращаемого значения, так и тип лямбда-переменной y . По сути, неизвестные типы заменяются переменными типа, которые записывают ограничения, которые должны быть соблюдены для успешной проверки типов. Решение этих ограничений определяет недостающий тип (12). Вывод локального типа должен построить наименьший верхний

Объектно-ориентированное программирование и дизайн

границы и максимальные нижние границы множеств типов. Здесь возникает проблема бесконечных аппроксимаций lub и glb. Компилятор Scala решает эту проблему, налагая предельный размер на типы, вычисляемые операциями lub и glb. В настоящее время он установлен на 10 уровнях. Если тип, вычисленный с помощью lub или glb, превышает этот предел, система ответит исключением. В результате компилятор Scala превращает потенциальную проблему неразрешимости вывода типа в другую проблему: локальный вывод типа теперь может не дать решения, даже если лучший тип существует. Тем не менее, всегда можно добавить больше аннотаций типов, чтобы избежать бесконечных приближений.

Как известно, в статически типизированном языке, таком как Scala, проверка типов выполняется перед запуском программы. В этом отличие от Clojure, который является динамически типизированным — типы проверяются во время выполнения. Проверка касается всех возможных исполнений программы и обнаруживает все нарушения спецификаций, которые она может выражать. Компромисс заключается в том, что проверка может отклонить действительные программы.

Более того, как легко догадаться из того факта, что Scala основана на Системе F, она поддерживает параметрический полиморфизм, известный как дженерики в объектно-ориентированном мире (11). Параметрический полиморфизм первого порядка теперь является стандартной функцией статически типизированных языков языка (12). Начиная с системы F и функциональных языков языка программирования, эти конструкции нашли свое применение в объектно-ориентированных языках, таких как Java, C# и многих других.

Фактически, дженерики можно рассматривать как обобщение типа массивов. Аналогично, полиморфный класс или метод может быть выражен через параметры его типа, которые представляют собой неизвестные типы, которые должны быть конкретизированы клиентами класса или метода. Это расширение прекрасно соответствует философии Scala, объединяющей функциональное и объектно-ориентированное программирование. Одной из стандартных областей применения дженериков являются коллекции. В качестве иллюстрации тип List[T] представляет списки заданного типа элемента T, которые можно выбирать свободно, полиморфный класс списков не чувствителен к точному типу своих элементов. Теперь с помощью параметрического полиморфизма такую согласованность можно получить без определения нового списка для каждого конкретного типа. Полиморфный список просто абстрагируется от конкретного типа своих элементов, используя параметр типа. Чтобы обеспечить согласованность, список элементов типа T позволяет добавлять в список только другой элемент типа T. В результате, когда извлекается первый элемент списка элементов типа T, список может гарантировать, что элемент имеет тип T. Без параметрического полиморфизма список может быть выражен только для хранения любого объекта (с использованием полиморфизма подтипа), и для каждого взаимодействия пользователь должен был проверять (посредством приведения), что ожидаемый тип объекта был получен. Без сомнения, интеграция универсальности в объектно-ориентированный язык делает систему типов более мощной.

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

Итак, Scala, как и Java, поддерживает дженерики или параметризованные типы. По сути, он работает более или менее так же, как в Java, с немного другим синтаксисом и более краткими средствами использования благодаря выводу типов в Scala. Вот пример, который демонстрирует это:

```
класс Стек[T] {
    элементы var : List[T] = Nil
    def push(x: T) { элементы = x :: элементы }
    защита сверху: T = items.head
    def peek: T = items.tail
    def pop() { items = items.tail }
}
```

Class Stack моделирует стеки произвольного типа T. Использование параметров типа позволяет проверить, что в стек помещаются только допустимые элементы (типа T). Вот несколько примеров использования:

```
val stack = новый стек[Int]
стек.push(1)
стек.push('d')
println(stack.top)//100
стек.pop()
println(stack.top)//1
```

Нам необходимо подчеркнуть, что подтипы универсальных типов инвариантны. Это значит, что если у нас есть стек символов типа Int, то его нельзя использовать как стек типа Char.

