Министерство науки и высшего образования Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

Ю. В. Морозова

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск
2018

УДК 004.42.045(075.8)
ББК 32.973.2-018я73
М 801

Рецензенты:

В. В. Герасименко, канд. техн. наук, заместитель начальника отдела информационных технологий ООО «КДВ Групп»;

П. В. Сенченко, канд. техн. наук, доцент кафедры автоматизации обработки информации Томского государственного университета систем управления и радиоэлектроники

Морозова Ю. В.

М 801 Объектно-ориентированный анализ и программирование : учебное пособие / Ю. В. Морозова. – Томск : Эль Контент, 2018. – 140 с.

ISBN 978-5-4332-0269-6

В пособии изложены основы объектно-ориентированного программирования (ООП). Подробно рассмотрены базовые принципы ООП: полиморфизм, наследование, инкапсуляция и абстракция. Даны основы языка Java.

Для студентов направлений «Программная инженерия» и «Бизнес-информатика», а также всех, кто начинает изучать основы объектно-ориентированного программирования на языке Java.

ISBN 978-5-4332-0269-6

Оглавление

Предисловие

Данное учебное пособие призвано раскрыть основные понятия объектно-ориентированного программирования (ООП). Оно будет полезно начинающим программистам и тем, кто хочет самостоятельно научиться программировать и освоить язык программирования Java.

В пособии рассмотрены основы языка программирования Java. Изучение начинается с азов объектно-ориентированного программирования и синтаксиса описания классов на языке Java. Подробно изложена лексика языка, рассмотрены элементы программного кода, их назначение и особенности применения. Отдельная глава посвящена обработке исключительных ситуаций. Значительное внимание уделено фундаментальному механизму наследования типов с учетом особенностей Java.

Почему объектно-ориентированный подход к программированию стал приоритетным при разработке большинства программных проектов? ООП предлагает новый мощный способ решения проблемы сложности программ. Вместо того чтобы рассматривать программу как набор последовательно выполняемых инструкций, в ООП программа представляется в виде совокупности объектов, обладающих сходными свойствами и набором действий, которые можно с ними производить. Возможно, изложенные тезисы будут казаться вам непонятными, пока вы не изучите соответствующий раздел программирования более подробно. Но со временем вы не раз сможете убедиться в том, что применение объектно-ориентированного подхода делает программы понятнее, надежнее и проще в использовании.

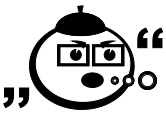
Язык Java представляет собой новейшую разработку в области объектно-ориентированных языков.

Соглашения, принятые в учебном пособии

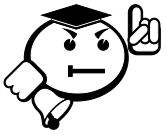
Для улучшения восприятия материала в данном учебном пособии используются пиктограммы и специальное выделение важной информации.

Эта пиктограмма означает определение или новое понятие.

Эта пиктограмма означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.

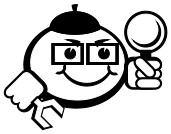


Эта пиктограмма означает цитату.



В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.

Эта пиктограмма означает совет. В данном блоке указаны более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.



Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Выводы

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по главе

1 Введение в методологию объектно-ориентированного программирования

Спасайся, кто может – началось
вторжение компьютеров.
Алан Купер

В настоящее время компьютеры используются во всех областях человеческой деятельности. Проникая во всевозможные сферы нашей жизни, компьютеры раздражают нас, приводят в ярость, а кое-кого даже убивают: жертвы социальных сетей, игроманы, интернет-зависимые и любители селфи. При этом мы испытаем соблазн уничтожить свои компьютеры, но не посмеем, потому что и сегодня уже полностью, необратимо зависим от этих многообещающих монстров, делающих современную жизнь возможной. Так вот, методология объектно-ориентированного программирования поможет нам сбежать из этой «психбольницы» [1], понять, как думают разработчики программ, и, наконец, самим стать разработчиками программ. Давайте рассмотрим, что вызвало появление объектно-ориентированного программирования (ООП).

1.1 Сложность программного обеспечения

В последние годы резко возросла сложность и размер решаемых задач. В этих условиях для увеличения эффективности решения различных задач уже недостаточно использования традиционных подходов. Более важным сейчас является выбор инструментария, т. е. системы программирования, языка программирования, т. к. современные системы содержат приемы, позволяющие эффективно решать большие задачи, контролировать и обрабатывать критические ситуации (исключения), работать с большими объемами данных.

Появление новых моделей процессоров и комплектующих, версий операционных систем и программного обеспечения происходит на фоне постоянного усложнения не только отдельных физических и программных компонентов, но и лежащих в их основе концепций. Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял функ-

циональные требования, что еще более усложняло процесс создания программного обеспечения.

Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 1980-х гг. возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало объектно-ориентированное программирование.

Таким образом, назрела настоятельная необходимость моделирования структуры и процесса функционирования программных систем до начала написания соответствующего кода. При этом непременным условием успешного завершения проекта стало построение предварительной *модели* программной системы.

Модель *(model) – абстракция физической системы, рассматриваемая с определенной точки зрения и представленная на некотором языке или в графической форме.*

С точки зрения общих принципов системного анализа одна и та же *физическая система* может быть представлена несколькими *моделями*. При этом назначение отдельной *модели* системы определяется характером решаемой проблемы. Основное требование к *модели* программной системы – она должна быть понятна заказчику и всем специалистам проектной группы, включая бизнес-аналитиков и программистов. Именно для разработки такой нотации потребовались усилия группы специалистов ведущих фирм – производителей программного и аппаратного обеспечения, которые привели к появлению языка *UML*.

UML *(Unified Modeling Language – унифицированный язык моделирования) – язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.*

UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели си-

стемы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования в основном программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

Средства UML-моделирования:

- Rational Rose;
- Microsoft Visual Studio .NET Enterprise Architect, Microsoft Visio.

Разработка и использование моделей языка UML осуществляется в рамках общей концепции *объектно-ориентированного анализа и проектирования*, которая, в свою очередь, является обобщением методологии *объектно-ориентированного программирования*.

Естественное стремление разработчиков программ – сократить время разработки, облегчить повторное использование отлаженных модулей и снизить издержки на сопровождение и модификацию программ. При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема в середине 1980-х гг. было предложено использовать объектный подход.

Объектно-ориентированное программирование *определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств* [2].

Основными достоинствами ООП по сравнению с модульным программированием является «более естественная» декомпозиция ПО, более полная локализация данных и интеграция их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов), новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Таким образом, любая работающая сложная система является результатом развития работавшей более простой системы.

Таким образом, программную систему можно разделить на подсистемы и совершенствовать независимо, при этом в уме не нужно держать информацию обо всей системе сразу.

1.2 Объектная декомпозиция

Декомпозиция *– разбиение сложной системы на части.*

В основе *структурного подхода* лежит декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших (до 40–50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название *процедурной декомпозиции*.

Процедурной декомпозицией называют процесс представления разрабатываемого программного обеспечения в виде совокупности вызывающих друг друга подпрограмм. Каждая подпрограмма в этом случае выполняет некоторую операцию, а вся совокупность подпрограмм решает поставленную задачу (рис. 1.1).

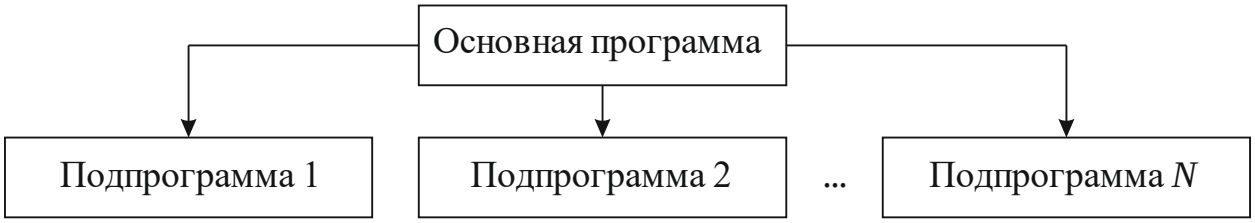


Рис. 1.1 – Процедурная декомпозиция

Объектной декомпозицией называют процесс представления предметной области задачи в виде совокупности функциональных элементов (*объектов*), обменивающихся в процессе выполнения программы входными воздействиями (*сообщениями*) (рис. 1.2).

Таким образом, при выполнении объектной декомпозиции определяют и описывают множество объектов предметной области и множество сообщений, которые формирует и получает каждый объект.

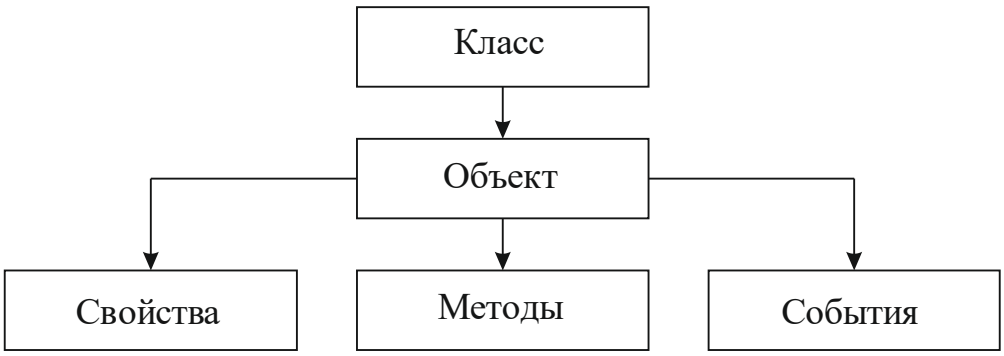


Рис. 1.2 – Объектная декомпозиция

1.3 Класс и объект

В объектно-ориентированном программировании базовыми единицами программ и данных являются объекты. Можно сказать, что в чисто объектно-ориентированной системе ничего, кроме объектов нет.

Объект *(object) – это осязаемая сущность, которая четко проявляет свое поведение* [2].

Объект состоит из следующих трех частей: имя объекта (*names*), состояние (*attributes*), поведение (*behaviors*) (рис. 1.3), т. е. объект обладает *состоянием*, *поведением* и *идентичностью*.

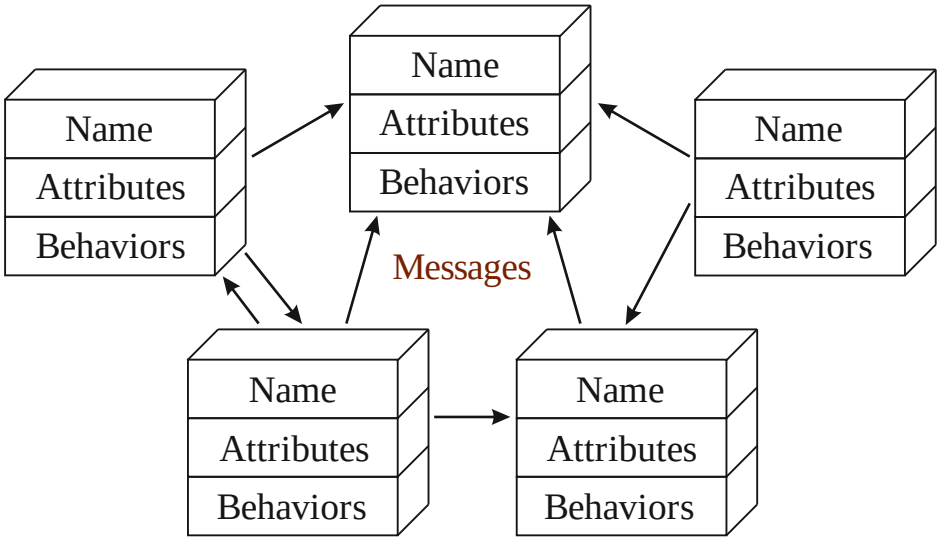


Рис. 1.3 – Представление объектов

Состояние *(attributes) объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.*

Поведение *(behaviors) – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.*

Идентичность *(names) – это такое свойство объекта, которое отличает его от всех других объектов* [2].

Основные черты ООП:

- все является объектом (за исключением примитивных типов данных);
- программа – это группа объектов, говорящих друг другу, что делать, посредством сообщений;
- каждый объект имеет собственную память, состоящую из других объектов;
- у каждого объекта есть тип. Каждый объект является экземпляром класса, здесь «класс» является аналогом слова «тип»;
- все объекты определенного типа могут получать одинаковые сообщения.

Идея классов отражает строение объектов реального мира – ведь каждый предмет или процесс обладает набором характеристик или отличительных черт.

Класс *(class) представляет собой набор объектов, которые обладают общей структурой и одинаковым поведением.*

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы.

Экземпляр класса *– это отдельная реализация класса, и все экземпляры класса имеют одинаковые свойства, которые описаны в определении класса.*

Объекты – это отдельные, четко обозначенные экземпляры некоторого класса. Класс дает общее описание объектов, указывает, «на что они похожи». Термины «экземпляр класса» и «объект» взаимозаменяемы.

1.4 Типы отношений между классами и объектами

Как правило, любая программа, написанная на объектно-ориентированном языке, представляет собой некоторый набор классов, связанных между собой (рис. 1.4).

Отношения *– определенная связь двух и более объектов.*

Рис. 1.4 – Типы отношений

Возможны следующие связи между классами в рамках объектной модели:

1) ассоциация (Association);
2) агрегация (Aggregation);
3) композиция (Composition);
4) обобщение/расширение/наследование (Inheritance).

Ассоциация

Если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера «владения» или контейнеризации, то такое отношение называют ***ассоциацией*** *(association).*

Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь друг с другом. Например, связь между объекта-

ми «Институт» и «Профессор» (рис. 1.5). Отношение ассоциации изображается линией, связывающей классы (простая, без ромбика).

Рис. 1.5 – Пример ассоциации

Мощность ассоциации (количество участников):

- «один-к-одному»;
- «один-ко-многим»;
- «многие-ко-многим».

Агрегация

Агрегация являются частными случаями ассоциации. Это более конкретизированные отношения между объектами.

*Отношение между классами типа «содержит» или «состоит из» называется **агрегацией** (aggregation) или **включением**.*

Например, «Факультет» входит в состав структуры «Института» (рис. 1.6). При агрегации объекты также имеют свой жизненный цикл, однако ограничены отношением принадлежности «HAS-A», то есть отношение «часть – целое» между двумя объектами.

Рис. 1.6 – Пример агрегации

Агрегация изображается линией с ромбиком на стороне того класса, который выступает в качестве владельца, или контейнера. Необязательное название отношения записывается посередине линии.

Число объектов, участвующих в отношении, записывается рядом с именем роли. Запись «0..n» означает «от ноля до бесконечности».

Приняты так же обозначения:

- «1..n» – от единицы до бесконечности;
- «0» – ноль;

- «1» – один;
- «n» – фиксированное количество;
- «0..1» – ноль или один.

Композиция

Композиция – еще более «жесткое» отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то.

Композиция *(Composition) – это разновидность жесткой взаимосвязи между объектами, составляющими класс. Когда объект уничтожается, объекты, составляющие его, также уничтожаются.*

Пример композиции – отношения объектов «Институт» и «Здание» (рис. 1.7).

Рис. 1.7 – Пример композиции

Наследование

Наследование *(inheritance) – это отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов.*

Наследование вводит иерархию «общее/частное», в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение (рис. 1.8).

Использование наследования способствует уменьшению количества кода, написанного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

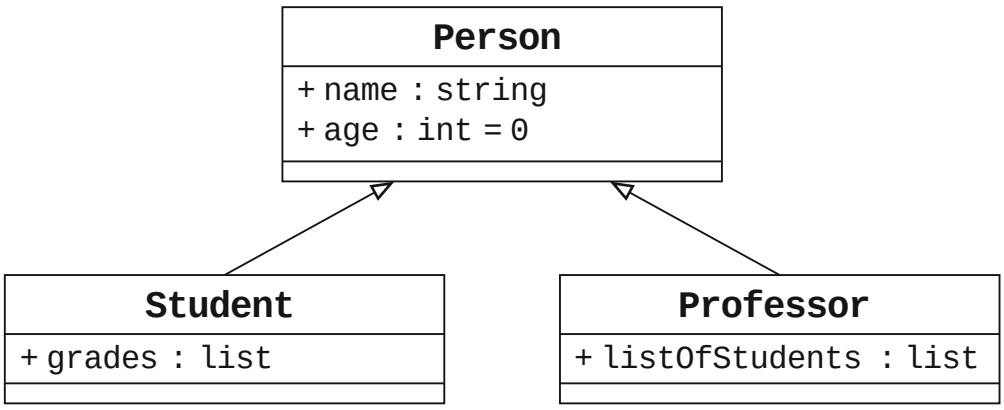


Рис. 1.8 – Пример наследования

1.5 Принципы ООП

ООП базируется на следующих основных принципах:

1) абстракция;
2) инкапсуляция;
3) наследование;
4) полиморфизм.

Абстракция

Абстрагирование является одним из основных методов, используемых для решения сложных задач.

Гради Буч дает следующее определение абстракции.

Абстракция *(abstraction) выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя* [2].

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Для этого необходимо *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции

сделает модель очень сложной, перегруженной деталями и потому непригодной.

Разница между классом и объектом такая же, как между абстрактным понятием и реальным объектом. Ниже приведены виды абстракций (степень связи с реальным объектом уменьшается с каждым пунктом):

1) *instance* (объект) – сущность, обладающая набором характеристик, свойственных конкретному экземпляру класса. Имеет конкретные значения полей (низший уровень, без абстракции);
2) *class* (класс) – описание множества объектов, схожих по свойствам и внутренней структуре (шаблон для создания объектов);
3) *abstract class* (абстрактный класс) – абстрактное описание характеристик множества классов (выступает как шаблон для наследования другими классами). Имеет высокий уровень абстракции, в связи с чем от абстрактного класса нельзя создавать объекты напрямую (только через создание объектов от классов-наследников);
4) *interface* (интерфейс) – это конструкция языка программирования Java, в рамках которой могут описываться только абстрактные публичные методы (*abstract public*) и статические константы свойства (*final static*). То есть так же, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.

Абстрактные классы и интерфейсы рассмотрим позже.

Так как *интерфейс* схож с *абстрактным классом*, но позволяет (неявно) выполнить множественное расширение, он имеет максимальный уровень абстракции.

Инкапсуляция

При использовании объектно-ориентированного подхода не принято использовать прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято использовать специальные методы этого класса для получения и изменения его свойств (*сеттеры* и *геттеры*). Внутри объекта данные и методы могут обладать различной степенью

открытости (или доступности). Инкапсуляция выступает договором для объекта, что он должен скрыть, а что открыть для доступа другими объектами.

Инкапсуляция *(encapsulation) – это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса).*

В Java используют модификаторы доступа, чтобы скрыть метод и ограничить доступ к переменной из внешнего мира. Java также располагает различными модификаторами доступа: `public` (по умолчанию), `protected`, `private`. Если рассматривать с позиции инкапсуляции, то модификаторы доступа позволяют ограничить нежелательный доступ к членам класса извне (рис. 1.9).

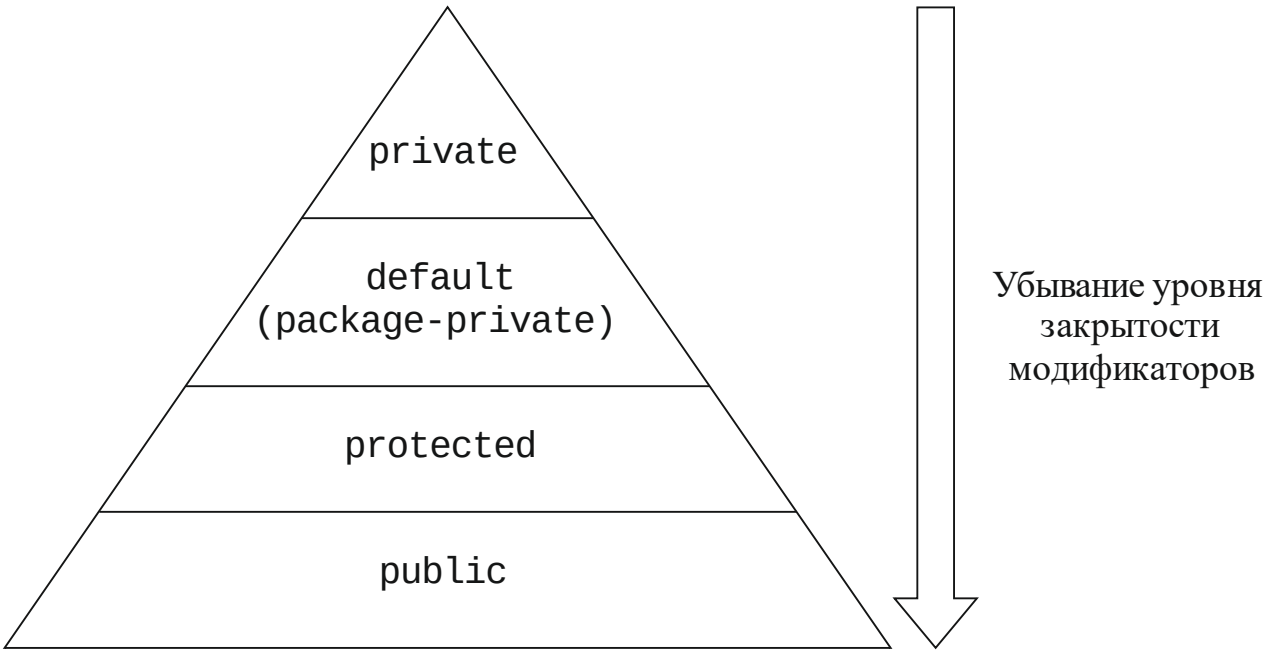
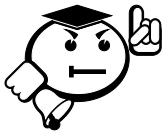


Рис. 1.9 – Уровни доступа



Абстракция и *инкапсуляция* дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством.

Наследование

Наследование – один из основополагающих принципов ООП.

Наследование *– свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется* ***базовым***, ***родительским*** *или* ***суперклассом****. Новый класс –* ***потомком***, ***наследником***, ***дочерним*** *или* ***производным классом****.*

Главное преимущество наследования в том, что оно обеспечивает формальный механизм повторного использования кода и избегает дублирования.

Унаследованный класс расширяет функциональность приложения благодаря копированию поведения родительского класса и добавлению новых функций. Это делает код сильно связанным. Если вы захотите изменить суперкласс, вам придется знать все детали подклассов, чтобы не разрушить код. Но наследование обладает и недостатком – *сильная связанность*: подкласс зависит от реализации родительского класса, что делает код сильно связанным.

Целью наследования является упорядочение классов в иерархическую древовидную структуру.

Иерархия классов *представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях.*

Рассмотрим некоторую иерархию классов, представленную на рисунке 1.10. На рисунке показаны классы иерархии, кто кого наследует и методы каждого класса.

Видим, что класс `Cat` является наследником класса `Pet`. `Pet`, в свою очередь, наследник класса `Animal`.

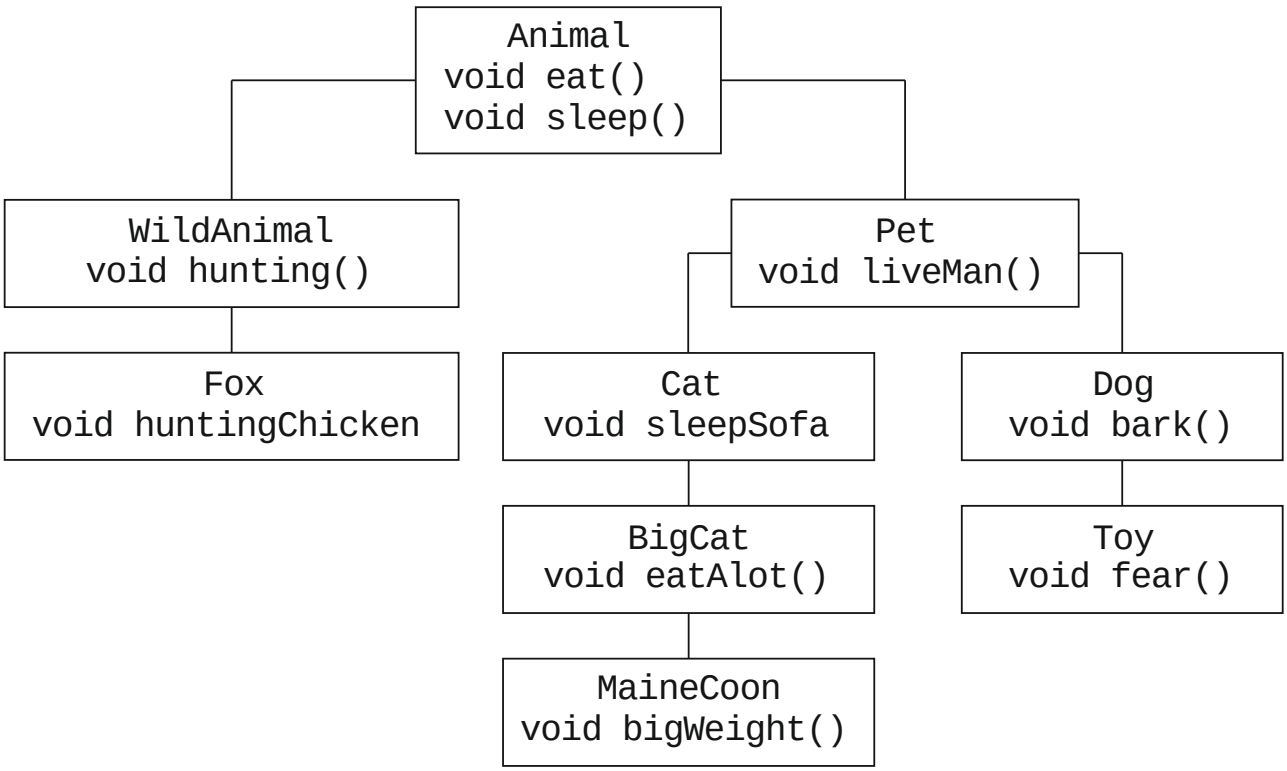


Рис. 1.10 – Пример иерархии классов

Полиморфизм

Если мы имеем объекты, которые принадлежат одной и той же ветви иерархии (были унаследованы), то для них можно использовать единый интерфейс, который будет для каждого объекта производить однотипное действие, но результат для каждого объекта будет различным (зависящим от этого конкретного объекта).

