

МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»
МИРЭА

Подлежит возврату
№

УЧЕБНОЕ ПОСОБИЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Учебное пособие для студентов, обучающихся по направлению
подготовки 09.03.04 Программная инженерия, 09.03.02
«Информационные системы и технологии», 38.03.05 «Бизнес-
информатика»

МОСКВА 2017

Автор Н. В. Зорина, Л. Б. Зорин, В.Л. Хлебникова

В авторской редакции

В учебном пособии рассматривается процесс разработки программ на основе объектной технологии, включая анализ, проектирование и разработку. Описываются основные приёмы проведения объектно-ориентированного анализа, рассматривается цикл разработки программного обеспечения (ПО). Большое внимание уделено назначению и содержанию этапов и роли анализа в процессе разработки программного обеспечения. Рассмотрены основные понятия объектно-ориентированного анализа: классы и объекты, идентификация, описание объектов и их поведения; отношениях, основные типы отношений между объектами. Уделено внимание основным средствам анализа и моделирования предметной области на языке UML. Рассмотрена объектно-ориентированная методология программирования на примере современного объектно-ориентированного языка программирования. Учебное пособие содержит теоретические сведения, примерные задания, а также примеры программ на языке Java. Учебное пособие предназначено для студентов, обучающихся по направлению «Программная инженерия» и профилю «Разработка программных продуктов и проектирование информационных систем», также может быть использовано для «Бизнес-информатика» и «Информационные системы и технологии» изучающих курс «Объектно-ориентированное программирование». Также будет полезным использование данного учебного пособия студентам других специальностей, изучающих объектно-ориентированное программирование на основе методологии объектно-ориентированного анализа и проектирования программ.

Рецензенты: к.т.н. А. В. Голышко

Глава 1. Объектно-ориентированная разработка, концепции и преимущества

1.1. Основные понятия и характеристики объектно-ориентированной разработки

Объектно-ориентированная разработка программного обеспечения ПО включает в себя:

- **объектно-ориентированный анализ (ООА)** - использование объектно-ориентированного подхода для проведения системного анализа;
- **объектно-ориентированный дизайн (ООД)** - использование объектно-ориентированного подхода для проектирования системы;
- **объектно-ориентированное программирование (ООП)** - использование объектно-ориентированного подхода при программировании;

Слово «design» переводится с английского как «проектирование», но в контексте объектно-ориентированной разработки используется слово дизайн как общепринятый термин в профессиональной среде, что кстати позволяет избежать путаницы в аббревиатурах с одноименным сокращением ООП, которым принято обозначать объектно-ориентированное программирование.

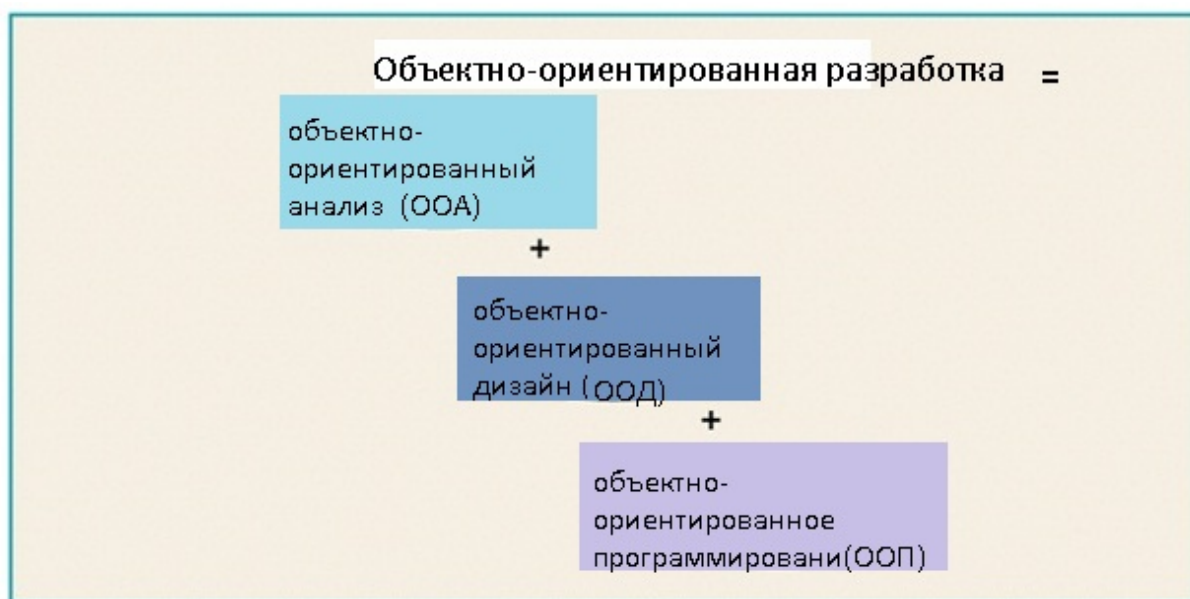


Рис.1.1. Объектно-ориентированная разработка.

Для проведения объектно-ориентированного анализа и дизайна используется унифицированный язык моделирования - UML (Unified Modeling Language).

- UML подходит для по нескольким причинам:
- использование стандартных нотаций моделирования для объектно-ориентированного анализа и дизайна;
- определён «трем китами» объектно-ориентированной парадигмы - Гради Бучем, Джеймсом Рэмбо и Иваром Якобсоном ;
- широкое использование для моделирования объектов предметной области:
 - Возможность создания графических моделей по системным требованиям и при проектировании системы (Enables creation of graphical models of the system requirements and system design).

Некоторые диаграммы UML, использующиеся при проведении объектно-ориентированного анализа и объектно-ориентированном проектировании:

- диаграммы классов (Class diagrams);
- диаграммы использования (Use Case diagrams);
- диаграммы последовательностей (Sequence diagrams);
- диаграммы состояний (Statecharts).

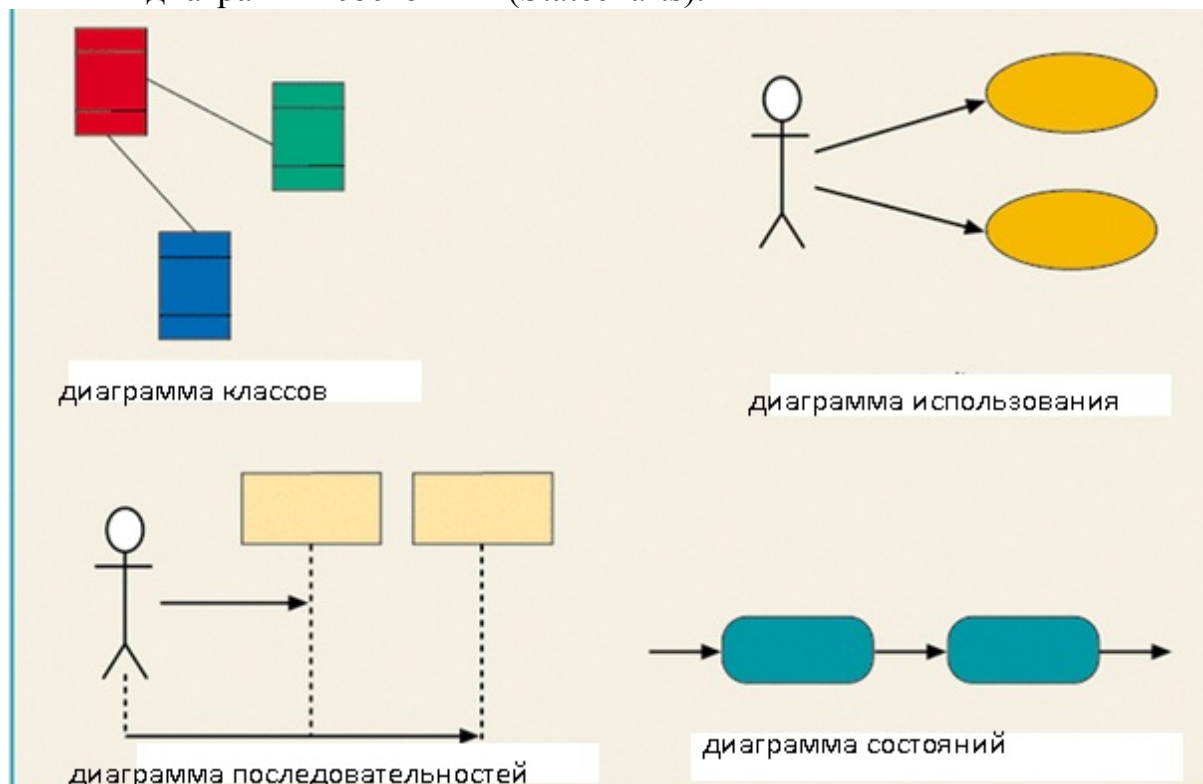


Рис.1.2. Диаграммы UML.

На рисунке 1.2. представлены основные диаграммы UML, которые используются при объектно-ориентированном анализе и объектно-ориентированном дизайне будущей программы.

Объектно-ориентированный анализ и дизайн необходим для того чтобы:

- осуществлять «прототипирование»;
- создавать работающие модели одной или нескольких частей системы для оценивания и внесения изменений на последующих итерациях;
- вести совместную разработку приложений JAD (Joint Application Development);
- организовать совместную работу постановщиков задач и системных архитекторов, чтобы можно было быстро составить требования к системе и осуществлять качественное проектирование .

Также объектно-ориентированный анализ и дизайн применяется для того, чтобы осуществлять:

- управление проектами;
- обследование системы;
- сбор данных проектирование пользовательского интерфейса тестирование;
- управление изменениями.

1.2. Роль анализа в процессе разработки программного обеспечения

Создание программного обеспечения для больших и сложных программных систем – сложная задача. Здесь необходимо отметить различные виды сложности – как сложность техническую, так и сложность программную.

Трудно недооценить роль анализа при проектировании системы. Когда мы анализируем сложную, большую систему, то в ней обнаруживается много составных структурных элементов, которые взаимодействуют между собой различными способами. Как сказал Эдзгер Дейкстра, «Способ управления сложными системами был известен ещё с древности – “divide et impera” (разделяй и властвуй)». Когда имеет место неопределённость проблемной ситуации или многокритериальность решаемой задачи, т.е. когда задача или проблема не может быть сразу представлена или решена с помощью формальных методов. Тогда на помощь приходит системный анализ. Основным методом проведения системного анализа является метод декомпозиции, т.е. расчленение сложной программной системы её удобно разделить на меньшие подсистемы, которые представляют из себя отдельные обозримые части, такие подсистемы лучше поддающиеся исследованию.

Г. Буч выделяет **два вида декомпозиции** (это связано скорее всего с двумя основными парадигмами программирования – структурной и объектно-ориентированной):

- **алгоритмическая декомпозиция.**
- **объектно-ориентированная декомпозиция**

Обе схемы решают одну и ту же задачу – проведения анализа, но делают это по-разному:

- основу первого составляет **функционально-модульный способ**. Это означает, что структура системы описывается в терминах иерархии ее функций и иерархии структур данных;
- основу второго составляет подход, что в качестве критерия декомпозиции выбирается принадлежность элементов системы к различным абстракциям конкретной предметной области.

Архитектура Программного Обеспечения (ПО) – набор ключевых правил, определяющих организацию системы :

- Совокупность структурных элементов системы и связей между ними
- Поведение элементов системы в процессе их взаимодействия
- Иерархия подсистем
- Архитектурный стиль

Каждое **архитектурное представление** – это модель системы с определенной точки зрения, в которой отражены лишь существенные аспекты и опущено все, что несущественно при данном взгляде на систему.

Модель ПО – это формализованное описание системы ПО на определенном уровне абстракции. Каждая модель описывает конкретный аспект системы, использует набор диаграмм или формальных описаний и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами. Модели служат полезным инструментом анализа проблем, обмена информацией между всеми заинтересованными сторонами, проектирования ПО. Моделирование способствует более полному усвоению требований, улучшению качества системы и повышению степени ее управляемости.

Гради Буч:

«Моделирование является центральным звеном всей деятельности по созданию **качественного** ПО. Модели строятся для того, чтобы понять и осмыслить структуру и поведение будущей системы, облегчить

управление процессом ее создания и уменьшить возможный риск, а также документировать принимаемые проектные решения».

Другими словами, если говорить об архитектуре системы, то **архитектурно значимый элемент** – это элемент, значительно влияющий на структуру системы, её функциональность, производительность, надежность, защищенность, возможность развития. Подсистемы, их интерфейсы, процессы и потоки управления являются архитектурно значимыми элементами.

Существуют различные графические модели, используемые при разработке ПО: блок-схемы, конечные автоматы, синтаксические диаграммы, семантические сети. Общее их достоинство графических моделей – наглядность.

Визуальное (графическое) моделирование позволяет описывать проблемы с помощью зримых абстракций, воспроизводящих понятия и объекты реального мира. Моделирование осуществляется при помощи языка моделирования, который включает в себя: элементы модели; нотацию (систему обозначений); руководство по использованию.

Моделирование не является целью разработки ПО. Диаграммы – это лишь наглядные изображения. Причины, побуждающие прибегать к их использованию:

Графические модели помогают получить общее представление о системе, сказать о том, какого рода абстракции существуют в системе и какие ее части нуждаются в дальнейшем уточнении.

Графические модели образуют внешнее представление системы и объясняют, что эта система будет делать, тем самым облегчают общение с заказчиком.

Графические модели облегчают изучение методов проектирования, в частности объектно-ориентированных методов.

В процессе создания ПО используются следующие виды моделей:

- модели деятельности организации (или модели бизнес-процессов):
 - модели "AS-IS" ("как есть"), отражающие существующее положение;
 - модели "AS-TO-BE" ("как должно быть"), отражающие представление о новых процессах и технологиях работы организации;
- модели проектируемого ПО, которые строятся на основе модели "AS-

ТО-BE", уточняются и детализируются до необходимого уровня.

Состав моделей, используемых в каждом конкретном проекте, и степень их детальности в общем случае зависят от следующих факторов: сложности проектируемой системы; необходимой полноты ее описания; знаний и навыков участников проекта; времени, отведенного на проектирование.

Для облегчения труда разработчиков и автоматизированного выполнения некоторых рутинных действий используются **CASE-средства (Computer Aided Software Engineering)**. В настоящее время CASE-средства обеспечивают поддержку большинства процессов жизненного цикла ПО, что позволяет говорить о CASE-технологиях разработки ПО. *CASE-технология* – это совокупность методов проектирования ПО и инструментальных средств для моделирования предметной области, анализа моделей на всех стадиях ЖЦ ПО и разработки ПО.

1.3. Жизненный цикл программы

Разработка программного обеспечения подчиняется определённому жизненному циклу. Дадим понятие определению жизненного цикла (lifecycle) программного обеспечения.

Жизненный цикл (ЖЦ) программного обеспечения это фактически упорядоченный набор видов деятельности, осуществляемый и управляемый в рамках каждого проекта по разработке программного обеспечения. **С временной точки зрения ЖЦ** – это период времени, с момента решения о необходимости создания ПО и до полного изъятия из эксплуатации. При этом начало разработки связывают с моментом возникновения потребности в разработке.

Основной документ регламентирующий жизненный цикл программного обеспечения – стандарт ISO/IEC 12207:1995 “Information Technology – Software Life Cycle Processes”(ГОСТ Р ИСО / МЭК 12207-99).

Вообще можно определить статическую структуру жизненного цикла ПО как некий набор процессов. Многие процессы протекают параллельно, отдельные следуют в некотором определённом порядке.

В течение жизненного цикла выделяют три взаимосвязанных базовых процесса: разработки, эксплуатации, сопровождения.

Разработка - это действия по непосредственному созданию программного средства.

Эксплуатация - это действия по непосредственному применению программного средства.

Сопровождение - это действия по поддержанию программного средства в работоспособном состоянии (обслуживанию), модернизации (совершенствованию) или устранению обнаруженных ошибок (модификации). Обычно разработка это процесс, связанный с эксплуатацией через сопровождение.

Классификация процессов ЖЦ:

- **основные** процессы - приобретение, поставка, разработка, эксплуатация, сопровождение;
- **вспомогательные** процессы - документирование, управление конфигурацией, обеспечение качества, верификацию, аттестацию, совместную оценку, аудит (проверку), разрешение проблем.
- **организационные** процессы - управление, создание инфраструктуры (обеспечение), усовершенствование, обучение.