Кроме того, методы в Scala также могут быть обобщены, как и в Java:

```
def doit[T](вещь: T):String = thing.toString
```

Система статических типов — важный инструмент эффективной разработки правильного программного обеспечения. Недавнее введение «общности» в объектно-ориентированные языки программирования значительно повысило выразительность их систем типов (12). Универсальность, которую также называют «параметрическим полиморфизмом», чрезвычайно полезна, поскольку позволяет определять полиморфные списки, которые используют параметр типа для абстрагирования типа своих элементов. Прелесть этого в том, что пользователю этих абстракций не нужно беспокоиться о своей внутренней работе. Интересно отметить, что помимо

Имея общего предка — Top, Scala также вводит тип Nothing, который является подтипов всех типов.

10.3 Смешение подходов к инкапсуляции

Как было сказано во введении, инкапсуляцию можно рассматривать как ключевой фактор упаковки и повторного использования компонентов. Важнейшая идея инкапсуляции заключается в том, что она позволяет скрыть детали реализации. В частности, абстрактные типы данных предоставляют общую сигнатуру типа данных без предоставления точных деталей реализации. Впоследствии эти абстрактные типы данных можно реализовать несколькими способами и скрыть от пользователя. Таким образом, опытный программист может инкапсулировать знания предметной области, чтобы основной программист, как пользователь этих абстракций, мог их обоснованно реализовать (10).

Scala предлагает новую модель компонентных систем. По сути, компоненты в этой модели — это классы, которые можно комбинировать с помощью вложенности и композиции миксинов. Кроме того, классы могут содержать абстрактные типы, экземпляры которых впоследствии могут быть созданы в подклассах. Преимущество этого подхода заключается в том, что относительно небольшого набора языковых конструкций достаточно для основного программирования, а также для определения компонентов и их структуры. Конструкции компонентов Scala обеспечивают золотую середину между мирами объектно-ориентированного программирования и системами функциональных модулей (11).

Если пойти глубже, то в Scala есть понятие абстрактных типов, которые предоставляют гибкий способ абстрагирования над конкретными типами, используемыми внутри объявления класса или типажа. Абстрактные типы используются для скрытия информации о реализации компонента. Как и любому другому члену класса, абстрактным типам в классе необходимо дать конкретные определения, прежде чем можно будет создать экземпляр класса. Так, например, List также можно определить как класс с членом абстрактного типа, а не как класс с параметризованным типом, например:

Список абстрактных классов {тип Elem}

Тогда конкретный экземпляр List можно определить как:

Список {тип Elem = String}

Более того, классы могут содержать члены типа. Член абстрактного типа аналогичен параметру типа. Основное различие между

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

параметры и члены — это их область видимости и видимость. Таким образом, параметр типа является частью типа, тогда как член типа инкапсулирован. Кроме того, члены типа наследуются, тогда как параметры типа являются локальными для своего класса.

Параметры типа конкретизируются с помощью приложения типа. Взаимодополняющие преимущества параметров типа и членов абстрактного типа являются ключевым ингредиентом рецепта Scala для масштабируемых абстракций компонентов (10).

В отличие от языков, поддерживающих только одиночное наследование, в Scala используется более общее понятие повторного использования классов. Scala позволяет повторно использовать определения новых членов класса в определении нового класса. Это делается с помощью композиции класса миксина (11). В качестве иллюстрации рассмотрим следующую абстракцию для итераторов¹⁷.

```
абстрактный класс AbstractIterator {
    ТИП Т
    def hasNextElement: логическое значение
    следующий : Т
}
```

Далее рассмотрим класс-примесь, который расширяет AbstractIterator методом foreach , который применяет заданную функцию к каждому элементу, возвращаемому итератором. Чтобы определить класс, который можно использовать в качестве примеси, мы используем ключевое слово типаж .

```
чертка IteratorWithForEach расширяет AbstractIterator {
    def foreach(f: Т => Unit) { while
        (hasNextElement) f(next)
    }
}
```

Вот конкретный класс итератора, возвращающий последовательные символы нить:

¹⁷ Представленный пример не является лучшей практикой в функциональном программировании, он представлен только ради объяснения композиций миксинов. Для дальнейшего чтения обратитесь к (10), (11), (12).

```
класс SimpleStringIterator (cur: String) расширяет
    AbstractIterator {
        тип Т = Символ
        индекс частной переменной = 0
        def hasNextElement = index < cur.length()
        защита следующий = {
            val temp = индекс Cur charAt; индекс += 1; температура
        }
    }
```