Полиморфизм *(polymorphism) – положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов.*

Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

Здесь действует принцип «Один интерфейс – много методов». Благодаря полиморфизму программы становятся менее сложными, так как для определения и выполнения однотипных действий служит единый интерфейс. Такой единый интерфейс применяется пользователем или программистом к объектам разного типа, а выбор конкретного метода для реализации соответствующей команды осуществляется компьютером в соответствии с типом объекта, для ко-

торого выполняется команда. Пожалуй, полиморфизм – это лучшее, что есть в объектно-ориентированном программировании.

Знакомство с ООП будет проходить с помощью языка программирования Java.

Контрольные вопросы по главе 1

1. Назовите принципы ООП и расскажите о каждом.
2. Дайте определение понятию «класс».
3. Что такое поле/атрибут класса?
4. Как правильно организовать доступ к полям класса?
5. Какие модификации уровня доступа вы знаете? Расскажите про каждый из них.
6. Дайте определение понятию «метод».
7. Какие задачи решает абстракция?
8. Чем отличается объект от экземпляра класса?
9. Чем отличается отношение композиция от ассоциации?
10. Приведите пример полиморфизма.

2 Основы языка Java

Когда вы произносите слово «Java», то, что бы ни скрывалось за ним, на первый план выступают позитивные аспекты фирменного названия.
Вы чувствуете: это нечто солидное, люди возбуждаются, стремятся приобщиться.
Даже таксисты знают: Java – это класс!

Эрик Шмидт, президент Novell

Крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки GitHub, который еще называют «социальной сетью для разработчиков» использует методику определения популярного языка программирования. Система под названием *PYPL* (PopularitY of Programming Languages) основана на количестве поисковых запросов руководств по конкретному языку программирования.

Согласно рейтингу языков программирования по версии GitHub, по состоянию на 2017 г. на первом месте Java – основной язык, используемый для разработки Android-приложений для смартфонов и планшетов (рис. 2.1) [3].

Популярность Java у разработчиков связана с простотой и надежностью языка, который обеспечивает долгосрочную совместимость написанных на нем продуктов.

Java *– кросс-платформенный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретенной компанией Oracle).*

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java. Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Еще во времена первой и открытой версии 1.02 этот язык покорил разработчиков своим дружественным

синтаксисом, объектно-ориентированной направленностью, управлением памятью и, что важнее всего, перспективой переносимости на разные платформы.

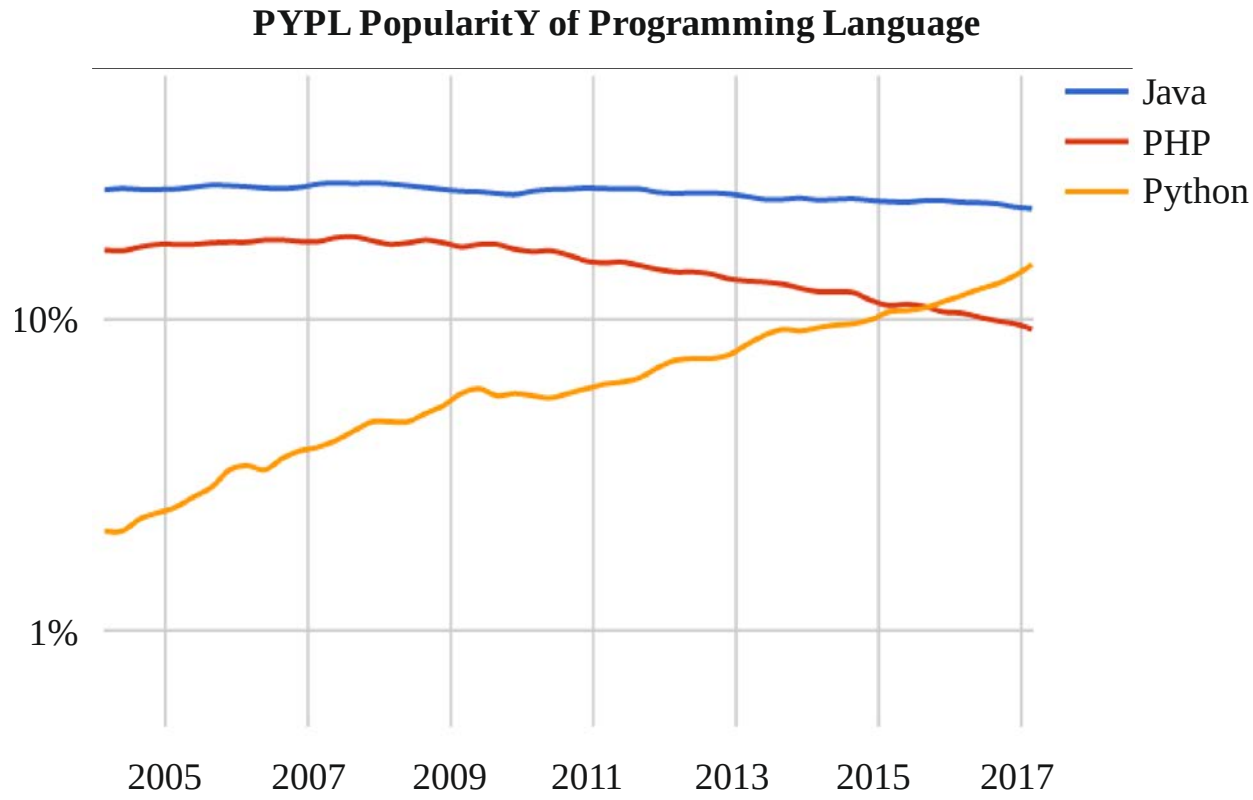


Рис. 2.1 – Рейтинг языков программирования по версии GitHub, по состоянию на 2017 г.

2.1 История создания Java

Возникновение языка Java приходится на драматичное время, когда бурно развивался Интернет и началась так называемая «война браузеров». Первоначально он предназначался для программирования бытовой техники.

Если поискать в Интернете историю создания Java, то можно выяснить, что изначально язык назывался Oak (Дуб), а работа по его созданию началась еще в 1990 г. с довольно скандальной истории внутри корпорации *Sun*. Фирма Sun была известна на рынке аппаратного и программного обеспечения такими продуктами, как процессоры на RISC-архитектуре и собственной операционной системой SOLARIS из семейства UNIX.

События, породившие нынешний язык Java, начались в компании Sun в 1990 г. Над разработкой программного обеспечения для бытовой техники трудилась группа под руководством Джеймса Гослинга [4].

Джеймс Гослинг – это один из наиболее известных людей в истории UNIX. Он много работал с машинами PDP-8, написал текстовый редактор EMACS, работал в компаниях DEC, IBM, Sun (с 1983 г.). 2 апреля 2010 г. уво-

лился из Sun Microsystems после того, как она была поглощена корпорацией Oracle.

В августе 1991 г. у Гослинга уже была графика, работающая в его новом языке, который он назвал Oak (Дуб) в честь дерева, видного из окна его офиса. Этот язык впоследствии был переименован в Java.

Демоверсия вышла в августе 1992 г. и произвела фурор. Это был объектно-ориентированный язык с возможностью сильно распределенной по сети работы. А кроме того, язык Oak, работая с распределенной вычислительной средой, содержит в своем ядре процедуры защиты, шифрования и аутентификации, так что, в сущности, система безопасности пользователям не видна.

Первоначально возникли большие проблемы с коммерческим продвижением новой разработки на рынке – производители бытовых приборов оказались не заинтересованными в новой технологии. За отсутствием прибыльной стратегии проект электронной приставки отложили на полку, а команде интерактивного телевидения, переименованной в Sun Interactive, поручили в содружестве с фирмой Thomson Consumer Electronics разрабатывать приставку меньших габаритов и видеосерверы – уже без Oak.

В январе 1995 г. новая написанная Гослингом версия Oak была наречена более привлекательным именем Java в честь названия популярного в США сорта кофе. Работа Ноутона вылилась в навигатор Web и впоследствии получила имя HotJava. Проект получил поддержку и одобрение профессионалов.

В 2010 г. корпорация Sun, а также все ее программные продукты и технологии (включая язык Java) стали принадлежать корпорации Oracle.

В самой первой версии Java Development Kit (JDK, средство разработки на Java) был пример апплета, представляющий простейшие электронные таблицы. Вскоре появился текстовый редактор, позволяющий менять стиль и цвет текста. Конечно, были игровые апплеты, обучающие, моделирующие физические и иные системы. Например, клиент, сделавший заказ в магазине или отправивший посылку почтой, получал возможность следить за доставкой через интернет.

В отличие от обычных программ, апплеты получили «в наследство» важное свойство HTML-страниц. Прочитав сегодня содержание страницы новостей, клиент не сохраняет ее на своем компьютере, а на следующий день читает обновленное содержание. Точно так же, скачав апплет и поработав с ним, можно удалить его, а в следующий раз получить более новую версию. Таким образом, программы появляются и исчезают с машины клиента безо всякого усилия,

не требуются ни специальные знания, ни действия, и при этом автоматически поддерживаются самые последние версии.

2.2 Технологии Java

Существует несколько основных семейств технологий Java, которые описаны в таблице 2.1 [4].

Таблица 2.1 – Технологии Java

Технология	Описание
Java SE (Java Standard Edition)	Основная технология Java, включающая компиляторы, API, Java Runtime Environment. Используется для создания пользовательских настольных приложений (desktop)
Java EE (Java Enterprise Edition)	Технология создания программного обеспечения уровня предприятия. Используется для разработки веб-приложений
Java ME (Java Micro Edition)	Технология создания программ для устройств, ограниченных по вычислительной мощности, например, мобильных телефонов
Java FX	Технология создания графических интерфейсов корпоративных приложений и бизнеса
Java Card	Технология создания программ для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объемом

Java активно используется для создания мобильных приложений для операционной среды Android.

2.3 Версии Java

Java 1.0

В 1996 г. была выпущена первая официальная версия – Java 1.0. Основными продуктами, доступными на тот момент в виде бета-версий, были:

- Java language specification, JLS, спецификация языка Java (описывающая лексику, типы данных, основные конструкции и т. д.);
- спецификация JVM;

- Java Development Kit – средство разработчика, состоящее в основном из утилит, стандартных библиотек классов и демонстрационных примеров.

Java 1.2

Обновленная спецификация JDK 1.2 была разработана в 1998 г. и вышла под наименованием Java 2. Язык практически не изменился, но платформа получила ряд дополнений:

- библиотеку Swing для разработки пользовательского интерфейса;
- набор коллекций;
- поддержку файлов и цифровых сертификатов пользователя;
- библиотеку Accessibility;
- Java 2D;
- поддержку технологии drag-and-drop;
- полную поддержку Unicode, включающую японский, китайский и корейский языки;
- поддержку воспроизведения аудиофайлов нескольких форматов;
- JIT-компилятор.

Java 5.0

В 2004 г. вышла спецификация Java 5.0. С разработки данной версии была изменена официальная индексация; вместо Java 1.5 правильно называть Java 5.0. Внутренняя же индексация Sun осталась без изменений – 1.x.

Минорные изменения включаются без изменения индексации. Для этого используется слово «Update», например, Java Development Kit 5.0 Update 25. Предполагается, что в обновления могут входить как исправления ошибок, так и небольшие добавления в API виртуальной машины JVM.

В версии Java 5.0 был внесен целый ряд принципиальных дополнений:

- перечислимые типы *enum*;
- аннотации – возможность добавления в текст программы метаданных, не влияющих на выполнение кода, но допускающих использование для получения различных сведений о коде и его исполнения;
- методы с неопределенным числом параметров;
- разрешен импорт статических полей и методов;
- в коллекции можно использовать *Iterator* объектов (*foreach*);

- использование *javadoc* комментариев для автоматического оформления документации;
- средства обобщенного программирования *generics*.

Java 6

В декабре 2006 г. вышел очередной релиз Java 6. Вместе с этим релизом внесены изменения в официальную индексацию – вместо Java 6.0 версия значится как Java 6. Минорные изменения, как и в предыдущей версии, вносятся в обычные обновления версии, например, Java Standard Edition Development Kit 6 Update 25.

В версии Java 6 внесены следующие изменения:

- в коллекции (наборы данных) добавлены интерфейсы для организации очереди;
- в Swing улучшена работоспособность OpenGL и DirectX;
- добавлен GifWriter для работы с файлами .gif;
- стали доступны классы-потоки для чтения и передачи сжатых данных, с возможностью передачи их по сети;
- архивация – сняты ограничения на количество файлов в архиве (ранее 64 Кб), увеличена длина названия файла (ранее 256 символов);
- сняты ограничения на количество одновременно открытых файлов (ранее было 2 000);
- организована система управления кешем и добавлена поддержка параметра no-cache в HTTP-запросе;
- наряду с уже существующими григорианским и буддийским календарями добавлена поддержка японского императорского календаря;
- можно использовать Java HTTP Server для создания полноценного HTTP-сервера с минимально необходимыми функциональными свойствами;
- увеличена скорость вычислений и скорость операций ввода-вывода.

Java 7

Выпуск релиза версии Java 7 состоялся в июле 2011 г.

В новой версии, получившей название Java Standard Edition 7, помимо исправления ошибок было представлено несколько новшеств:

- добавлен новый более быстрый верификатор типов, получивший название «проверяющий типы» (typechecking verifier);
- в коллекции (наборы данных) добавлены интерфейсы для организации очереди;
- модификация загрузчика классов (class-loader);
- URLClassLoader – освобождение ресурсов, которые держит class-loader, методом `close()`;
- JDBC обновлен до релиза 4.1, Rowset до версии 1.1;
- добавлен новый look-and-feel следующего поколения;
- nio.2 – новые интерфейсы для доступа к файловой системе, масштабируемого асинхронного IO-взаимодействия, полноценной работы с zip/jar архивами как с файловой системой;
- поддержка версии Unicode 6.0;
- generic – изменение вывода типа при создании объекта;
- Locale – разделены локали пользователя и графического интерфейса.

Java 8

Выпуск релиза версии Java 8 состоялся в марте 2014 г. Были внесены следующие изменения:

- полноценная поддержка лямбда-выражений;
- ключевое слово `default` в интерфейсах для поддержки функциональности по умолчанию;
- ссылки на методы;
- функциональные интерфейсы (предикаты, поставщики и т. д.);
- потоки (*stream*) для работы с коллекциями;
- новое API для работы с датами.

Java 9

Релиз Java 9 был несколько раз отложен. Oracle должен был исправить некоторые проблемы безопасности и критические технические вопросы. В итоге первый релиз состоялся в сентябре 2017 г., а в октябре, после исправления 12 багов, произошел переход на новую систему нумерации – Java SE 9.0.1.

Если Java 8 может быть описана как версия с упором на лямбды, стримы и изменения API, то Java 9 полностью посвящена Jigsaw (основная цель – раз-

бить JRE и принести модульность в компоненты ядра Java), дополнительным утилитам и изменениям ядра.

Java 10

В марте 2018 г. вышла новая версия, в которой можно выделить следующие новшества:

- вывод типа локальной переменной: улучшает язык Java, позволяя сократить объявление переменной до ключевого слова `var`;
- Parallel Full GC для G1: усовершенствованный сборщик мусора;
- Application Class-Data Sharing: ряд наработок, позволяющих классам приложений размещаться в одном архиве;
- экспериментальный JIT-компилятор на базе Java: новый JIT-компилятор Java VM.

2.4 Платформа Java

Язык программирования Java является полностью объектно-ориентированным. Это означает, что программа, написанная на языке Java, должна строго соответствовать парадигме объектно-ориентированного программирования (ООП). Следует понимать, что принципы ООП не просто определяют структуру программы. Это некий фундаментальный подход, если угодно, философия программирования, на которой имеет смысл остановиться подробнее перед непосредственным изучением основ языка Java.

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java.

Виртуальная машина Java *(Java Virtual Machine, JVM) – это программа (виртуальный компьютер), которая обрабатывает байт-код и передает инструкции оборудованию как интерпретатор.*

Одним из основных достоинств данного способа выполнения программ является полная независимость от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина.

. . . . . .

Компиляция *(compilation) – преобразование программы, написанной на языке программирования, в программу на другом языке (как правило, машинном) путем последовательной замены операторов исходной программы эквивалентной последовательностью команд машинного языка. Вычислительная машина выполняет скомпилированную программу вместо исходной.*

Интерпретация *(interpretation) – пооператорное выполнение исходной программы с помощью программы-интерпретатора. Интерпретатор анализирует каждый оператор исходной программы и непосредственно реализует эквивалентную последовательность команд машинного языка.*

. . . . . .

Исходные коды (тексты программ) Java переводятся компилятором не в машинный код, а в так называемый байт-код. Инструкции в байт-коде выполняет виртуальная машина Java.

. . . . . .

Байт-код Java *(byte-code) – программа, созданная компилятором на языке JVM.*

. . . . . .

Компиляция не зависит от типа какого-либо конкретного процессора и архитектуры конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы, программу не надо перекомпилировать под разные платформы. Байт-коды записываются в одном или нескольких файлах, могут храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодами. Затем полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM. Так достигается переносимость программы на Java: один и тот же байт-код одинаково работает на любой платформе, где установлена нужная версия виртуальной машины.

При этом устранение большинства ошибок происходит на этапе компиляции. Java ограничивает вас в нескольких ключевых областях и таким образом способствует обнаружению ошибок на ранних стадиях разработки программы. В то же время в ней отсутствуют многие источники ошибок, свойственных другим языкам программирования (строгая типизация, например). Большинство используемых сегодня программ «отказывают» в одной из двух ситуаций: при

выделении памяти либо при возникновении исключительных ситуаций. В традиционных средах программирования распределение памяти является довольно нудным занятием – программисту приходится самому следить за всей используемой в программе памятью, не забывая освобождать ее по мере того, как потребность в ней отпадает. Зачастую программисты забывают освобождать захваченную ими память или, что еще хуже, освобождают ту память, которая все еще используется какой-либо частью программы. Исключительные ситуации в традиционных средах программирования часто возникают в таких случаях, как деление на ноль или попытка открыть несуществующий файл, и их приходится обрабатывать с помощью неуклюжих и нечитабельных конструкций (кроме Delphi). Java фактически снимает обе эти проблемы, используя сборщик мусора (*garbage collector*) для освобождения незанятой памяти и встроенные объектно-ориентированные средства для обработки исключительных ситуаций. Java поддерживает механизм исключений.

Фирма Sun и ее партнеры создали JVM практически для всех современных операционных систем. Когда речь идет о браузере с поддержкой Java, подразумевается, что в нем имеется встроенная виртуальная машина.

При этом не важен ни тип процессора, ни архитектура компьютера. Так реализуется принцип Java «Write once, run anywhere» – «Написано однажды, выполняется где угодно».

Основное достоинство языка Java – именно в его кросс-платформенности. Байт-код не зависит от оборудования и легко переносим. Также основным достоинством Java является поддержка многопоточных программ, включая возможность запретить выполнение одного и того же задания двумя потоками одновременно. В Java есть встроенная поддержка многопотокового выполнения приложений.

2.5 Разработка программ на языке Java

На рисунке 2.2 приведена структурная схема жизненного цикла разработки и запуска программы на языке Java.

Для разработки программ на языке Java потребуется специальное программное обеспечение.

Минимальный комплект для разработки программ на Java (рис. 2.3):

- JRE (Java Runtime Environment) – среда выполнения;

- JDK (Java Development Kit) – компилятор и библиотеки;
- JVM.

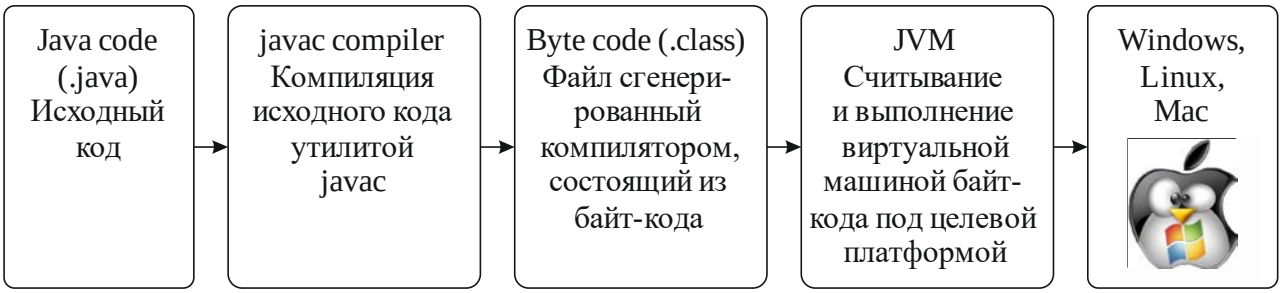


Рис. 2.2 – Схема жизненного цикла разработки и запуска программы на языке Java

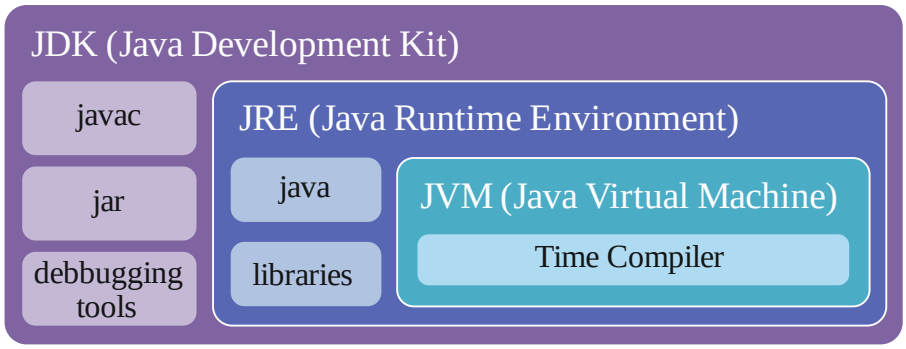


Рис. 2.3 – Минимальный комплект для разработки программ на Java

Java Development Kit (JDK)

***JDK** (Java Development Kit) – комплект разработки программного обеспечения (компилятор, стандартные библиотеки и т. п.).*

Набор программ и классов JDK:

- компилятор из исходного текста в байт-коды javac;
- интерпретатор java, содержащий реализацию JVM;
- облегченный интерпретатор jre (в последних версиях отсутствует);
- программа просмотра апплетов appletviewer, заменяющая браузер;
- отладчик jdb;
- дизассемблер javap;
- программа архивации и сжатия jar;
- программа сбора и генерирования документации javadoc;
- программа генерации заголовочных файлов языка C для создания «родных» методов javah;

- программа генерации электронных ключей keytool;
- программа native2ascii, преобразующая бинарные файлы в текстовые;
- программы rmic и rmiregistry для работы с удаленными объектами;
- программа serialver, определяющая номер версии класса;
- библиотеки и заголовочные файлы «родных» методов;
- библиотека классов Java API (Application Programming Interface).

Компания Sun Microsystems активно развивала и обновляла JDK, почти каждый год выходили новые версии. В первых версиях JDK все пакеты библиотеки Java API были упакованы в один архивный файл classes.zip и вызывались непосредственно из этого архива, его не нужно было распаковывать. Затем набор инструментальных средств JDK был сильно переработан.

В версии J2SE JDK 1.5.0, вышедшей в конце 2004 г., было уже под сотню пакетов, составляющих Core API (Application Programming Interface). В упакованном виде это файл размером около 46 Мбайт и необязательный файл с упакованной документацией такого же размера. В это же время произошло очередное переименование технологии.

В шестой версии, вышедшей в начале 2007 г., из названия технологии убрали цифру 2 и стали писать *Java Platform, Standard Edition 6*, сокращенно – Java SE 6 и JDK 6.

В марте 2018 г. вышел релиз JDK 10.0.1. Релиз одиннадцатой версии состоялся в сентябре 2018 г.

Java Runtime Environment (JRE)

***JRE** (Java Runtime Environment) – это программа для запуска и исполнения программ (среда выполнения Java).*

Хотя JRE входит в состав JDK, корпорация Oracle распространяет этот набор и отдельным файлом. JRE состоит из виртуальной машины Java, бинарных файлов и других классов. JRE не содержит инструменты для разработки (компилятор Java, отладчик и т. д.).

Самые новые версии системного программного обеспечения, необходимого для поддержки, можно загрузить с сайта компании Sun (http://java.sun.com/).

Интегрированные среды разработки (IDE)

Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ (Integrated Development Environment, IDE) для Java, и их число все время возрастает. Некоторые из них, такие как Eclipse, IntelliJ IDEA, NetBeans, являются просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор. Eclipse содержит собственный компилятор.

Другие интегрированные среды содержат JDK в себе или имеют собственный компилятор, например JBuilder фирмы Embarcadero или IBM Rational Application Developer. Их можно устанавливать, не имея под руками JDK. Надо заметить, что перечисленные продукты сами написаны полностью на Java.

Большинство интегрированных сред являются средствами визуального программирования и позволяют быстро создавать пользовательский интерфейс, т. е. относятся к классу средств RAD (Rapid Application Development).

Выбор какого-либо средства разработки диктуется, во-первых, возможностями вашего компьютера, ведь визуальные среды требуют больших ресурсов; во-вторых, личным вкусом; в-третьих, уже после некоторой практики, достоинствами компилятора, встроенного в программный продукт.