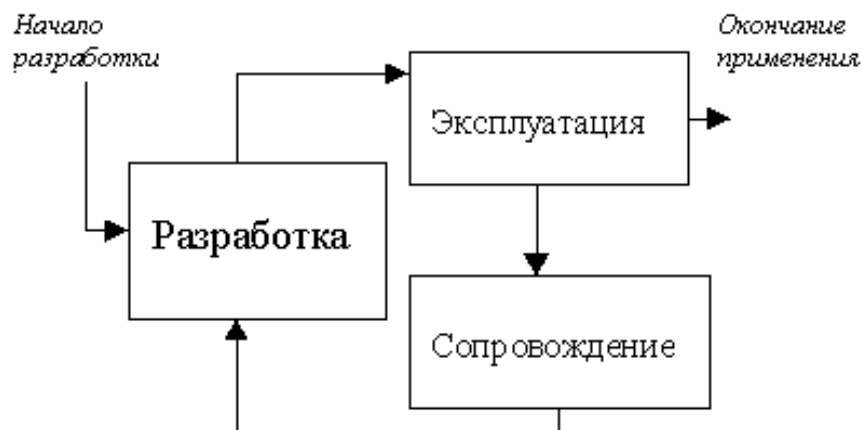


Рис 1.3. Схема модели ЖЦ

Рассмотрим кратко отдельные процессы. Задачи разработки, решаемые во время этого процесса:

- Выбор модели ЖЦ ПО и согласование с заказчиком
- Определение требований к ПО: функциональных и нефункциональных
- Определение компонентов ПО и создание документации по каждому компоненту, моделирование

- Создание исходных текстов программ, поиск и исправление ошибок в ПО и программной документации
- Сборка ПО, развертывание, оценка

Процесс эксплуатации:

- Эксплуатационное тестирование
- Эксплуатация
- поддержка пользователей

Процесс сопровождения:

- анализ проблемы запросов на модификацию ПО;
- проверка и приемка;
- перенос ПО в другую среду;
- снятие ПО с эксплуатации.

В данном случае основные действующие лица здесь будут служба сопровождения и пользователи. Задачи, решаемые на этом этапе:

- выработка плана сопровождения;
- оценка целесообразности модификации ПО;
- принятие решения в ПО;
- поиск ошибок;
- проверка целостности ПО;
- архивирование при снятии с эксплуатации;
- обучение пользователей.

Процесс документирования:

- Проектирование и разработка документации
- Выпуск документации
- Сопровождение

1.4. Основные этапы разработки программного обеспечения (ПО) и их назначение

Разработка программного обеспечения является одним из динамично развивающихся видов бизнеса, любой бизнес должен быть управляемым, чтобы приносить прибыль. Что именно же такое вкладывают в понятие программного обеспечения?

Программное обеспечение (ПО) системы включает в себя программы, документацию, данные необходимые для правильной работы программ.

Когда говорят о разработке программного обеспечения, то имеют в виду проект по разработке ПО. Дадим определение этому понятию.

Проектирование ПО – это процесс создания спецификаций (требований) ПО на основе исходных требований к нему.

Проект по созданию ПО – это совокупность спецификаций ПО, включая модели и проектную документацию, обеспечивающих создание ПО в конкретной программно-технической среде.

1.5. Современные стандарты на разработку ПО и использование типовых приёмов работы при проектировании и разработке ПО

Классификация методологий разработки приложений:

- **Международные** (*IDEF, ISO*)
- **Федеральные** (*ГОСТ*)
- **Корпоративные** (*IBM RUP, Microsoft MSF, Oracle CDM*)
- **«Мелкие и средние»/прочие:** *Agile, XP*

Основные методологии разработки приложений:

- IDEF
- ГОСТ / ISO
- IBM **RUP** (итеративный подход; быстрое прототипирование; этапы ЖЦ = потоки; роли);
- Microsoft **MSF** (синтез каскадной и спиральной моделей, процессный подход, синхростабилизация);
- Oracle **CDM** (каскадная модель, жестко детерминированные этапы ЖЦ, прототипирование)
- «Гибкие»: **Agile, XP** (итеративность, неформальные критерии «лучшие практики», рефакторинг – улучшение кода)

Модели и методы разработки ПО

Как модели, так и методы разработки ПО изучают вопросы, связанные с производством ПО. По аналогии с моделями и процессами ЖЦ ПО модели и методы разработки представляют собой общую концепцию, которой должны придерживаться разработчики, а не жёсткий стандарт, чётко регламентирующий все действия. Следовательно, организации, работающие над проектом, могут смешивать элементы разных моделей и методов, изобретая собственные процессы ЖЗ. Современные процессы разработки носят итеративный и поступательный характер. Процесс разработки состоит из нескольких итераций(повторяющиеся шаги). Результатом выполнения каждой итерации является улучшенная версия ПО, что по большому счёту означает добавление функциональности к существующей версии ПО.

Наиболее распространённые модели и методы разработки ПО:

- каскадная (водопадная) модель;
- спиральная модель;

- итерационная модель (Унифицированный процесс RUP, архитектура, управляемая моделями);
- ускоренная разработка ПО;
- аспектно-ориентированная разработка ПО.

Связи будем рассматривать между этапами в следующих популярных моделях:

1) Каскадная модель

Каскадная модель жизненного цикла («модель водопада», англ. *waterfall model*) была предложена в 1970 г. Уинстоном Ройсом. Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Этапы проекта в соответствии с каскадной моделью:

- формирование требований;
- проектирование;
- реализация;
- тестирование;
- внедрение;
- эксплуатация и сопровождение.
- снятие с эксплуатации

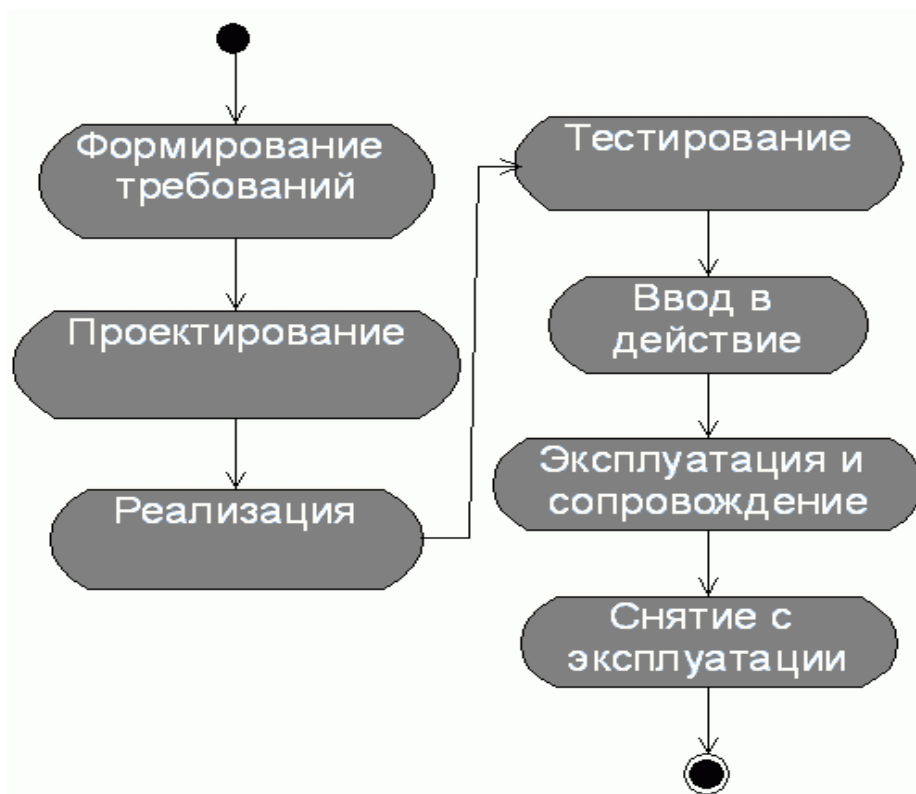


Рис 1.4. Каскадная модель.

Неизбежные возвраты на предыдущие стадии в каскадной модели.

Особенности **модели ЖЦ, основанной на формальных преобразованиях**: использование специальных нотаций для формального описания требований; кодирование и тестирование заменяется процессом преобразования формальной спецификации в исполняемую программу.

Достоинства: формальные методы гарантируют соответствие ПО спецификациям требований к ПО, т.о. вопросы надежности и безопасности решаются сами собой.

Недостатки: большие системы сложно описать формальными спецификациями; требуются специально подготовленные и высоко квалифицированные разработчики; есть зависимость от средств разработки и нотации спецификаций.

2) Спиральная модель

Является базовой для все итеративных моделей ПО. Эта модель используется довольно-таки давно, ещё с эпохи структурного программирования.

Спиральная модель (англ. *spiral model*) была разработана в середине 1980-х годов Барри Бозмом. Она основана на классическом цикле Деминга PDCA (plan-do-check-act). При использовании этой модели ПО создается в несколько итераций (витков спирали) методом прототипирования.

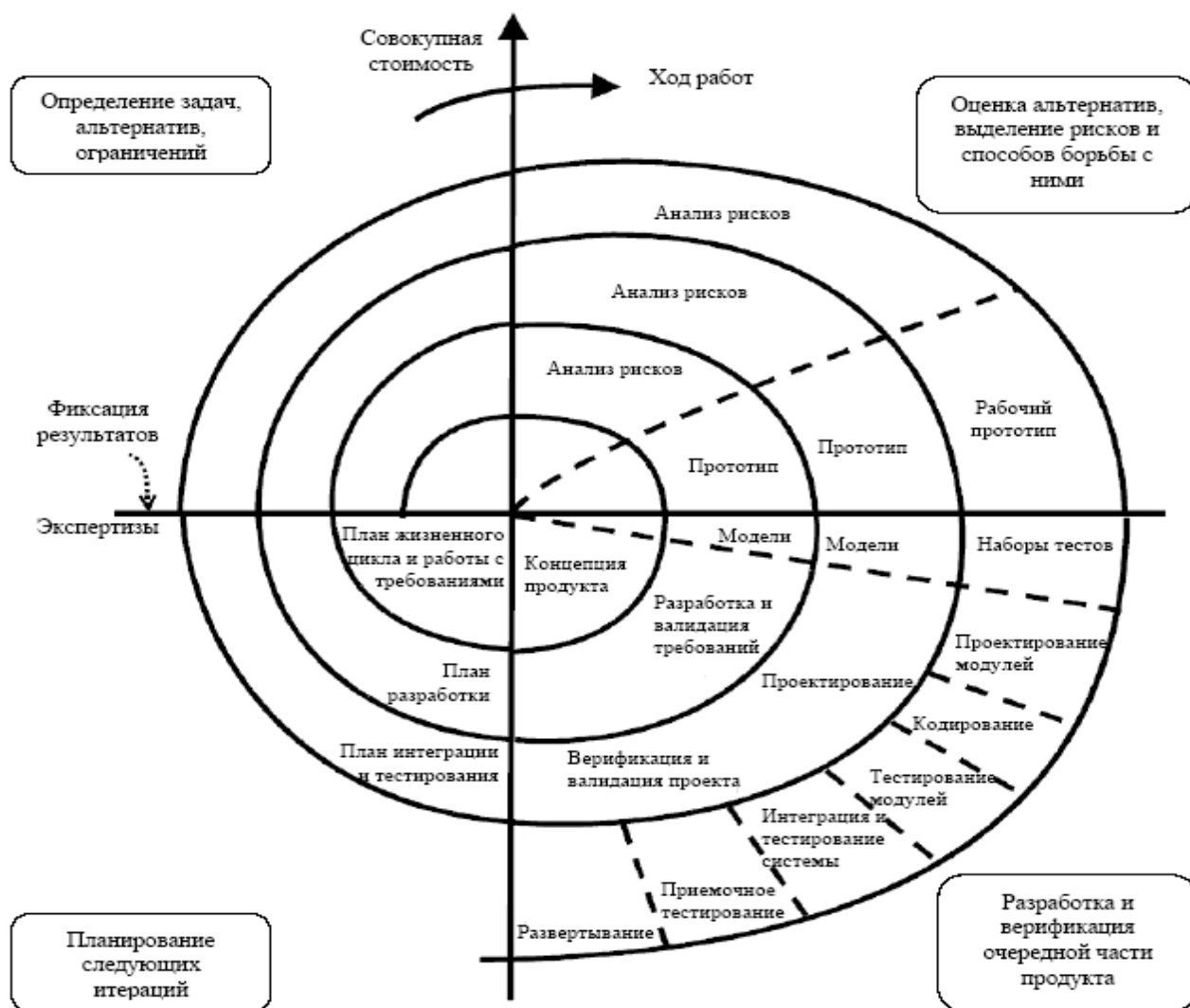


Рис 1.5. Спиральная модель.

Прототип — действующий компонент ПО, реализующий отдельные функции и внешние интерфейсы. Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации.

На каждой итерации оцениваются:

- риск превышения сроков и стоимости проекта;

- необходимость выполнения еще одной итерации;
- степень полноты и точности понимания требований к системе;
- целесообразность прекращения проекта.

Один из примеров реализации спиральной модели — ***RAD*** (англ. ***Rapid Application Development***, метод быстрой разработки приложений).

Особенности спиральной модели:

общая структура действий на каждой итерации – планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов;

решение о начале новой итерации принимается на основе результатов предыдущей;

досрочное прекращение проекта в случае обнаружения его нецелесообразности;

в конце «миникаскад», завершающийся выпуском финальной версии ПС.

3) Итерационная модель

Естественное развитие каскадной и спиральной моделей привело к их сближению и появлению современного итерационного подхода, который представляет рациональное сочетание этих моделей. Самый распространенный подход на нынешний день.

1.6. Разработка объектно-ориентированных систем, основные понятия разработки объектно-ориентированных систем

Смысл объектно-ориентированного подхода, применяемого при разработке, состоит в том, что систему можно представить как коллекцию объектов, которые работают совместно.

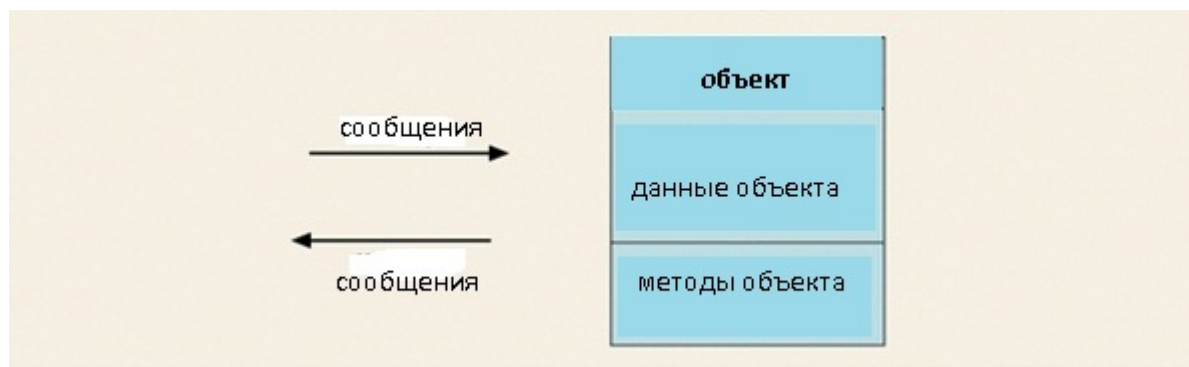


Рис.1.6. Схема объектно-ориентированной обработки данных.

Т.е. объекты выполняют некоторые согласованные действия предопределённые назначением системы:

- объекты выполняют некоторые действия, когда это от них требуется;
- каждый объект производит обработку своих собственных данных.

Из схемы, представленной на рис.1.6. видно, что объект инкапсулирует свои данные, и методы посредством которых эти данные обрабатываются. Объектно-ориентированный подход рассматривает

Суть процедурного подхода состоит в том, что система определяется как набор процедур, которые взаимодействуют через общие данные, причём, данные существуют отдельно от процедур.

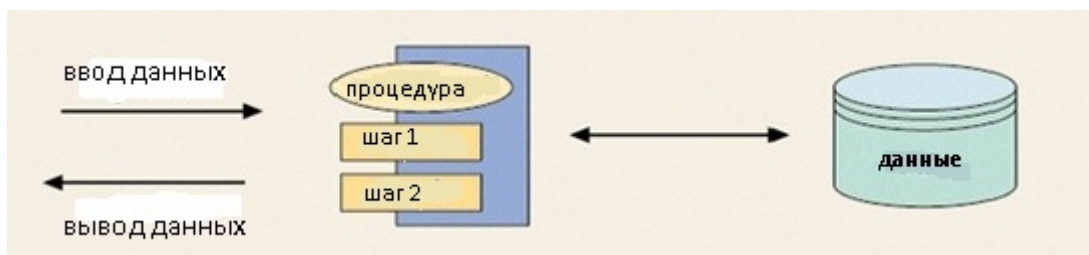


Рис. 1.7. Схема процедурной обработки данных.

Из схемы, представленной на рис.1.7. видно, что данные отделены от процедур.

Итак, мы выяснили, что концепция объектно-ориентированной разработки «стоит на трёх китах»: объектно-ориентированный анализ(ООА), объектно-ориентированное проектированиеили дизайн (ООД) и объектно-ориентированное программирование(ООП).

ООА – «полужформальная» технология (результат зависит от квалификации аналитика).

В отличие от **структурного анализа** (ранний «дообъектный» подход к проектированию ПО) при проведении объектно-ориентированного анализа фокус – на данных и действиях (они равнозначны при проектировании ПО)

Средства моделирования действий:

- DFD;
- Конечные автоматы.

ООА учитывает динамику взаимосвязей данных и действий (атрибутов и методов) при проектировании ПО .

Большинство аналитиков пользуются при проведении объектно-ориентированного анализа средствами UML:

- **моделирование прецедентов** – способов вычисления ПО различных результатов в форме диаграмм прецедентов и набора описаний сценариев
(более «от действий»)
- **моделирование классов** – представление информации в виде классов, их атрибутов и связей в форме диаграмм классов (чисто «от данных»)
- **моделирование поведения** – представление действий объектов каждого класса в форме диаграмм состояний (чисто «от действий»).