Теперь предположим, что мы хотели бы объединить функциональность, предоставляемую SimpleStringIterator и IteratorWithForEach, в один класс. При одиночном наследовании и интерфейсах (например, в Java или C#) это совершенно невозможно, поскольку оба класса содержат реализации членов с помощью кода. Однако в Scala реальность благодаря составу миксинов-классов позволяет повторно использовать дельту (все новые определения, которые не унаследованы) определения класса. Этот механизм дает возможность комбинировать наши классы — SimpleStringIterator с IteratorWithForEach, как это сделано в следующей тестовой программе, которая печатает все символы заданной строки.

```
класс MyCoolIterator
расширяет SimpleStringIterator("бла-бла") с помощью
    IteratorWithForEach
val итератор = новый MyCoolIterator
итератор foreach println
```

Класс MyCoolIterator создан на основе смеси родительских элементов SimpleStringIterator и IteratorWithForEach с ключевым словом with. Первый панцирь называется суперклассом MyCoolIterator, а второй — миксином. Этот пример демонстрирует, что объединение полиморфизма и абстрактных типов данных обеспечивает мощные абстракции данных.

Итак, как мы видели, в Scala класс может наследовать от другого класса одну или несколько характеристик. Признак — это класс, который можно объединить с другими признаками, используя композицию миксинов — ограниченную форму множественного наследования (10). Основное отличие абстрактного класса от типажа заключается в том, что последний можно составить с использованием смешанного наследования. Еще одно отличие состоит в том, что черты не могут определять конструкторы. Кроме того, тип пересечения, например A с B можно понимать как тип, который является подтипом как A, так и B (12).

Итак, мы узнали, что вместо интерфейсов в Scala используется более общая концепция трейтов. Как и интерфейсы, черты можно использовать для определения абстрактных методов. Тем не менее, в отличие от интерфейсов, черты также могут определять

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

конкретные методы (12). Давайте посмотрим на некоторые другие примеры, иллюстрирующие, что признаки можно комбинировать с помощью композиции миксинов, что делает возможной безопасную форму множественного наследования:

```
черта Привет {
    val hi = "Привет"
}

чертa Любопытный {
    val вопрос = "Ты любишь яблоки?"
    защитa Ask() = println(вопрос)
}

чертa Злой {
    def крик(ы: Стока): Единица измерения
}

чертa AngryPerson расширяет Hello c Curious c Angry{
    val приветствие = привет+ ", "+вопрос
    def Shout(s: String) =
        println(s.toUpperCase()+" !!!")
}
```

В приведенном выше примере мы используем типажи для объявления как абстрактных методов, таких как `Shout()`, так и конкретных методов, таких как `Ask()`. Например, в Java или C# невозможно определить подкласс, объединяющий в себе функциональность четырех вышеупомянутых блоков кода. Тем не менее, композиция примесей позволяет комбинировать любое количество признаков: признак `AngryPerson` наследует методы от `Hello`, `Curious`, реализует метод `Shout()` из `Angry` и определяет значение приветствия, сочетающее в себе функциональность обоих предков.

Теперь давайте посмотрим на еще один привлекательный пример миксинов. Предположим, у нас есть следующие черты характера: Студент, Сотрудник, Мать, Молодой, Шахматист. Как мы могли бы объявить класс `BuzyPerson` со всеми этими характеристиками?

```
класс BuzyPerson расширяет возможности
    ученика ,
    сотрудника и матери
    с Янгом c
    ChessPlayer
```

Как видно из приведенного выше кода, это легко сделать, поскольку при объявлении класса вы просто используете ключевое слово `with` так часто, как захотите. Также

если вы не хотите иметь класс, который всегда использует одни и те же черты, вы можете использовать их позже:

класс YoungMother расширяет возможности Young with Mother
вал Пакита = новая молодая мать с сотрудником и плитой

Наконец, давайте создадим тип MyBool, который имитирует предопределенные логические значения. Интересно отметить, что такие операторы, как «||» и «&&», также могут быть представлены как методы, поскольку Scala позволяет передавать аргументы по имени. В таблице 18 ниже представлен сам признак MyBool, а также два его канонических экземпляра.

Как видно из этих реализаций, правый operand операции && оценивается только в том случае, если левый operand является объектом True.