К технологии Java подключились и разработчики CASE-средств. Например, популярный во всем мире продукт Rational Rose может сгенерировать код на Java.

Для изучения Java, пожалуй, удобнее всего интегрированная среда Eclipse, в которой будут показаны все примеры данного пособия.

Контрольные вопросы по главе 2

1. В каком году версия Oak была наречена более привлекательным именем Java?
2. Перечислите технологии Java.
3. Перечислите основные новшества в версиях Java 8 и 9.
4. Чем отличается компиляция от интерпретации?
5. Во что преобразуются компилятором при компиляции исходные коды (тексты программ) Java?
6. Что входит в состав программ и классов JDK?
7. Что входит в состав программ и классов JRE?

8. Для чего нужна JVM?
9. Для чего нужны IDE?
10. Назовите наиболее популярные IDE.

3 Синтаксис и структура языка Java

По давней традиции, восходящей к языку Си, учебники по языкам программирования начинаются с программы «Hello, World!». Не будем нарушать эту традицию.

```
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Всякая программа, написанная на языке Java, представляет собой один или несколько классов, в этом простейшем примере только один *класс* (*class*) (рис. 3.1).

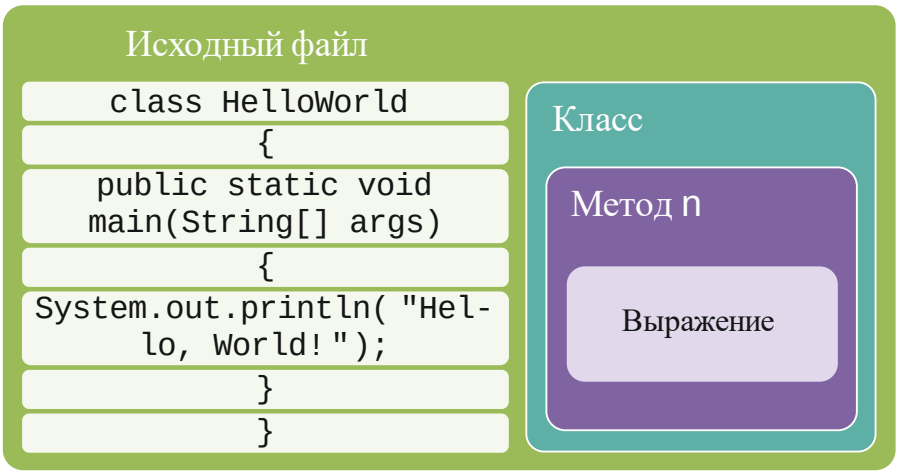


Рис. 3.1 – Программа, написанная на языке Java

Начало класса отмечается служебным словом `class`, за которым следует имя класса, выбираемое произвольно, в данном случае это имя `HelloWorld`. Все, что содержится в классе, записывается в фигурных скобках и составляет *тело класса* (*class body*) (рис. 3.2).

Все действия в программе производятся с помощью методов обработки информации, коротко говорят просто *метод* (*method*). Методы используются в объектно-ориентированных языках вместо функций, применяемых в процедурных языках.

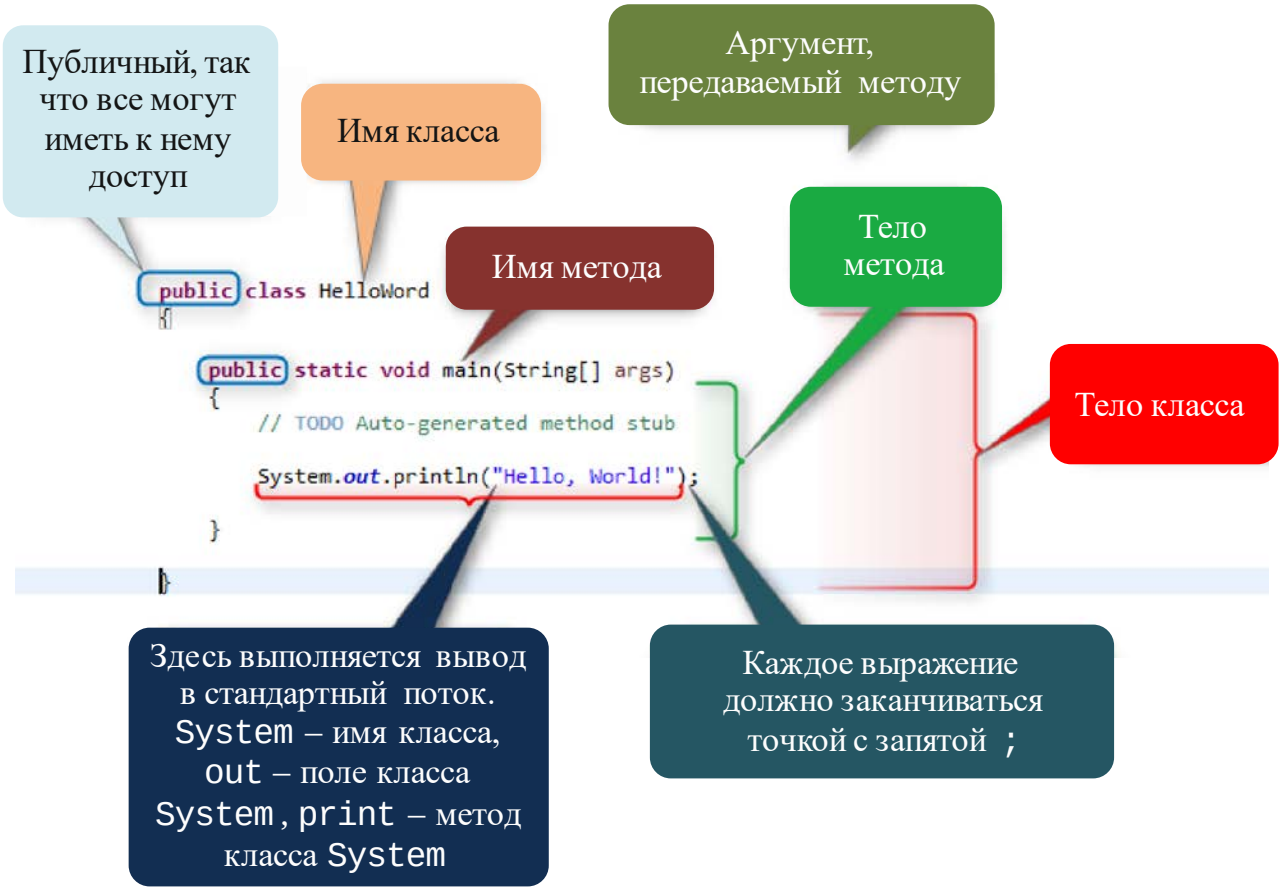


Рис. 3.2 – Программа, написанная на языке Java

Методы различаются по именам и параметрам. Один из методов обязательно должен называться `main`, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя его `main` (исключение составляют апплеты – у них метода `main()` нет). Метод `main()` иногда называют главным методом программы, поскольку во многом именно с этим методом отождествляется сама программа.

Ключевые слова `public`, `static` и `void` перед именем метода `main()` означают буквально следующее: `public` – метод доступен вне класса, `static` – метод статический и для его вызова нет необходимости создавать экземпляр класса (то есть объект), `void` – метод, который не возвращает результат. Модификаторы и уровни доступа рассмотрим немного позже.

Инструкция `String[] args` в круглых скобках после имени метода `main()` означает тип аргумента метода: формальное название аргумента `args`, и этот аргумент является текстовым массивом (тип `String`).

Все, что содержит метод, *тело метода* (*method body*), записывается в фигурных скобках.

Фигурными скобками в языке программирования Java (как и C++ и C#) отмечаются блоки программного кода. Программный код размещается между открывающей (символ `{`) и закрывающей (символ `}`) фигурными скобками.

В примере выше использовано две пары фигурных скобок. Первая, внешняя, пара использована для определения программного кода класса, вторая – для определения метода этого класса.

Единственное действие, которое выполняет метод `main()` в нашем примере, заключается в вызове другого метода со сложным составным именем `System.out.println` и передаче ему на обработку одного аргумента – строкового литерала `"Hello, World!"`. Строковые литералы записываются в кавычках, которые являются только ограничителями и не входят в текст.

Составное имя `System.out.println` означает, что в классе `System`, входящем в Java API, определяется переменная с именем `out`, содержащая экземпляр одного из классов Java API, класса `PrintStream`, в котором есть метод `println()`. Все это станет ясно позднее, а пока просто будем писать это длинное имя.

Сделаем сразу важное замечание. Имена `main`, `Main`, `MAIN` различны с точки зрения компилятора Java. В примере важно писать `String`, `System` с заглавной буквы, а `main` – со строчной.

Язык Java различает прописные и строчные буквы.

В именах нельзя оставлять пробелы. Свои имена можно записывать как угодно, можно было бы дать классу имя `helloworld` или `helloWorld`, но между Java-программистами заключено соглашение, называемое Code Conventions for the Java Programming Language [5].

Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной (заглавной) буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы;

- имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается с прописной буквы;
- имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания.

Имя файла должно в точности совпадать с именем класса, содержащего метод `main()`. Данное правило очень желательно выполнять. При этом система исполнения Java будет быстро находить метод `main()` для начала работы, просто отыскивая класс, совпадающий с именем файла. Расширение имени файла должно быть `.java`.

Называйте файл с программой именем класса, содержащего метод `main()`, соблюдая регистр букв.

В нашем примере сохраним программу в файле с именем `HelloWorld.java` в текущем каталоге (рис. 3.3).

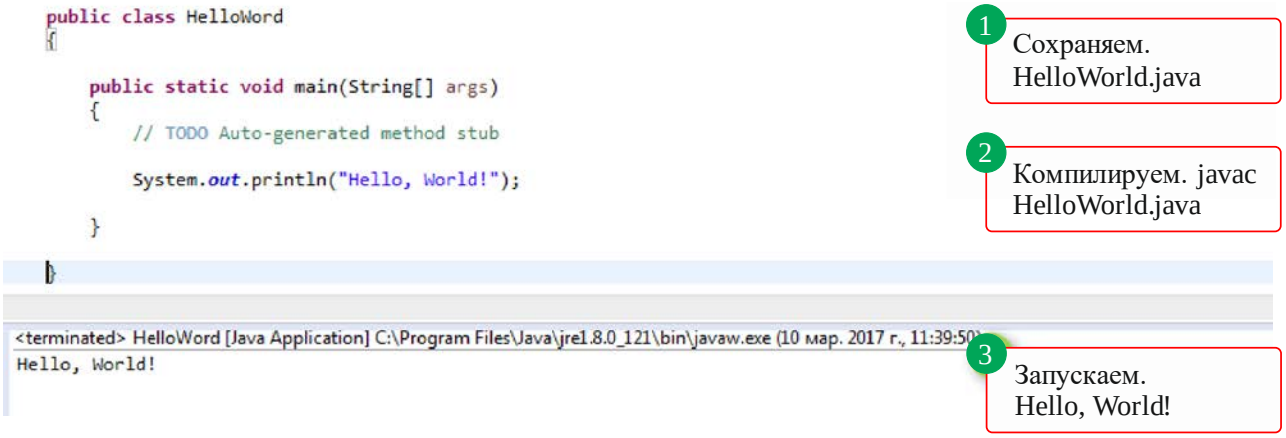


Рис. 3.3 – Запуск программы, написанной на языке Java

Затем вызовем компилятор, передавая ему имя файла в качестве аргумента:

```
javac HelloWorld.java
```

Компилятор создаст файл с байт-кодами, даст ему имя `HelloWorld.class` и запишет этот файл в текущий каталог.

Осталось вызвать интерпретатор байт-кодов, передав ему в качестве аргумента имя класса (а не файла!):

```
java HelloWorld
```

На экране появится строка:

```
Hello, World!
```

Выводы

Основы основ синтаксиса языка Java:

- Язык Java различает прописные и строчные буквы. Это означает, что имена всех функций и ключевые слова следует записывать в точности так, как они значатся в примерах и справочниках.
- Каждая команда (оператор) в языке Java должна заканчиваться точкой с запятой.
- Программа на Java состоит из одного или нескольких классов. Абсолютно вся функциональная часть программы (т. е. то, что она делает) должна быть помещена в методы тех или иных классов.
- Классы группируются в пакеты.
- Хотя бы в одном из классов должен существовать метод `main()`, в точности такой, как в рассмотренном нами примере (Eclipse все сгенерирует сам, если поставить нужную галочку). Именно этот метод и будет выполняться первым.

Программы на Java могут быть написаны с использованием набора символов Unicode, каждый символ в котором представляется при помощи 16 бит.

Unicode *– это стандарт кодирования множества национальных алфавитов, видов письма, технических и математических символов, разработанный для того, чтобы решить проблемы интернационализации в многоязычной компьютерной среде.*

Более ранний стандарт ASCII, в котором каждый символ представлен одним байтом, был пригоден только для европейских языков. Таблица ASCII вошла в набор Unicode как его подмножество. Собственно Unicode – это набор (последовательность) символов, который можно закодировать разными способами. Язык Java использует кодировку UTF-8, которая отводит по одному байту для букв западноевропейских алфавитов, по два байта – для восточноевропейских, а для экзотических языков – три и более байта.

Лексически программы на Java состоят из разделителей строк (символы возврата каретки, перевода строки или их комбинация), комментариев, пустых символов (пробелы и табуляции), идентификаторов, ключевых слов, литералов, операторов и разделительных символов.

В Java ключевые слова не могут использоваться в качестве идентификаторов.

В настоящее время в языке Java определено 50 ключевых слов.

3.1 Комментарии

В текст программ Java, как и в текст программ на других языках программирования, можно вставить свои комментарии.

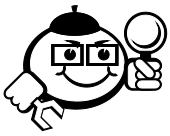
Комментарии *(comments) – некоторые пояснения, напоминания или просто текст, предназначенный для пользователя. Важно отметить, что комментарий предназначен не для компилятора, поэтому компилятором он игнорируется.*

Обычно комментарии описывают работу функций, назначение классов, содержат сведения о лицензии, на правах которой предоставляется этот исходный код, об авторе и т. п. Кроме того, часто комментарии используют для временного исключения какой-либо части программы вместо удаления. Опытные программисты предпочитают при ненадобности «закомментировать» часть программы, чтобы обеспечить возможность восстановления этой части при необходимости. Комментарием считается все, что находится:

- за двумя наклонными чертами, набранными без пробела (//). Это однострочный комментарий;
- между символами `/*` и `*/`. Текст комментария между открывающим и закрывающим символами может занимать несколько строк;
- между символами `/**` и `*/`. Такая форма комментария используется для формирования документации.

Благодаря хорошо написанным комментариям программа становится самодокументированной, а благодаря комментариям третьего типа и программе `javadoc` из комплекта JDK можно создать к программе документацию в фор-

мате HTML. Если добавить комментарии в простую программу, рассматриваемую выше, получим следующий исходный код.



Пример

```
    /** HelloWorld - тестовый класс для демонстрации простого приложения
    * данный класс хранит в себе минимум функционала
    */
    class HelloWorld {
    /** main - функция, которая должна быть в каждой программе на Java */
        public static void main(String[] args) {
            // Вывести на экран строку текста
            System.out.println("Hello, World!");
        }
    }
```

В данном примере добавлено три комментария. Один (однострочный) – информирует о том, что на экран будет выведена строка. Этот комментарий не попадет в документацию. Два других – попадут, поскольку начинаются с последовательности символов `/**`.

Хороший комментарий существенно улучшает читабельность программного кода и позволяет избежать многих неприятностей.

3.2 Аннотации

Аннотации *(annotations) – это пометки, с помощью которых программист указывает компилятору Java и средствам разработки, что делать с участками кода помимо исполнения программы.*

Аннотации используются для отслеживания ошибок, анализа кода, компиляции или выполнения. Аннотируемы пакеты, классы, методы, переменные и параметры.

Развивая эту идею, разработчики добавили в компилятор возможность обрабатывать эти указания начиная с пятой версии Java SE. Аннотации объявляются интерфейсами специального вида, помеченными символом at-sign («собачкой»).

Аннотации позволяют:

- автоматически создавать конфигурационные XML-файлы и дополнительный Java-код на основе исходного аннотированного кода;
- документировать приложения и базы данных параллельно с их разработкой;
- проектировать классы без применения маркерных интерфейсов;
- быстрее подключать зависимости к программным компонентам;
- выявлять ошибки, незаметные компилятору;
- решать другие задачи по усмотрению программиста.

Примеры аннотаций, которые будут использованы в данном пособии:

`@Override` – проверяет, переопределен ли метод. Когда метод суперкласса будет удален или изменен, компилятор выдаст сообщение об ошибке. Очень важно всегда использовать аннотацию `@Override` при переопределении метода;

`@Deprecated` – отмечает, что метод устарел. Вызывает предупреждение компиляции, если метод используется.

Аннотации записываются не внутри комментариев, а непосредственно в том месте, где возникает необходимость в их использовании. Например, при указании аннотации `@Deprecated` перед каким-либо методом, мы заставим компилятор при использовании этого метода выводить сообщение о том, что данный метод устарел и вместо него следует использовать другой.

Некоторые аннотации объявлены прямо в компиляторе и их количество растет от версии к версии. Можно также добавлять свои аннотации.

3.3 Имена

Имена (*names*) переменных, классов, методов и других объектов могут быть *простыми* (общее название – *идентификаторы* (*identifiers*)) и *составными* (*qualified names*).

Идентификаторы *– это имена переменных, подпрограмм-функций и других элементов языка программирования.*

Идентификаторы в Java составляются из так называемых *букв Java* (*Java letters*) и арабских цифр 0–9, причем первым символом идентификатора не может быть цифра. В набор букв Java обязательно входят прописные и строчные латинские буквы, знак доллара ($) и знак подчеркивания (_), а также символы национальных алфавитов.

Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Служебные слова Java, такие как `class`, `void`, `static`, зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

Составное имя *(qualified name) – это несколько идентификаторов, разделенных точками, без пробелов.*

Пример составного имени: `System.out.print()`.

3.4 Переменные

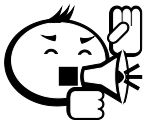
Обычно программы пишут для того, чтобы обрабатывать данные. Вспомним, что такое переменная.

Переменная *(variables) – именованная область памяти ЭВМ, в которой программа может хранить данные определенного типа (называемые значением переменной) и обращаться к этим данным, используя имя переменной.*

Каждая переменная в Java имеет конкретный тип, который определяет размер и размещение ее в памяти, диапазон значений, которые могут храниться в памяти, и набор операций, которые могут быть применены к переменной. Прежде чем использовать переменные, необходимо их объявить, т. е. указать тип данных и имя.

Тип данных *(data types) – это характеристика переменной или константы, определяющая, какого рода значение хранится в отведенной для нее области памяти: числовое, символьное, логическое, объект какого-либо класса.*

В Java все данные можно разделить на *примитивные типы* (*primitive types*) и *ссылочные типы* (*reference types*).



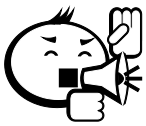
Язык Java относится к строго типизованным языкам. Это означает, что любая переменная в программе относится к определенному типу данных – одному и только одному.

Ссылочные данные реализуются через иерархию классов. Ссылочные типы включают *массивы* (*arrays*), *классы* (*classes*) и *интерфейсы* (*interfaces*). Начиная с Java SE 5 появился *перечислимый тип* (*enum*).

Простые данные – это скорее дань традиции. Забегая наперед, отметим, что для простых типов данных существуют ссылочные аналоги, описанные в классах-оболочках.

Разница между простыми и ссылочными типами на практике проявляется при передаче аргументов методам. Простые типы данных передаются по значению, ссылочные – через ссылку. Эту разницы мы разберем позже на примере сравнения двух символов и строк.

3.5 Литерал



Любая переменная должна быть объявлена до ее первого использования.

Переменной рано или поздно придется присвоить значение, т. е. переменную необходимо инициализировать. В ходе работы программы значение переменной может изменяться. Делается это с помощью литералов.

Литерал *(literal) – это постоянное значение, предназначенное для восприятия человеком, которое не может быть изменено в программе.*

Тип литерала определяется его значением. Чаще всего литералы встречаются в выражениях, аргументах методов. Тип литерала компилятор распознает по его значению. По умолчанию целочисленные литералы имеют тип `int`. Логические литералы могут быть двух значений: `true` или `false`. Логические литералы могут быть присвоены переменным типа `boolean`. Символьные литералы – это символы, которые поддерживают набор символов «Юникод» (Unicode). Строковые литералы берутся в двойные кавычки.

3.6 Константы

Константы *(constants) представляют собой данные, которые остаются неизменными на всем протяжении работы программы.*

Формат записи констант в Java практически полностью соответствует записям констант в языке программирования Си.

Литерал отличается от константы и переменной тем, что ему не может быть присвоено значение. Переменным и константам в качестве значений обычно присваиваются литералы.

При объявлении констант в Java используют модификатор `final`, который показывает, что литерал не должен меняться. Именовать константы принято заглавными буквами со знаком подчеркивания вместо пробела.

```
final String MY_CONST="Значение константы";
```

3.7 Примитивные типы

В Java существует группа типов, называемых ***примитивными*** *(primitive types). Это типы, представляющие простые (скалярные) значения – числа или символы.*

Переменные таких типов являются «автоматическими», для их создания не нужно слово `new`. Такие переменные хранятся в стеке. Размер каждого примитивного типа в Java задается явно и не изменяется с изменением машинной

архитектуры. Эта неизменность размера является одной из основных причин переносимости программ на Java.

Примитивные типы имеют классы-оболочки. Классы-оболочки используются в тех случаях, когда переменную соответствующего типа необходимо рассматривать как объект. Для проведения арифметических операций повышенной точности в Java имеются два класса: `BigInteger` и `BigDecimal`. Класс `BigInteger` поддерживает целые числа произвольной точности. Аналогично, `BigDecimal` представляет числа с фиксированной запятой произвольной точности. Эти классы мы позже рассмотрим более подробно.

В Java существует 8 примитивных типов данных:

- логический (иногда говорят *булев*) тип, называемый `boolean`;
- 2 вещественных типа (*floating-point*): `float` и `double`;
- символьный тип `char`;
- 4 целых типа (*integral*): `byte`, `short`, `int`, `long`.

Эти восемь типов служат основой для всех остальных типов данных.

Поскольку по имени переменной невозможно определить ее тип, все переменные обязательно должны быть описаны перед использованием. Описание заключается в том, что записывается имя типа, затем через пробел список имен переменных, относящихся к этому типу. Имена в списке разделяются запятой. Для всех или некоторых переменных можно указать начальные значения после знака равенства, которыми могут служить любые константные выражения того же типа. Описание каждого типа завершается точкой с запятой.

Примитивные типы обладают явным диапазоном допустимых значений:

- `byte` – диапазон допустимых значений от –128 до 127;
- `short` – диапазон допустимых значений от –32768 до 32767;
- `int` – диапазон допустимых значений от –2147483648 до 2147483647;
- `long` – диапазон допустимых значений от –9223372036854775808 до 9223372036854775807;
- `float` – диапазон допустимых значений от $\sim 1{,}4 \cdot 10^{-45}$ до $\sim 3{,}4 \cdot 10^{38}$;
- `double` – диапазон допустимых значений от $\sim 4{,}9 \cdot 10^{-324}$ до $\sim 1{,}8 \cdot 10^{308}$;
- `boolean` предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений `true` или `false`;

- `char` – символьный тип данных, представляет собой один 16-битный символ Unicode. Он имеет минимальное значение ‘\ u0000’ (или 0) и максимальное значение ‘\ uffff’ (или 65535 включительно).

Символы `char` можно задавать также при помощи соответствующих чисел. Так, символ ‘Ы’ соответствует числу 1067. Рассмотрим на примере:

```
char symb1=1067; char symb2='Ы';
System.out.println("symb1 contains "+ symb1);
System.out.println("symb2 contains "+ symb2);
```

Вывод этой программы:

```
symb1 contains Ы
symb2 contains Ы
```

Небольшой пример того, как узнать, какому числу соответствует символ.