ООД:

- анализ и спецификация требований,
- извлечение существенных,
- модульная декомпозиция,
- эскизное/первичное/детальное проектирование,
- реализация, модульное тестирование)

ООП:

- программирования на объектно-ориентированных языках программирования в соответствии с основными принципами ООП(абстрагирование, инкапсуляция, наследование, полиморфизм).

1.7. Основные этапы жизненного цикла объектно-ориентированных систем

Объектно-ориентированная система проходит в своём развитии все этапы ЖЦ, особенность заключается в том, что такие приложения громоздки, имеют большой объём источников кода, развитой графический интерфейс пользователя, обширную бизнес-логику, возможно распределённую структуру и работают с БД.

Трудоемкость создания современных приложений на начальных этапах проекта, как правило, оценивается значительно ниже реально затрачиваемых усилий. Это приводит к следующим неизбежным последствиям:

- Затягиваются окончательные сроки готовности программ

- Увеличиваются незапланированные расходы
- В процессе разработки приложений изменяются функциональные требования заказчика, что еще более отдаляет момент окончания работы программистов
- Увеличение размеров программ вынуждает привлекать сверхштатных программистов, что, в свою очередь, требует дополнительных ресурсов для организации их согласованной работы, то это влечёт за собой неизбежные трудности и на всех основных этапах ЖЦ.

В процессе эксплуатации программного обеспечения современных информационных систем возникают проблемы связанные с иерархией наследования, особенно если проект живёт длительное время. В один «прекрасный» момент новичок (совсем необязательно), принятый в команду внесёт небольшие изменения, и вся иерархия классов может рухнуть. Кроме того, если в процессе эксплуатации заказчик захочет внести изменения или дополнения, а программная система изначально плохо спроектирована, то придётся переписывать большую часть классов.

В процессе работы над проектом заказчик часто может менять требования, такой вид разработки программного обеспечения стал популярным в последнее время, и называется экстремальное программирование (XP). Появились даже специальные техники по экстремальному программированию, а разработка систем в таком ключе получила название гибкой разработки (Agile development)/

Таким образом, нельзя недооценивать важность объектно-ориентированного анализа и объектно-ориентированного проектирования для разработки объектно-ориентированных систем, иначе проект наверняка выйдет за рамки бюджета. Другими словами при разработке объектно-ориентированных систем не годятся традиционные способы, такие как структурный анализ или процедурно-ориентированное проектирование, так-как объектно-ориентированное программное обеспечение оперирует такими понятиями как объекты, а не процессы.

С точки зрения ООАиД, *жизненный цикл системы SDLC* (System Development Life Cycle) можно рассматривать как:

- фреймворк управления проектом, который определяет все этапы проекта и действия)
- процесс, включающий в себя следующие этапы:
 - **Планирование** (Planning)
 - **Анализ**(Analysis)
 - **Проектирование**(Design)

- **Реализация**(Implementation)
- **Сопровождение**(Support)

Различные варианты итерационного подхода реализованы в большинстве современных технологий и методов (**RUP, MSF, XP**):

- **Rational Unified Process (RUP)** - методология разработки программного обеспечения, созданная компанией [Rational Software](#);
- **Microsoft Solutions Framework (MSF)** — это методология разработки программного обеспечения от Microsoft.
- **XP** (Extremal programming) или Ускоренная разработка.

MSF опирается на практический опыт корпорации Майкрософт и описывает управление людьми и рабочими процессами в процессе разработки решения. MSF представляет собой согласованный набор концепций, моделей и правил. В последние годы появилась новая модель разработки ПО, которая приобретает большое число поклонников. Это так называемое экстремальное программирование, которое основано на **манифесте гибкой разработки ПО**.

Термины, определяющие этапы ЖЦ проекта заменяются на новые, такие как:

- пользовательские истории (user stories) вместо анализа требований;
- приёмочные тесты(acceptance tests) вместо спецификаций программ, рефакторинг (refactoring);
- разработка через тестирование(test-driven development) и непрерывную интеграцию (continuous integration) вместо тестирования.

Ускоренная разработка заменяет этапы моделирования и реализации проекта на приёмочные тесты. Разработка программного кода производится парами программистов, т.е. двое людей в прямом смысле слова пишут программу на одном компьютере (“прямо как Ильф и Петров”). Такое программирование позволяет сразу же обмениваться идеями, отслеживать логику, а кроме того уже не один человек обладает правами собственности на код и понимает его. Ускоренная разработка всегда основывается на рефакторинге.

Рефакторинг кода это улучшение кода программы путём реструктуризации его без существенных изменений в бизнес логике.

Манифест гибкой разработки ПО:

- Индивидуумы и взаимодействия ценятся выше процессов и инструментов.
- Работающее ПО ценится выше всеобъемлющей документации.

- Сотрудничество с заказчиками ценится выше согласования условий договора.
- Реагирование на изменения ценится выше соблюдения плана.

1.8. Обзор объектно-ориентированного развития системы

Приложения могут образовывать большие программные системы со сложными схемами взаимодействия. Разработка объектно-ориентированных приложений – многоаспектный, замкнутый, итерационный процесс. В ходе разработки приложение последовательно проходит целый ряд стадий своего развития – ЖЦ. Методология (в идеале) включает модели, методы и средства.

Поскольку ЖЦ – непрерывный, замкнутый, итерационный **процесс**, то соответственно в своём развитии система проходит несколько итераций

Рассмотрим *стадии* ЖЦ ПО, *не зависящие от методологий*:

- Анализ требований
- Подготовка спецификаций
- Проектирование (эскизное, детальное, рабочее)
- Реализация
- Тестирование
- Интеграция (сборка – ср. assembly в MS .NET)
- Сопровождение
- Снятие с эксплуатации

Все стадии, кроме последней, обязательно включают в себя процесс создания документации на соответствующем этапе или документирования.

Анализ требований:

- встреча разработчика и заказчика;
- достижение общего понимания задачи, для решения которой будет разработано ПО ;
- выявление и обсуждение требований и ограничений заказчика к ПО (посредством собеседования).

Результат, получаемый на этом этапе - формализованное описание требований (ТЗ) или краткий список требований (requirements checklist)

Подготовка проектных спецификаций готовится разработчиком на основе описания требований и содержит:

- описание всей функциональности проекта;
- выбранную методологию/модель разработки ПО (следует определить как можно раньше!);

- оценку сроков проекта;
- оценку стоимости проекта.

Детальное проектирование выполняется разработчиком на основе проектных спецификаций и содержит:

- Описания всех программных модулей
- Описание программной архитектуры: интеграция компонент проекта (модулей и интерфейсов- мы рассматриваем ООП) с программной средой заказчика (например—Java / NetBeans IDE / pluginUML for NetBeans)

Реализация содержит отдельные программные модули, например на Java, готовится программистами разработчика или субподрядчика и производится на основе:

- документов детального проектирования,
- общего плана проекта (глобальные ограничения сроков и стоимости, важнейшие функциональные параметры и ограничения)

В результате каждый программный модуль реализован и протестирован (**пока по отдельности!**)

В процессе интеграции:

- разработчик производит сборку модулей в общую архитектурную схему
- Разработчик и заказчик проводят тестирование

В результате этого этапа ПО разворачивается на оборудовании у заказчика, и если все приемочные тесты (проведенные заказчиком на реальном АО и ПО согласно функциональным требованиям) будут успешными, то ПО передается заказчику.

В процессе эксплуатации могут вскрыться скрытые ошибки или просчёты в требованиях, соответственно разработчик обязан удовлетворить требования заказчика и в соответствии с условиями договора сопровождать ПО, в том числе и осуществлять апгрейд и апдейт существующей системы.

1.9. Объектно-ориентированное программирование

Объектно-ориентированная методология программирования возникла довольно-таки давно, а именно в 1960-е в Норвегии появился язык программирования Simula. Язык Simula-67 был первым языком программирования, в котором нашли своё отражение идеи построения программ на основе объектов и данных. Объекты в этом языке могли обрабатывать значения своих собственных данных и могли

взаимодействовать независимо друг от друга. В 1970-х в недрах компании Херох PARC появился новый язык программирования, фактически – первый объектно-ориентированный язык Smalltalk. В настоящее время объектно-ориентированная методология программирования стала общепризнанной парадигмой. Существует довольно-таки много объектно-ориентированных языков программирования.

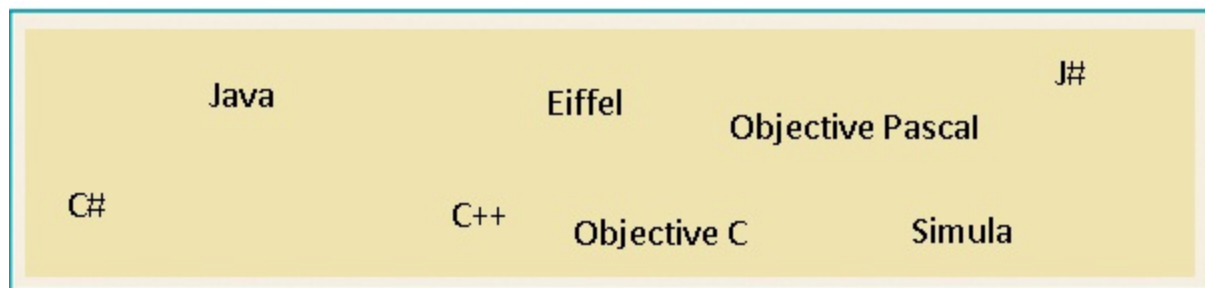


Рис.1.8. Объектно-ориентированные языки программирования.

Программа, написанная на объектном языке, представляет собой совокупность объектов, каждый из которых принадлежит определённому типу или классу объектов, причём каждый объект имеет набор методов, которые представляют интерфейс для взаимодействия объектов друг с другом в виде послышки сообщений.

На рис.1.8. представлены некоторые объектно-ориентированные языки программирования. В последнее время очень популярным языком у разработчиков стал объектно-ориентированный язык программирования Java. Этот язык программирования был разработан компанией Sun Microsystems и представляет собой “чисто” объектно-ориентированный язык. Синтаксис языка Java очень похож на язык C++. Язык является кроссплатформенным, разрабатывался под девизом “сделано однажды, работает всегда”, отлично подходит для разработки Web-ориентированных приложений.

Объектная модель является концептуальной базой объектно-ориентированного подхода (ООП).

Проблемы, стимулировавшие развитие ООП:

- необходимость повышения производительности разработки за счет многократного (повторного) использования ПО;
- необходимость упрощения сопровождения и модификации разработанных систем (локализация вносимых изменений);
- облегчение проектирования систем (за счет сокращения семантического разрыва между структурой решаемых задач и

структурой ПО).

Забегая вперед, скажем, какие решения данных проблем дает ООП. При ООП изменения локализуются внутри класса (компоненты или пакета, если изменяются несколько классов). Семантический разрыв ликвидируется, поскольку сущности предметной области представляются объектами, следовательно, разработчик и заказчик (пользователь) оперируют схожими понятиями. Повторное использование достигается за счет построения систем с использованием библиотек готовых компонент – модулей (заимствовано из структурного или функционального подхода).

Краткая история ООП:

- 1967: язык Simula – первый среди объектно-ориентированных;
- 1970-е: Smalltalk – получил довольно широкое распространение;
- 1980-е: Теоретические основы, C++, Objective-C;
- 1990-е: Методы ООА и OOD (Booch, OMT), появился язык Java;
- 1997: Принят стандарт OMG UML 1.1.

В основе объектно-ориентированного подхода лежит объектная декомпозиция, при этом статическая структура ПО описывается в терминах объектов и связей между ними, а динамический аспект ПО описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Объектная модель является естественным способом представления реального мира. Она является концептуальной основой ООП.

Основными принципами ее построения являются:

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Дополнительные принципы:

- типизация;
- параллелизм;
- устойчивость (persistence).

Абстрагирование – это выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других видов объектов, важных с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей. Абстракцией является любая модель, включающая наиболее важные, существенные или отличительные характеристики некоторого объекта, и игнорирующая

менее важные или незначительные детали. Абстрагирование позволяет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстракция зависит от предметной области и точки зрения – то, что важно в одном контексте, может быть неважно в другом. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. **Основные абстракции** предметной области - **объекты и классы**.

Модульность – это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (частей). Модульность снижает сложность системы, позволяя выполнять независимую разработку ее отдельных частей.

Иерархия – ранжированная или упорядоченная система абстракций, расположение их по уровням в виде древовидной структуры. Элементы, находящиеся на одном уровне иерархии, должны также находиться на одном уровне абстракции. Основными видами иерархических структур сложных систем являются структура классов и структура объектов. Иерархия классов строится по наследованию, а иерархия объектов – по агрегации.

Тип – точная характеристика некоторой совокупности однородных объектов, включающая структуру и поведение.

Типизация – способ защититься от использования объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием.

При строгой типизации (например, в языке Оберон) запрещается использование объектов неверного типа, требуется явное преобразование к нужному типу. При менее строгой типизации такого рода запреты ослаблены. В частности, допускается полиморфизм – многозначность имен. Одно из проявлений полиморфизма, использование объект подтипа (наследника) в роли объекта супертипа (предка).

Параллелизм – наличие в системе нескольких потоков управления одновременно. Объект может быть активен, т. е. может породить отдельный поток управления. Различные объекты могут быть активны одновременно.

Устойчивость – способность объекта сохранять свое существование во времени и/или пространстве (адресном, в частности при перемещении между узлами вычислительной системы). В частности, устойчивость объектов может быть обеспечена за счет их хранения в базе данных.

Переходим к основным понятиям объектно-ориентированного подхода (*элементам объектной модели*). К ним относятся: **объект; класс; атрибут; операция; полиморфизм; наследование; компонент; пакет; подсистема; связь.**

1.10. Основные понятия объектно-ориентированной концепции

Как уже упоминалось выше, реальный мир в объектно-ориентированной системе представлен объектами, которые состоят друг с другом в определённых отношениях.

В объектно-ориентированном мире можно мыслить следующими категориями: **объекты, атрибуты и методы.**

Объект характеризуется:

- атрибутами (характеристики объекта, которые принимают некоторые значения);
- поведением (за это отвечают методы, которые описывают, что объект умеет делать).

В качестве примеров можно привести объекты графического интерфейса пользователя (например для работы с Windows) и объекты, относящиеся к некоторой предметной области.

Таблица 1.1. Атрибуты и методы GUI объектов.

Название Объекта GUI	Атрибуты объекта	Методы
Button	Форма, размер, цвет, положение, название	Клик, включить, выключить, показать, спрятать
Label	Форма, размер, цвет, положение, текст	Установить текст, получить текст, спрятать, показать
Form	Ширина, высота, стиль оформления	Изменение размера, минимизация, максимизация

	рамки, цвет фона	размера, сделать видимым, сделать невидимым
--	------------------	---

Объекты взаимодействуют друг с другом и обмениваются сообщениями.

Сообщения - это средства с помощью которых объекты взаимодействуют.

Например:

- пользователь инициирует взаимодействие с системой путём сообщений, посылаемых объектам GUI;
- GUI объекты взаимодействуют с объектами, описывающими предметную область через сообщения;
- объекты предметной области взаимодействуют друг с другом через сообщения;
- объектам GUI реагируют на действия пользователя путем посылки сообщений.

На рис 1.9.показана общая схема обработки заказов, где объекты взаимодействуют путём посылки сообщений.

Инкапсуляция и сокрытие информации

Инкапсуляция означает, что объекты комбинируют атрибуты и методы для работы с ними в одном модулеи (Objects have attributes and methods combined into one unit)

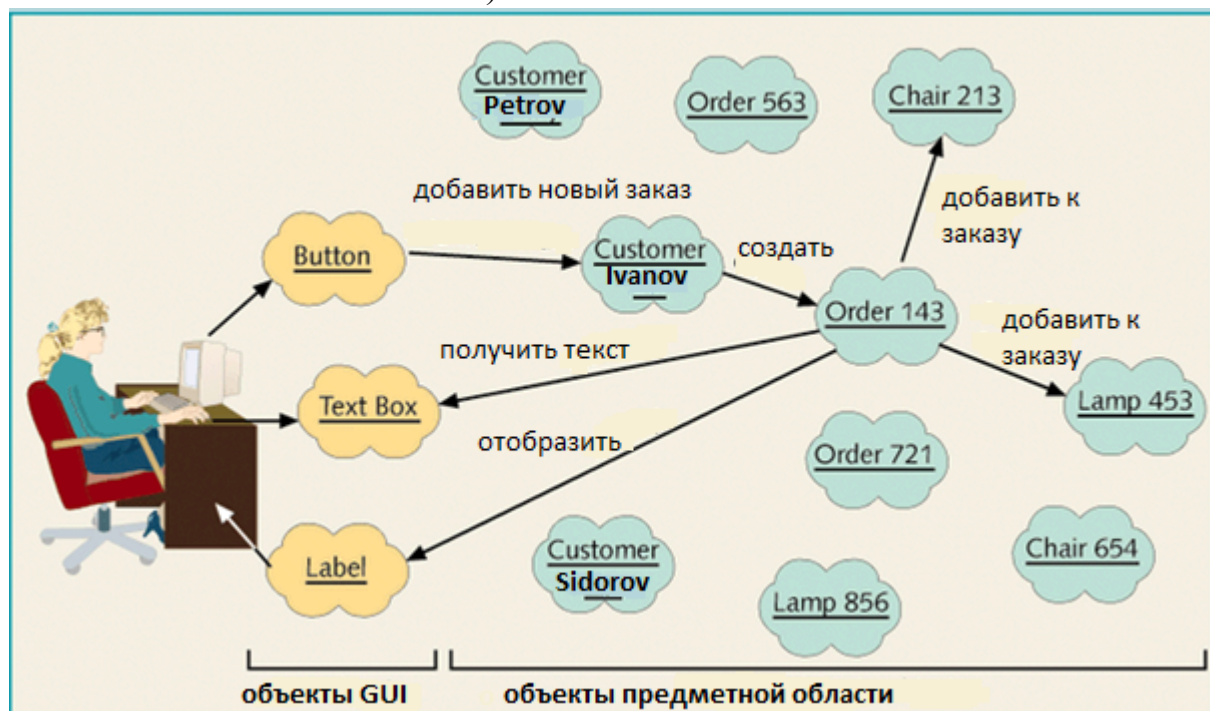


Рис.1.9. Система обработки заказов

Соккрытие информации означает соккрытие внутренней структуры объекта (реализация), защищая объект извне от несанкционированного доступа, доступ к объекту можно получить только через определённый интерфейс.