Соответственно, правый operand || операция оценивается, если левый operand имеет значение False. Таким образом, в Scala можно определить каждый оператор как метод и рассматривать каждую операцию как вызов метода.

Черта MyBool	ЛОЖЬ	Истинный
<pre>trait MyBool { def && (x: MyBool): MyBool def (x: MyBool): MyBool }</pre>	<pre>object False extends MyBool { def && (x: MyBool): MyBool = thi def (x: MyBool): MyBool = x; }</pre>	<pre>object True extends MyBool { def && (x: MyBool): MyBool = x; def (x: MyBool): MyBool = thi }</pre>

Таблица 18. Тракт MyBool

Теперь давайте обсудим взаимодействие между Scala и Java. Как мы упоминали ранее, Scala ориентирована на промышленное использование: это ключевая цель разработки Scala заключается в том, что она должна легко взаимодействовать с основными языками, такими как Java и C#, что делает их многочисленные библиотеки легко доступными для программистов Scala (10). Именно поэтому в настоящее время он реализован на платформах Java и .NET. Он разделяет с этими языками большинство основных операторов, типов данных и структур управления.

В частности, Scala использует большинство управляющих структур и конструкций Java, таких как классы, абстрактные классы, подтипы и наследование, но в ней отсутствует традиционный для Java оператор for. Вместо этого существуют for-комплементации, которые позволяют перебирать элементы списка напрямую без необходимости индексации. Кроме того, в Scala параметры конструктора следуют за именем класса, поэтому в теле нет отдельного определения конструктора класса. Scala также включает в себя некоторые менее распространенные концепции. В частности, существует конкретное понятие объекта, а интерфейсы заменяются более

Глава 10. Последние достижения в области компонентного программного обеспечения: Scala превосходит Java?

общее понятие черт, которые, как мы уже видели (11), можно составить с помощью миксиновой композиции. Наконец, некоторые типы для Java и Scala пишутся по-разному: Object становится Any, Unit — это обычный тип, соответствующий ключевому слову void в Java, а Nothing — это подтип всех типов, который не может быть выражен в Java. Однако в целом синтаксис Scala относительно знаком Java-программистам. Рассмотрим код ниже:

```
if (имя начинается с «A» ) // синтаксический сахар
    System.out.println("A"+name.substring(1));
```

Этот небольшой пример доказывает, что, несмотря на различия в синтаксисе, программы Scala могут без проблем взаимодействовать с программами Java. В приведенном выше примере программа Scala вызывает методы startWith и подстроку String , которая является классом из библиотеки Java. Он также обращается к статическому полю out класса Java Java и вызывает его метод println .

Более того, классы и объекты Scala также могут наследовать классы Java и реализовывать интерфейсы Java. Это позволяет использовать код Scala в среде Java (10). В общем, вы можете использовать классы Scala из Java (а также классы Java из Scala), даже не зная, что они были определены на другом языке. Ключевым фактором этого является то, что, несмотря на совершенно разный синтаксис, программы Scala и Java при компиляции создают почти идентичный байт-код. В качестве простой иллюстрации давайте создадим простой класс в Scala:

```
класс Человек {
    var Name = "Нет имени"
    def getDescription() = "Имя: "+ Имя
}
```

Впоследствии, после компиляции этого класса, будет создан файл класса Person.class, содержащий байт-код. Если вы программист на Java, Scala или C#, вы можете быть пользователем этого класса. Давайте посмотрим, как его использовать в Java:

```
Человек p = новый Человек ();
String desc = p.getDescription();
```

В результате мы можем заключить, что Scala имеет единые и мощные концепции абстракции как для типов, так и для значений. Более того, он имеет гибкие симметричные конструкции миксинов для составления классов и признаков.

Наконец, программы Scala во многом напоминают программы Java, и очень полезно то, что они могут взаимодействовать с кодом, написанным на Java.

10.4 Сблизились ли объектно-ориентированные и функциональные языки?