```
char ch = 'J';
int intCh = (int) ch;
System.out.println("J = "+ intCh);
```

Результат:

```
J = 74
```

3.8 Преобразование типов в Java

Очень часто при программировании бывает необходимо присвоить переменной одного типа значение другого типа. Это легко сделать, если диапазон допустимых значений типа переменной, которой присваивается значение, больше диапазона типа присваиваемого значения.

В Java принято явное и неявное приведение типов.

Явное приведение *– присвоение переменной значения несовместимого типа (или типа с более широким диапазоном).*

Неявное приведение *(автоматическое) – присвоение переменной значения, которое уже приведено к нужному типу данных.*

Неявное приведение типов без потерь в точности возможно для совместимых типов данных, если диапазон значений целевого типа шире, чем диапазон исходного (рис. 3.4).

При приведении числовых типов данных действует правило *повышения* (*promotion*), в соответствии с которым диапазон значений целевого типа должен

покрывать диапазон типа присваиваемого значения. Следующие преобразования называются расширяющими преобразованиями примитивов:

- `byte` → `short`, `int`, `long`, `float`, `double`;
- `short` → `int`, `long`, `float`, `double`;
- `char` → `int`, `long`, `float`, `double`;
- `int` → `long`, `float`, `double`;
- `long` → `float`, `double`;
- `float` → `double`.

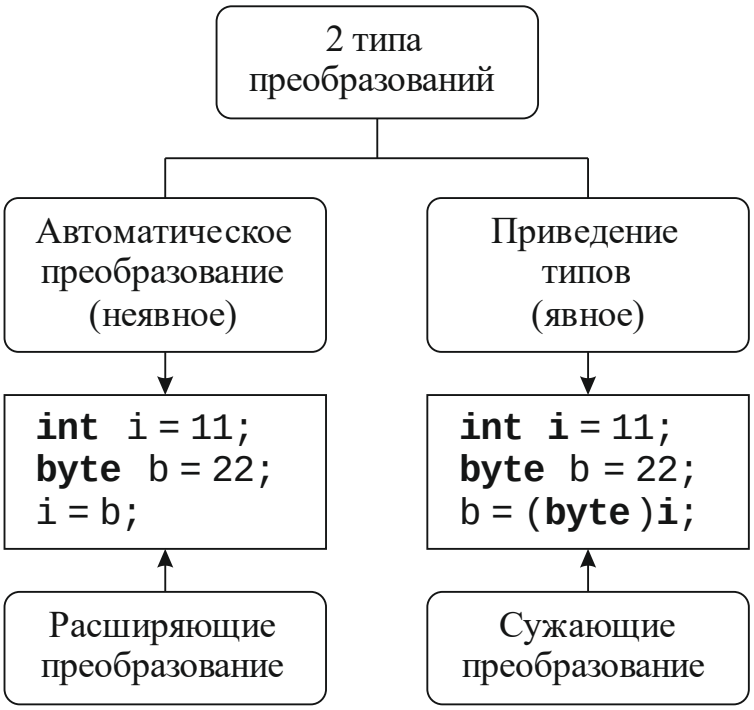


Рис. 3.4 – Неявное и явное приведение типов

Иногда применяется и *сужение* (*narrowing*) типа `int` до типа `short`. Сужение осуществляется просто отбрасыванием старших битов, что необходимо учитывать для больших значений. Следующие преобразования называются сужающими преобразованиями примитивов:

- `short` → `byte`, `char`;
- `char` → `byte`, `short`;
- `int` → `byte`, `short`, `char`;
- `long` → `byte`, `short`, `char`, `int`;
- `float` → `byte`, `short`, `char`, `int`, `long`;
- `double` → `byte`, `short`, `char`, `int`, `long`, `float`.

Сужающее преобразование примитивов может привести к потере точности и даже к получению совсем другого числа из-за выхода за границу размерности.

Процесс автоматического преобразования типов подчиняется нескольким базовым правилам:

- типы переменных, входящих в выражение, должны быть совместимыми. Например, целое число можно преобразовать в формат действительного числа, чего не скажешь о текстовой строке;
- целевой тип (тип, к которому выполняется приведение) должен быть «шире» исходного типа. Другими словами, преобразование должно выполняться без потери данных;
- перед выполнением арифметической операции типы `byte`, `short` и `char` расширяются до типа `int`.

3.9 Операторы

Действия или поведения объектов в методах можно записать операторами.

Оператор *(operator) – это средство языка программирования, с помощью которого записывается алгоритм программы.*

Обычно это символ, сообщающий компилятору, какую математическую или логическую операцию должен выполнить компьютер.

Все операторы Java можно разделить на следующие группы: арифметические, логические, побитовые, сравнения, присваивания, тернарный.

Арифметические операторы

К арифметическим операторам относятся:

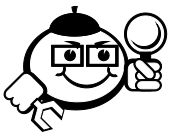
- сложение – + (плюс);
- вычитание – - (дефис);
- умножение – * (звездочка);

- деление – `/` (наклонная черта, слеш);
- взятие остатка от деления (деление по модулю) – `%` (процент);
- инкремент (увеличение на единицу) – `++`;
- декремент (уменьшение на единицу) – `--`.

Между сдвоенными плюсами и минусами нельзя вставлять пробелы.

Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает только целое (так называемое *целочисленное деление*), например 5/2 даст в результате 2, а не 2.5, а 5/(–3) даст –1. Дробная часть попросту отбрасывается, происходит так называемое усечение частного.

В Java принято целочисленное деление.



Пример

```
int a1=5;
int a2=2;
double y1=a1/a2;
double y2=(double)a1/a2;
double y3=(double)(a1/a2);
    System.out.println(5/2);
    System.out.println(5/2.0);
    System.out.println(5.0/2);
    System.out.println(5.0/2.0);
    System.out.println(a1/a2);
    System.out.println(y1);
    System.out.println(y2);
    System.out.println(y3);
```

Результат выполнения программы:

```
2
2.5
2.5
2.5
```

```
2
2.0
2.5
2.0
```

Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать 5/2.0, или 5.0/2, или 5.0/2.0, и получим 2.5 как результат деления вещественных чисел.

*Операторы **инкремент** и **декремент** означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать 5++ или (a + b)++.*

Операторы инкремента и декремента часто используются в операторе цикла `for`. Они применимы к переменной целочисленного типа. Оба эти оператора могут стоять как до аргумента (префиксная форма), так и после (постфиксная форма).

Разница проявится только в выражениях: при первой форме записи (*постфиксной*) в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи (*префиксной*) сначала изменится переменная, и ее новое значение будет участвовать в выражении.

Например,

```
int x=10;
int y=++x; // новое значение y==11; новое значение x==11
```

или

```
int x=10;
int y=x++; // новое значение y==10; новое значение x==11
```

Оператор присваивания

***Оператор присваивания** (simple assignment operator) записывается знаком равенства (=), слева от которого стоит переменная, а справа – выражение, совместимое с типом переменной.*

Оператор присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом будет приведенное значение правой части. После присваивания `x = y` изменится переменная `x`, став равной `y`, а после `y = x` изменится переменная `y`.

Кроме оператора присваивания есть еще 11 *составных* операторов присваивания (*compound assignment operators*): `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`.

Символы записываются без пробелов, нельзя переставлять их местами.

Логические операторы и операторы сравнения

Логические операторы применяются к операндам типа `boolean`.

В Java логический и числовые типы нельзя преобразовывать друг к другу.

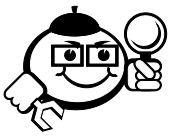
Логические операторы:

- отрицание (NOT) `!`;
- конъюнкция (AND) `&`;
- дизъюнкция (OR) `|`;
- исключающее ИЛИ (XOR) `^`;
- сокращенная конъюнкция `&&`;
- сокращенная дизъюнкция `||`.

У логических операторов следующий приоритет: отрицание, конъюнкция, дизъюнкция.

Сокращенные операторы `&&` и `||`, называемые также *условными*, применяются в условиях `if`, `while` или `do` для вычисления значения истинности условия.

Так же как и в случае с арифметическими операторами, для коррекции приоритета используются круглые скобки. Если одна пара скобок вложена в другую пару скобок, то сначала вычисляется значение во внутренних скобках.



Пример

```
boolean a = true;
boolean b;
b = a || true; // b истинно
b = !b; // b ложно
System.out.println(b); // выведет false
a = a || b; // a истинно
boolean c;
c = a && (a||b); // c истинно
System.out.println(c); // выведет true
```

В языке Java шесть обычных операторов сравнения целых чисел по величине:

- больше – >;
- меньше – <;
- больше или равно – >=;
- меньше или равно – <=;
- равно – ==;
- не равно – !=.

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись => будет неверной.

Результат сравнения – логическое значение: `true`, например в результате сравнения `3 != 5`; или `false`, например в результате сравнения `3 == 5`.

Побитовые операторы

Побитовые операции могут выполняться над операндами целочисленных типов: `byte`, `char`, `short`, `int` и `long`.

В языке Java есть три операции сдвига двоичных разрядов:

- дополнение ~;
- побитовая конъюнкция &;
- побитовая дизъюнкция |;

- побитовой исключающее ИЛИ ^;
- сдвиг влево – <<;
- сдвиг вправо – >>;
- беззнаковый сдвиг вправо – >>>.

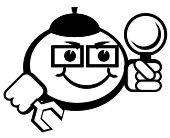
Первые четыре побитовых оператора обозначаются такими же символами, как и логические операторы, но они обрабатывают свои операнды бит за битом.

Тернарный оператор

Тернарным оператор называется потому, что он имеет три операнда:

```
Выражение1 ? выражение2 : выражение3;
```

Тернарный оператор служит для организации ветвления программы, заменяя громоздкий оператор `if`. При этом `выражение1` является условием ветвления, оно должно возвращать значение типа `boolean`. Если `выражение1` возвращает значение `true`, то вычисляется значение `выражения2` и оператор возвращает его значение, иначе вычисляется и возвращается значение `выражения3`. Отсюда следует, что `выражение2` и `выражение3` должны возвращать одинаковый тип.



Пример

```
int x = 7;
int y = 10;
int z = x < y ? y : x;
System.out.println(z);
```

Результат будет:

```
10
```

Приоритет операторов

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

Приоритеты операторов в Java:

1. Круглые скобки `( )`, квадратные скобки `[ ]` и оператор «точка».
2. Инкремент `++`, декремент `--`, отрицания `~` и `!`.
3. Умножение `*`, деление `/` и вычисление остатка `%`.

4. Сложение + и вычитание -.
5. Побитовые сдвиги >>, << и >>>.
6. Больше >, больше или равно >=, меньше или равно <= и меньше <.
7. Равно == и неравно !=.
8. Побитовое И &.
9. Побитовое исключающее ИЛИ ^.
10. Побитовое ИЛИ |.
11. Логическое И &&.
12. Логические ИЛИ ||.
13. Тернарный оператор ?:.
14. Присваивание = и сокращенные формы операторов вида op=.

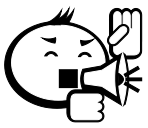
В случаях, когда возникают сомнения в приоритете операторов и последовательности вычисления выражений, рекомендуется использовать круглые скобки.

3.10 Управляющие конструкции

Порядок выполнения программы определяется операторами. Операторы могут содержать в себе другие операторы или выражения.

Набор операторов языка Java включает:

- условный оператор `if`;
- три оператора цикла `while`, `do-while`, `for`;
- оператор варианта `switch`;
- операторы перехода `break`, `continue` и `return`;
- блок, выделяемый фигурными скобками;
- пустой оператор – просто точка с запятой.



В языке Java нет оператора `goto`.

Всякий оператор завершается точкой с запятой.

Область видимости локальных переменных и классов ограничена блоком, в котором они определены.

Блок *– это последовательность операторов, объявлений локальных классов или локальных переменных, заключенных в скобки.*

Например: `{x = 5; y = 7;}`. Можно записать и пустой блок, просто пару фигурных скобок `{}`.

Блоки операторов часто применяются для ограничения области действия переменных, а иногда просто для улучшения читаемости текста программы.

Операторы в блоке выполняются слева направо, сверху вниз. Если все операторы (выражения) в блоке выполняются нормально, то весь блок выполняется нормально. Если какой-либо оператор (выражение) завершается ненормально, то весь блок завершается ненормально.

Нельзя объявлять несколько локальных переменных в пределах видимости блока.

Приведенный ниже код вызовет ошибку времени компиляции.

```
for(int i = 0; i < 5; i++) {
    int sum=sum+i;
}
System.out.print(sum);
```

Не вызовет ошибок следующий код:

```
int sum=0;
for(int i = 0; i < 5; i++) {
    sum=sum+i;
}
System.out.print(sum);
```

В то же время не следует забывать, что локальные переменные перекрывают видимость переменных-членов. Например, этот пример отработает нормально.

```
static int x = 5;
public static void main(String[] args) {
    int x = 1;
    System.out.println("X = " + x);
}
```

И на консоль будет выведено `X = 1`.

Следует напомнить, что перекрытие локальными переменными области видимости глобальных переменных является частой, но трудно обнаруживаемой ошибкой.

3.11 Нормальное и прерванное выполнение операторов

Последовательность выполнения операторов может быть непрерывной, а может и прерываться (при возникновении определенных условий).

Выполнение оператора может быть прервано. Если в потоке вычислений будут обнаружены операторы

- `break`;
- `continue`;
- `return`,

то управление будет передано в другое место.

Нормальное выполнение оператора может быть прервано и при возникновении исключительных ситуаций, которые будут рассмотрены позже.

Оператор `continue` используется только в операторах цикла. Он имеет две формы.

Первая форма состоит только из слова `continue` и осуществляет немедленный переход к следующей итерации цикла.

Рассмотрим пример, когда оператор `continue` позволяет обойти деление на ноль:

```
int j = 5;
for (int i = 0; i < 10; i++) {
    if (i == j) continue;
    System.out.println(i);
}
```

Вторая форма содержит метку. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Метка не требует описания и не может начинаться с цифры.

```
M: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (j > i) {
            continue M;
        }
        System.out.println("i="+i+"; "+"j="+j+"\n");
    }
}
```

Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из циклов, а именно помеченного меткой цикла.

Оператор `break` используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    System.out.println("i="+i+"\n");
}
```

Помеченный оператор `break` применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за этих операторов.

```
M: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (j > i) {
            continue M;
        }
        if (j > 5) {
            break M;
        }
        System.out.println("i="+i+"; "+"j="+j+"\n");
    }
}
```

Оператор `return` предназначен для возврата управления из вызываемого метода в вызывающий. Если в последовательности операторов выполняется `return`, то управление немедленно (если это не оговорено особо) передает управление в вызывающий метод.

Если при объявлении метода использован тип `void`, то `return` не может иметь аргументов, в противном случае будет получена ошибка времени компиляции.

3.12 Условный оператор

Условный оператор предназначен для организации разветвлений в программе. На языке Java он записывается так:

```
if (логическое выражение) оператор1 else оператор2;
```

Если результат вычисления `true`, то действует `оператор1` и на этом работа условного оператора завершается, `оператор2` не действует. Далее будет выполняться оператор, следующий за оператором `if`. Если результат логического выражения `false`, то действует `оператор2`, при этом `оператор1` вообще не выполняется.

Условный оператор может быть сокращенным, без ветви `else`.

```
int a, b;
    a = 5;
    b = 5;
if (a == b) {
    // Если a равно b - выводим сообщение
    System.out.println("a и b равны!");
}
```

Соглашения Code Conventions рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
int a, b;
    a = 1;
    b = 5;
if (a == b) {
    // Если a равно b - выводим сообщение
    System.out.println("a и b равны!");
}
else {
    System.out.println("a не равно b!");
}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма.

Операторы `if-else` могут каскадироваться.

```
String test = "Василий";
if (test.equals("Мурзик")) {
    System.out.println("Это котик Мурзик");
} else if (test.equals("Барсик")) {
    System.out.println("Это котик Барсик, а не Василий");
} else if (test.equals("Муся")) {
```

```
        System.out.println("Это вообще кошка Муся!");
    } else {
        System.out.println("Василия не нашли");
    }
```

3.13 Операторы цикла

Основной оператор цикла – оператор `while` – выглядит так:

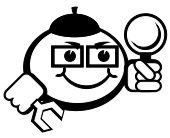
```
while (логическое выражение) оператор;
```

Вначале вычисляется логическое выражение `логВыр`. Если его значение `true`, то выполняется `оператор`, образующий цикл. Затем снова вычисляется `логВыр` и действует `оператор`, и так до тех пор, пока не получится значение `false`. Если `логВыр` изначально равняется `false`, то `оператор` не будет выполнен ни разу. Предварительная проверка условия выполнения цикла обеспечивает безопасность работы цикла, позволяет избежать переполнения, деления на ноль и зацикливания. Оператор `while` еще называют *оператором с предусловием*.

Можно организовать и бесконечный цикл:

```
while (true) оператор;
```

Конечно, из такого цикла следует предусмотреть какой-то выход, например оператором `break`.

.....  Пример .....

```
int i = 1;
while (true) {
    System.out.print(i + " ");
    i++;
    if (i==10) break;
}
```

Результат выполнения:

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

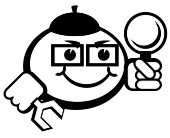
.....

Второй оператор цикла – оператор `do-while`. Он имеет вид:

```
do оператор while (логВыр);
```

Здесь сначала выполняется оператор, а потом происходит вычисление логического выражения логВыр. Цикл выполняется, пока логВыр остается равным true.

Существенное различие между этими двумя операторами цикла только в том, что в цикле do-while оператор обязательно выполнится хотя бы один раз.



Пример

```
int i = 1;
do {
    i++;
    System.out.print(i + " ");
} while (i < 5);
```

Результат выполнения:

```
2 3 4 5
```

```
int i = 1;
do {
    System.out.print(i + " ");
    i++;
} while (i > 5);
```

Результат выполнения:

```
1
```

Оператор do-while еще называют *оператором с постусловием*.

Третий оператор цикла – оператор for – выглядит так:

```
for (списокВыр1; логВыр; списокВыр2) оператор;
```

Перед выполнением цикла вычисляется список выражений списокВыр1. Это ноль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла. Затем вычисляется логическое выражение логВыр. Если оно истинно, true, то действует оператор, потом вычисляются слева направо выражения из списка выражений списокВыр2.

Следующий код программы выводит на экран числа от 1 до 100:

```
for (int i = 1; i <= 100; i++) {
```

```
    System.out.print(i + " "); }
```

Следующий код программы выводит на экран числа от 10 до −10:

```
for (int s = 10; s > -11; s--) {
    System.out.print(s + " "); }
```

Следующий код программы выводит на экран нечетные числа от 1 до 33:

```
for (int i = 1; i <= 33; i = i + 2) {
    System.out.print(i + " "); }
```

3.14 Оператор `switch`

Оператор `switch()` используется в случае необходимости множественного выбора. Выбор осуществляется на основе целочисленного значения.

Оператор варианта `switch` организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме `long`) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так:

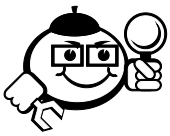
```
switch (выражение) {
    case констВыр1: оператор1;
    case констВыр2: оператор2;
    case констВырN: операторN;
    default: операторDef;
}
```

Причем фраза `default` не является обязательной.

Стоящее в скобках `выражение` может быть простого целого типа `byte`, `short`, `int`, `char`, но не `long`. Кроме простых целых типов допускаются их классы-оболочки, перечисления и строки символов типа `String`.

После выполнения одного варианта оператор `switch` продолжает выполнять все оставшиеся варианты. Чаще всего необходимо пройти только одну ветвь операторов. В таком случае используется оператор `break`, сразу же прекращающий выполнение оператора `switch`.

С другой стороны, может понадобиться выполнить один и тот же оператор в разных ветвях `case`. В этом случае ставим несколько меток `case` подряд.



Пример

```
switch (day){
    case 1: case 2: case 3: case 4: case 5:
    System.out.println("Рабочий день"); break;
    case 6: case 7:
    System.out.println("Выходной день"); break;
    default:
    System.out.println("Неправильно задан день недели");
}
```

Контрольные вопросы по главе 3

1. Из каких символов может состоять имя переменной (корректный идентификатор)?
2. Сколько ключевых слов зарезервировано языком Java? Что это за слова, какие из них не используются?
3. Что такое типизация? Что такое тип данных?
4. Что такое объявление переменной?
5. Что такое «инициализация» в Java?
6. Что такое литералы и какие они бывают?
7. На какие основные группы можно поделить типы данных?
8. Какие примитивные типы вы знаете?
9. Что вы знаете о явном и неявном преобразовании типов?
10. Расскажите о сужении и повышении типов. Приведите примеры.

4 Основы объектно-ориентированного программирования

Объекты *– это отдельные, четко обозначенные экземпляры некоторого класса.*

4.1 Класс и его структура

Класс дает общее описание объектов, указывает, «на что они похожи». Класс изображается в виде прямоугольника, состоящего из трех частей (рис. 4.1).

```
Class_modifiers class Class_name {
// attributes
// methods          } Class_body
// constructors
}
```

Рис. 4.1 – Структура класса

В верхней части помещается название класса, в средней – свойства объектов класса, в нижней – действия, которые можно выполнять с объектами данного класса (методы).

В Java имя класса также является именем нового ссылочного типа.

Как и другие идентификаторы Java, имена классов имеют следующие ограничения:

- должны начинаться с буквы, символа подчеркивания или знака $;
- должны содержать только Unicode-символы;
- не должны совпадать с ключевыми словами Java.

Соглашения Code Conventions рекомендуют начинать имя класса с заглавной буквы.

Перед словом `class` можно записать модификаторы класса (*class modifiers*).

Модификатор *(modifier) – это ключевое слово языка, которое может каким-либо образом изменить смысл некоторого определения (например, класса или метода).*

Модификаторы доступа: `public`, `protected`, `private`. Без модификатора – `package`.

У класса могут быть следующие модификаторы:

- `abstract` – модификатор реализации класса – указывает, что класс не может иметь экземпляров. Класс, имеющий хотя бы один абстрактный метод (т. е. метод без реализации), должен быть объявлен как `abstract`;
- `final` – модификатор ограничения иерархии классов – указывает, что класс не может иметь производных классов;
- `static` – модификатор для обозначения вложенных классов (nested static class), которые не имеют доступа к нестатическим полям и методам внешнего класса, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр внешнего класса;
- `strictfp` – модификатор, ограничивающий точность вычислений с `float` и `double`. В классе он обозначает, что все методы на всех системах будут возвращать вещественные числа с одинаковой точностью.

Давайте создадим класс `Cat`.

Определение класса производится через указание полей (данных) и методов класса:

```
public class Cat {
}
```

Тело класса, в котором в любом порядке перечисляются поля, методы, вложенные классы и интерфейсы, заключается в фигурные скобки.

Атрибутами класса `Cat` могут быть `name`, `weight`, `color`.

Поле (атрибут) класса *– это характеристика объекта, описывающая его свойство.*

Атрибуты – это переменные класса, которые объявляются следующим образом:

```
public class Cat {
    private int weight; // вес кота
    private String name; // имя кота
    private String color; // окрас кота
}
```

Описание поля может начинаться с одного или нескольких необязательных модификаторов: `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`.

Если поле класса объявлено как `static`, то будет существовать ровно одно значение этого поля, не зависимо от того, сколько экземпляров класса будет создано, даже если не будет создано ни одного экземпляра. Такие статические поля еще называют переменными уровня класса (*class variable*).

Поле с модификатором `final` не может поменять свое значение после инициализации.

Для указания того, что во время сериализации объекта некоторое поле нужно игнорировать, используется модификатор `transient`.

`volatile` означает, что потоки могут хранить значения некоторых переменных в некотором своем локальном контексте. Если один поток изменит значение такого поля, то другой поток может об этом не узнать (так как хранит копию). Для полей с модификатором `volatile` есть гарантия, что все потоки всегда будут видеть актуальное значение такой переменной.

При этом поле не может быть помечено как `abstract`, `synchronized`, `strictfp`, `native`.

Если надо поставить несколько модификаторов, то перечислять их JLS рекомендует в указанном порядке, поскольку некоторые компиляторы требуют определенного порядка записи модификаторов.

Помимо полей определим методы для класса `Cat`.

Метод *(method) – это последовательность команд, которые вызываются по определенному имени.*

По большому счету метод – это то, что может делать объект соответствующего класса (или что можно делать с объектом).

В Java нет вложенных процедур и функций, в теле метода нельзя описать другой метод, есть вложенные классы.

При описании метода указывается тип возвращаемого им значения или слово `void`, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках расписывается выполняемый метод.

Описание метода может начинаться с модификаторов `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`.

Метод с модификатором `abstract` может быть объявлен в пределах абстрактного класса (или интерфейса). В этом случае тело метода отсутствует, а реализация может быть предоставлена в классах-наследниках.

Метод, объявленный с модификатором `final`, не может быть переопределен в наследниках.

Метод с модификатором `static` относится к классу в целом, а не к его экземплярам, то есть в него не передается текущий объект. Такой метод может быть вызван без объекта.

Модификатор `native` перед объявлением метода указывает, что он является специфическим для операционной системы. Как и у абстрактного метода, у него тоже нет тела, а реализация находится в скомпилированном виде в файлах JVM.

Модификатор `synchronized` у метода говорит о том, что перед его выполнением должен быть захвачен монитор объекта (для нестатического метода) либо монитор, связанный с классом (для статического метода).

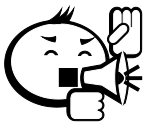
Важным понятием является *сигнатура метода*.

Сигнатура *(signature) определяется именем метода и его аргументами (количеством, типом, порядком следования).*

Если для полей запрещается совпадение имен, то для *методов* в классе запрещено создание двух *методов* с одинаковыми *сигнатурами*.

В сигнатуру метода не входит возвращаемое значение, бросаемые им исключения, а также модификаторы.

Перед началом работы метода для каждого параметра выделяется ячейка оперативной памяти, в которую копируется значение параметра, заданное при обращении к методу. Такой способ называется передачей параметров *по значению.*



В языке Java применяется только передача аргументов по значению.

Научим нашего кота *есть* и *разговаривать*. Опишем это поведение с помощью методов.

```
// кот ест
public void eat() {
    System.out.print("Ам-ам…");
}
```

И немного иначе опишем метод – *кот говорит:*

```
// кот говорит
public String voice(String words) {
    String phrase = "Котик говорит" + words;
    return phrase;
}
```

В данном методе необходимо передать аргумент (параметр) типа `String`, а он свою очередь вернет значение типа `String`.

Чтобы наш котик стал *есть* и *говорить*, нужно создать конкретные *объекты*, *экземпляры класса* (*instances*) описанного класса. Создание экземпляров производится в три этапа. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них. Затем операцией `new` определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

Операция `new` применяется для выделения памяти массивам и объектам. Результатом операции `new` будет ссылка на созданный объект. Эта ссылка может быть присвоена переменной ссылочного типа на данный тип:

```
Cat kitty = new Cat();
```

но может использоваться и непосредственно:

```
new Cat().eat();
```

Здесь после создания безымянного объекта сразу выполняется его метод `eat()`. Это возможно потому, что приоритет операции `new` выше, чем приоритет операции обращения к методу, обозначаемой точкой.

Далее происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции `new` повторяется имя класса со скобками `Cat()`. Это так называемый *конструктор класса*.

4.2 Конструкторы

Каждый класс может также иметь специальные методы, которые автоматически вызываются при создании и уничтожении объектов этого класса:

- конструктор (*constructor*) – выполняется при создании объектов;
- деструктор (*destructor*) – выполняется при уничтожении объектов.