Инкапсуляция и соккрытие информации гарантируют:

- Идентификацию объекта
 - Уникальная ссылка для каждого объекта ;
- Целостность объекта
 - Определяет возможность использования объекта даже через определённый промежуток времени.

1.10.1. Объект, класс, экземпляр, атрибуты, методы и инкапсуляция

Что понимается под объектом в реальном мире?

Объект – осязаемая сущность (tangible entity) – предмет или явление (процесс), имеющие четко выраженные границы, индивидуальность и поведение. Любой объект обладает состоянием, поведением и индивидуальностью.

Состояние объекта определяется значениями его свойств (атрибутов) и связями с другими объектами, оно может меняться со временем.

Поведение объекта определяет действия объекта и его реакцию на запросы от других объектов. Поведение представляется с помощью набора сообщений, воспринимаемых объектом (операций, которые может выполнять объект).

Индивидуальность объекта – это свойства объекта, отличающие его от всех других объектов.

Структура и поведение схожих объектов определяют общий для них класс.

Класс – это множество объектов, связанных общностью свойств, поведения, связей и семантики.

Любой объект является **экземпляром класса**. Определение классов и объектов – одна из самых сложных задач объектно-ориентированного проектирования.

Атрибут – поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства. Атрибуты могут быть скрыты от других классов, это определяет видимость атрибута: *public* (общий, открытый); *private* (закрытый, секретный); *protected* (защищенный).

Требуемое **поведение системы** реализуется через взаимодействие объектов. **Взаимодействие объектов** обеспечивается механизмом пересылки сообщений. Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется операцией или посылкой сообщения. Сообщение может быть послано только вдоль **соединения между объектами**. В терминах программирования **соединение** между объектами существует, если **один объект имеет ссылку на другой**.

Операция – это реализация услуги, которую можно запросить у любого объекта данного класса. Операции реализуют связанное с классом поведение, его обязанности.

Описание операции включает четыре части:

- имя;
- список параметров;
- тип возвращаемого значения;
- видимость.

Результат операции зависит от текущего состояния объекта. Виды операций:

- операции реализации (implementor operations) – реализуют требуемую функциональность;
- операции управления (manager operations) управляют созданием и уничтожением объектов (конструкторы и деструкторы);
- операции доступа (access operations) – так называемые, get-теры, set-теры – дают доступ к закрытым атрибутам.

Вспомогательные операции (helper operations) – “непубличные операции” (с ограничением доступа), служат для реализации операций других видов.

Объект может быть абстракцией некоторой сущности предметной области (объект реального мира) или программной системы (архитектурный объект).

Понятие полиморфизма может быть интерпретировано, как способность объекта принадлежать более чем одному типу.

Полиморфизм – способность скрывать множество различных реализаций под единственным общим именем или интерфейсом.

Интерфейс – это совокупность операций, определяющих набор услуг класса или компонента.

Интерфейс не определяет внутреннюю структуру, все его операции открыты. Пример, одна и та же операция *рассчитатьЗарплату* может иметь три различные реализации в трех различных классах: СлужащийСПочасовойОплатой, СлужащийНаОкладе, ВременныйСлужащий.

Компонент – это относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры.

Компонент представляет собой физическую реализацию проектной абстракции и может быть: компонентом исходного кода (src-файл); компонентом времени выполнения (dll, ActiveX и т. п.); исполняемый компонентом (exe-файл). Компонент обеспечивает физическую реализацию набора интерфейсов.

Компонентная разработка (component-based development) представляет собой создание программных систем, состоящих из компонентов (не путать с объектно-ориентированным программированием ООП).

Объектно-ориентированное программирование – способ создания программных компонентов, базирующихся на объектах, а компонентная разработка – технология, позволяющая объединять объектные компоненты в систему.

Пакет – это общий механизм для организации элементов в группы. Это элемент модели, который может включать другие элементы. Каждый элемент модели может входить только в один пакет.

Пакет является:

- средством организации модели в процессе разработки, повышения ее управляемости и читаемости;
- единицей управления конфигурацией.

Подсистема – это комбинация пакета (может включать другие элементы модели) и класса (обладает поведением). Подсистема реализует один или более интерфейсов, определяющих ее поведение. Она используется для представления компонента в процессе проектирования.

Зависимости – связи между двумя элементами модели, при которых изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе. Например, пакет, который импортирует классы другого пакета, является зависимым от него. Зависимость изображается как пунктирная стрелка с обычным наконечником.

Зависимость между классами возникает в следующих случаях:

- в сигнатуре операции одного класса есть аргумент – объект другого класса;
- в методе одного класса есть локальный объект другого класса;
- результатом операции одного класса является экземпляр другого класса.

Реализация – связь между контрактом (интерфейсом, вариантом использования) и его исполнением (классом, подсистемой, компонентой и т. п.). Изображается пунктирной стрелкой с треугольным наконечником, исходящая из исполнения и указывающая на контракт.

Полиморфизм – это положение теории типов, согласно которому имена (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. Один интерфейс – много методов.

1.10.2. Взаимодействие объектов через методы и ассоциативные отношения

Объекты в понятиях объектно-ориентированной концепции могут взаимодействовать только через методы, которые представляют интерфейс для взаимодействия или могут наследовать методы и интерфейсы через ассоциативные отношения.

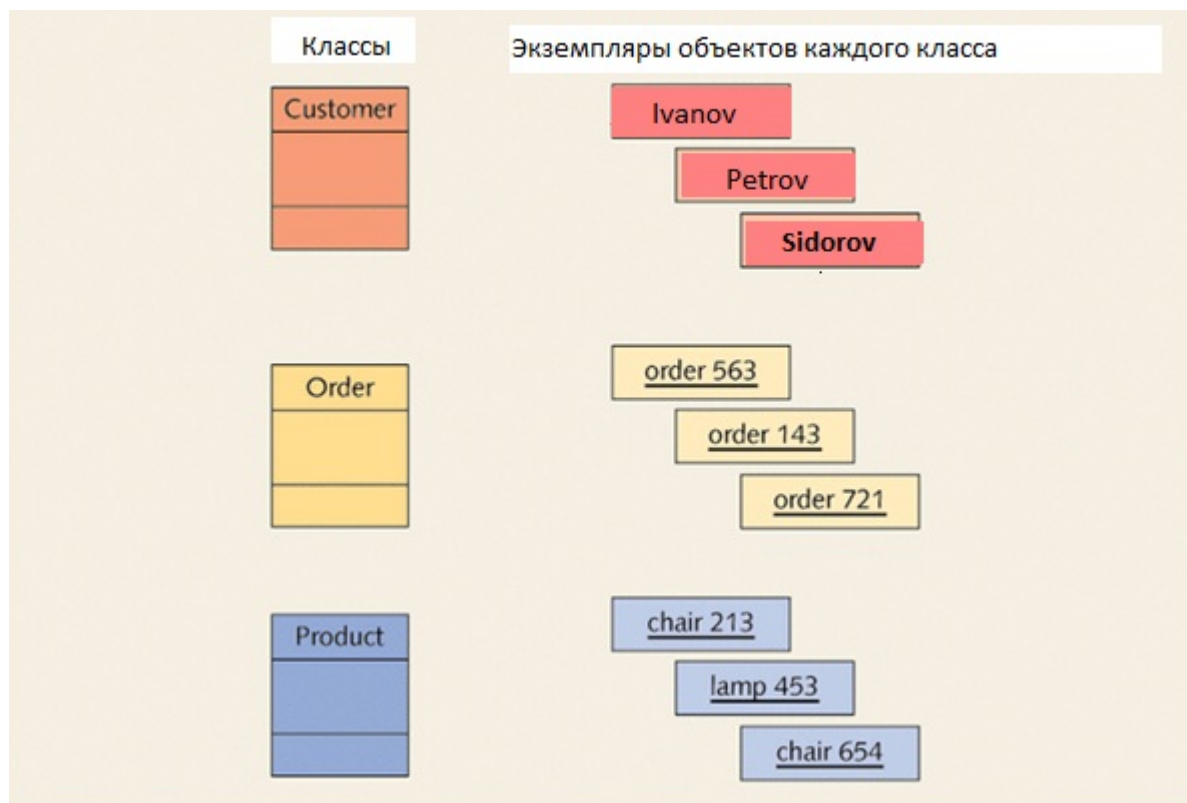


Рис.1.10. Классы и экземпляры объектов.

Сущности ООП:

- **классы**
 - Определяют каким образом все объекты определённого класса будут представлены;
- **экземпляры классов**
 - объекты определённого класса;
- **ассоциации**

То есть экземпляр объекта принадлежит определённому типу объектов типа класс. Тип класса один, а объектов определённого типа может быть много, их называют экземплярами объектов, типа класс.

Ассоциативные отношения или ассоциации означают:

- каждый объект связан с другими объектами следующими отношениями или связями
 - один к одному
 - один ко многим
- существует множество таких ассоциаций
 - для задания количества ассоциаций или связей используется в терминология UML

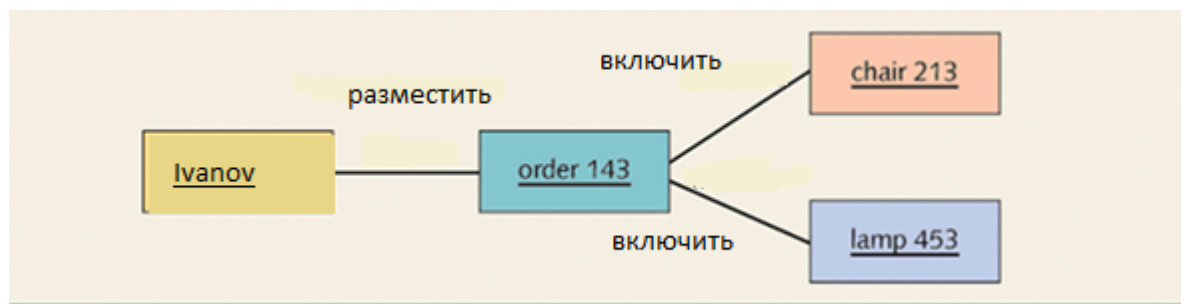


Рис.1.11. Ассоциативные связи объектов с другими объектами.

То есть, ассоциации это связи между классами или иначе, ассоциативные связи между классами. Всего есть два вида связи между классами: **ассоциации** и **подтипы**.

Допустим, у нас есть два класса – Клиент и Заказ. Нам нужно отобразить эти два класса на диаграмме и показать, как они между собой связаны. Диаграмма будет выглядеть как на рис.1.12.

С двумя блоками мы уже знакомы, а вот линия, которая их соединяет – это и есть ассоциация. На каждом конце линии на рис.1.12. расположены знаки: 1 и * – они определяют кратность конца ассоциации. Другими словами, на диаграмме показано, что у 1-го Клиента может быть много Заказов.

Кроме таких кратностей бывают еще и другие:

- 0..1 – необязательная кратность (одна или вообще связей нет)
- 0..* – ни одной или любое кол-во связей
- 1..* – одна или любое кол-во связей (0 быть не может)

Это наиболее часто используемые кратности отношений.

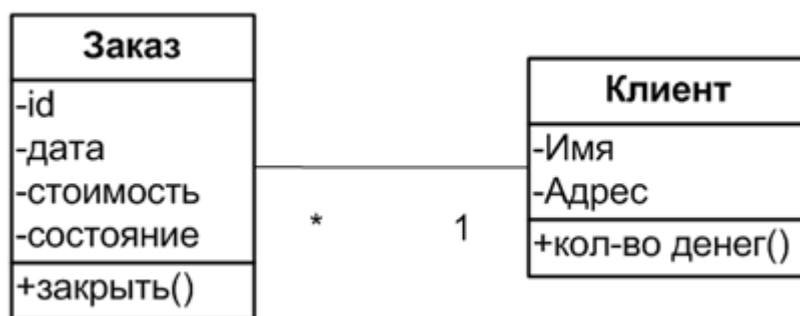


Рис.1.12. Диаграмма ассоциативных связей между объектами.

1.10.3. Концепция наследования применительно к классам объектов.

Наследование – это построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций). Изображается как стрелка с

треугольным наконечником, исходящая из наследника и указывающая на родителя.

Смысл наследования в том, что:

- один класс объектов заимствует необходимые характеристики из любого другого класса, расширяя и дополняя их
- родительский или базовый класс -> производный или дочерний класс (*superclass* → *subclass*)

Таким образом получаются обобщённые/специализированные иерархии классов. Т.е. схема, по которой происходит расширение базовых классов в производные классы называется иерархией наследования. Говорят, что результатом иерархии наследования происходит расширение класса в более специфические подклассы.

На рис.1.13. показано иерархия наследования, которая образуется из расширения родительского суперкласса некоторыми характеристиками и получения дочерних подклассов. В этой иерархии есть один суперкласс Person, от которого порождаются два производных подкласса Customer и Sales Manager, которые наследуют атрибуты (полями данных в терминах ООП) и методы этого класса. Суперкласс Person описывает обобщённые характеристики сотрудника. Подкласс Sales Manager, полученный с помощью наследования, это новый тип класса, который описывает специфицированный тип сотрудника, наделённый некоторыми новыми характеристиками и содержащий обобщённые характеристики, унаследованные им от суперкласса.

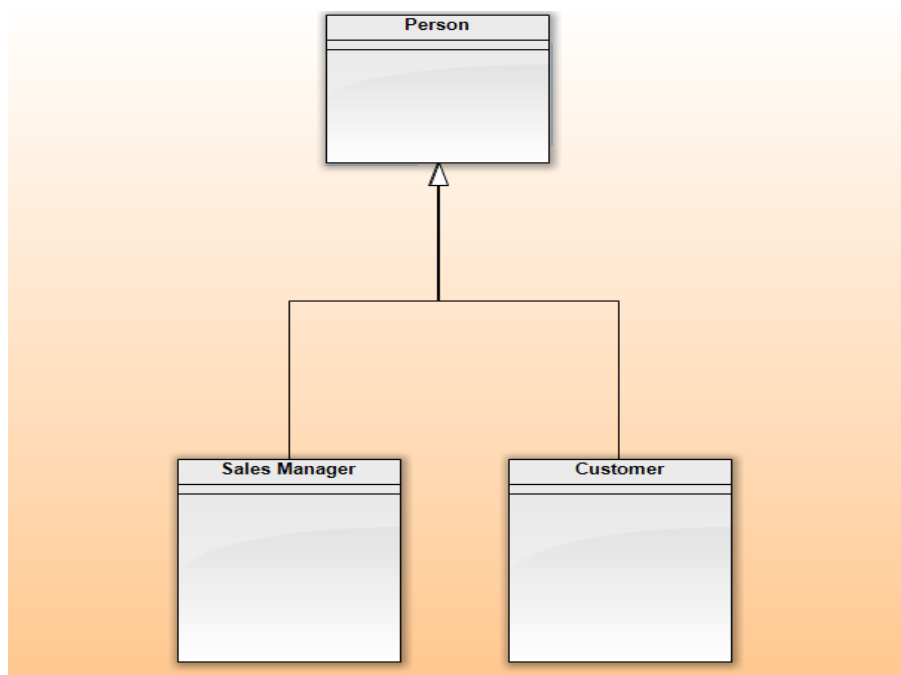


Рис.1.13. Иерархия наследования.

Тоже можно сказать и про подкласс Customer.

Смысл полиморфизма в том:

- Много форм
- Различные объекты могут отвечать своим собственным способом на некоторые сообщения

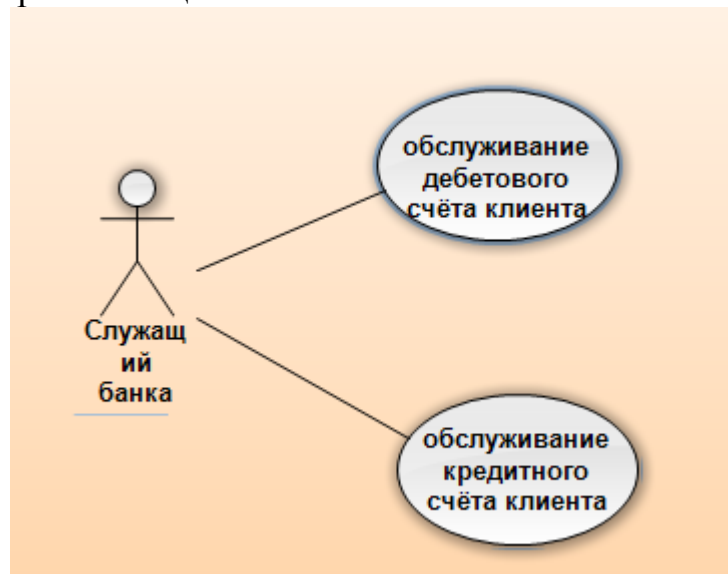


Рис.1.14. Диаграмма использования.