В заключение можно сказать, что проведя это небольшое исследование возможностей Scala и сравнив ее с Java PL, мы можем сказать, что Scala стремится объединить объектно-ориентированное и функциональное программирование. Этот относительно новый язык обеспечивает плавную интеграцию функциональной и объектно-ориентированной парадигм. Scala не только предоставляет эквиваленты всех необходимых функций функционального программирования, таких как свертывание, функции высшего порядка, параметрический полиморфизм, классы типов и конструкторов, но также предоставляет наиболее полезные функции объектно-ориентированных языков, такие как переопределение и перегрузка. подтиповование, традиционное одинарное наследование и множественное наследование в виде миксинов. Поэтому некоторые утверждают, что Scala — это объектно-ориентированный язык, обладающий некоторыми функциональными особенностями. Третьи выдвигают точку зрения, что Scala — это функциональный язык, обладающий объектно-ориентированными функциями. Действительно, он предлагает лучшее из обоих миров. Таким образом, он занимает уникальное положение как функционального, так и объектно-ориентированного языка.

Как мы знаем, Scala нацелена на построение компонентов и компонентных систем. Очевидно, это и было основной мотивацией объединения объектно-ориентированного и функционального программирования в статически типизированном языке программирования. Итак, отвечая на вопрос, поставленный в этой задаче – «Сконвергировались ли объектно-ориентированные и функциональные языки?» , мы можем сказать: «Да, они сошлись» . На языке программирования Scala» .

Conclusion

После прочтения этой книги вы должны хорошо понять основные концепции объектно-ориентированного программирования.

Тем не менее, по-прежнему существует ряд вопросов, по которым нет единого мнения. в объектно-ориентированном сообществе. Текущие проблемы объектно-ориентированного программирования, с которыми вы можете столкнуться, применяя теорию в реальных ситуациях.

Напомним:

- Общая концептуальная основа, лежащая в основе объектно-ориентированного моделирования, должна быть сформулирована независимо от языков. •

Механизмы абстракции. Несмотря на то, что существует некоторое базовое соглашение

относительно основных механизмов абстракции, между языками все еще существуют значительные различия. Например, одиночное и множественное наследование, типы и классы, инкапсуляция, динамическая типизация и статическая типизация и т. д. • Хранение данных представляет собой общую проблему,

поскольку большинство организаций используют реляционные базы данных для хранения данных, а реляционные базы данных обычно не подходят для хранения данных. хранение предметов. Вы, как программист, должны уметь разрешать конфликты такого рода. Лучший вариант — использовать объектно-реляционное сопоставление¹⁸ (например, Hibernate), которое создает своего рода базу данных виртуальных объектов. Другие программисты предпочитают решать эту проблему с помощью объектно-ориентированных баз данных, но в целом это редкий случай, когда компания использует объектно-ориентированную базу данных. Самый простой способ, который часто используется на практике, — это написание функций, которые преобразуют объекты в модель реляционной базы данных и наоборот.

Сбалансируйте необходимость придерживаться философии объектно-ориентированного проектирования и потребность в несложном приложении. Например, если единственной целью определения объекта является размещение процедуры, рассмотрите возможность определения процедуры как отдельной процедуры, а не определения другого объекта. Автономная процедура приемлема в объектно-ориентированной программе, если она имеет смысл для приложения.

- Обозначение предметов. Хотя идентификация объектов в реальном мире кажется очень естественной и интуитивной, в мире программирования это может оказаться сложной задачей, поскольку некоторые концепции реального мира можно интерпретировать как два или более объектов. Мы советуем использовать «сохраните это» .

¹⁸

Объектно-реляционное сопоставление – метод, используемый для преобразования данных между несовместимыми системами типов в ОО языках. В частности, между объектами и записями таблиц.

Объектно-ориентированное программирование и дизайн

простой» принцип – избегайте создания слишком большого количества объектов в вашей системе, если они все не являются критически важными, не создавайте десятки подклассов. Вместо этого вы можете создать один объект и выделить различия между разными типами объектов с помощью полей и методов. В качестве простой иллюстрации рассмотрим объект `Student`.

Поскольку существует несколько типов студентов (например, бакалавриат, аспирант, аспирант и т. д.), вам может потребоваться определить класс для каждого студента. Однако гораздо лучше использовать для этой цели атрибуты, которые могут описать тип ученика (например, посредством перечисления).

- Сложная иерархия. Иногда новички в программировании с диким энтузиазмом создают сложную иерархию для своих систем, упуская из виду цель приложения. Конечно, многоуровневое наследование технически эффективно, но на практике его поддержка представляет собой сложную задачу. Поэтому избегайте создания ненужных сложных структур наследования, особенно если они включают более трех уровней.