Как и в C++, в классах Java имеются конструкторы. Их назначение полностью совпадает с назначением аналогичных методов C++.

Конструктор *(constructor) – это особенный метод класса, который вызывается автоматически в момент создания объектов этого класса. Имя конструктора совпадает с именем класса.*

Конструкторы добавляются в класс, если в момент создания объекта нужно выполнить какие-то действия (начальную настройку) с его данными (полями).

По синтаксису конструктор похож на метод без возвращаемого значения. Также выделяют заголовок и тело конструктора. Заголовок состоит из модификаторов доступа (никакие другие модификаторы недопустимы). Тело конструк-

тора пустым быть не может и поэтому всегда описывается в фигурных скобках (для простейших реализаций скобки могут быть пустыми).

Класс обязательно должен иметь конструктор, иначе невозможно порождать объекты ни от него, ни от его наследников. Поэтому если в классе не объявлен ни один конструктор, компилятор добавляет один по умолчанию. Это public-конструктор без параметров и с телом, описанным парой пустых фигурных скобок.

Автоматический вызов конструктора по умолчанию позволяет избежать ошибок, связанных с использованием неинициализированных переменных. Этот конструктор не имеет параметров, все что он делает – вызывает конструктор без параметров класса-предка.

Вернемся к нашему классу `Cat()`. Изменим немного метод `eat()`.

```
class Cat {
    // поля
    int age; // возраст
    String name; // кличка
    // кот ест
    public void eat() {
        for (int i = 1; i <= age; i++) {
            System.out.println("ам-ам");
        }
    }
    // кот говорит
    public String voice(String words) {
        String phrase = name+"говорит"+words;
        return phrase;
    }
}
```

Добавим конструктор с двумя параметрами, который при создании нового кота позволяет сразу задать его кличку и возраст.

```
public Cat (String n, int a) {
    name = n;
    age = a;
}
```

Теперь, когда у нас есть такой конструктор, мы можем им воспользоваться:

```
Cat cat = new Cat("Василий", 2);
```

В результате переменная `cat` будет указывать на конкретного кота по кличке Василий, которому 2 года.

Переменную `cat` теперь можно использовать для *вызова методов класса* `Cat`, например:

```
cat.eat();
System.out.print(cat.voice("мяу"));
```

Как и обычному методу, конструктору можно передавать аргументы. Передаются и используются они по той же схеме, что и для прочих методов класса. Однако теперь при создании объекта необходимо передать аргументы для конструктора. Аргументы передаются в круглых скобках после имени класса в команде создания объекта.

Если в классе не определен конструктор без аргументов (но определен хотя бы один конструктор), рассчитывать на конструктор по умолчанию (конструктор без аргументов) нельзя – необходимо передавать аргументы в соответствии с описанным вариантом конструктора.

Конструктор не может быть `static`, `abstract` или `final`.

В отличие от других объектно-ориентированных языков в Java процесс удаления ненужных объектов происходит автоматически самой виртуальной машиной и называется он «сборка мусора» (*garbage collection*).

В Java нет деструкторов, но существует возможность объявлять методы с именем `finalize`. Методы `finalize` аналогичны деструкторам в C++ (ключевой знак ~) и Delphi (ключевое слово `destructor`).

Несколькими предложениями этот процесс можно описать так:

У каждого объекта имеется счетчик ссылок, указывающих на него. В момент создания объекта счетчик инициализируется единицей, и впоследствии во время работы программы, если появляются новые ссылки на этот объект, счетчик увеличивается. Когда какая-то ссылка становится «ненужной», счетчик уменьшается. Виртуальная машина время от времени проверяет все объекты

программы, и для тех объектов, чьи счетчики ссылок имеют нулевое значение, очищает память. Естественно, описанный механизм весьма абстрактно описывает процесс очистки.

4.3 Наследование

Для того чтобы один класс был потомком другого, необходимо при его объявлении после имени класса указать ключевое слово `extends` и название суперкласса. Он должен быть доступным классом и не иметь модификатора `final`. Если ключевое слово `extends` не указано, считается, что класс унаследован от универсального класса `Object`.

В Java класс-наследник может иметь только одного родителя. Множественное наследование в Java реализуется только через интерфейсы.

```
class Имя_класса_наследника extends Имя_класса_родителя {
    // реализация
}
```

Например, создадим отдельный класс для породы кошек `MaineCoon`:

```
class MaineCoon extends Cat {
    // дополнительные поля и методы
}
```

4.4 Геттеры и сеттеры

Следующее понятие из ООП, которое следует рассмотреть, – это геттеры и сеттеры (*get* – получать, *set* – устанавливать). Это общепринятый способ *вводить данные* (*set*) или *получать данные* (*get*). Геттеры и сеттеры выполняют важную миссию *защиты* данных.

Сеттеры и геттеры реализуют главный принцип ООП – принцип инкапсуляции – сокрытие данных [6].

Метод *геттер* не имеет параметров (т. е. в скобках ничего не пишется) и возвращает значение одной переменной (одного поля). Метод *сеттер* всегда имеет модификатор `void` и только один параметр, для изменения значения одного поля.

Например, у нас есть класс `Cat`. Зададим, используя *setter*, имя котика. А потом эти данные можно получить с помощью *getter*. Поле в нашем классе `Cat`, которое мы хотим защитить, пометим модификатором `private`. Добавим метод геттер/сеттер для этого поля.

```
class Cat {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String a) {
        name = a;
    }
}
```

Обычно имя геттера состоит из слова `get+название_переменной`. Имя сеттера тоже состоит из слова `set+название_переменной`.

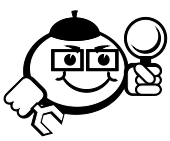
4.5 Перегрузка методов

Часто одно и то же слово имеет несколько разных значений – оно *перегружено*. Можно создавать методы с одинаковыми именами, но с разным набором аргументов. Java позволяет создавать несколько методов с одинаковыми именами, но разными сигнатурами.

Различные реализации методов с одинаковыми именами, но разными сигнатурами в Java называются ***перегрузкой методов.***

Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров.

Перегрузка методов реализует такое важное свойство в программировании, как полиморфизм.



Пример

```
class Cat {
    void eat() {
```

```
        // параметры отсутствуют
    }
    void eat(int count) {
        // используется один параметр типа int
    }
    void eat(int count, int pause) {
        // используются два параметра типа int
    }
    long eat(long time) {
        // используется один параметр типа double
        return time;
    }
}
```

Для объекта `cat` можно вызвать любой метод из класса:

```
Cat cat = new Cat();
cat.eat();
cat.eat(3);
cat.eat(3, 2);
cat.eat(4500.25);
```

Аналогично перегрузка используется и для *конструкторов*.

4.6 Ключевые слова `this` и `super`

`this` *и* ***`super`*** *– это два специальных ключевых слова в Java, которые представляют соответственно текущий экземпляр класса и его суперкласса.*

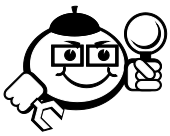
`this` представляет текущий экземпляр класса, в то время как `super` – текущий экземпляр родительского класса.

Внутри класса для вызова своего конструктора без аргументов используется `this()`, тогда как `super()` используется для вызова конструктора без аргументов, или, как его еще называют, конструктора по умолчанию родительского класса.

`this` и `super` в Java используются для обращения к переменным экземпляра класса и его родителя.

При этом следует учесть, что `this` и `super` – это нестатические переменные, соответственно их нельзя использовать в статическом контексте, а это означает, что их нельзя использовать в методе `main()`. То же самое произойдет, если в методе `main()` воспользоваться ключевым словом `super`.

Приведем пример использования ключевых слов `this` и `super`.




```
public class Pet {
    protected String name;
    protected int age;
    public Pet() { this("Unnamed"); }
    public Pet(String name) { this.name=name; }
    public Pet(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void setName(String name){ this.name = name; }
    public String getName() { return name; }
    public String getType() { return "Домашнее животное"; }
}
```

Далее создадим класс наследника:

```
public class Cat extends Pet {
    private String name;
    private int age;
    public Cat() {
        name= "Unnamed";
        age = 0;
    }
    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Cat(String name) {
        super(name);
```

```
    }
}
```

В главном методе `main()` создадим два объекта класса `Cat` и вызовем для них методы текущего и родительского класса `Pet`.

```
public static void main(String[] args) {
    Cat cat1 = new Cat("Василий");
    Cat cat2 = new Cat("Мася",4);

    cat1.setAge(2);
    System.out.println(cat1.getType()+ " " +
cat1.getName() + " "+ cat1.getAge());
    System.out.println(cat2.getType()+ " " +
cat2.name + " " + cat2.age);
}
```

Результат

```
Домашнее животное Василий 2
Домашнее животное Мася 4
```

Как видим, основное назначение `this` и `super` – вызывать один конструктор из другого и ссылаться на переменные экземпляра, объявленные в текущем классе и его родительском классе.

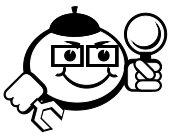
4.7 Переопределение методов

Кроме перегрузки существует также *переопределение методов* (*overriding*).

Изменить работу любого из методов, унаследованных от класса-предка, класс-потомок может, описав новый метод с точно таким же именем и параметрами. Это называется ***переопределением****.*

Замещение происходит, когда класс-потомок (подкласс) определяет некоторый метод, который уже есть в родительском классе (суперклассе), таким образом новый метод заменяет метод суперкласса. У нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса.

Начинается такой метод с аннотации `@Override`. В этом случае компилятор получает возможность проверить, что вы переопределили метод, а не написали новый. Таким образом можно избежать некоторых ошибок из-за невнимательности. Аннотации появились только в версии Java 1.5 и более ранние версии их не поддерживают. Всегда используйте аннотацию `@Override`, когда вы пытаетесь переопределить метод суперкласса.

 Пример

```
public class Cat {
    public String name;
    public int age;

    public void voice(String name) {
        this.name=name;
        System.out.println(name+"говорит Мяу");
    }
}
public class Kitten extends Cat {
    public String name;
    // переопределим родительский метод
    @Override
    public void voice(String name) {
        this.name="My kitty"+name;
        System.out.println(name+"говорит Мяв-мяв");
        // Вызов версии метода родительского класса
        super.voice(name);
    }
    public static void main(String[] args) {
        Kitten cat1=new Kitten();
        cat1.voice("Мася"); // дочерний метод
        Catt cat2=new Cat();
        cat2.voice("Василий"); // родительский метод
    }
}
```

Результат:

```
Мася говорит Мяв-мяв
```

```
Мася говорит Мяу
Василий говорит Мяу
```

- Переопределение метода происходит только тогда, когда имена и сигнатуры типов этих двух методов идентичны. Если это не так, то оба метода просто перегружены.
- Поля нельзя переопределить, их можно только скрыть.
- Если пометить метод модификатором `final`, то метод не может быть переопределен.
- Переопределенные методы позволяют поддерживать полиморфизм времени выполнения.

4.8 Вложенные и внутренние классы

Синтаксис языка Java позволяет объявлять классы внутри другого класса, такие классы называются внутренними или вложенными.

Вложенный класс не может существовать независимо от класса, в который он вложен. Следовательно, область действия *вложенного класса* ограничена его *внешним классом*, однако вложенные классы имеют доступ к методам и переменным внешнего.

Вложенные классы делятся на два вида: статические и нестатические (рис. 4.2).

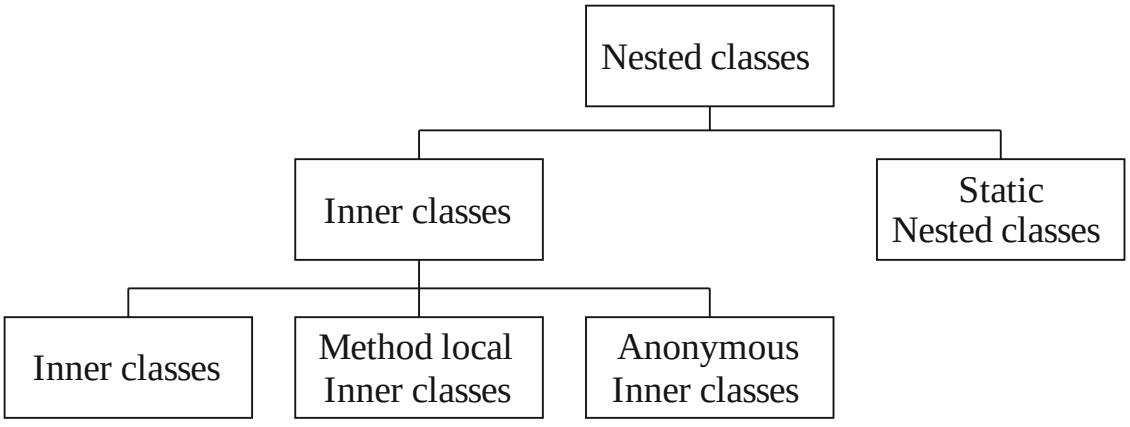
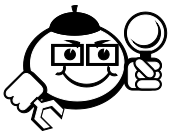


Рис. 4.2 – Вложенные классы

Вложенные классы, объявленные как статические, называются ***вложенными статическими*** *(static nested classes). Вложенные классы, объявленные БЕЗ ключевого слова* `static`*, называются* ***внутренними классами*** *(inner classes).*



Пример

```
class OuterClass {
    // Простой вложенный класс
    static class StaticNestedClass {
        public void show() {
            System.out.println("Метод вложенного
        класса");
        }
    }
    // Простой внутренний класс
    class InnerClass {
        public void show() {
            System.out.println("Метод внутреннего
        класса");
        }
    }
    public static void main(String[] args) {
        OuterClass.InnerClass inner = new Outer-
    Class().new InnerClass();
        OuterClass.StaticNestedClass nested = new
    OuterClass.StaticNestedClass();
        inner.show();
        nested.show();
    }
}
```

Нестатические классы имеют доступ к полям содержащего класса, даже если они объявлены как `private`. Статический внутренний класс (вложенный)

видит только статические переменные внешнего класса. Другие члены внешнего класса доступны ему посредством ссылки на объект.

Как и другие поля класса, вложенные классы могут быть объявлены как `private`, `public`, `protected` или `package private`. Внешний класс (*OuterClass*) может быть только `public` или `package-private`.

Внутренние (нестатические) классы, как переменные и методы, связаны с *объектом внешнего класса*. Внутренние классы также имеют прямой доступ к полям внешнего класса. Такие классы не могут содержать в себе статические методы и поля. Внутренние классы не могут существовать без экземпляра внешнего.

Если необходимо получить ссылку на объект внешнего класса, то следует использовать наименование внешнего класса с ключевым словом `this`, разделенные точкой, например: `OuterClass.this`.

Внутренние классы бывают трех типов:

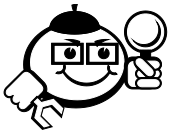
- внутренний класс;
- локальный;
- анонимный.

В Java можно написать класс в методе, и это будет *локальный класс*. Как локальные переменные, область внутреннего класса ограничивается в рамках метода.

Метод локального внутреннего класса может быть реализован только в методе, в котором определяется внутренний класс. Нижеприведенная программа показывает, как использовать метод локального внутреннего класса.

Локальные классы не могут иметь никаких модификаторов доступа: `private`, `protected`, `public`, `static` и `transient`, но могут быть помечены как `abstract` и `final`, но не оба одновременно.

В следующем примере внутренний класс расположен в методе `outMethod()`.

.....  Пример .....

```
public class Outer {
    void outMethod() {
        System.out.println("Метод внешнего класса");
        /* Внутренний класс - локальный для метода
    outMethod() */
```

```
        class Inner {
            public void innerMethod() {
                System.out.println("Метод внутреннего
            класса");
            }
        }
        Inner inner = new Inner();
        inner.innerMethod();
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outMethod();
    }
}
```

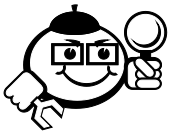
Вывод:

```
Метод внешнего класса
Метод внутреннего класса
```

.....

Анонимные внутренние классы (*anonymous Inner classes*) объявляются без указания имени класса. Они могут быть созданы двумя путями:

1. Как наследник определенного класса.

.....  Пример .....

```
public class Outer {
    // Анонимный класс наследуется от класса Demo
    static Demo demo = new Demo() {
        @Override
        public void show() {
            super.show();
            System.out.println("Метод анонимного клас-
        са");
        }
    };
    public static void main(String[] args) {
        demo.show();
    }
```

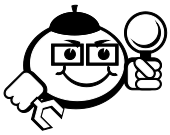
```
}

class Demo {
    public void show() {
        System.out.println("Метод суперкласса");
    }
}
```

Вывод:

```
Метод суперкласса
Метод внутреннего анонимного класса
```

В коде выше класс Demo является суперклассом, от которого наследуется анонимный класс, и оба они имеют метод show(). В анонимном классе метод show() будет переопределен.

2. Как реализация определенного интерфейса. Создаем объект анонимного внутреннего класса, но этот анонимный внутренний класс является реализацией интерфейса Hello.




```
public class Outer {
    /* Анонимный класс, который реализует интерфейс
Hello */
    static Hello h = new Hello() {
        public void show() {
            System.out.println("Метод анонимного
        класса");
        }
    };
    public static void main(String[] args) {
        h.show();
    }
}
interface Hello {
    void show();
}
```

Вывод:

```
Метод внутреннего анонимного класса
```

Любой анонимный внутренний класс за одно действие может либо расширить класс, либо реализовать интерфейс, но не одновременно.

Анонимный класс никогда не может быть `abstract`.

Анонимный класс всегда неявно `final`.

Классический пример анонимного класса:

```
new Thread(new Runnable() {
    public void run() {
        ...
    }
}).start();
```

На основании анонимного класса создается поток и запускается с помощью метода `start()` класса `Thread`. Синтаксис создания анонимного класса базируется на использовании оператора `new` с именем класса (интерфейса) и телом созданного анонимного класса.

4.9 Абстрактные классы

По мере изучения особенностей наследования объектов в языке Java может понадобиться создать класс, характеризующий объект обобщенно, на базе которого впоследствии можно создать подклассы. Подклассы будут уточнять свойства объектов, формируя таким образом иерархию классов.

Самый общий класс в данном случае будет абстрактным. Он формирует «внешнюю оболочку» для подклассов, и только подклассы наполняют эту оболочку конкретным содержанием – кодом, реализующим задачи программы. Абстрактный класс, как правило, содержит один или несколько абстрактных методов.

Абстрактный метод *(abstract method) – это метод, реализация которого неизвестна на данный момент. Известно только то, что этот метод должен быть у всех наследников.*

Перед таким абстрактным методом указывается `abstract`, а заканчивается описание сигнатуры метода в классе традиционно – точкой с запятой:

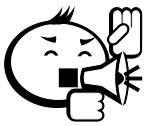
```
abstract void method();
```

Абстрактный метод не может быть `private`, `native`, `static`, `final`.

Класс, который содержит хотя бы один абстрактный метод, называется ***абстрактным****. Иначе:* ***абстрактный класс*** *– это класс, экземпляр которого нельзя создать.*

Класс, содержащий один или несколько абстрактных методов, должен быть также объявлен как абстрактный с использованием того же спецификатора `abstract` в объявлении класса.

Поскольку абстрактный класс не определяет реализацию полностью, у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора `new` приведет к возникновению ошибки во время компиляции. Подкласс, наследующий абстрактный класс, должен реализовать все абстрактные методы суперкласса. В противном случае он также должен быть определен как абстрактный. Таким образом, атрибут `abstract` наследуется до тех пор, пока не будет достигнута полная реализация класса.



Абстрактные классы реализуют на практике один из принципов ООП – *полиморфизм* [6].

Рассмотрим пример иерархии класса `Animal` (рис. 4.3).

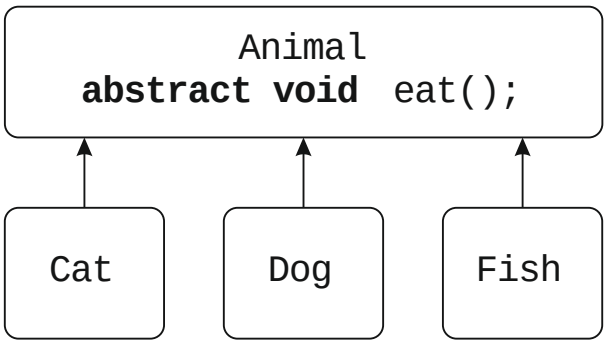
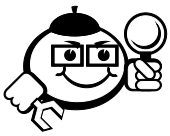


Рис. 4.3 – Иерархия класса `Animal`



Пример

```
public abstract class Animal {
    String name;
    int age;
    void sleep() {
        System.out.println("i am sleeping");
    }
    abstract void eat(); // абстрактный метод
    // переопределим метод toString() класса Object
    @Override
    public String toString() {
        return "name: " + name + " age: " + age;
    }
}
public class Cat extends Animal { // класс-наследник
    /* переопределим абстрактный метод eat() класса
Animal для класса Cat и реализуем его */
    @Override
    void eat() {
        System.out.println("Скушает мышку");
    }
}
public class Main {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.name = "Мурка";
        cat.age = 11;
        System.out.println(cat);
        cat.eat();
    }
}
```

4.10 Интерфейсы

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, в Java можно наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование. Java множественное наследование не поддерживает.

Множественным наследованием *называется ситуация, когда класс наследует от двух или более классов.*

В языке Java подобную проблему позволяют решить *интерфейсы*.

Интерфейс *(interface) – это явно указанная спецификация набора абстрактных методов, которые должны быть представлены в классе, реализующем эту спецификацию.*

Любой *интерфейс* (*interface*) может иметь много реализаций. Любой класс может реализовывать несколько интерфейсов, при этом интерфейсы не входят в иерархию классов. Реализовать один и тот же интерфейс имеют право классы, никак не связанные друг с другом. Интерфейсы в Java являются ссылочными типами, как классы, но они могут содержать в себе только *константы, сигнатуры методов*.

Интерфейс может быть только реализован каким-либо классом либо наследоваться другим интерфейсом.

Интерфейсы реализуют на практике один из принципов ООП – *полиморфизм.*

Когда класс реализует интерфейс и не обеспечивает выполнения или тела метода по отношению к любому из абстрактных методов интерфейса, то класс становится *абстрактным*.

Добавим в наш пример иерархии `Animal` интерфейс `CanMove`, научим нашего котика бегать (`run()`) с определенной скоростью (`getVelocity()`) и передвигаться назад (`back()`).

```
public interface CanMove {
    // Методы в Interface все являются
    // абстрактными и public
```

```
    public abstract void run();
    /* Даже если вы не пишете 'public abstract' то
java все равно понимает как абстрактный */
    void back();
    public int getVelocity();
}
```

Чтобы использовать интерфейс, нужно объявить класс, *реализующий этот интерфейс*, с помощью ключевого слова `implements`.

```
/* Interface CanMove имеет 3 абстрактных метода.
Этот класс выполняет только 1 метод.
Поэтому он должен быть объявлен как abstract.
Остальные абстрактные методы будут выполнены подклас-
сами */
public abstract class Animal implements CanMove {
    String name;
    int age;
    void sleep() { System.out.println("мур-мур"); }
    abstract void eat();
    // Выполнить метод run() в interface CanMove
    @Override
    public void run() {
        System.out.println("бежать…"); }
    @Override
    public String toString() {
        return "name: " + name + " age: " + age;
    }
}
public class Cat extends Animal implements CanMove {
    private String name;
    public Cat(String name) {
        this.name = name; }
    public String getName() {
        return this.name; }
    // Выполняет метод интерфейса (interface) CanMove
    @Override
    public void back() {
```

```
        System.out.println(name + " бежать
    назад..."); }
    // Выполняет метод интерфейса (interface) CanMove
    @Override
    public int getVelocity() {
        return 110; }
    // Выполняет абстрактный метод eat()
    @Override
    public void eat() {
        System.out.println(name + " ам-ам..."); }
}
public class Main {
    public static void main(String[] args) {
        Cat cat = new Cat("Мурка");
        cat.eat();
        /* Создает объект CanMove.
        Объект объявлен как CanMove.
        Но на самом деле является Cat, так как вызы-
    вается конструктор Cat */
        CanMove cancat = new Cat("Tom");
        // Java всегда знает вид объекта ==> Tom ам-ам ...
        cancat.eat();
    }
}
```

У интерфейса могут быть следующие модификаторы:

- `public` (если он есть, то интерфейс доступен отовсюду, если его нет – доступен только в данном пакете);
- `abstract` (так как интерфейс всегда абстрактный, то модификатор обычно опускается);
- `strictfp` – все позже реализуемые методы должны будут работать с числами с плавающей точкой аналогично на всех машинах Java;
- интерфейсом могут расширяться многие классы;
- интерфейс может сам расширяться несколькими интерфейсами;
- интерфейс могут использовать сразу несколько классов, независимых друг от друга;
- один класс может применить множество интерфейсов.

С выпусков Java 8 интерфейсы получили новые очень интересные возможности: статические методы, методы по умолчанию и автоматическое преобразование из лямбд (функциональные интерфейсы). Начиная с JDK 8 в интерфейсах доступны статические методы, они аналогичны методам класса. С методом по умолчанию все иначе: интерфейс может отметить метод ключевым словом `default` и обеспечить реализацию для него [7]. Например:

```
public interface InterfaceWithDefaultMethods {
    void performAction();
    default void DefaulMethod() {
        // Implementation here
    }
}
```

4.11 Коллекции

В пакете `java.util` содержится библиотека *коллекций* (*collection*), которая предоставляет большие возможности для работы с множествами, хеш-таблицами, векторами, различными списками и т. д.