На рис.1.14 представлена диаграмма использования, из которой видно что полиморфизм может быть очень полезным, например при работе объекта СлужащийБанка с объектом СчетВБанке.

На верхнем уровне иерархии отображаются - общие атрибуты, операции и/или отношения. В объектной модели наследование может быть множественным (в разных языках программирования реализуется по-разному, например в Java множественное наследование возможно только на уровне интерфейсов). На связи могут накладываться ограничения. Например, если необходимо, множественное наследование в некоторой иерархии классов может быть запрещено (над связью указывается ключевое слово: **{disjoint}**).

В разных языках программирования может быть разная поддержка наследования. В языке C++ разрешено множественное наследование от нескольких классов. В языке Java может быть только одиночное наследование для классов и множественное для интерфейсов, при этом классы могут реализовать несколько интерфейсов.

1.10.4. Преимущества использования объектно-ориентированной разработки.

Основные преимущества объектно-ориентированной разработки:

- объекты являются естественными для восприятия человеком
 - естественность основывается на том, что большинство людей думают об окружающем мире с использованием терминологии объектов
 - естественно определять в виде классов объекты реального мира
- классы объектов являются повторно используемыми
 - классы объектов, которые однажды были разработаны, могут многократно использоваться при решении похожих задач
 - во время проведения анализа, проектирования, и программирования
 - ничего не нужно ничего знать об исходных кодах для повторно используемых классов (то есть нам не нужны сведения о реализации), просто нужно знать как устроен интерфейс

Глава 2. Основы программирования на Java

2.1. Введение в программирование на Java

Язык программирования Java это новый, чисто объектно-ориентированный язык, достаточно лёгкий в изучении и позволяющий создавать объектно-ориентированные программы. Ещё с Java связана тема кофе (это и само название продуктов, использование логотипа и т.п.). Кроме того в Java есть сборщик мусора, и в отличие от программистов на C++ программистам на Java не нужно искать утечки памяти в своих программах. Также не секрет, что язык Java широко используется для написания интернет-приложений. Необходимо также отметить, Java это компилируемый и интерпретированный язык одновременно. Создавая программное обеспечение на каком-либо языке программирования, разработчики не обходят вниманием вопрос: под какой операционной системой и на какой аппаратной платформе будет выполняться программа. С учётом этих вопросов используется соответствующая среда разработки создавая приложения на языке Java можно не задумываться над тем какое операционное окружение будет использоваться. Java свой собственный набор машинно-независимых библиотек, которые называются

пакетами(packages). Таким образом, можно сказать, что приложения написанные на языке программирования Java, обладают переносимостью или портируемостью. Всё дело в том, что компилятор Java генерирует некий промежуточный код, или байт-код (bytecodes), а виртуальная машина Java (Java Virtual Machine-JVM), которая устанавливается на компьютер, где будет выполняться приложение интерпретирует этот байт код в выполнение программы. Поскольку ядро виртуальной машины Java имеет реализацию практически для любого типа компьютеров, то приложение фактически является кроссплатформенным.

История создания и развития языка JAVA

Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ для Java, и их число все время возрастает. Некоторые из них являются просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор. Эти интегрированные среды требуют предварительной установки JDK. Другие содержат JDK в себе или имеют собственный компилятор, например, Java Workshop фирмы SUN Microsystems, JBuilder фирмы Inprise, Visual Age for Java фирмы IBM и множество других программных продуктов. Надо заметить, что перечисленные продукты написаны полностью на Java. Самые распространенные у разработчиков Java программ Eclipse и IntelliJ IDEA, есть ещё бесплатно распространяемая NetBeans.

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

Приложения на языке Java выполняются в специальной универсальной среде Virtual Machine, JVM отдельно для каждой платформы

Java это ***компилируемый и интерпретированный язык одновременно.***

Программа на языке Java – это обычный текст файл с расширением java. Файлы подаются на вход Java компилятор, который переводит их в специальный Java код. Результат работы компилятора сохраняется в одинаковых файлах с расширением (точка). class. Java машина начинает их

исполнять или интерпретировать. Java – *чисто объектно-ориентированный* язык со строгой типизацией, в нём существует только 8 типов данных. В Java отсутствуют деструкторы, есть встроенная сбойка мусора (Garbage collector).

2.2. Лексика языка Java

Технология Java, как платформа, изначально спроектированная для Глобальной сети Internet, должна быть многоязыковой, а значит, обычный набор символов ASCII, включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и т.д.), недостаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Пробелы.

Пробелами в данном случае называют все символы, разбивающие текст программы на лексемы. Это как сам символ пробела (space, \u0020, десятичный код 32), так и знаки табуляции и перевода строки. Они используются для разделения лексем, а также для оформления кода, чтобы его было легче читать. Пример, приведённый ниже по тексту, отформатирован, в соответствии с конвенцией кода т.е. код выровнен при помощи пробелов, табуляций и символов окончания строки. Такой код хорошо читается, приятен для просмотра.

```
if (x > 0)
    return 0;
return -1;
```

А вот этот пример показывает, как не следует поступать.

```
if(x>0)return 0;return -1;
```

Одним из пунктов «культуры» программирования, является форматирование кода. Существуют множество приемов форматирования исходного кода, которые являются частью конвенций. Каждый выбирает свой, понравившийся способ и, как правило, придерживается его всю свою сознательную, трудовую деятельность. О конвенциях подробнее будет рассказано чуть позже.

Как бы ни был записан исходный код программы, форматированный или в кучу, компилятор создаст один и тот же байт код. Ему не интересны «пробельные» символы. Ему интересны лексемы, составные части языка. Лексемами являются: идентификаторы, ключевые слова, литералы.

Идентификаторы.

Это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные. Идентификаторы

можно записывать символами Unicode, то есть на любом удобном языке. Длина имени не ограничена.

`Internet, Интернет, _Internet, iNtErNeT, INTERENET, internet,`
`ЭтоСуперДлинныйИНикомуНеНужныйИдентификатор_КоторыйНичегоНеОбозначаетИНигдеНеПрименяется`

Для составления идентификатора нужна только фантазия программиста и некоторые правила: идентификатор должен начинаться только с буквы или нижнего подчеркивания, в последующем могут присутствовать и цифры; идентификатор не должен быть ключевым словом; идентификатор не должен содержать специальные символы или пробелы. В большинстве случаев идентификатор составляется по ранее выбранной конвенции или по заданным правилам разработки.

Литералы.

Они позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов. Всего в Java определено 6 видов литералов:

- ***Целочисленный***
- ***Дробный***
- ***Булевский***
- ***Символьный***
- ***Строковый***
- ***Null литерал***

Целочисленные литералы позволяют задавать целочисленные значения в десятичном, восьмеричном и шестнадцатеричном виде. Десятичный формат традиционен и ничем не отличается от правил, принятых в других языках. Значения в восьмеричном виде начинаются с нуля, и, конечно, использование цифр 8 и 9 запрещено. Запись шестнадцатеричных чисел начинается с 0x или 0X (цифра 0 и латинская ASCII-буква X в произвольном регистре). Таким образом, ноль можно записать тремя различными способами:

```
0;  
00;  
0x0;
```

Шестнадцатеричные цифры:

```
0xabcd;  
0xffff;
```

Целочисленные:

```
0L;  
9223372036854775807L;  
0xffffffffffffffffL;
```

Дробные:

```
3.14;  
2.;  
.5;  
7e10;  
3.1E-20;
```

Булевские:

```
true;  
false;
```

Символьные:

```
'a';  
'\u0044';
```

Строковые:

```
"";  
"Это \u0056 строковый \"литерал\", \tkоторый включает спец. символы.\r\n";
```

Служебные (ключевые) слова.

Это зарезервированные слова, состоящие из ASCII-символов и выполняющие различные задачи языка. Вот их полный список (48 слов):

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

Null литералы:

```
null;
```

Операторы (всего 36):

```
= > < ! ~ ? :  
== <= >= != && || ++ --  
+ - * / & | ^ % << >> >>>  
+= -= *= /= &= |= ^= %= <<= >>= >>>=
```

Комментарии.

Никак не влияют на исполнение программы и на байткод, компилятор их пропускает. Они служат только для написания пояснений в исходном коде.

```
/* Многострочный, в одну строку */

// Однострочный

/*
    Многострочный
*/

/**
 * @lecture Многострочный
 *      для документации
 *      JavaDoc
 * <p> </br> <i> <b> <link>
 */
```

2.3. Основные типы данных и операции над ними

Java является строго типизированным языком. Это означает, что любая переменная и любое выражение имеют известный тип еще на момент компиляции. Такое строгое правило позволяет выявлять многие ошибки уже во время компиляции.

Все типы данных в языке разделяются на **две группы**.

Первую группу составляют 8 простых, или **примитивных**, по другому **встроенных типов данных**. Они подразделяются на три подгруппы:

- *Целочисленные*

```
byte
short
int
long
char
```

- *Дробные*

```
float
double
```

- *Булевский*

```
boolean
```

Вторая группа – все остальные.

- *объекты,*
- *интерфейсы,*
- *массивы,*

- *перечисления.*

2.4. Объявление переменных и констант

Переменные являются контейнерами для значений. Когда объявляется переменная, то в памяти компьютера отводится определённое место для хранения значения такого типа, которого переменная объявлена.

Существуют определённые правила для объявления переменных:

- при объявлении указывается её тип;
- имя переменной должно нести смысловую нагрузку;
- желательно сразу инициализировать конкретным значением.

Кроме того имена переменных должны удовлетворять требованиям которые предъявляются к идентификаторам (см. п.2.3.) .Имя переменной должно быть уникально и не должно совпадать со служебными словами Java (см. п.2.3)

Например: *int* daysInWeek = 5;
double payRate;
double taxRate = 0.67;

Рекомендуется объявлять и инициализировать переменные в начале блока кода, где они используются.

В языке Java существуют четыре вида переменных:

- локальные переменные;
- переменные экземпляра;
- переменные формального параметра;
- статические переменные.

Константы

Для определения констант в JAVA используется ключевое слово *final*. Оно обычно задается при объявлении переменной и указывает на то, что заданная переменная может определяться только один раз, т.е. является, по сути, константой. При объявлении такой переменной ее сразу следует определить, задать значение или определить ее в конструкторе, в других случаях компилятор выдаст ошибку. Применение данного ключевого слова почти всегда гарантирует, что переменная определена и инициализирована.

Пример:

final int MONTH_IN_YEAR = 12;

Подводя итог по примитивным типам данных следует упомянуть то, что в JAVA нет **беззнаковых типов** данных, в отличие например от C/C++.

2.5. Выражения и операторы

Приоритеты операций Java, от высшего к низшему, описаны ниже. Обратите внимание, что в первой строке указаны элементы, которые, как правило, не считают символами операций: круглые и квадратные скобки и символ точки. С технической точки зрения они являются разделителями, но в выражениях они действуют подобно операциям. Круглые скобки используют для изменения порядка выполнения операций. Квадратные скобки служат для индексации массивов. А символ точки используется для разыменования объектов. В таблице 2.1. представлены сгруппированы операции в порядке понижения приорита, от высшего к низшему.

Таблица 2.1. Порядок приоритета операций

№	Тип операции
1	() [] .
2	++ -- ~!
3	* / %
4	+ -
5	>> >>> <<
6	== !=
7	&
8	^
9	
10	&&
11	
12	=

Пример использования операций:

```

package ru.mirea.lecture.primitive;

public class PrimitiveSampleObjeration {

    public static void main(String[] args) {
        int x = 1;
        int y = ++x;

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

        x = 1;
        y = x++;

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

        x = 2 + 2 * 2;
        System.out.println("x = " + x);

        x = (2 + 2) * 2;
        System.out.println("x = " + x);
    }
}

```

Результат выполнения фрагмента программы:

```

x = 2, y = 2
x = 2, y = 1
x = 6
x = 8

```

В третьем случае, т.к. приоритет операции `*` выше чем у операции `+`, то результат будет равен числу 6, используя круглые скобки, мы явным образом повысили приоритет операции `+` и получили число 8.

2.6. Основные алгоритмические конструкции

2.6.1. Инструкции выбора

Ветвление.

Пожалуй, наиболее часто встречающейся конструкцией в Java, как и в любом другом структурном языке программирования, является оператор условного перехода.

В общем случае конструкция **полного ветвления** выглядит так:

```

if (логическое выражение) {блок операторов 1}
else {блок операторов 2}

```

Логическое выражение записывается с помощью знаков логических операций и операндов и может быть любой языковой конструкцией, которая возвращает булевский результат. В отличие от языка C, в котором в качестве логического выражения может быть использованы различные типы данных, где отличное от нуля выражение трактуется как истинное

значение, а ноль как ложное. В Java возможно использование только логических выражений.

Возможно использование *неполного ветвления*, если ветвь *else* отсутствует. Например:

```
if ( x<0 ) x *= -1;
```

Множественный выбор

Оператор `switch` позволяет осуществлять в случае необходимости множественный выбор. Выбор осуществляется на основе целочисленного значения.

Структура оператора:

```
switch(int value){  
case const1:  
  {выражение или блок 1 }  
case const2:  
  {выражение или блок 2}  
case constN:  
  {выражение или блок N}  
default:  
  {выражение или блок}  
}
```

Причем ветка *default* не является обязательной.

В качестве параметра `switch` может быть использована переменная типа `byte`, `short`, `int`, `char` или выражение. Выражение должно в конечном итоге возвращать параметр одного из указанных ранее типов.

2.6.2. Работа с циклами

В языке Java имеется три основных конструкции для работы с циклами:

- цикл `while`
- цикл `do`
- цикл `for`

Цикл с предусловием while

Основная форма цикла `while` может быть представлена как *while*(логическое выражение){
выражение или блок;
}

В данной языковой конструкции выражение или блок операторов , будут исполняться до тех пор, пока логическое выражение будет иметь истинное значение.

Выражение или блок представляющий тело цикла будет завершён раньше времени по следующей по причине:

- внутри блока операторов встретился оператор `continue`, то часть тела цикла, следующая за оператором *continue* будет

пропущена и выполнение цикла начнется со следующей итерации;

- встретился оператор **break**, то выполнение цикла будет сразу прекращено;
- если возникла исключительная ситуация то выполнение while тоже будет прекращено.

Цикл с постусловием do-while

do{

повторяющееся выражение или блок;

}

while(логическое выражение)

В отличие от цикла while() {}, цикл do {} while() будет выполняться до тех пор, пока логическое выражение будет ложным. Вторым важным отличием от цикла с предусловием является то, что тело цикла do {} while() будет выполнен по крайней мере хотя бы один раз.

Итерационный цикл for

Довольно часто, необходимо изменять значение какой-либо переменной в заданном диапазоне, и выполнять повторяющуюся последовательность операторов с использованием этой переменной. Для выполнения этой последовательности действий как нельзя лучше подходит конструкция цикла for. Для организации итерационного цикла используется переменная особого вида, которая называется - счётчик. Счётчик может быть только целочисленным значением. Основная форма цикла for выглядит следующим образом:

for(иниц_счётчика_ц; усл_выполнения_ц; измен_счётчика_ц)

{

повторяющееся выражение или блок операторов;

}

Ключевыми элементами данной языковой конструкции являются предложения заключенные в круглых скобках и разделенные точкой с запятой.

иниц_счётчика_ц означает инициализация значения счетчика цикла
усл_выполнения_ц означает проверка условия выполнения цикла(пока истина - работаем).

измен_счётчика_ц означает изменение переменной счётчика во избежание закливания.

Пример:

int sum = 0;

for (int i = 0; i < 10 ; i++) sum += i;

2.7. Объявление массивов и доступ к ним

В отличие от обычных переменных, которые хранят только одно значение, массивы используются для хранения целого набора значений. Количество значений в массиве называется его длиной, сами значения — элементами массива. Значений может не быть вовсе, в этом случае массив считается пустым, а его длина равной нулю.