Как мы узнали, объектно-ориентированный подход подчеркивает важность объектов, в отличие от процедурного или императивного программирования, которое подчеркивает выполнение последовательных команд. В свою очередь, функциональное программирование делает упор на определении функций.

Как вы узнали из предыдущей главы, в настоящее время наиболее популярные подходы к программированию рассматривают сочетание объектно-ориентированного и функционального программирования, которые вместе обеспечивают мощные механизмы абстракции данных. Ярким примером таких подходов является недавно появившийся язык программирования `Scala`, который был разработан для хорошей работы с `Java` и `C#`. Он заимствует большую часть синтаксиса и систем типов этих языков.

`Scala` обеспечивает плавную интеграцию функциональной и объектно-ориентированной парадигм. `Scala` не только предоставляет эквиваленты всех необходимых функций функционального программирования, таких как свертывание, функции высшего порядка, параметрический полиморфизм, классы типов и конструкторов, но также предоставляет наиболее полезные функции объектно-ориентированных языков, такие как переопределение и перегрузка, подтиповование, традиционное одинарное наследование и множественное наследование, которое в языке программирования `Scala` представлено в виде примесей.

Acknowledgements

Мне приятно отметить усилия людей, чьи имена не указаны на обложке, но чье сотрудничество, помощь и понимание сыграли решающую роль в создании этого учебника:

Научные рецензенты:

Л.Б. Атымтаева, доктор физико-математических наук

И.М. Уалиева, кандидат физико-математических наук

Р. Ж. Сатыбалдиева, кандидат технических наук

Р. Хорн, доктор компьютерных наук

Дизайнер обложек книг :

Айтпаева А.А. – 2nd студент ФИТ года

Кроме того, хочу выразить благодарность 2-му , а на данный момент 3-му Студенты ФИТ (ФИТ-11, ФИТ-10), обучающиеся по курсу «Объектно-ориентированное программирование и проектирование» в 2012-2013 и 2011-2012 учебных годах соответственно. Эта книга была бы невозможна без их любезных предложений, мотивирующих комментариев, сложных вопросов и разумных исправлений, которые я получал в ходе курса.

Мы будем искренне признательны за все ваши комментарии, критику, исправления и предложения по улучшению текста и примеров. Пожалуйста, не стесняйтесь направлять все комментарии по следующему адресу:

pakita.shamoi@gmail.com

Электронная почта

References

1. Эк, Дэвид Дж. Введение в программирование с использованием Java. sl: Интернет-книга: <http://math.hws.edu/javanotes>, 2006.
2. Кей С. Хорстманн, Гэри Корнелл. Core Java, Том 1 — Основы. sl: Прентис Холл (Sun Microsystems Press), 2008.
3. Перри, Дж. Стивен. Введение в программирование на Java, Часть 1: Java. sl: Корпорация IBM, 2010.
4. Р. Морелли, Р. Вальде. Java, Java, объектно-ориентированное решение задач Java. sl: Pearson Education, Inc., 2012.
5. Хортон, Айвор. Начиная с Java 2, JDK. sl: Wiley Publishing, 2005.
6. Дейтел, Х.М. Java – Как программировать. sl: Прентис Холл, 2004.
7. Хорстманн, Кей. Большая Ява. sl: Wiley Publishing, 2008.
8. Джеймс Гослинг, Билл Джой, Гай Стил. Спецификация языка Java. сл: Аддисон-Уэсли, 2011.
9. java.sun.com/j2se/1.4.2/docs/api. Документация по API Java. sl: Официальный Java-сайт.
10. Одерский, Мартин. Обзор языка программирования Scala. EPFL Лозанна, Швейцария: sn, 2004. IC/2004/64.
11. Профессор, доктор ир. В. ЙООСЕН, профессор, доктор медицинских наук. Ф. ПИССЕНС. Полиморфизм конструктора типов для Scala: теория и практика. sl: Католический университет Левена, 2009.
12. Лука Карделли, Симоне Мартини, Джон К. Митчелл и Андре Щедров. Расширение системы f с подтипованием. . sl: Теоретические аспекты компьютерного программного обеспечения, том 526 конспектов лекций по информатике, Pringer Berlin Heidelberg. , 1991.
13. Открытые проблемы объектно-ориентированного программирования. Мэдсен, Оле Лерманн. Кафедра компьютерных наук, Орхусский университет, Дания: sn