Коллекциями *(collection) называют структуры, предназначенные для хранения однотипных данных.*

Простейшей коллекцией является массив. Но массив имеет ряд недостатков. Один из самых существенных – размер массива, который фиксируется до начала его использования. То есть необходимо заранее знать или подсчитать, сколько потребуется элементов коллекции до начала работы с ней. Зачастую это неудобно, а в некоторых случаях невозможно. Поэтому все современные базовые библиотеки различных языков программирования имеют тот или иной вариант поддержки коллекций объектов. Коллекции обладают одним важным свойством – их размер не ограничен. Выделение необходимых для коллекции ресурсов спрятано внутри соответствующего класса.

В библиотеке коллекций Java существуют два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций (рис. 4.4).

1. `Collection` – коллекция содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (*add*, *addAll*), удаление (*remove*, *removeAll*, *clear*), поиск (*contains*).

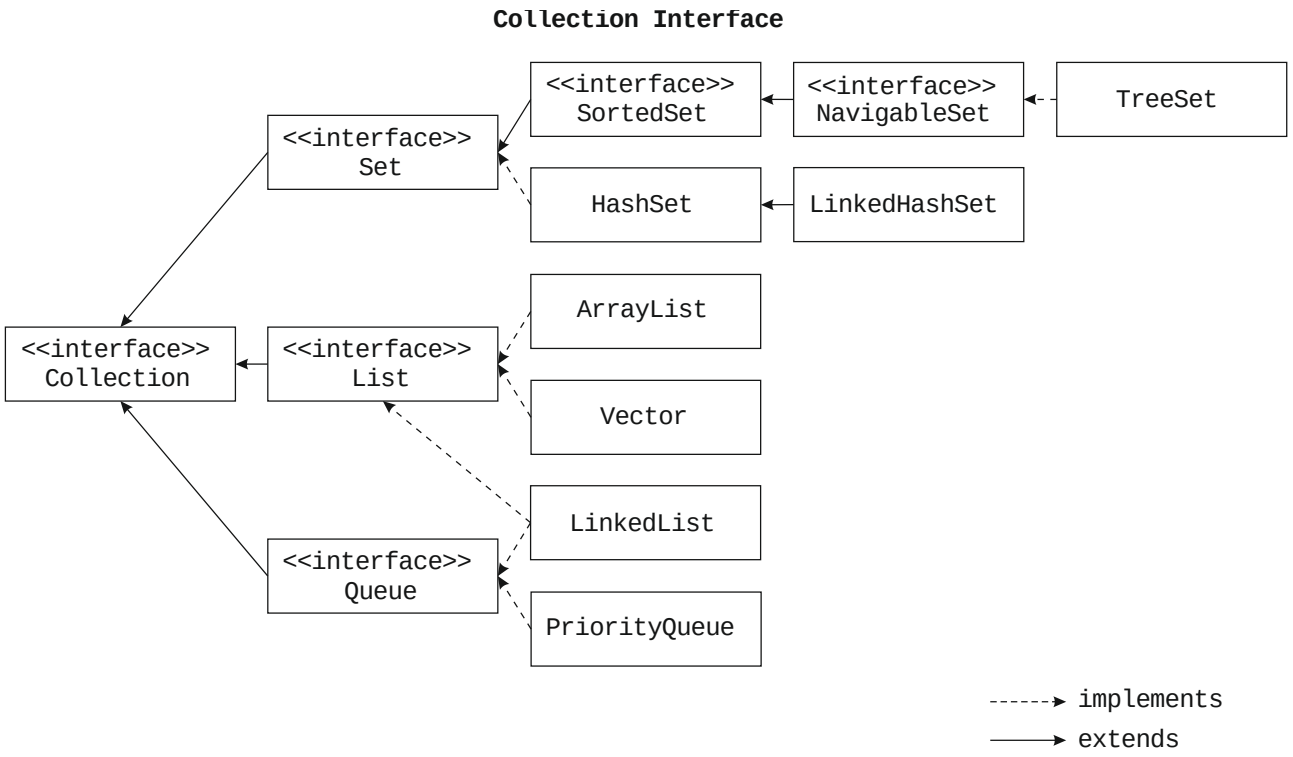


Рис. 4.4 – Иерархия `Collection`

Интерфейс `Collection` расширяют интерфейсы `List`, `Set` и `Queue`:

- `List` (*список*) представляет собой упорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (*sequence*). Элементы такой коллекции пронумерованы, начиная от ноля, к ним можно обратиться по индексу;
- `Set` (*множество*) описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов;
- `Queue` (*очередь*) предназначен для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса `Collection` очередь предоставляет дополнительные операции вставки, получения и контроля.

2. `Map` описывает коллекцию, состоящую из пар «ключ – значение» (рис. 4.5). У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (*map*). Такую коллекцию также часто называют словарем (*dictionary*) или ассоциативным массивом (*associative array*). Словарь может содержать произвольное число элементов, никак НЕ относится к интерфейсу `Collection` и является самостоятельным.

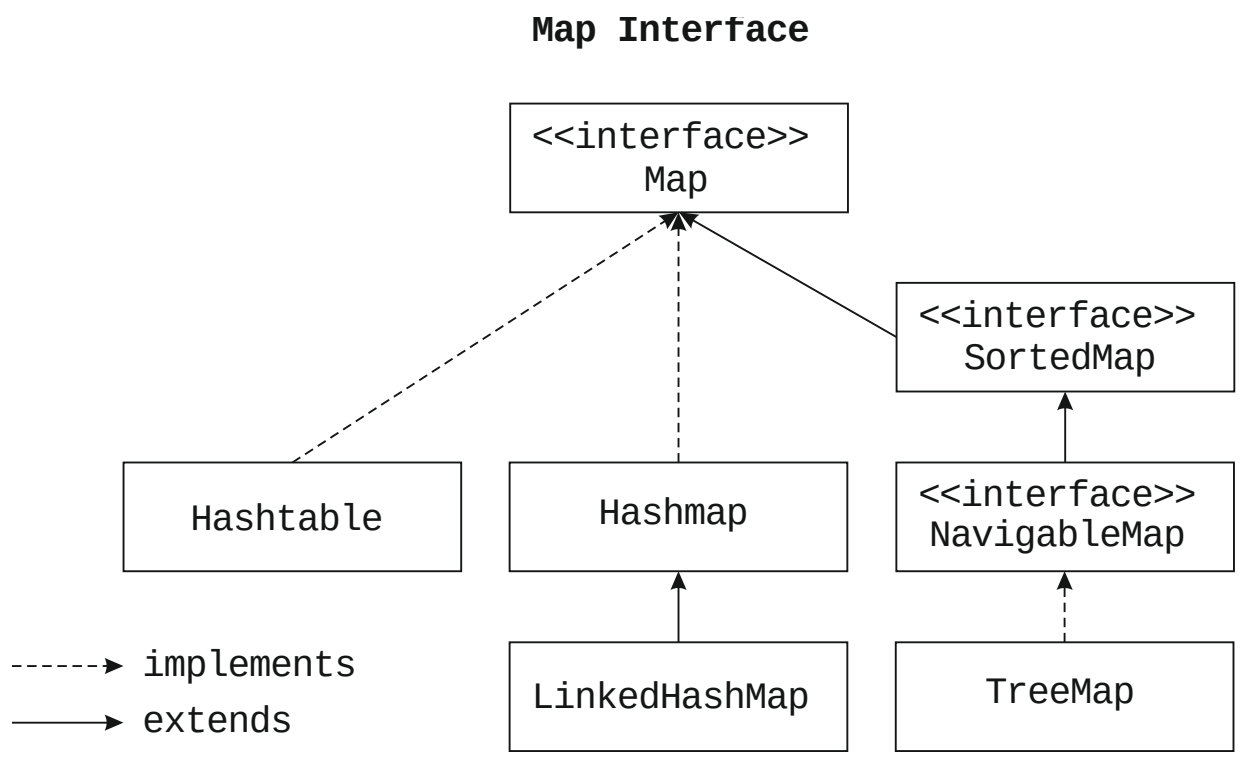


Рис. 4.5 – Иерархия `Map`

Интерфейсы `Collection` и `Map` являются базовыми, но не единственными. Их расширяют другие интерфейсы, добавляющие дополнительный функционал.

`Map` не наследует интерфейс `Collection`, так как они несовместимы. В интерфейсе `Collection` описан метод `add(Object o)`. Словари не могут содержать этот метод, потому что работают с парами ключ/значение. Также словари имеют представления `keySet`, `valueSet`, которых нет в коллекциях. Эти различия обусловили то, что интерфейс `Map` не может наследовать интерфейс `Collection` и представляет собой отдельную ветвь иерархии.

В таблице 4.1 представлены классы наборов данных [8].

Таблица 4.1 – Классы коллекций

Класс	Описание
`ArrayList`	Индексируемая последовательность, размер которой может увеличиваться и уменьшаться

Класс	Описание
`LinkedList`	Упорядоченная последовательность, обеспечивающая эффективное выполнение операций включения или удаления элемента в любой позиции
`HashSet`	Неупорядоченный набор, не допускающий дублирования элементов
`TreeSet`	Сортированное множество элементов
`EnumSet`	Набор значений нумерованного типа
`LinkedHashSet`	Множество, которое помнит порядок, в котором элементы были включены в него
`PriorityQueue`	Набор, обеспечивающий эффективное удаление наименьшего элемента
`HashMap`	Карта, которая хранит связи ключ/значение
`TreeMap`	Карта, в которой ключи отсортированы
`EnumMap`	Карта, в которой ключи принадлежат нумерованному типу
`LinkedHashMap`	Карта, которая помнит порядок включения элементов в нее
`WeakHashMap`	Карта, не используемые значения которой могут быть обработаны системой сборки мусора
`IdentityHashMap`	Карта, для сравнения ключей которой может быть использована операция `==`

Так как большинство коллекций параметризированы (начиная с Java 5), то при описании интерфейсов и классов принято указывать `T`, что означает любой класс, которым параметризируете коллекцию. Использование `<E>` – это указание типа объекта, который коллекция может содержать. Это помогает сократить ошибки времени выполнения с помощью проверки типов объектов во время компиляции.

Параметризованные типы (настраиваемые типы, *generic types*):

- позволяют создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр;
- позволяют создавать более компактный код, чем универсальные (обобщенные) типы, использующие ссылки типа `Object`;
- обеспечивают автоматическую проверку и приведение типов;
- позволяют создавать хороший, годный повторно используемый код.

Методы интерфейсов и их описание представлены в таблице 4.2 [8].

Таблица 4.2 – Основные методы коллекций

Интерфейс	Методы
`Collection`	`add(T e)` – добавить элемент `e`
	`clear()` – очистить
	`addAll(Collection col)` – добавить все элементы другой коллекции, со схожим типом данных
	`contains(Object o)` – содержит ли коллекция элемент
	`isEmpty()` – возвращает, пуста ли коллекция
	`remove(Object o)` – удаляет элемент
	`removeAll(Collection col)` – удаляет все элементы, которые есть в коллекции `col`
	`size()` – возвращает количество элементов в коллекции
	`containsAll(Collection col)` – возвращает, содержатся ли все элементы `col` в коллекции
	`toArray(T[] a)` – возвращает массив, который содержит все элементы коллекции, на вход принимает массив, который будет заполнен ими
	`retainAll(Collection col)` – удаляет все элементы, не принадлежащие `col`
	`toArray()` – возвращает массив объектов, который содержит все элементы коллекции
`Set`	Содержит все операции интерфейса `Collection`, но некоторые из них имеют другой смысл. Например, операция `add` добавляет только уникальные элементы, зато операции поиска элемента происходят быстрее, чем в списке
`List`	Все методы интерфейса `Collection`, а также следующие: • `get(int index)` – получает элемент по индексу; • `add(int index, T e)` – вставляет элемент в позицию; • `indexOf(Object obj)` – возвращает первое вхождение элемента в список; • `lastIndexOf(Object obj)` – возвращает последнее вхождение; • `set(int index, T e)` – заменяет элемент в позиции `index`; • `subList(int from, int to)` – возвращает новый

Интерфейс	Методы
	список, представляющий собой часть главного
Map	`size()` – возвращает количество элементов
	`containsKey(Object key)` – проверяет наличие ключа
	`containsValue(Object value)` – проверяет наличие значения в ассоциативном массиве
	`get(Object key)` – возвращает значение по ключу
	`put(K key, V value)` – кладет в ассоциативный массив значение `value` по ключу `key`. В случае наличия уже такого ключа происходит замена значения
	`values()` – возвращает значения всех элементов в виде коллекции
	`remove(Object key)` – удаляет элемент с ключом `key`, возвращая значение этого элемента (вернет `null` в случае отсутствия)
	`clear()` – удаляет все элементы из массива
	`isEmpty()` – возвращает, не пуст ли массив
	`size()` – возвращает количество элементов
	`containsKey(Object key)` – проверяет наличие ключа

Перебор элементов коллекций

1. Перебор с помощью итератора.

Интерфейс `Collection` расширяет интерфейс `Iterable`, у которого есть только один метод `iterator()`. Объект типа `Iterator` может использоваться для последовательного перебора элементов коллекции.

Интерфейс `Iterator` имеет следующее определение:

```
public interface Iterator <E> {
    E next();
    boolean hasNext();
    void remove();
}
```

Реализация интерфейса предполагает, что с помощью вызова метода `next()` можно получить следующий элемент. С помощью метода `hasNext()` можно узнать, есть ли следующий элемент и не достигнут ли конец коллекции. И если элементы еще имеются, то `hasNext()` вернет значение `true`. Метод

hasNext() следует вызывать перед методом next(), так как при достижении конца коллекции метод next() выбрасывает исключение NoSuchElement-Exception. И метод remove() удаляет текущий элемент, который был получен последним вызовом next().

Используем итератор для перебора коллекции ArrayList:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
    ArrayList<String> cats = new ArrayList<String>();
        states.add("Мася");
        states.add("Василий");
        states.add("Мурзик");
        states.add("Барсик");

        Iterator<String> iter = cats.iterator();
        while(iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор – интерфейс ListIterator. Данный итератор используется классами, реализующими интерфейс List, то есть классами LinkedList, ArrayList и др.

Интерфейс ListIterator расширяет интерфейс Iterator и определяет ряд дополнительных методов:

- void add(E obj): вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом next();
- boolean hasNext(): возвращает true, если в коллекции имеется следующий элемент, иначе возвращает false;
- boolean hasPrevious(): возвращает true, если в коллекции имеется предыдущий элемент, иначе возвращает false;
- E next(): возвращает следующий элемент, если такого нет, то генерируется исключение NoSuchElementException;

- `E previous()`: возвращает предыдущий элемент, если такого нет, то генерируется исключение `NoSuchElementException`;
- `int nextIndex()`: возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка;
- `int previousIndex()`: возвращает индекс предыдущего элемента. Если такого нет, то возвращается число –1;
- `void remove()`: удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов `next()` или `previous()`, иначе будет сгенерировано исключение `IlligalStateException`;
- `void set(E obj)`: присваивает текущему элементу, выбранному вызовом методов `next()` или `previous()`, ссылку на объект `obj`.

Используем `ListIterator`:

```
import java.util.*;
public class CollectionApp {
  public static void main(String[] args) {
    ArrayList<String> cats = new ArrayList<String>();
        cats.add("Мася");
        cats.add("Василий");
        cats.add("Мурзик");
        cats.add("Барсик");
    ListIterator<String> listIter = cats.listIterator();

        while(listIter.hasNext()) {
            System.out.println(listIter.next());
        }
        // сейчас текущий элемент - Мася
        // изменим значение этого элемента
        listIter.set("Муся");
        // пройдемся по элементам в обратном порядке
        while(listIter.hasPrevious()) {
            System.out.println(listIter.previous());
        }
  }
}
```

2. С помощью цикла `forEach`.

В Java 8 у интерфейса `Iterable` появился метод `forEach`, принимающий лямбда-выражение и применяющий это выражение на каждый элемент коллекции:

```
Map<String, Integer> cats = new HashMap<>();
cats.put("Васька", 10);
cats.put("Мася", 15);
cats.put("Пушистик", 6);
cats.put("Барсик", 2);
cats.put("Муся", 5);
cats.put("Мышка", 7);
cats.forEach((key, value) -> {
    System.out.println(key + " == " + value);
});
```

3. С помощью цикла `for`.

```
List<String> cats = Arrays.asList("Васька", "Муся", "Барсик", "Рыжик", "Пушок");
```

Рассмотрим операцию перебора элементов коллекции в старых версиях Java:

```
for (int i = 0; i < cats.length; i++) {
    System.out.println(cats.get(i))
}
```

Java также предлагает более современный подход:

```
for (String cat_name :cats) {
    System.out.println(cat_name);
}
```

4. С помощью расширенного цикла `for`.

5. С помощью цикла `while`.

Рассмотрим пример перебора на примере очереди (*Queue*), у которой есть дополнительные методы по добавлению, извлечению и проверке элементов. Чаще всего порядок выдачи элементов соответствует FIFO (firstin, first-out), но в общем случае определяется конкретной реализацией. Очереди не могут хранить `null`. У очереди может быть ограничен размер.

Основные методы очереди:

- `element();` – возвращает, но не удаляет головной элемент очереди;

- `offer(E o);` – добавляет в конец очереди новый элемент и возвращает `true`, если вставка удалась;
- `peek();` – возвращает первый элемент очереди, не удаляя его;
- `poll();` – возвращает первый элемент и удаляет его из очереди;
- `remove();` – возвращает и удаляет головной элемент очереди.

```
Queue<String> queue = new LinkedList<String>();
queue.offer("Мася");
queue.offer("Василий");
queue.offer("Том");
queue.offer("Тимка");
while (queue.size() > 0) {
    System.out.print(queue.remove() + " ");
}
```

Интерфейс `Deque` позволяет реализовать двунаправленная очередь, разрешающую вставку и удаление элементов в два конца очереди.

Методы `addFirst(e)`, `addLast(e)` вставляют элементы в начало и в конец очереди соответственно. Метод `add(e)` унаследован от интерфейса `Queue` и абсолютно аналогичен методу `addLast(e)` интерфейса `Deque`.

```
Deque<String> deque = new LinkedList<String>();
deque.offer("Мася");
deque.offer("Василий");
deque.addFirst("Том");
deque.offer("Тимка");
while (deque.size() > 0) {
    System.out.print(deque.remove() + " ");
}
```

Выводы

Главные преимущества коллекций:

- уменьшаются затраты времени на написание кода;
- улучшается производительность благодаря использованию высокоэффективных алгоритмов и структур данных;
- коллекции являются универсальным способом хранения и передачи данных, что упрощает взаимодействие разных частей кода;

- простота в изучении, потому что необходимо выучить только самые верхние интерфейсы и поддерживаемые операции;
- реализуется поддержка многопоточного доступа.

4.12 Потоки

Ввод/вывод относится к различным способам получения и передачи информации: к чтению и записи дискового файла, символов с клавиатуры либо получению данных из сети. Эти абстракции дают удобную возможность для работы с вводом-выводом (I/O), не требуя при этом, чтобы каждая часть вашего кода понимала разницу между, скажем, клавиатурой и сетью. В Java эта абстракция называется потоком (*stream*) и реализована в нескольких классах пакета `java.io`.

Поток данных *(stream) представляет собой абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.*

Поток прячет детали того, что случается с данными внутри реального устройства ввода-вывода. При этом потоки делятся на *байтовые* и *символьные*. Единицей обмена для байтовых потоков является байт, для символьных – символ Unicode.

На вершине иерархии байтовых потоков находятся два абстрактных класса: `InputStream` и `OutputStream` (рис. 4.6). В этих классах определены методы `read()` и `write()`, предназначенные для чтения данных из потока и записи данных в поток соответственно.

Иерархия классов для символьных потоков ввода-вывода начинается с абстрактных классов `Reader` и `Writer` (рис. 4.7). В этих классах определены методы `read()` для считывания символьных данных из потока и `write()` для записи символьных данных в поток.

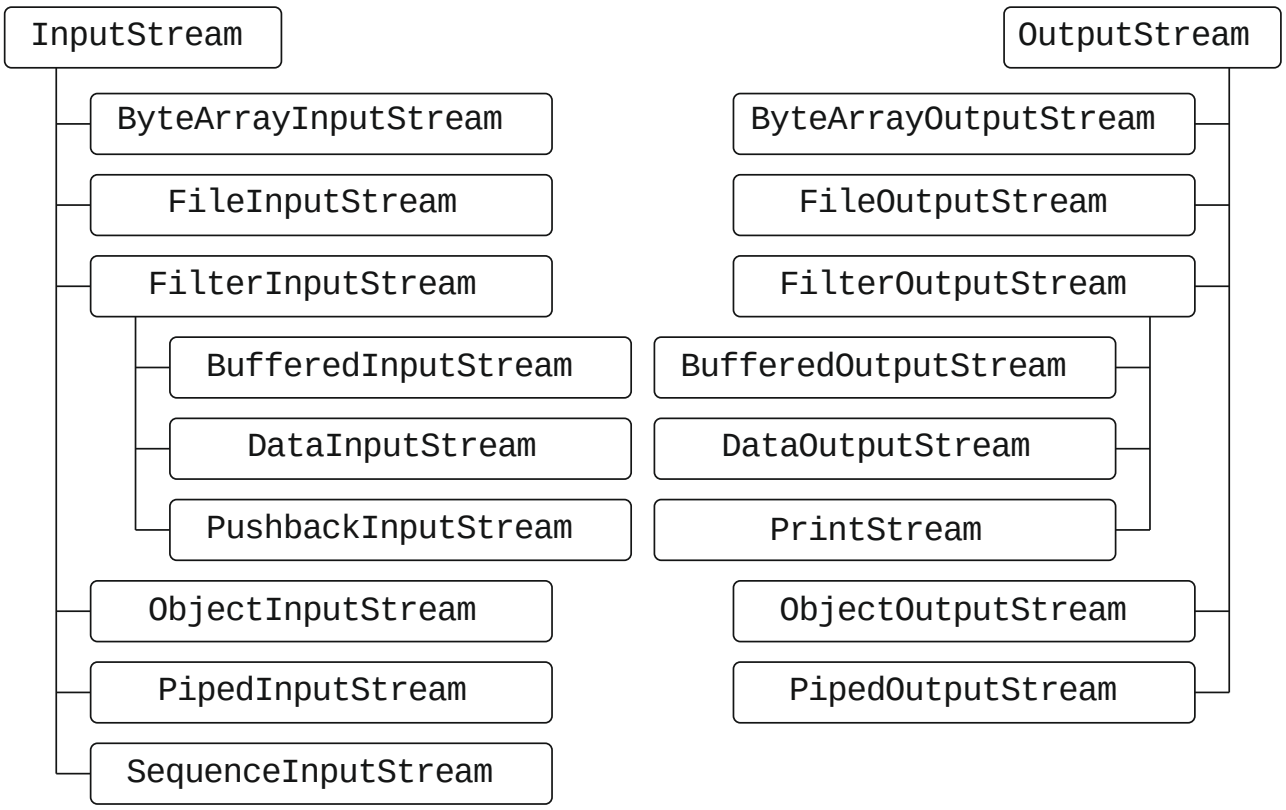


Рис. 4.6 – Иерархия байтовых потоков

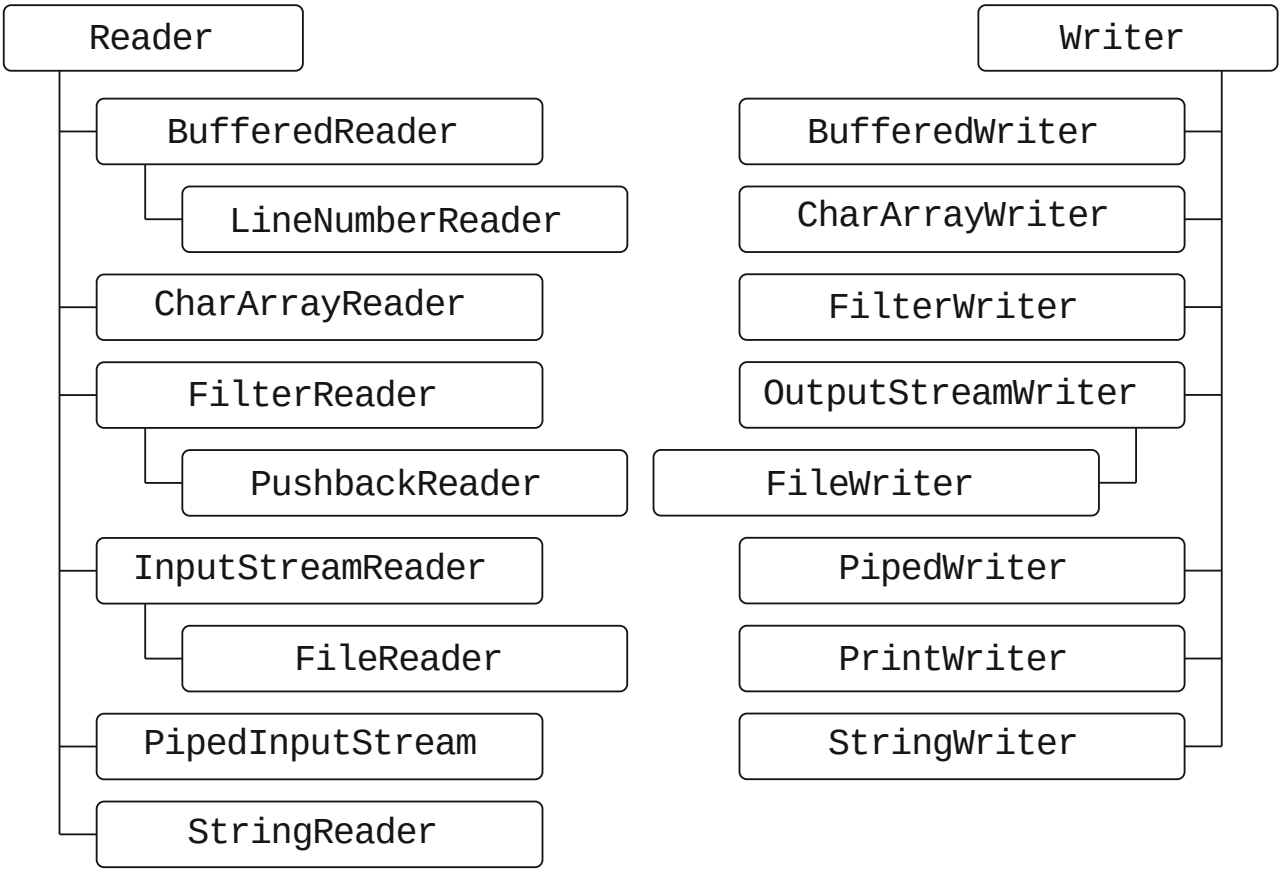


Рис. 4.7 – Иерархия символьных потоков

Бывают ситуации, когда требуется совместно использовать байтовые и символьные потоки данных, например, если данные должны поступить из источника в виде набора битов, быть каким-либо образом обработаны и переданы в другое место также в виде битов, при этом известно, что данные являются текстовой информацией. В таких ситуациях более удобно на этапе обработки данных перейти от байтовых потоков к символьным, так как последние предоставляют более приспособленный к обработке текста инструментарий. Для преобразования между байтовыми и символьными потоками используются *классы-«мосты»*: `InputStreamReader` – для преобразования от байтового потока к символьному для чтения данных и `OutputStreamWrite` – от символьного потока к байтовому для записи данных.