Объявление массивов

```
int x []  
int [] y  
int [] [] z  
int [] h []  
int g [] []
```

Доступ к элементам массива осуществляется по индексу. В свою очередь индекс не может быть меньше нуля.

```

package ru.mirea.lecture.primitive;

public class PrimitiveSampleArray {

    public static void initArray(int[] array, int value) {
        for (int index = 0; index < array.length; ++index)
            array[index] = value;
    }

    public static void printArray(String name, int[] array) {
        System.out.print(name + ": [");
        for (int index = 0; index < array.length; ++index) {
            if (index > 0)
                System.out.print(", ");
            System.out.print(array[index]);
        }
        System.out.println("]");
    }

    public static void main(String[] args) {

        int array0[] = new int[10];
        int array1[] = new int[10];

        System.out.println("-----");
        initArray(array0, 2);
        initArray(array1, 3);
        printArray("Array0", array0);
        printArray("Array1", array1);
        System.arraycopy(array0, 0, array1, 0, array1.length);
        printArray("Array0", array0);
        printArray("Array1", array1);

        System.out.println("-----");
        initArray(array0, 2);
        initArray(array1, 3);
        printArray("Array0", array0);
        printArray("Array1", array1);

        System.arraycopy(array0, 5, array1, 5, 3);
        printArray("Array0", array0);
        printArray("Array1", array1);
    }
}

```

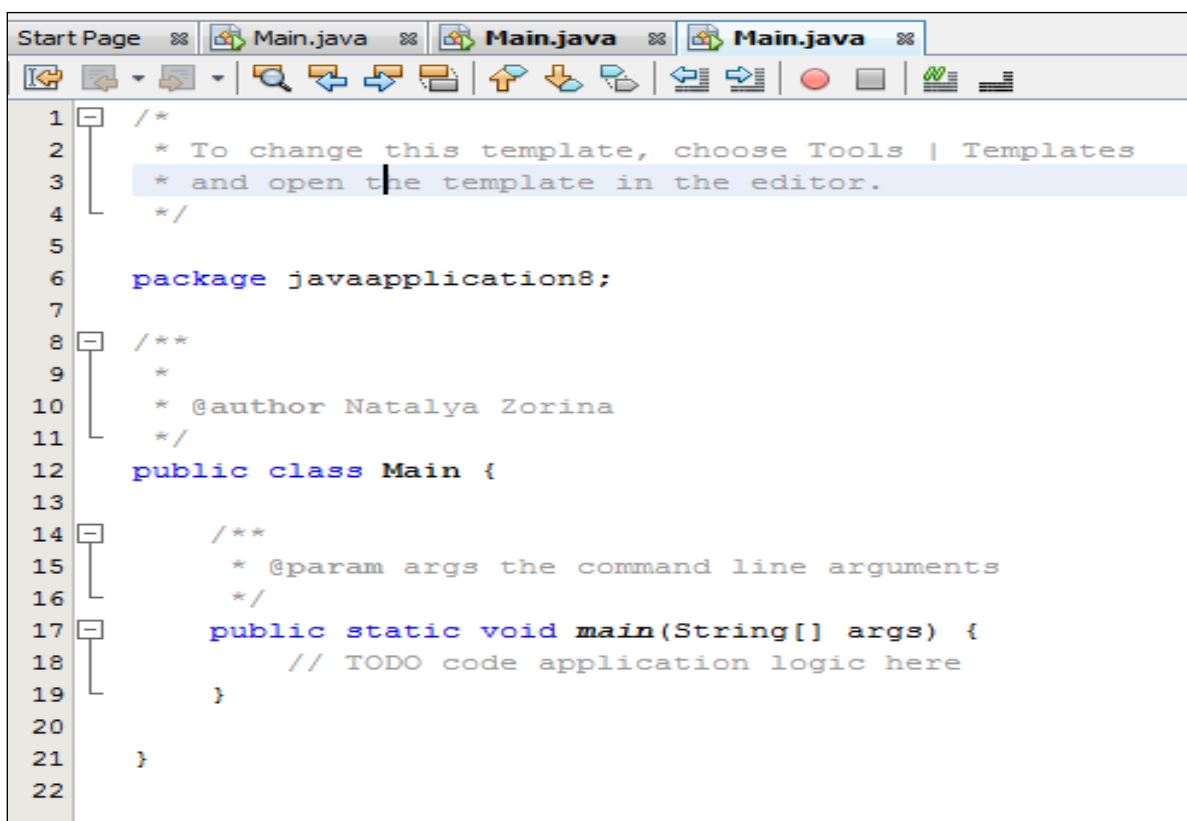
На примере выше, показаны простые способы задания массивов и специальные методы по работе с ними. Вызов `System.arraycopy()` служит для простого и быстрого копирования одного массива в другой, с заданной позиции положения и заданного размера. В случае возникновения исключительной ситуации (например, неправильный индекс, выход за границу массива), будет выдана ошибка (Exception).

Результат работы кода, представленного выше. Из него видно, что в первом случае мы просто скопировали один массив в другой. Во втором же, мы скопировали 3 элемента из одного массива, с позиции 5, во второй.

```
-----
Array0: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Array1: [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
Array0: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Array1: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
-----
Array0: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Array1: [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
Array0: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Array1: [3, 3, 3, 3, 3, 2, 2, 2, 3, 3]
```

2.8. Структура кода и вывод результата работы программы на экран

Структура исходного кода приложения на Java представлена на примере:



```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5
6  package javaapplication8;
7
8  /**
9   *
10   * @author Natalya Zorina
11   */
12  public class Main {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19      }
20
21  }
```

Внутри многострочного комментария `/** ... */`, предназначенного для автоматического создания документации по классу присутствует инструкция задания метаданных с помощью выражения `@author` – информация об авторе проекта для утилиты создания документации `javadoc`. Метаданные – это некая информация, которая не относится к

работе программы и не включается в неё при компиляции, но сопровождает программу и может быть использована другими программами для проверки прав на доступ к ней или её распространения, проверки совместимости с другими программами, указания параметров для запуска класса и т.п. Далее следует объявление класса `Main`, который является главным классом приложения, в нём объявлен метод `Main` (как `public`). Метод `Main` - главный метод приложения, он управляет работой запускаемой программы и вызывается при запуске приложения

Напишем традиционный пример, который обычно используется для изучающих программирование, а именно добавим в тело метода `Main` следующую строку:

```
System.out.println("Hello, world!");
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package lab1;

/**
 *
 * @n.v.zorina
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // TODO code applicat:
    }
}
```

Метод `println()` выводит строку "Hello, world!" на экран и осуществляет перевод каретки. Каждый класс помещается в пакет, вообще по правилам сначала создается пакет, а затем класс, но хотя бы один класс из программы должен содержать метод `main()`. Ключевое слово `public` означает, что метод свободно вызывается. Ключевое слово `static` означает, что этот метод является общим, а не таким который используется только с этим объектом.

Таким образом откомпилировав программу и убедившись, что в ней нет ошибок мы можем запустить приложение и увидим результат как на примере ниже. Таким образом мы создали первое консольное приложение, без использования GUI.

```
run:
Hello, world!
BUILD SUCCESSFUL (total time: 4 seconds)
```

Класс *System* содержит набор полезных статических методов и полей. Экземпляр этого класса нельзя получить для использования. Среди прочих полезных средств, предоставляемых этим классом, особо стоит отметить потоки стандартных ввода и вывода, поток для вывода ошибок. Конечно, наиболее широко используемым является стандартный вывод, доступный через объект *System.out*.

2.9. Ввод данных с клавиатуры

Следуя модели стандартных потоков, берущей своё начало ещё со времён UNIX в Java тоже есть стандартные потоки *System.out* *System.in* *System.err* для стандартного вывода, стандартного ввода и поток ошибок.

System.out и *System.err* можно использовать напрямую, а *System.in* приходится настраивать. Так как чтение осуществляется, как правило построчно, то есть смысл буферизовать стандартный ввод *System.in* посредством *BufferedReader*. Для этого поток *System.in* следует конвертировать в считывающее устройство *Reader* посредством класса преобразователя *InputStreamReader*. Или есть ещё более простой способ, как на примере внизу:

```
package ru.mirea;
import java.lang.*;
import java.util.Scanner;

/**
 * Author: ZorinaN
 */
public class exampleOutput {

    public static void main(String [] args){

        Scanner myScanner = new Scanner(System.in);
        System.out.println(myScanner.nextLine());

    }

}
```

Программа демонстрирует эхо-вывод пользователя. Нужно только импортировать соответствующий пакет *java.util.Scanner*. Для этого у

класса `Scanner` имеется перегруженный конструктор. Как это работает? Хорошо видно на рис.2.1.

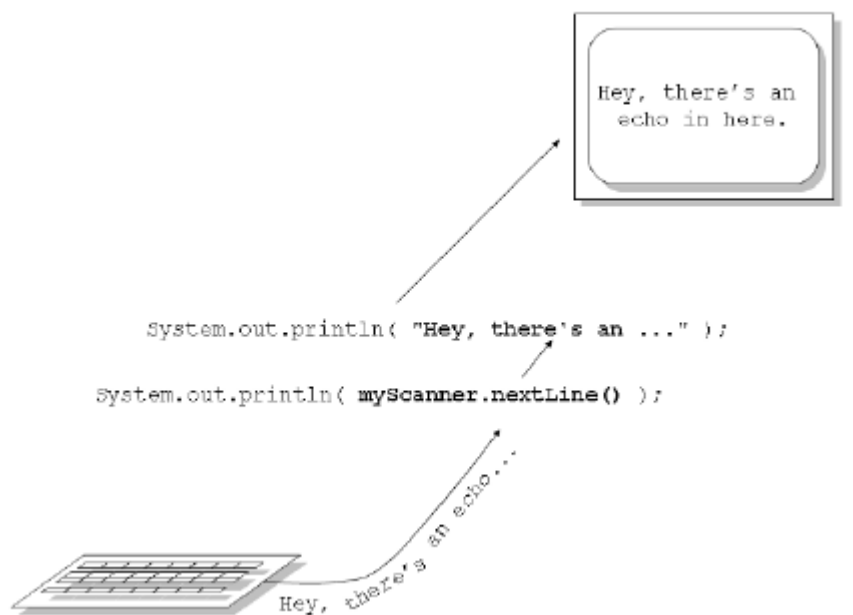


Рис 2.1. Схема чтения ввода с клавиатуры.

Глава 3. Базовые классы Java

3.1. Пакеты и классы, поддерживаемые в JDK

Конфликты имен становятся источником серьезных проблем при разработке повторно используемого кода. Как бы тщательно вы ни подбирали имена для своих классов и методов, кто-нибудь может использовать это же имя для других целей. При использовании простых названий проблема лишь усугубляется — такие имена с большей вероятностью будут задействованы кем-либо еще, кто также захочет пользоваться простыми словами. Такие имена, как *set*, *get*, *clear* и т. д., встречаются очень часто, и конфликты при их использовании оказываются практически неизбежными.

Во многих языках программирования предлагается стандартное решение — использование —префикса пакета перед каждым именем класса, типа, глобальной функции и так далее. Соглашения о префиксах создают *контекст имен (naming context)*, который предотвращает конфликты имен одного пакета с именами другого. Обычно такие префиксы имеют длину в несколько символов и являются сокращением названия пакета — например, `Xt` для —X Toolkit или `WIN32` для 32-разрядного Windows API.

Если программа состоит всего из нескольких пакетов, вероятность конфликтов префиксов невелика. Однако, поскольку префиксы являются

сокращениями, при увеличении числа пакетов вероятность конфликта имен повышается.

В Java принято более формальное понятие пакета, в котором типы и подпакеты выступают в качестве членов, т.е. входят в состав пакетов. Пакеты являются именованными и могут импортироваться. Имена пакетов имеют иерархическую структуру, их компоненты разделяются точками. При использовании компонента пакета необходимо либо ввести его полное имя, либо **импортировать пакет** — целиком или частично. Иерархическая структура имен пакетов позволяет работать с более длинными именами. Кроме того, это дает возможность избежать конфликтов имен — если в двух пакетах имеются классы с одинаковыми именами, можно применить для их вызова форму имени, в которую включается имя пакета.

Приведем пример метода, в котором полные имена используются для вывода текущей даты и времени с помощью вспомогательного класса Java с именем Date:

```
class Date1 {  
    public static void main(String[] args) {  
        java.util.Date now = new java.util.Date();  
        System.out.println(now);  
    }  
}
```

Теперь сравните этот пример с другим, в котором для объявления типа Date используется ключевое слово import:

```
import java.util.Date;  
class Date2 {  
    public static void main(String[] args) {  
        Date now = new Date();  
        System.out.println(now);  
    }  
}
```

Пакеты Java не до конца разрешают проблему конфликтов имен. Два различных проекта *могут* присвоить своим пакетам одинаковые имена. Эта проблема решается только за счет использования общепринятых соглашений об именах. По наиболее распространенному из таких соглашений в качестве префикса имени пакета используется перевернутое имя домена организации в Internet. Например, если институт mirea содержит в Internet домен с именем mirea.ru, то разработанные в институте пакеты будут иметь имена типа ru.mirea.package.

Точки, разделяющие компоненты имени пакета, иногда могут привести к недоразумениям, поскольку те же самые точки используются при вызове методов и доступе к полям в ссылках на объекты. С другой стороны, `java.util` является пакетом, так что допускается импорт `java.util.Date` (или `java.util.*`, если вы хотите импортировать все содержимое пакета). Если у вас возникают проблемы с импортированием чего-либо, остановитесь и убедитесь в том, что вы импортируете тип.

Классы Java всегда объединяются в пакеты. Имя пакета задается в начале файла:

```
package com.sun.games;  
class Card  
{  
// ...  
}  
// ...
```

Если имя пакета не было указано в объявлении `package`, класс становится частью **безымянного пакета**. Хотя это вполне подходит для приложения (или апплета), которое используется отдельно от другого кода, все классы, которые предназначены для использования в библиотеках, должны включаться в именованные пакеты.

3.1. Классы работы со строками и их методы.

Для работы с последовательностями символов в Java предусмотрены тип объектов `String` и языковая поддержка при их инициализации. Класс `String` предоставляет разнообразные методы для работы с объектами `String`.

Примеры литералов типа `String` уже встречались нам в примерах — в частности, в программе `HelloWorld`. Когда в программе появляется оператор следующего вида:

```
System.out.println("Hello, world");
```

компилятор Java на самом деле создает объект `String`, присваивает ему значение указанного литерала и передает его в качестве параметра методу `println`.

Объекты типа `String` отличаются от массивов тем, что при их создании не нужно указывать размер. Создание нового объекта `String` и его инициализация выполняются всего одним оператором, как показывает следующий пример:

```
class StringDemo {  
static public void main(String args[]) {  
String myName = "Ulius";  
myName = myName + " Cesar";  
System.out.println("Name = " + myName);
```

```
}  
}
```

Мы создаем объект String с именем myName и инициализируем его строковым литералом. Выражение с оператором конкатенации +, следующее за инициализацией, создает новый объект String с новым значением. Наконец, значение myName выводится в стандартный выходной поток. Результат работы приведенной выше программы будет таким:

```
Name = Ulius Cesar
```

Кроме знака +, в качестве оператора конкатенации можно использовать оператор += как сокращенную форму, в которой название переменной размещается в левой части оператора. Усовершенствованная версия приведенного выше примера выглядит так:

```
class BetterStringDemo {  
    static public void main(String args[]) {  
        String myName = "Ulius";  
        String occupation = "Reorganization Specialist";  
        myName = myName + " Cesar";  
        myName += " ";  
        myName += "(" + occupation + ")";  
        System.out.println("Name = " + myName);  
    }  
}
```

Теперь при запуске программы будет выведена следующая строка:

```
Name = Petronius Arbiter (Reorganization Specialist)
```

Объекты String содержат метод length, который возвращает количество символов в строке. Символы имеют индексы от 0 до length()-1.

Объекты String являются *неизменяемыми*, или *доступными только для чтения*; содержимое объекта String никогда не меняется. Когда в программе встречаются операторы следующего вида:

```
str = "redwood";  
// ... сделать что-нибудь со str ...  
str = "oak";
```

второй оператор присваивания задает новое значение *ссылки на объект*, а не *содержимого* строки. При каждом выполнении операции, которая на первый взгляд изменяет содержимое объекта (например, использование выше +=), на самом деле возникает новый объект String, также доступный только для чтения, — тогда как содержимое исходного объекта String остается неизменным. Класс StringBuffer позволяет создавать строки с изменяющимся содержимым;

Самый простой способ сравнить два объекта String и выяснить, совпадает ли их содержимое, заключается в использовании метода equals:

```
if (oneStr.equals(twoStr))  
foundDuplicate(oneStr, twoStr);
```

3.2. Объявление и доступ к строковым массивам

В Java можно создавать строковые массивы, т.е. массивы, в качестве элементов которых будут содержаться строки. Пример объявления такого массива:

```
String stringArray[] = new String[4];
```

В результате такого объявления создается массив из четырёх объектов типа строка. Все элементы этого массива являются объектами – строками. В этом примере элемент массива инициализируется строкой

```
stringArray[0] = new String( "Hello" );
```

Must be performed for each element

Объекты строки можно, например сравнивать, т.е. производить действия которые можно производить со строками. Но, внимание! Поскольку мы работаем не с простыми данными, а со ссылочными переменными, то нельзя сравнивать с помощью знака (==). Нужно использовать специальные методы работы со строками, например:

- equals();
- equalsIgnoreCase();

3.3. Классы Vector, Calendar, и Date

Класс Vector содержится в пакете java.util и может быть импортирован в программу. Представляет собой массив- вектор, который может динамически изменять размер по мере необходимости. Внутри вектора могут храниться в качестве элементов различные типы данных. Не может содержать по определению примитивные типы данных, нужно использовать классы обёртки.

Пример объявления:

```
Vector v = new Vector(3);
```

В результате работы участка кода (конструктор получает в качестве параметра число создаваемых элементов вектора) создается вектор с тремя элементами.

В таблице 3.1. представлены некоторые методы для работы с вектором.

таблица 3.1. Методы для работы с вектором.

Название метода	Описание
add(obj)	Помещает ссылку на экземпляр объекта obj в

	следующий доступный элемент вектора
capacity ()	Емкость вектора, общее количество элементов
contains(obj)	Производит поиск в массиве элемента obj и возвращает в зависимости от удачного или нет поиска значение true/ false
get (i)	Возвращает содержимое элемента вектора с индексом i
remove (i)	Удаляет элемент из вектора, т.е. устанавливает значение элемента с индексом i в null
size()	Возвращает размер

Представляют собой классы для работы со значениями дат. Для работы с ними нужно импортировать пакет java.text package

Класс *Date* как и класс *Calendar* предназначены, как ясно из названия для работы с датами и находятся в пакете java.util package.

Класс Calendar. Содержит методы и константы.

Класс Date. Экземпляры соответствующего класса содержат актуальные значения дат. Классы для работы со значениями дат находятся в пакете java.text package:

Класс DateFormat. Экземпляры этого класса обеспечивают представление дат в нескольких общепринятых форматах для различных целей.

3.4. Классы обертки

Так как Java чисто объектно-ориентированный язык, то в нем предусмотрена возможность с помощью специальных классов, называемых классами обертками конвертации примитивных типов данных в классы обертки и наоборот:

- Примитивы в обёртки
 - Инициализируется экземпляр объекта типа соответствующий класс обёртка, используя примитивные переменные как аргументы
- Обёртки в примитивы
 - Используется метод экземпляра класса (инстантный метод), который называется *xxxValue()* (где xxx – примитивный тип данных)

Названия используются такие же как и у примитивных типов данных с использованием в названии первой заглавной буквы (исключая целые и символы)

Конвертация строк в примитивы и обратно

Строки конвертируются в примитивы с помощью инстантных методов с именем *parsexxx* (где xxx- примитивный тип данных).

Класс обёртка в примитив:

- Используется инстантный метод с названием *toString()*, который создаёт экземпляр класса *String*, содержащий примитивное значение

Конвертация *String* в обёртку и обратно:

- *String* в обёртку:
 - Используется статический метод обертки с названием *valueOf()*, который создает экземпляр обертки из экземпляра *String*
- Обёртку в *String*:
 - Используется метод обертки с названием *toString()*, который создает экземпляр *String* из экземпляра обертки

Глава 3. Базовые классы Java

3.2. Пакеты и классы, поддерживаемые в JDK

Конфликты имен становятся источником серьезных проблем при разработке повторно используемого кода. Как бы тщательно вы ни подбирали имена для своих классов и методов, кто-нибудь может использовать это же имя для других целей. При использовании простых названий проблема лишь усугубляется — такие имена с большей вероятностью будут задействованы кем-либо еще, кто также захочет пользоваться простыми словами. Такие имена, как *set*, *get*, *clear* и т. д., встречаются очень часто, и конфликты при их использовании оказываются практически неизбежными.

Во многих языках программирования предлагается стандартное решение — использование —префикса пакета перед каждым именем класса, типа, глобальной функции и так далее. Соглашения о префиксах создают *контекст имен (naming context)*, который предотвращает конфликты имен одного пакета с именами другого. Обычно такие префиксы имеют длину в несколько символов и являются сокращением названия пакета — например, *Xt* для —X Toolkit или *WIN32* для 32-разрядного Windows API.

Если программа состоит всего из нескольких пакетов, вероятность конфликтов префиксов невелика. Однако, поскольку префиксы являются сокращениями, при увеличении числа пакетов вероятность конфликта имен повышается.

В Java принято более формальное понятие пакета, в котором типы и подпакеты выступают в качестве членов, т.е. входят в состав пакетов. Пакеты являются именованными и могут импортироваться. Имена пакетов имеют иерархическую структуру, их компоненты разделяются точками. При использовании компонента пакета необходимо либо ввести его полное

имя, либо **импортировать пакет** — целиком или частично. Иерархическая структура имен пакетов позволяет работать с более длинными именами. Кроме того, это дает возможность избежать конфликтов имен — если в двух пакетах имеются классы с одинаковыми именами, можно применить для их вызова форму имени, в которую включается имя пакета.

Приведем пример метода, в котором полные имена используются для вывода текущей даты и времени с помощью вспомогательного класса Java с именем Date:

```
class Date1 {  
    public static void main(String[] args) {  
        java.util.Date now = new java.util.Date();  
        System.out.println(now);  
    }  
}
```

Теперь сравните этот пример с другим, в котором для объявления типа Date используется ключевое слово import:

```
import java.util.Date;  
class Date2 {  
    public static void main(String[] args) {  
        Date now = new Date();  
        System.out.println(now);  
    }  
}
```

Пакеты Java не до конца разрешают проблему конфликтов имен. Два различных проекта *могут* присвоить своим пакетам одинаковые имена. Эта проблема решается только за счет использования общепринятых соглашений об именах. По наиболее распространенному из таких соглашений в качестве префикса имени пакета используется перевернутое имя домена организации в Internet. Например, если институт mirea содержит в Internet домен с именем mirea.ru, то разработанные в институте пакеты будут иметь имена типа ru.mirea.package.

Точки, разделяющие компоненты имени пакета, иногда могут привести к недоразумениям, поскольку те же самые точки используются при вызове методов и доступе к полям в ссылках на объекты. С другой стороны, java.util является пакетом, так что допускается импорт java.util.Date (или java.util.*, если вы хотите импортировать все содержимое пакета). Если у вас возникают проблемы с импортированием чего-либо, остановитесь и убедитесь в том, что вы импортируете тип.

Классы Java всегда объединяются в пакеты. Имя пакета задается в начале файла:

```
package com.sun.games;  
class Card  
{  
    // ...  
}  
// ...
```

Если имя пакета не было указано в объявлении `package`, класс становится частью **безымянного пакета**. Хотя это вполне подходит для приложения (или апплета), которое используется отдельно от другого кода, все классы, которые предназначаются для использования в библиотеках, должны включаться в именованные пакеты.

3.5. Классы работы со строками и их методы.

Для работы с последовательностями символов в Java предусмотрены тип объектов `String` и языковая поддержка при их инициализации. Класс `String` предоставляет разнообразные методы для работы с объектами `String`.

Примеры литералов типа `String` уже встречались нам в примерах — в частности, в программе `HelloWorld`. Когда в программе появляется оператор следующего вида:

```
System.out.println("Hello, world");
```

компилятор Java на самом деле создает объект `String`, присваивает ему значение указанного литерала и передает его в качестве параметра методу `println`.

Объекты типа `String` отличаются от массивов тем, что при их создании не нужно указывать размер. Создание нового объекта `String` и его инициализация выполняются всего одним оператором, как показывает следующий пример:

```
class StringDemo {  
    static public void main(String args[]) {  
        String myName = "Ulius";  
        myName = myName + " Cesar";  
        System.out.println("Name = " + myName);  
    }  
}
```

Мы создаем объект `String` с именем `myName` и инициализируем его строковым литералом. Выражение с оператором конкатенации `+`, следующее за инициализацией, создает новый объект `String` с новым значением. Наконец, значение `myName` выводится в стандартный выходной поток. Результат работы приведенной выше программы будет таким:

Name = Ulius Cesar

Кроме знака +, в качестве оператора конкатенации можно использовать оператор += как сокращенную форму, в которой название переменной размещается в левой части оператора. Усовершенствованная версия приведенного выше примера выглядит так:

```
class BetterStringDemo {  
    static public void main(String args[]) {  
        String myName = "Ulius";  
        String occupation = "Reorganization Specialist";  
        myName = myName + " Cesar";  
        myName += " ";  
        myName += "(" + occupation + ")";  
        System.out.println("Name = " + myName);  
    }  
}
```

Теперь при запуске программы будет выведена следующая строка:

Name = Petronius Arbiter (Reorganization Specialist)

Объекты String содержат метод length, который возвращает количество символов в строке. Символы имеют индексы от 0 до length()-1.

Объекты String являются *неизменяемыми*, или *доступными только для чтения*; содержимое объекта String никогда не меняется. Когда в программе встречаются операторы следующего вида:

```
str = "redwood";  
// ... сделать что-нибудь со str ...  
str = "oak";
```

второй оператор присваивания задает новое значение *ссылки на объект*, а не *содержимого* строки. При каждом выполнении операции, которая на первый взгляд изменяет содержимое объекта (например, использование выше +=), на самом деле возникает новый объект String, также доступный только для чтения, — тогда как содержимое исходного объекта String остается неизменным. Класс StringBuffer позволяет создавать строки с изменяющимся содержимым;

Самый простой способ сравнить два объекта String и выяснить, совпадает ли их содержимое, заключается в использовании метода equals:

```
if (oneStr.equals(twoStr))  
    foundDuplicate(oneStr, twoStr);
```

3.6. Объявление и доступ к строковым массивам

В Java можно создавать строковые массивы, т.е. массивы, в качестве элементов которых будут содержаться строки. Пример объявления такого массива:

```
String stringArray[] = new String[4];
```

В результате такого объявления создается массив из четырёх объектов типа строка. Все элементы этого массива являются объектами – строками. В этом примере элемент массива инициализируется строкой

```
stringArray[0] = new String("Hello");
```

Must be performed for each element

Объекты строки можно, например сравнивать, т.е. производить действия которые можно производить со строками. Но, внимание! Поскольку мы работаем не с простыми данными, а со ссылочными переменными, то нельзя сравнивать с помощью знака (==). Нужно использовать специальные методы работы со строками, например:

- equals();
- equalsIgnoreCase();

3.7. Классы Vector, Calendar, и Date

Класс Vector содержится в пакете java.util и может быть импортирован в программу. Представляет собой массив- вектор, который может динамически изменять размер по мере необходимости. Внутри вектора могут храниться в качестве элементов различные типы данных. Не может содержать по определению примитивные типы данных, нужно использовать классы обёртки.

Пример объявления:

```
Vector v = new Vector(3);
```

В результате работы участка кода (конструктор получает в качестве параметра число создаваемых элементов вектора) создается вектор с тремя элементами.

В таблице 3.1. представлены некоторые методы для работы с вектором.

таблица 3.1. Методы для работы с вектором.

Название метода	Описание
add(obj)	Помещает ссылку на экземпляр объекта obj в следующий доступный элемент вектора
capacity ()	Емкость вектора, общее количество элементов
contains(obj)	Производит поиск в массиве элемента obj и возвращает в зависимости от удачного или нет поиска значение true/ false
get (i)	Возвращает содержимое элемента вектора с индексом i
remove (i)	Удаляет элемент из вектора, т.е. устанавливает

	значение элемента с индексом <i>i</i> в null
size()	Возвращает размер

Представляют собой классы для работы со значениями дат. Для работы с ними нужно импортировать пакет `java.text package`

Класс ***Date*** как и класс ***Calendar*** предназначены, как ясно из названия для работы с датами и находятся в пакете `java.util package`.

Класс Calendar. Содержит методы и константы.

Класс Date. Экземпляры соответствующего класса содержат актуальные значения дат. Классы для работы со значениями дат находятся в пакете `java.text package`:

Класс DateFormat. Экземпляры этого класса обеспечивают представление дат в нескольких общепринятых форматах для различных целей.

3.8. Классы обертки

Так как Java чисто объектно-ориентированный язык, то в нем предусмотрена возможность с помощью специальных классов, называемых классами обертки конвертации примитивных типов данных в классы обертки и наоборот:

- Примитивы в обёртки
 - Инициализируется экземпляр объекта типа соответствующий класс обёртка, используя примитивные переменные как аргументы
- Обёртки в примитивы
 - Используется метод экземпляра класса (инстантный метод), который называется *xxxValue()* (где *xxx* – примитивный тип данных)

Названия используются такие же как и у примитивных типов данных с использованием в названии первой заглавной буквы (исключая целые и символы)

Конвертация строк в примитивы и обратно

Строки конвертируются в примитивы с помощью инстантных методов с именем *parsexxx* (где *xxx*- примитивный тип данных).

Класс обёртка в примитив:

- Используется инстантный метод с названием *toString()*, который создаёт экземпляр класса `String`, содержащий примитивное значение

Конвертация `String` в обёртку и обратно:

- `String` в обёртку:

- Используется статический метод обертки с названием *valueOf()*, который создает экземпляр обертки из экземпляра *String*
- Обёртку в *String*:
 - Используется метод обертки с названием *toString()*, который создает экземпляр *String* из экземпляра обертки

Глава 4. Основные понятия и концепция объектно-ориентированного анализа и объектно-ориентированного дизайна

4.1. Объектно-ориентированный анализ и спецификация требований к ПО

Существенной особенностью инженерного подхода к созданию программных средств является наличие стадии **анализа** и стадии **проектирования** или синтеза.

Целью анализа является преобразование общих и, как правило, неполных, нечетко сформулированных, противоречивых, а может быть и нереализуемых требований к программному комплексу в достаточно полные, точные, непротиворечивые и реализуемые формулировки требований, которые называют системными.

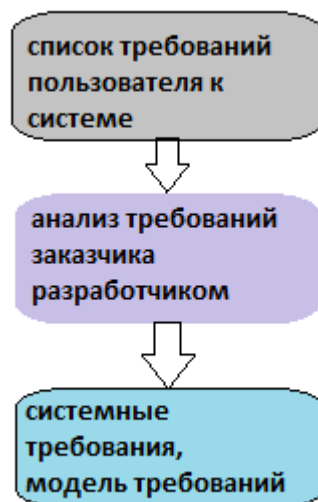
Независимо от выбранного подхода к проектированию процесс разработки программного обеспечения начинают с **этапа выработки требований** и затем, их **анализа**.

Список требований заказчика к системе включает: внешние условия, функции и ограничения

Как правило, **анализ требований** к программному комплексу ограничивается спецификацией требований к *функциям, данным и управлению*, составляющим, так называемую **модель требований** будущего ПО.

Список общих требований, определенных в техническом задании, должен содержать следующие сведения:

- внешние условия, при которых будет применяться система, а именно: аппаратные и программные ресурсы так называемого операционного окружения, режимы применения, состав пользователей и оборудование их рабочих мест, разграничение доступа к системе;
- выполняемые функции, входные и выходные данные;
- ограничения на сроки и порядок разработки системы, в том числе состав программной и эксплуатационной документации



Анализ требований к программному средству в процессе разработки включает два вида деятельности:

- анализ предметной области;
- планирование вариантов использования (сценариев).

Системные требования включают: согласованные с заказчиком требования к архитектуре, согласованные функции, согласованные требования к интерфейсу пользователя и т.д.

В результате согласованных системных требований вырабатывается **модель требований**.

4.1. Основные понятия объектно-ориентированного анализа и объектно-ориентированного дизайна

Как мы уже обсуждали в первой главе синоним термина дизайн – проектирование. Так вот, объектно-ориентированный анализ и объектно-ориентированный дизайн - понятия тесно связаны между собой.

Как мы уже выяснили, вначале проводится:

- **Системный анализ**

- Чтобы изучить, понять и определить требования к системе;

И определяются:

- **Системные требования**

- Чтобы определить, какие системные нужды будут соответствовать пользователям в терминах ведения их бизнеса

Составляются **модели требований**

Модели представляют описание некоторых требуемых аспектов системы

Логические модели

Логическая модель показывает, что требуется в системе, независимо от технологии обычно применяемой для реализации этих свойств.

Процесс разработки логической модели называют внешним (логическим), или *эскизным проектированием*

Физические модели

Физическая модель показывает, как реализовать и интегрировать системные компоненты, используя специфическую технологию.

Процесс разработки детальной модели программного комплекса называют внутренним (физическим, детальным), или ***техническим проектированием***

Создание физической модели гораздо более сложный процесс, чем создание логической.

Во время проведения объектно-ориентированного анализа строится модель требований, а в результате проектирования, опирающегося на требования, строится, согласованная по требованиям ***модель реализации*** в рамках выделенных ресурсов и времени.

4.3. Объектно-ориентированный анализ (ООА) и объектно-ориентированный дизайн (ООД)

Объектно-ориентированный анализ - это методология, при которой требования к программе формулируются с точки зрения объектов предметной области, вовлекаемых в решение задачи, соответствующей предназначению программы.

Объектно-ориентированный дизайн (проектирование) - это методология, соединяющая в себе объектную декомпозицию цели разработки и методы представления объектно-ориентированных моделей проектируемой программы (комплекса, системы) или синтеза программы с точки зрения структурной ее организации.

Поскольку язык UML является концептуально полным и универсальным языком, то такие графические нотации, как ERD (диаграммы «сущность-связь», средства ARIS, ERwin), SADT (диаграммы функционального моделирования) и DFD (диаграммы потоков данных), а так же языки, возникшие на базе SATD языки IDEF0 и IDEFIX (стандарт FIPS, средство BPwin) рассматриваться не будут. Причина – отсутствие объектной ориентации.