`InputStream`

Работа `InputStream` состоит в представлении классов, которые производят ввод от различных источников. Каждый из них имеет ассоциированный подкласс `InputStream`.

Все методы этого класса при возникновении ошибки возбуждают исключение `IOException`. Ниже приведен краткий обзор методов класса `InputStream`:

- `read()` возвращает представление очередного доступного символа во входном потоке в виде целого;
- `read(byte b[])` пытается прочесть максимум `b.length` байтов из входного потока в массив `b`. Возвращает количество байтов, в действительности прочитанных из потока;
- `read(byte b[], int off, int len)` пытается прочесть максимум `len` байтов, расположив их в массиве `b`, начиная с элемента `off`. Возвращает количество реально прочитанных байтов;
- `skip(long n)` пытается пропустить во входном потоке `n` байтов. Возвращает количество пропущенных байтов;
- `available()` возвращает количество байтов, доступных для чтения в настоящий момент;
- `close()` закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению `IOException`;

- `mark(int readlimit)` ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано `readlimit` байтов;
- `reset()` возвращает указатель потока на установленную ранее метку;
- `markSupported()` возвращает `true`, если данный поток поддерживает операции `mark`/`reset`.

`OutputStream`

Эта категория включает классы, которые решают, куда будет производиться вывод: в массив байтов, в файл. Все методы этого класса имеют тип `void` и возбуждают исключение `IOException` в случае ошибки. Ниже приведен список методов этого класса:

- `write(int b)` записывает один байт в выходной поток. Обратите внимание – аргумент этого метода имеет тип `int`, что позволяет вызывать `write`, передавая ему выражение, при этом не нужно выполнять приведение его типа к `byte`;
- `write(byte b[])` записывает в выходной поток весь указанный массив байтов;
- `write(byte b[], int off, int len)` записывает в поток часть массива – `len` байтов, начиная с элемента `b[off]`;
- `flush()` очищает любые выходные буферы, завершая операцию вывода;
- `close()` закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать `IOException`.

Символьные потоки

Для работы с символьными потоками в Java существуют два базовых класса – `Reader` и `Writer`.

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

- `absract void close()`: закрывает поток ввода;
- `int read()`: возвращает целочисленное представление следующего символа в потоке. Если таких символов нет и достигнут конец файла, то возвращается число –1;

- `int read(char[] buffer)`: считывает в массив `buffer` из потока символы, количество которых равно длине массива `buffer`. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;
- `int read(CharBuffer buffer)`: считывает в объект `CharBuffer` из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;
- `absract int read(char[] buffer, int offset, int count)`: считывает в массив `buffer`, начиная со смещения `offset`, из потока символы, количество которых равно `count`;
- `long skip(long count)`: пропускает количество символов, равное `count`. Возвращает число успешно пропущенных символов.

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

- `Writer append(char c)`: добавляет в конец выходного потока символ `c`. Возвращает объект `Writer`;
- `Writer append(CharSequence chars)`: добавляет в конец выходного потока набор символов `chars`. Возвращает объект `Writer`;
- `abstract void close()`: закрывает поток;
- `abstract void flush()`: очищает буферы потока;
- `void write(int c)`: записывает в поток один символ, который имеет целочисленное представление;
- `void write(char[] buffer)`: записывает в поток массив символов;
- `absract void write(char[] buffer, int off, int len)`: записывает в поток только несколько символов из массива `buffer`. Причем количество символов равно `len`, а отбор символов из массива начинается с индекса `off`;
- `void write(String str)`: записывает в поток строку;
- `void write(String str, int off, int len)`: записывает в поток из строки некоторое количество символов, которое равно `len`, причем отбор символов из строки начинается с индекса `off`.

Специализированные потоки

В пакет `java.io` входят потоки для работы с основными типами источников и приемников данных (табл. 4.3).

Таблица 4.3 – Специализированные потоки

Тип данных	Байтовые потоки		Символьные потоки	
	Входной	Выходной	Входной	Выходной
Файл	`FileInput-Stream`	`FileOutput-Stream`	`FileReader`	`FileWriter`
Массив	`ByteArray-InputStream`	`ByteArray-OutputStream`	`CharArray-Reader`	`CharArray-Writer`
Строка	_	_	`String-Reader`	`String-Writer`
Конвейер	`PipedInput-Stream`	`PipedOutput-Stream`	`Piped-Reader`	`Piped-Writer`

Конструкторы этих потоков в качестве аргумента принимают ссылку на источник или приемник данных – файл, массив, строку.

Файловые потоки

Java обеспечивает ряд классов и методов, которые позволяют читать и записывать файлы. Для Java все файлы имеют байтовую структуру, а Java обеспечивает методы для чтения и записи байтов в файл. Кроме того, Java позволяет упаковывать байтовый файловый поток в символьно-ориентированный объект.

Для создания байтовых потоков, связанных с файлами, чаще всего используются два поточных класса – `FilelnputStream` и `FileOutput-Stream`. Для открытия файла создается объект одного из этих классов с указанием имени файла как аргумента конструктора.

Стандартные потоки ввода-вывода

Термин *стандартный ввод-вывод* относится к концепции Unix (которая в некоторой форме была воспроизведена в Windows и многих других операционных системах) единого потока информации, который используется программой. Весь ввод программы может вестись через *стандартный ввод*, весь вывод может идти в *стандартный вывод*, а все сообщения об ошибках могут посылаться в *стандартный* поток *ошибок*. Значение стандартного ввода-вывода

в том, что программы вместе могут представлять цепочку и стандартный вывод одной программы может стать стандартным вводом для другой.

Чтение из стандартного ввода

Часть возможностей ввода-вывода может быть реализована посредством класса `System`.

Класс `java.lang.System` является `final`, все поля и методы являются статическими (`static`), поэтому нельзя создать подкласс и переопределить его методы, используя наследование. Класс Java `System` не предоставляет каких-либо публичных конструкторов, поэтому мы не можем создать экземпляр этого класса.

Класс `System` содержит три переменных потока: `in`, `out` и `err`. Эти поля имеют атрибуты `public` и `static`:

- поле `System.out` – поток стандартного вывода. По умолчанию он связан с консолью. Поле `System.out` является объектом класса `PrintStream`;
- поле `System.in` – это поток стандартного ввода. По умолчанию он связан с клавиатурой. Поле является объектом класса `InputStream`;
- поле `System.err` – это стандартный поток ошибок. По умолчанию поток связан с консолью. Поле является объектом класса `PrintStream`.

Для вывода на консоль ранее использовался метод `println()` класса `PrintStream` без определения экземпляров этого класса. Можно использовать статическое поле `out` класса `System`, которое является объектом класса `PrintStream`. Исполняющая система Java связывает это поле с консолью.

Можно писать `System.out.println()`, определяя новую ссылку на `System.out`, например:

```
PrintStream pr = System.out;
```

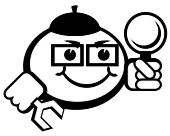
и писать просто

```
pr.println();
```

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты.

Для символов Latin 1 с кодами '\u0000'–'\u00FF' при этом просто откидывается нулевой старший байт и выводятся байты '0x00'–'0xFF'. Для кодов кириллицы, которые лежат в диапазоне '\u0400'–'\u04FF' кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали.

Пожалуй, наиболее широко используемой возможностью, предоставляемой `System`, является стандартный вывод, доступный через переменную `System.out`. Ее тип – `PrintStream`. Стандартный вывод можно перенаправить в другой поток (файл, массив байт и т. д., главное, чтобы это был объект `PrintStream`).


Пример

```
System.out.println("Наши котики сохранены в файл");
try {
    PrintStream print = new PrintStream(new FileOut-
    putStream("c:\\file.txt"));
    System.setOut(print);
    System.out.println("Наши котики сохранены");
    for(MyClass mc : list) {
        System.out.println("Item:" + mc);
    }
} catch(FileNotFoundException e) {
    e.printStackTrace();
}
```

Ввод с консоли

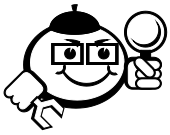
Для ввода данных используется класс `Scanner` из библиотеки пакетов Java.

Этот класс надо импортировать в той программе, где он будет использоваться. Это делается до начала открытого класса в коде программы.

В классе есть методы для чтения очередного символа заданного типа со стандартного потока ввода, а также для проверки существования такого символа.

.....

Для работы с потоком ввода необходимо создать объект класса `Scanner`, при создании указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом – `System.in`. А стандартный поток вывода (дисплей) – уже знакомым объектом `System.out`. Есть еще стандартный поток для вывода ошибок – `System.err`, но работа с ним выходит за рамки нашего курса.

.....  Пример .....

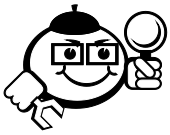
```
import java.util.Scanner; // импортируем класс
public class Cat {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in); /* создаем объект класса Scanner */
        int i;
        System.out.print("Введите возраст кота в виде целого числа: ");
        if(in.hasNextInt()) { /* возвращает истину, если с потока ввода можно считать целое число */
            i = in.nextInt(); /* считывает целое число с потока ввода и сохраняет в переменную */
            System.out.println(i);
        } else {
            System.out.println("Вы ввели не целое число");
        }
    }
}
```

.....

Метод `hasNextDouble()`, примененный к объекту класса `Scanner`, проверяет, можно ли считать с потока ввода вещественное число типа `double`,

а метод `nextDouble()` – считывает его. Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит).

Метод `nextLine()` позволяет считывать целую последовательность символов, т. е. строку, а значит, полученное через этот метод значение нужно сохранять в объекте класса `String`. В следующем примере создается два таких объекта, потом в них поочередно записывается ввод пользователя, а далее на экран выводится одна строка, полученная объединением введенных последовательностей символов.



Пример

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s;
        System.out.print("Введите имя кота: ");
        s = in.nextLine();
        System.out.println(s);
    }
}
```

Существует и метод `hasNext()`, проверяющий, остались ли в потоке ввода какие-то символы.

Как видно, связь с консолью средствами классов-потоков весьма сложна. Начиная с Java SE 6 в пакет `java.io` добавлен класс `Console`, облегчающий эту задачу.

Поскольку программа связывается с той консолью, в которой запущена виртуальная машина Java, единственный объект класса `Console` создается статическим методом `console()` класса `System`, например:

```
Console cons = System.console();
```

Метод возвращает `null`, если виртуальная машина Java запущена не из консоли, а каким-нибудь приложением.

Форматированный вывод

На технологию Java традиционно переходит очень много программистов, прежде писавших программы на языке C. Им очень не хватает функции `printf()`, позволяющей самому программисту как-то оформить вывод информации: задать количество цифр при выводе вещественных чисел, точно указать количество пробелов между данными.

Начиная с JDK 1.5 методы `printf()`, очень похожие на одноименные функции языка C, появились в классах `PrintStream` и `PrintWriter`. Кроме них в эти классы введены методы `format()`, выполняющие те же действия. Последние методы заимствованы из класса `Formatter`, находящегося в пакете `java.util` и специально предназначенного для форматирования.

Заголовки методов форматированного ввода/вывода класса `PrintStream` выглядят так:

- `PrintStream format(Local l, String format, Object ... args);`
- `PrintStream format(String format, Object ... args);`
- `PrintStream printf(Local l, String format, Object ... args);`
- `PrintStream printf(String format, Object ... args).`

В классе `PrintWriter` такие же методы возвращают ссылку на свой экземпляр класса `PrintWriter`.

Строка символов `format` описывает шаблон для вывода данных, перечисленных в следующих аргументах метода, а также содержит надписи, которые должны появиться на консоли. Например, тот же самый вывод на консоль, который мы до сих пор делали методом

```
System.out.println("x = " + x + ", y = " + y);
```

можно сделать методом

```
System.out.printf("x = %d, y = %d\n", x, y);
```

В строке формата мы пишем поясняющий текст `"x = , y = \n"`, который будет просто выводиться на консоль. В текст вставляем *спецификации формата* `"%d"`. На место этих спецификаций во время вывода будут подставлены значения данных, перечисленных в следующих аргументах метода.

Класс `File`

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

Для создания объектов класса `File` можно использовать один из следующих конструкторов:

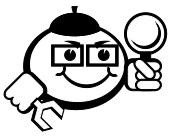
- `File(File dir, String name)` – указывается объекта класса `File` (каталог) и имя файла;
- `File(String path)` – указывается путь к файлу без указания имени файла;
- `File(String dirPath, Sring name)` – указывается путь к файлу и имя файла;
- `File(URI uri)` – указывается объекта `URI`, описывающий файл.

Методы класса `File`

Класс `File` может использоваться для создания каталога или дерева каталогов. Также можно узнать свойства файлов (размер, дату последнего изменения, режим чтения/записи), определить, к какому типу (файл или каталог) относится объект `File`, удалить файл. У класса очень много методов, перечислим некоторые:

- `getAbsolutePath()` – абсолютный путь файла начиная с корня системы. В Android корневым элементом является символ «слеш» (`/`);
- `canRead()` – доступно для чтения;
- `canWrite()` – доступно для записи;
- `exists()` – файл существует или нет;
- `getName()` – возвращает имя файла;
- `getParent()` – возвращает имя родительского каталога;
- `getPath()` – путь;
- `lastModified()` – дата последнего изменения;
- `isFile()` – объект является файлом, а не каталогом;
- `isDirectory` – объект является каталогом;
- `isAbsolute()` – возвращает `true`, если файл имеет абсолютный путь;

- `renameTo(File newPath)` – переименовывает файл. В параметре указывается имя нового имени файла. Если переименование прошло неудачно, то возвращается `false`;
- `delete()` – удаляет файл. Также можно удалить *пустой* каталог;
- `length()` – получить длину в байтах;
- `setReadable()`, `setWritable()`, `setExecutable()` – позволяют установить их для всех пользователей или только для владельца файла или каталога.



Пример

```
import java.io.File;
import java.io.IOException;

public class File_App {

  public static void main(String[] args) {
    // определяем объект для каталога
    File myFile = new File("C://work//oop.txt");
    System.out.println("Имя файла: " +
myFile.getName());
    System.out.println("Родительский каталог: " +
myFile.getParent());
    if(myFile.exists())
        System.out.println("Файл существует");
    else
        System.out.println("Файл еще не создан");

    System.out.println("Размер файла: " +
myFile.length());
    if(myFile.canRead())
    System.out.println("Файл доступен для чтения");
    else
    System.out.println("Файл не доступен для чтения");

    if(myFile.canWrite())
```

```
        System.out.println("Файл доступен для записи");
        else
        System.out.println("Файл не доступен для записи");

        // создадим новый файл
        File newFile = new File("C://work//MyFile");
        try {
            boolean created = newFile.createNewFile();
            if(created)
                System.out.println("Файл создан");
        }
        catch(IOException ex) {
            System.out.println(ex.getMessage());
        }
      }
    }
```

.....

Классы системы ввода-вывода NIO

Начиная с версии 1.4 в Java предоставляется вторая система ввода-вывода под названием NIO (сокращение от New I/O – новый ввод-вывод). В этой системе поддерживается канальный подход к операциям ввода-вывода, ориентированный на применение буферов [9].

Поточная модель использует потоки и байты; модель NIO использует каналы и буфера. NIO использует один тип объектов канала (*Channel*) вместо `InputStream` и `OutputStream`.

Java NIO позволяет управлять несколькими каналами (сетевыми соединениями или файлами), используя минимальное число потоков выполнения. Однако ценой такого подхода является более сложный, чем при использовании блокирующих потоков, парсинг данных. Система ввода-вывода Java NIO2 в дополнение к прежним классам предоставляет программисту широкий выбор методов работы с файлами.

При этом Java NIO отнюдь не является заменой Java IO. Его стоит рассматривать как усовершенствование – инструмент, позволяющий значительно расширить возможности по организации ввода-вывода. Грамотное использова-

ние мощности обоих подходов позволит вам строить хорошие высокопроизводительные системы.

Выводы

Библиотека потоков ввода-вывода Java удовлетворяет основным требованиям: вы можете выполнить чтение и запись с консолью, файлом, блоком памяти или даже через интернет. С помощью интерфейсов вы можете создать новые типы объектов ввода и вывода. Вы также можете использовать простую расширяемость объектов потоков, имея в виду, что метод `toString()` вызывается автоматически, когда вы передаете объект в метод, который ожидает `String` (ограничение Java на автоматическое преобразование типов).

Контрольные вопросы по главе 4

1. Дайте определение понятию «класс».
2. Что такое поле/атрибут класса?
3. Как правильно организовать доступ к полям класса?
4. Дайте определение понятию «конструктор».
5. Чем отличаются конструкторы по умолчанию, копирования и конструктор с параметрами?
6. Какие модификации уровня доступа вы знаете, расскажите про каждый из них.
7. О чем говорят ключевые слова `this`, `super`, где и как их можно использовать?
8. Дайте определение понятию «метод».
9. Что такое сигнатура метода?
10. Какие методы называются перегруженными?
11. Расскажите про переопределение методов.
12. Где и для чего используется модификатор `abstract`?
13. Дайте определение понятию «коллекция».
14. Назовите преимущества использования коллекций.
15. Какова иерархия `Collection`?
16. Что вы знаете о коллекциях типа `List`?

17. Что вы знаете о коллекциях типа `Set`?
18. Что вы знаете о коллекциях типа `Queue`?
19. Что вы знаете о коллекциях типа `Map`, в чем их принципиальное отличие?
20. Какие существуют виды потоков ввода-вывода?
21. Назовите основные предки потоков ввода-вывода.
22. Что общего и чем отличаются следующие потоки: `InputStream`, `OutputStream`, `Reader`, `Writer`?

5 Обработка исключений

Один из основополагающих принципов философии Java состоит в том, что «плохо написанная программа не должна запускаться» [9].

В идеале ошибки должны обнаруживаться во время компиляции, перед запуском программы. Однако не все ошибки удается выловить в процессе отладки. *Механизм исключений* значительно упрощает процесс отладки и восстановление после ошибок, позволяет передавать информацию о возникших проблемах в клиентский код.

Исключениями *или* ***исключительными ситуациями*** *(exception) называются ошибки, возникшие в программе во время ее работы.*

Исключения могут возникать во многих случаях, например:

1. Пользователь ввел некорректные данные.
2. Файл, к которому обращается программа, не найден.
3. Арифметические ошибки, деление на ноль.
4. Переполнение массива.
5. Сетевое соединение с сервером было утеряно во время передачи данных и т. д.

В тех языках программирования, где не поддерживается обработка исключений, ошибки должны проверяться и обрабатываться вручную, как правило, благодаря использованию кодов ошибок и т. п. Такой подход является обременительным и хлопотным. Обработка исключений в Java позволяет избежать подобных трудностей, а кроме того, переносит управление ошибками, возникающими во время выполнения, в область ООП. Используя подсистему обработки исключений Java, можно управлять реакцией программы на появление ошибок во время выполнения.

Обработка исключительных ситуаций *(exception handling) – механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы.*

Чтобы обеспечить требуемую реакцию на конкретную ошибку, в программу следует включить соответствующий обработчик событий. Исключения широко применяются в стандартной библиотеке Java API.

5.1 Иерархия классов исключений

В Java все исключения представлены отдельными классами. Все классы исключений являются потомками класса `Throwable` (рис. 5.1). Так, если в программе возникнет исключительная ситуация, будет сгенерирован объект класса, соответствующего определенному типу исключения. У класса `Throwable` имеются два непосредственных подкласса: `Exception` и `Error`. Исключения типа `Error` относятся к ошибкам, возникающим в виртуальной машине Java, а не в прикладной программе. Контролировать такие исключения невозможно, поэтому реакция на них в прикладной программе, как правило, не предусматривается.

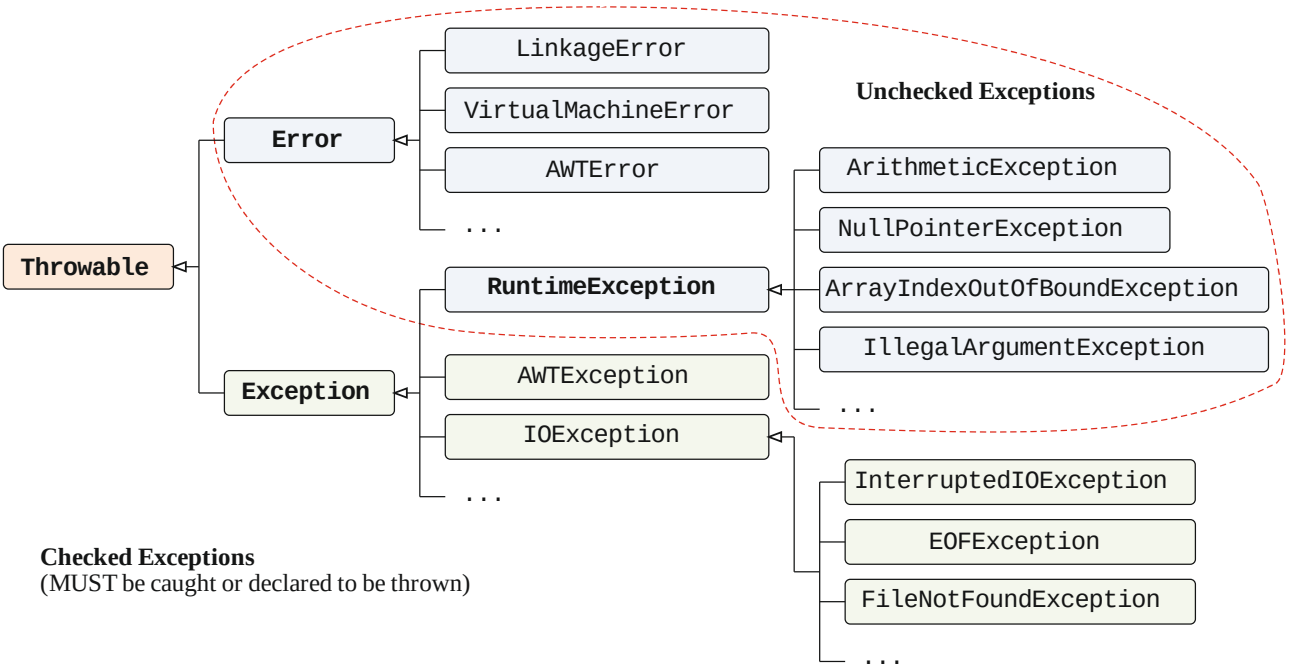


Рис. 5.1 – Иерархия классов исключений

Ошибки, связанные с работой программы, представлены отдельными подклассами, производными от класса `Exception`. В частности, к этой категории относятся ошибки деления на ноль, выхода за границы массива и обращения к файлам. Подобные ошибки следует обрабатывать в самой программе. Важным подклассом, производным от `Exception`, является класс `Runtime-`

`Exception`, который служит для представления различных видов ошибок, часто встречающихся во время выполнения программ.

Исключение в Java представляет собой объект, описывающий исключительную (т. е. ошибочную) ситуацию, возникающую в определенной части программного кода. Когда возникает такая ситуация, в вызвавшем ошибку методе генерируется объект, который представляет исключение. Этот метод может обработать исключение самостоятельно или же пропустить его. Так или иначе, в определенный момент исключение *перехватывается* и *обрабатывается*. Исключения могут генерироваться автоматически исполняющей системой Java или вручную в прикладном коде. Исключения, генерируемые исполняющей системой Java, имеют отношение к фундаментальным ошибкам, нарушающим правила языка Java или ограничения, накладываемые исполняющей системой Java. А исключения, генерируемые вручную, обычно служат для уведомления вызывающего кода о некоторых ошибках в вызываемом методе.

Все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком.

Исключения:

- `Throwable` – *проверяемое*;
- `Error` – *непроверяемое*;
- `Exception` – *проверяемое*;
- `RuntimeException` – *непроверяемое*.

Исключения:

1. `Checked` исключения – это те, которые должны обрабатываться блоком `catch` или описываться в сигнатуре метода. `Unchecked` могут не обрабатываться и не быть описанными.

2. `Unchecked` исключения в Java – наследованные от `RuntimeException`, `checked` – от `Exception` (не включая `unchecked`).

После создания исключения Java *Runtime Environment* пытается найти *обработчик исключения*.

Обработчик исключения *– блок кода, который может обрабатывать объект-исключение.*

Полезные методы класса `Throwable`:

1) `public String getMessage()` – возвращает сообщение, которое было создано при создании исключения через конструктор;
2) `public String getLocalizedMessage()` – метод, который переопределяют в подклассах для локализации конкретного сообщения об исключении. В реализации `Throwable` класса этот метод просто использует метод `getMessage()`, чтобы вернуть сообщение об исключении (`Throwable` на вершине иерархии – ему нечего локализировать, поэтому он вызывает `getMessage()`);
3) `public synchronized Throwable getCause()` – возвращает причину исключения или идентификатор в виде `null`, если причина неизвестна;
4) `public String toString()` – возвращает информацию о `Throwable` в формате `String`;
5) `public void printStackTrace()` – выводит информацию трассировки стека в стандартный поток ошибок. Этот метод перегружен, и мы можем передать `PrintStream` или `PrintWriter` в качестве аргумента, чтобы написать информацию трассировки стека в файл или поток.

5.2 Обработка исключений

Для обработки исключений в Java предусмотрены пять ключевых слов: `try`, `catch`, `throw`, `throws` и `finally`. Они образуют единую подсистему, в которой использование одного ключевого слова почти всегда автоматически влечет за собой употребление другого. Операторы, в которых требуется отслеживать появление исключений, помещаются в блок `try`. Если в блоке `try` будет сгенерировано исключение, его можно перехватить.

Обработка исключений:

1) `try` – данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке;
2) `catch` – ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений;
3) `finally` – ключевое слово для отметки начала блока кода, которое является дополнительным. Этот блок помещается после последнего

блока `catch`. Управление обычно передается в блок `finally` в любом случае;

4) `throw` – служит для генерации исключений;
5) `throws` – ключевое слово, которое прописывается в сигнатуре метода и обозначает, что метод потенциально может выбросить исключение с указанным типом.

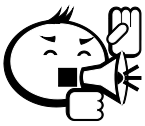
Операторы `try-catch`

Основными языковыми средствами обработки исключений являются ключевые слова `try` и `catch`. Они используются совместно. Это означает, что в коде нельзя указать ключевое слово `catch`, не указав ключевое слово `try`. Ниже приведена общая форма записи блоков `try-catch`, предназначенных для обработки исключений.

```
try {
/* здесь код, который потенциально может привести к ошибке */
}
catch(SomeException e) {
/* в скобках указывается класс конкретной ожидаемой ошибки, описываются действия, направленные на обработку исключений */
}
finally {
/* выполняется в любом случае (блок finally не обязателен) */
}
```

Следует иметь в виду, что если исключение не генерируется, то блок `try` завершается обычным образом и ни один из его операторов `catch` не выполняется. Выполнение программы продолжается с первого оператора, следующего за последним оператором `catch`.

Таким образом, операторы `catch` выполняются только при появлении исключения.



Начиная с версии JDK 7 используется новая форма оператора `try`, поддерживающая автоматическое управление ресурсами и называемая оператором `try` с ресурсами.

Откомпилируем и запустим такую программу:

```
class Main {
    public static void main(String[] args) {
        int a = 4;
        System.out.println(a/0);
    }
}
```

В момент запуска на консоль будет выведено следующее сообщение:

```
Exception in thread "main" java.lang.ArithmeticException: /
by zero at Main.main(Main.java:4)
```

Когда исполняющая система Java обнаруживает попытку деления на ноль, она создает новый объект исключения, а затем генерирует исключение. В данном примере обработчик исключений отсутствует, и поэтому исключение перехватывается стандартным обработчиком, предоставляемым исполняющей системой Java (рис. 5.2). Любое исключение, не перехваченное прикладной программой, в конечном итоге будет перехвачено и обработано этим стандартным обработчиком.

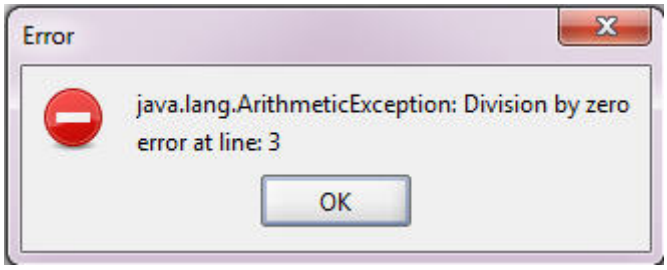


Рис. 5.2 – Исключение `Arithmetic Exception`

Следует также иметь в виду, что сгенерированное исключение относится к подклассу `Arithmetic Exception`, производному от класса `Exception` и точнее описывающему тип возникшей ошибки.

Добавим в код обработку ошибки:

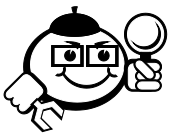
```
class Main {
    public static void main(String[] args) {
```

```
            int a = 4;
            try {
                System.out.println(a/0);
            } catch (ArithmeticException e) {
                System.err.println("Произошла недопусти-
            мая арифметическая операция");
            }
        }
    }
```

Эта программа выводит следующий результат:

```
Произошла недопустимая арифметическая операция
```

Иногда в одном фрагменте кода может возникнуть не одно исключение. Чтобы справиться с такой ситуацией, можно указать два или больше операторов `catch`, каждый из которых предназначается для перехвата отдельного типа исключения.



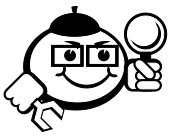
Пример

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        int[] m = {-1,0,1};
        Scanner sc = new Scanner(System.in);
        try {
            int a = sc.nextInt();
            m[a] = 4/a;
            System.out.println(m[a]);
        } catch (ArithmeticException e) {
            System.out.println("Произошла недопусти-
        мая арифметическая операция");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Ошибка индексации за
        пределами массива");
        }
    }
}
```

Если, запустив представленную программу, пользователь введет с клавиатуры 1 или 2, то программа отработает без создания каких-либо исключений.

Если пользователь введет 0, то возникнет исключение класса `ArithmeticException` и оно будет обработано первым блоком `catch`. Если пользователь введет 3, то возникнет исключение класса `ArrayIndexOutOfBoundsException` (выход за приделы массива) и оно будет обработано вторым блоком `catch`. Если пользователь введет нецелое число, например 1.5, то возникнет исключение класса `InputMismatchException` (несоответствие типа вводимого значения) и оно будет выброшено в формате стандартной ошибки.

Операторы `try` могут быть вложенными. Это означает, что один оператор `try` может находиться в блоке другого оператора `try`. Всякий раз, когда управление передается блоку оператора `try`, контекст соответствующего исключения размещается в стеке. Если во вложенном операторе `try` отсутствует оператор `catch` для перехвата и обработки конкретного исключения, стек развертывается и проверяется на соответствие оператор `catch` из внешнего блока оператора `try`, и так до тех пор, пока не будет найден подходящий оператор `catch` или не будут исчерпаны все уровни вложенности операторов `try`. Если подходящий оператор `catch` не будет найден, то возникшее исключение обработает исполняющая система Java. Ниже приведен пример программы, демонстрирующий применение вложенных операторов `try`.



Пример

```
public class Main {
  public static void main(String[] args) {
    try {
        int a = args.length;
        int b = 4/a;
        System.out.println("a"+a);
        try { // вложенный блок try
            if (a==1) a=a/(a-a); // деление на ноль
            if (a==2) {
                int c[]={1};
                c[4]=99;
```

```
            }
        }
        // здесь генерируется исключение в связи
        // с выходом за пределы массива
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Индекс за пределами
        массива:"+e);
        }
    }
    catch(ArithmeticException e) {
        System.out.println("Деление на ноль:"+e);
    }
  }
}
```

Как видите, в этой программе один блок `try` вложен в другой. Программа работает следующим образом. Когда она запускается на выполнение без аргументов командной строки, во внешнем блоке оператора `try` генерируется исключение в связи с делением на ноль.

```
Деление на ноль:java.lang.ArithmeticException: / by
zero
```

А если программа запускается на выполнение с одним аргументом, то исключение в связи с делением на ноль генерируется во вложенном блоке оператора `try`. Но поскольку это исключение не обрабатывается во вложенном блоке, то оно передается внешнему блоку оператора `try`, где и обрабатывается.

```
a = 1
Деление на ноль:java.lang.ArithmeticException: / by
zero
```

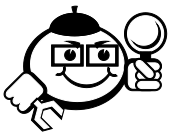
Если же программе передаются два аргумента командной строки, то во внутреннем блоке оператора `try` генерируется исключение в связи с выходом индекса за пределы массива.

```
a = 2
Индекс за пределами
массива:java.lang.ArrayIndexOutOfBoundsException: 4
```

Оператор `finally`

Когда генерируется исключение, выполнение метода направляется по нелинейному пути, резко изменяющему нормальную последовательность выполнения операторов в теле метода. В зависимости от того, как написан метод, исключение может даже стать причиной преждевременного возврата из метода. В некоторых методах это может вызвать серьезные осложнения. Так, если файл открывается в начале метода и закрывается в конце, то вряд ли кого-нибудь устроит, что код, закрывающий файл, будет обойден механизмом обработки исключений. Для таких непредвиденных обстоятельств и служит оператор `finally`.

Оператор `finally` образует блок кода, который будет выполнен по завершении блока операторов `try-catch`, но перед следующим за ним кодом. Блок оператора `finally` выполняется независимо от того, сгенерировано ли исключение. Если исключение сгенерировано, блок оператора `finally` выполняется, даже при условии, что ни один из операторов `catch` не совпадает с этим исключением.




```
public class Main {
  public static void main(String[] args) {
    int a = 4;
    try {
        System.out.println(a/0);
    } catch (ArithmeticException e) {
        System.err.println("Произошла недопустимая
    арифметическая операция");
    }

    finally {
        System.out.println("Блок finally");
    }

    System.out.println("Программа завершена");
  }
}
```

```
Произошла недопустимая арифметическая операция
Блок finally
Программа завершена
```

. . . . . .

В любой момент, когда метод собирается возвратить управление вызывающему коду из блока оператора `try-catch` (через необработанное исключение или явным образом через оператор `return`), блок оператора `finally` выполняется перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были выделены в начале метода и должны быть освобождены перед возвратом из него. Указывать оператор `finally` необязательно, но каждому оператору `try` требуется хотя бы один оператор `catch` или `finally`.

Оператор `throw`

В блоке `try-catch` перехватывались только те исключения, которые генерировала исполняющая система Java. Но исключения можно генерировать и непосредственно в прикладной программе, используя оператор `throw`:

- `throw объектThrowable;`
- `throw new IOException();`
- `throw new IllegalArgumentException(); /* генерация исключения */`

Таким образом, мы хотим прервать работу функций, но вместе с тем получить информацию о том, где и какая ошибка произошла. Для этого в Java можно поступить следующим образом:

```
int calculate(int a, int b) {
    if (b != 0) {
        return a / b;
    } else {
        throw new ArithmeticException();
    }
}
```

В этот момент создастся новый объект типа `ArithmeticException`. Бросая его (*throw*), функция прерывается (никаких значений она при этом вообще не возвращает), прерывается и та функция, которая ее вызвала и т. д. Условно говоря, этот `ArithmeticException` поднимается вверх по стеку

вызовов функций и в него записывается, где мы «вывалились» (из какого метода, в какой строчке и с какими параметрами). Когда произойдет вылет из функции `main`, работа программы завершится и будет выведен `stacktrace` – информация о том, работа каких функций и в каких местах привела к ошибке.

Оператор `throws`

Иногда исключения нецелесообразно обрабатывать в том методе, в котором они возникают. В таком случае их следует указывать с помощью ключевого слова `throws`. Ниже приведена общая форма объявления метода, в котором присутствует ключевое слово `throws`. Ключевое `throws` указывает на то, что метод может сгенерировать (выбросить) одно из перечисленных в списке исключений.

```
import java.util.*;
import java.io.*;
public class Main {
    public static void main(String[] args) {
        FileInputStream f = new FileIn-
    putStream("input.txt");
    }
}
```

```
Ошибка компиляции time: 0.04 memory: 711168 signal:-1
Main.java:15: error: unreported exception FileNot-
FoundException; must be caught or declared to be thrown
FileInputStream f = new FileInputStream(“input.txt");
1 error
```

Пробросим исключение:

```
import java.util.*;
import java.io.*;
public class FileWithThrow {
    public static void main(String[] args) throws
FileNotFoundException {
        FileInputStream f = new FileIn-
    putStream("input.txt");
    }
}
```

```
    Exception in thread "main" java.io.FileNotFound-
Exception: input.txt (Не удается найти указанный файл)
    at java.io.FileInputStream.open0(Native Method)
```

Добавим блок обработки исключения:

```
    import java.util.*;
    import java.io.*;
    public class Main {
        public static void main(String[] args) throws
    FileNotFoundException {
            try {
                FileInputStream f = new FileIn-
            putStream("input.txt");
            }
            catch (FileNotFoundException e) {
                System.out.print(e);
            }
        }
    }
    java.io.FileNotFoundException: input.txt (Не удается
найти указанный файл)
```

Или

```
    import java.util.*;
    import java.io.*;
    public class Main {
        public static void main(String[] args) throws
    FileNotFoundException {
            try {
                FileInputStream f = new FileIn-
            putStream("input.txt");
            }
            catch (FileNotFoundException e) {
                System.err.println("FileNotFound-
            Exception has been caught.");
            }
        }
    }
    FileNotFoundException has been caught.
```

5.3 Системные исключения

Существует несколько готовых системных исключений. Большинство из них являются подклассами типа `RuntimeException`, и их не нужно включать в список `throws`.

С появлением версии JDK 7 механизм обработки исключений в Java был значительно усовершенствован благодаря включению в него трех новых средств. Первое из них поддерживает *автоматическое управление ресурсами*, позволяющее автоматизировать процесс освобождения таких ресурсов, как файлы, когда они больше не нужны [7].

Второе новое средство называется *групповым перехватом*, а третье – *окончательным* или *уточненным повторным генерированием* исключений.

Групповой перехват позволяет перехватывать два или более исключения одним оператором `catch`. В приведенной ниже строке кода показывается, каким образом групповой перехват исключений `ArithmeticException` и `ArrayindexOutOfBoundsException` указывается в одном операторе `catch`.

```
    catch(ArithmeticException |
ArrayindexOutOfBoundsException e) {
        // что-то сделать с перехваченной ошибкой...
    }
```

В данном примере программы исключение `ArithmeticException` генерируется при попытке деления на ноль, а исключение `ArrayindexOutOfBoundsException` – при попытке обращения за границы массива. Оба исключения перехватываются одним оператором `catch`.

5.4 Непроверяемые исключения

В стандартном пакете `java.lang` определены некоторые классы, представляющие стандартные исключения Java. Часть из них использовалась в предыдущих примерах программ. Наиболее часто встречаются исключения из подклассов стандартного класса `RuntimeException`. А поскольку пакет `java.lang` импортируется по умолчанию во все программы на Java, то исключения, производные от класса `RuntimeException`, становятся доступными автоматически (табл. 5.1). Их даже обязательно включать в список опера-

тора `throws`. В терминологии языка Java такие исключения называют *непроверяемыми* (*unchecked*), поскольку компилятор не проверяет, обрабатываются или генерируются подобные исключения в методе.

Таблица 5.1 – Непроверяемые исключения, определенные в пакете `java.lanq`

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка, например попытка деления на ноль
`ArrayindexOutOfBounds-Exception`	Попытка обращения за границы массива
`ArrayStoreException`	Попытка ввести в массив элемент, несовместимый с ним по типу
`ClassCastException`	Недопустимое приведение типов
`EnumConstNotPresent-Exception`	Попытка использования нумерованного значения, которое не было определено ранее
`IllegalArgument-Exception`	Недопустимый параметр при вызове метода
`IllegalMonitorState-Exception`	Недопустимая операция контроля, например, ожидание разблокировки потока
`IllegalStateException`	Недопустимое состояние среды выполнения или приложения
`IllegalThreadState-Exception`	Запрашиваемая операция несовместима с текущим состоянием потока
`IndexOutOfBounds-Exception`	Недопустимое значение индекса
`NegativeArraySize-Exception`	Создание массива отрицательного размера
`NullPointerException`	Недопустимое использование пустой ссылки
`NumberFormatException`	Неверное преобразование символьной строки в число
`SecurityException`	Попытка нарушить систему защиты
`StringindexOutOfBounds`	Попытка обращения к символьной строке за ее границами
`TypeNotPresent-Exception`	Неизвестный тип
`UnsupportedOperation-Exception`	Неподдерживаемая операция

5.5 Проверяемые исключения `java.lang`

Исключения из пакета `java.lang`, которые следует непременно включать в список оператора `throws` при объявлении метода, если, конечно, в методе содержатся операторы, способные генерировать эти исключения, а их обработка не предусмотрена в теле метода. Такие исключения принято называть *проверяемыми* (*checked*) (табл. 5.2).

Таблица 5.2 – Проверяемые исключения, определенные в пакете `java.lanq`

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`CloneNotSupported-Exception`	Попытка клонирования объекта, не реализующего интерфейс `Cloneable`
`IllegalAccessException`	Доступ к классу запрещен
`InstantiationException`	Попытка создания объекта абстрактного класса или интерфейса
`InterruptedException`	Прерывание одного потока другим
`NoSuchFieldException`	Требуемое поле не существует
`NoSuchMethodException`	Требуемый метод не существует
`ReflectiveOperation-Exception`	Суперкласс исключений, связанных с рефлексией (добавлен в версии JDK 7)

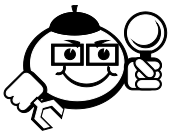
5.6 Собственные исключения

Встроенные в Java исключения позволяют обрабатывать большинство распространенных ошибок. Тем не менее в прикладных программах возможны особые ситуации, требующие наличия и обработки соответствующих исключений.

Для того чтобы создать класс собственного исключения, достаточно определить его как производный от класса `Exception`, который, в свою очередь, является производным от класса `Throwable`. В подклассах собственных исключений совсем не обязательно реализовать что-нибудь. Их присутствия в системе типов уже достаточно, чтобы пользоваться ими как исключениями [8].

Всем классам исключений, в том числе и создаваемым самостоятельно, доступны методы, определенные в классе `Throwable`.

Один или несколько этих методов можно также переопределить в своих классах исключений.



Пример

```
import java.io.*;
class ArithmeticExp extends Exception {
    private int number;
    public int getNumber() { return number; }
    public ArithmeticExp(String message, int num) {
        super(message);
        number=num;
    }
}

public class Main {
    public static int get_d(int x, int y) throws
ArithmeticExp {
        int result=1;
        if(y==0) throw new ArithmeticExp("Знаменатель
    не может быть равен 0", y);
        result=x/y;
        return result;
    }

    public static void main(String[] args) throws
IOException {
        int a=5;
        int b=0;
        try {
            int result = get_d(a,b);
            System.out.println(result);
        }
        catch(ArithmeticExp ex) {
            System.out.println(ex.getMessage());
            System.out.println(ex.getNumber());
```

```
            }
        }
    }
Знаменатель не может быть равен 0
0
```

Здесь для определения ошибки, связанной с делением на ноль, определен класс `ArithmeticExp`, который наследуется от `Exception` и содержит всю информацию о вычислении. Для генерации исключения в методе вычисления дроби выбрасывается исключение с помощью оператора `throw`: `throw new ArithmeticExp ("Знаменатель не может быть равен 0", y)`.

Контрольные вопросы по главе 5

1. Дайте определение понятию «исключение».
2. Какова иерархия исключений?
3. Можно и нужно ли обрабатывать ошибки JVM?
4. Какие существуют способы обработки исключений?
5. О чем говорит ключевое слово `throws`?
6. В чем особенность блока `finally`? Всегда ли он исполняется?
7. Может ли не быть ни одного блока `catch` при отлавливании исключений?
8. Могли бы вы придумать ситуацию, когда блок `finally` не будет выполнен?
9. Может ли один блок `catch` отлавливать несколько исключений (с одной и разных веток наследований)?
10. Что вы знаете об обрабатываемых и необрабатываемых (`catched`/`uncatched`) исключениях?
11. В чем особенность `RuntimeException`?
12. Как написать собственное («пользовательское») исключение? Какими мотивами вы будете руководствоваться при выборе типа исключения: `checked`/`unchecked`?

Заключение

Объектно-ориентированное программирование обеспечивает правильные методики проектирования, переносимость кода и его повторное использование, однако для того, чтобы все это полностью понять, необходимо изменить свое мышление [10].

Высокий уровень доступных в настоящее время средств Java определяет выбор данного языка в качестве инструмента для создания научных и коммерческих программ. Интересным и перспективным является использование Java для создания приложений для устройств с ограниченными ресурсами – мобильных телефонов и компьютеров. Это направление очень активно развивается и становится одним из самых актуальных в эволюции языка Java.

Литература

1. Купер А. Психбольница в руках пациентов / А. Купер. – СПб. : Символ-Плюс, 2005. – 336 с.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Гради Буч. – 3-е изд. – М. : ООО «ИД «Вильямс», 2008. – 720 с.
3. PYPL (PopularitY of Programming Language) [Электронный ресурс] // GitHub. – Режим доступа: http://pypl.github.io/PYPL.html (дата обращения: 02.10.2018).
4. Гослинг Дж. Язык программирования Java SE 8. Подробное описание / Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли. – 5-е изд. – М. : ООО «ИД «Вильямс», 2015. – 672 с.
5. Code Conventions for the Java Programming Language [Электронный ресурс] // Oracle. – Режим доступа: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html (дата обращения: 02.10.2018).
6. Мейер Б. Почувствуй класс / Б. Мейер ; пер. с англ. под ред. В. А. Биллига. – М. : Национальный открытый университет «ИНТУИТ» : БИНОМ. Лаборатория знаний, 2011. – 775 с.
7. Шилдт Г. Java 8: руководство для начинающих : пер. с англ. / Г. Шилдт. – 6-е изд. – М. : ООО «ИД «Вильямс», 2015. – 720 с.
8. Блинов И. Н. Java 2 : практ. руководство / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2005. – 400 с.
9. Вайсфельд М. Объектно-ориентированное мышление / М. Вайсфельд. – СПб. : Питер, 2014. – 304 с.
10. Эккель Б. Философия Java. Библиотека программиста / Б. Эккель. – 4-е изд. – СПб. : Питер, 2009. – 640 с.

Глоссарий

Абстрактный класс (*abstract class*) – класс, экземпляр которого нельзя создать.

Абстракция (*abstraction*) выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Агрегация (*aggregation*), или *включение*, – отношение между классами типа «содержит» или «состоит из».

Ассоциация (*association*) – отношение, если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера «владения» или контейнеризации.

Вложенные классы (*static nested classes*) – это классы, объявленные внутри внешнего класса как статические.

Внутренние классы (*inner classes*) – это классы, объявленные внутри внешнего класса без ключевого слова `static`.

Идентичность (*names*) – это такое свойство объекта, которое отличает его от всех других объектов.

Иерархия классов (*class-hierarchy*) представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях.

Инкапсуляция (*encapsulation*) – это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса).

Интерфейс (*interface*) – это явно указанная спецификация набора абстрактных методов, которые должны быть представлены в классе, реализующем эту спецификацию.

Исключениями или *исключительными ситуациями* (*exception*) называются ошибки, возникшие в программе во время ее работы.

Класс (*class*) представляет собой набор объектов, которые обладают общей структурой и одинаковым поведением.

Коллекциями (*collection*) называют структуры, предназначенные для хранения однотипных данных.

Композиция (*composition*) – это разновидность жесткой взаимосвязи между объектами, составляющими класс. Когда объект уничтожается, объекты, составляющие его, также уничтожаются.

Конструктор (*constructor*) – это особенный метод класса, который вызывается автоматически в момент создания объектов этого класса.

Метод (*method*) – это последовательность команд, которые вызываются по определенному имени.

Модификатор (*modifier*) – это ключевое слово языка, которое может каким-либо образом изменить смысл некоторого определения (например, класса или метода).

Наследование (*inheritance*) – это отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов.

Обработка исключительных ситуаций (*exception handling*) – механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы.

Объект (*object*) – это осязаемая сущность, которая четко проявляет свое поведение.

Объектно-ориентированное программирование (*object-oriented programming*) – технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств.

Объектной декомпозицией (*Object decomposition*) называют процесс представления предметной области задачи в виде совокупности функциональных элементов (объектов), обменивающихся в процессе выполнения программы входными воздействиями (сообщениями).

Перегрузка методов (*method overloading*) – различные реализации методов с одинаковыми именами, но разными сигнатурами в Java.

Переменная (*variables*) – именованная область памяти ЭВМ, в которой программа может хранить данные определенного типа (называемые значением переменной) и обращаться к этим данным, используя имя переменной.

Переопределение метода (*method overriding*) – изменение работы метода, унаследованного от класса-предка классом-потомком, путем описания нового метода с точно такими же именем и параметрами.

Поведение (*behaviors*) – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Полиморфизм (*polymorphism*) – положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов.

Поток данных (*stream*) представляет собой абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

Состояние (*attributes*) объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Тип данных (*data types*) – это характеристика переменной или константы, определяющая, какого рода значение хранится в отведенной для нее области памяти: числовое, символьное, логическое, объект какого-либо класса.

Экземпляр класса (*instance of a class*) – это отдельная реализация класса. Все экземпляры класса имеют одинаковые свойства, которые описаны в определении класса.

Учебное издание

Юлия Викторовна Морозова

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Учебное пособие

Корректор А. Н. Миронова
Оригинал-макет А. А. Кусаиновой

Подписано в печать 05.12.2018. Формат 60x84 1/16.
Бумага офсетная. Гарнитура Times.
Усл. печ. л. 8,13.
Тираж 150 экз. Заказ № .

Издательство «Эль Контент»
634061, г. Томск, ул. Киевская, д. 57, оф. 27

ISBN 978-5-4332-0269-6