Глава 5. Использование Java для создания пользовательских классов

5.1. Создание пользовательских классов

Для того, чтобы написать приложение придётся запрограммировать работу объектов, описывающих предметную область. Выделять объекты и проектировать структуру класса мы научились, теперь нужно взяться за

дело и написать пользовательские классы, т.е. свои собственные. Для того, чтобы написать пользовательский класс, а это означает написание класса самим программистом, а не воспользоваться уже готовым классом, например библиотечным. Необходимо прежде всего написать определение класса, затем определить атрибуты, после этого написать методы класса, а затем уже использовать экземпляры класса, после всего этого нужно ещё написать класс тестер или тестирующий класс, с помощью которого можно запускать приложение. Но это ещё не все, нужно написать конструкторы. А ещё необходимо поразмыслить, возможно нужно продумать работу с несколькими экземплярами типа пользовательский класс, который мы создали, и возможно их нужно где-то будет хранить. Возможно нам понадобится контейнерный класс из классов – коллекций Java.

Нужно помнить о том, что приступая к написанию кода программист должен изучить конвенцию кода по языку Java и впредь её придерживаться, для того чтобы программа была удобочитаемой и понятной не только самому “писателю”, но и возможным “читателям”. Вообще же существует такое понятие как хороший стиль программирования. В кратком изложении конвенция кода гласит:

- название класса должно начинаться с заглавной буквы, например Customer, TaxSalary и т.п.;
- Имена атрибутов или полей данных класса должны начинаться с прописной буквы и представлять собой содержательные названия описываемого атрибута объекта, возможно составленные из нескольких слов, объединённых в одну строку firstName, secureCode, hardDiskCapacity или number;
- Названия методов должны начинаться тоже с прописной буквы и образовываться также как и имена у атрибутов, но должны содержать глаголы, поясняющие, что метод делает, например, getPhoneNumber(), createNewDesktop() или getPrice().

5.2. Класс, его структура и определение в Java программе (использование конвенции по созданию кода на Java)

Любая программа на Java это фактически набор классов. Объявление класса состоит из заголовка и тела класса. Заголовок класса включает служебное слово *class* и имя класса. Имя строится по правилам создания идентификаторов (имён) для данного языка.

```
class имя_класса {  
    модификатор доступа:  
    тип_переменной имя_переменной; // поле данных класса  
    тип_возвращаемого_значения имя_метод(список_параметров);  
}
```

Например:

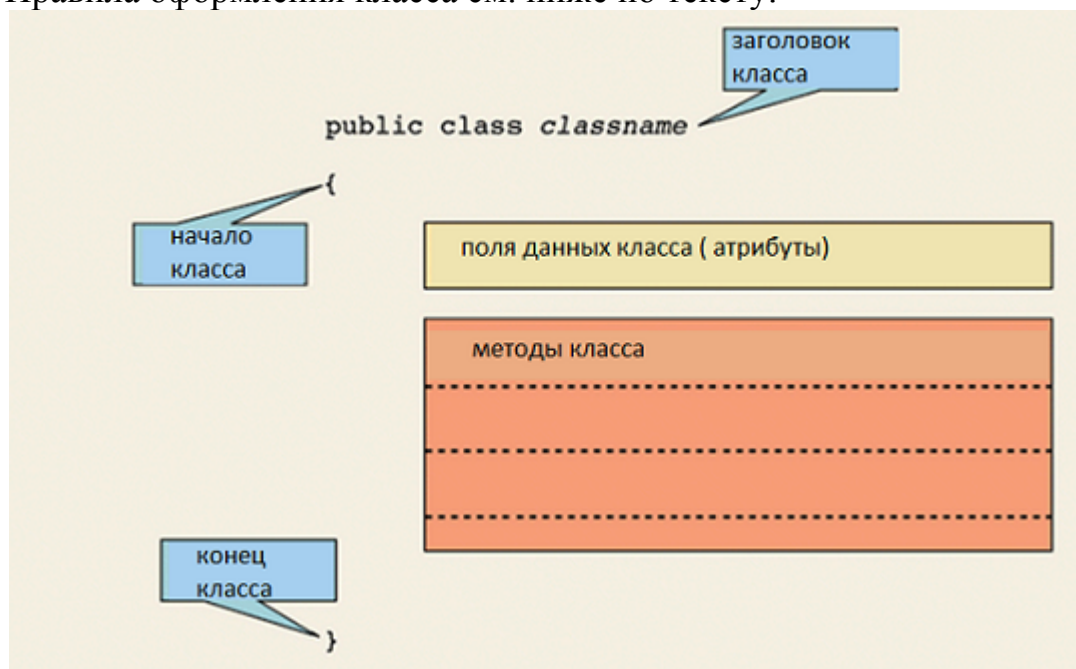
```
class Human {
private:
int age;
public:
int getAge ();
}
```

Данный Java код представляет класс. Название класса должно отражать некоторые характеристики объекта, который идентифицирует класс.

Модификаторы доступа используются для разграничения доступа к полям данных и методам класса, а кроме того к самим классам.

```
public class Human {
    public int age;
}
```

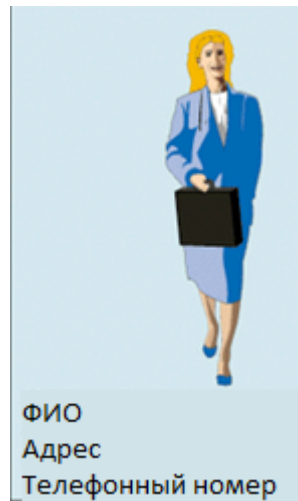
Правила оформления класса см. ниже по тексту.



Сначала выбираем название, затем определяем необходимые атрибуты, потом добавляем необходимые методы. Тело класса заключается в фигурные скобки.

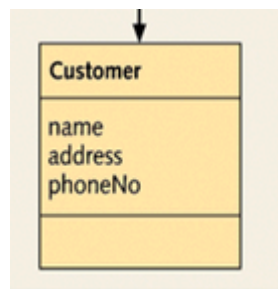
5.3. Определение атрибутов класса - полей данных класса

Допустим при проектировании программы мы выделили объект *Заказчик* с атрибутами *ФИО*, *адрес*, *телефонный номер*.



Поскольку заказчиков может быть много, но у всех есть общие характеристики, которые объединяют эти объекты в один класс объектов.

Соответственно получаем диаграмму класса:



Кроме того, нужно определить права доступа к полям данных класса.

Разграничение прав доступа(уровень доступа - свойства)

В Java модификаторы доступа используются для:

1. Типов (классов и интерфейсов, объявления верхнего уровня)
2. Элементов ссылочных типов (полей, методов, внутренних типов)
3. Конструкторов классов

Все 4 уровня доступа имеют только элементы типов и конструктор

(1,3)

- ***public***
- ***private***
- ***protected***
- ***default*** (если не указан ни один из трех)// по умолчанию

Модификатор ***protected*** – может быть указан для наследника из другого пакета, а доступ по умолчанию позволяет обращение из классов не наследников, если они находятся в том же пакете

Модификаторы доступа, которые возможны для различных элементов языка:

- ***Пакеты*** всегда доступны => нет модификатора доступа у них, могут быть вызваны из у точки программы

- Типы (**классы и интерфейсы**) верхнего уровня объявления. Можно указать либо `public` либо не указывать (то это значит, что он доступен внутри пакета `package`, в котором он объявлен);
- **Массив** имеет тот же уровень доступа, что и тип, на основании которого он объявлен;
- **Элементы и конструкторы объектных типов**;

В примере ниже, все элементы интерфейса являются `public`:

```
public class Wheel {
    private double radius ;
    public double getRadius ()
        {return radius;
        }
}
```

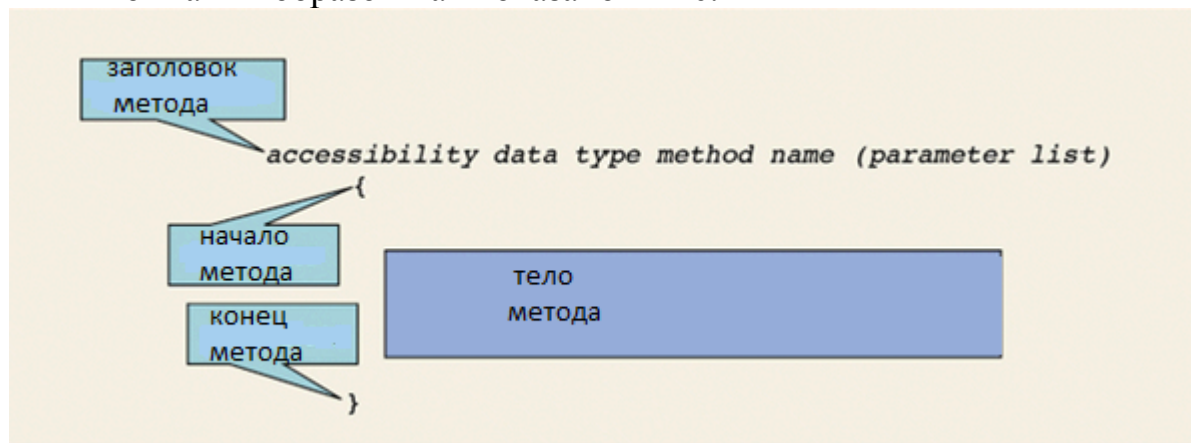
5.4. Написание методов класса

Метод состоит из заголовка и тела, которое заключается в парные фигурные скобки.

Заголовок метода состоит из четырёх частей:

- **Спецификатор доступа:** *public, private, или protected (accessibility)*
- **тип данных:** *void* или тип данных, возвращаемого значения (*data type*)
- **имени метода (method name)**, в соответствии с конвенцией
- **список формальных параметров**, т.е. переменные, объявленные в соответствии с типами аргументов (*parameter list*)

Вот таким образом как показано ниже:



5.5. Методы для получения доступа к полям класса (методы—«аксессоры»)

Для класса также нужно написать, так называемые методы «аксессоры», от слова *access*, что в переводе с английского означает доступ. Суть этих методов в том, чтобы получить доступ к полям данных

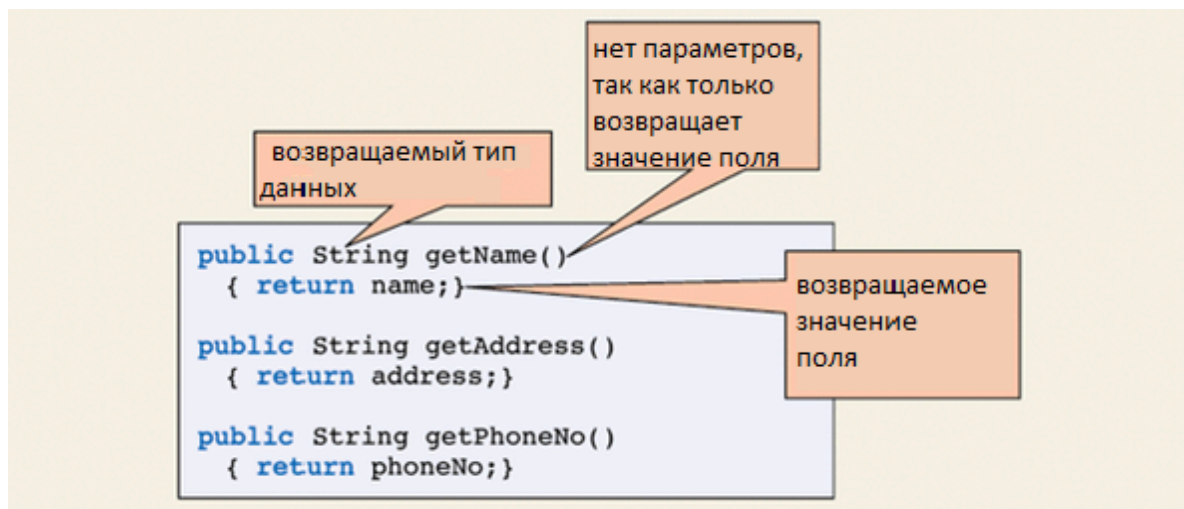
класса. Их ещё называют стандартными методами. Всего таких методов два вида:

- **get** метод (*getters*)

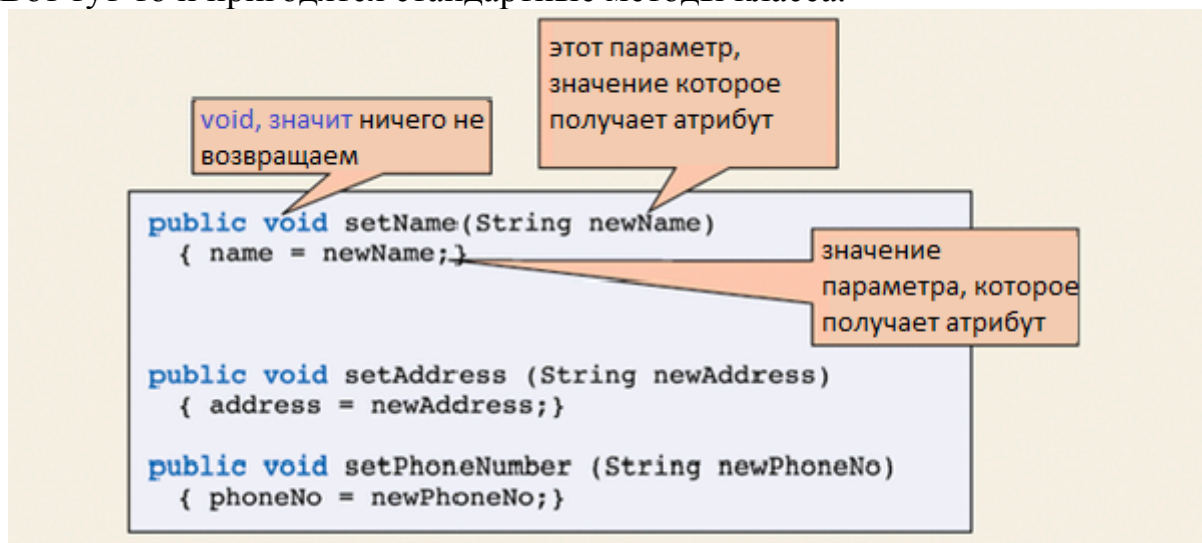
- в названии этого метода присутствует префикс называется “get”, за ним следует имя поля (атрибут), возвращает

- **set** метод (*setters*)

- В названии этого метода присутствует префикс называется “set”, за ним следует имя поля(атрибут), у которого изменяем значение

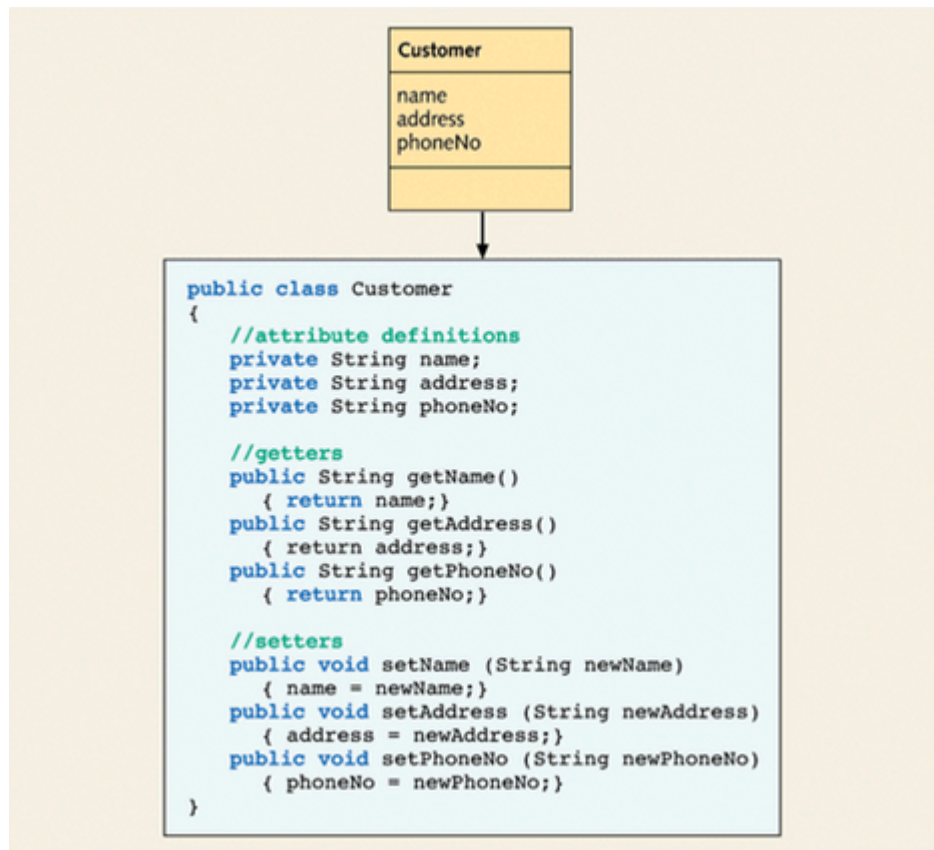


Для чего же нужны дополнительные методы? Все дело в том, что один из принципов ООП – инкапсуляция, если к полю данных закрыт доступ (*private*)? То, каким образом, получить к этому полю доступ извне класса, считать или записать значение в переменную поле данных класса. Вот тут-то и пригодятся стандартные методы класса.



Метод *get()* и используется, чтобы считать значение поля данных класса, поэтому он возвращает значение, а метод *set()* используется наоборот, чтобы записать некоторое значение в поле данных класса,

поэтому у него есть формальный параметр такого же типа как и поле данных для записи, но он ничего не возвращает(возвращает *void*).



5.6. Создание экземпляров объектов типа класс

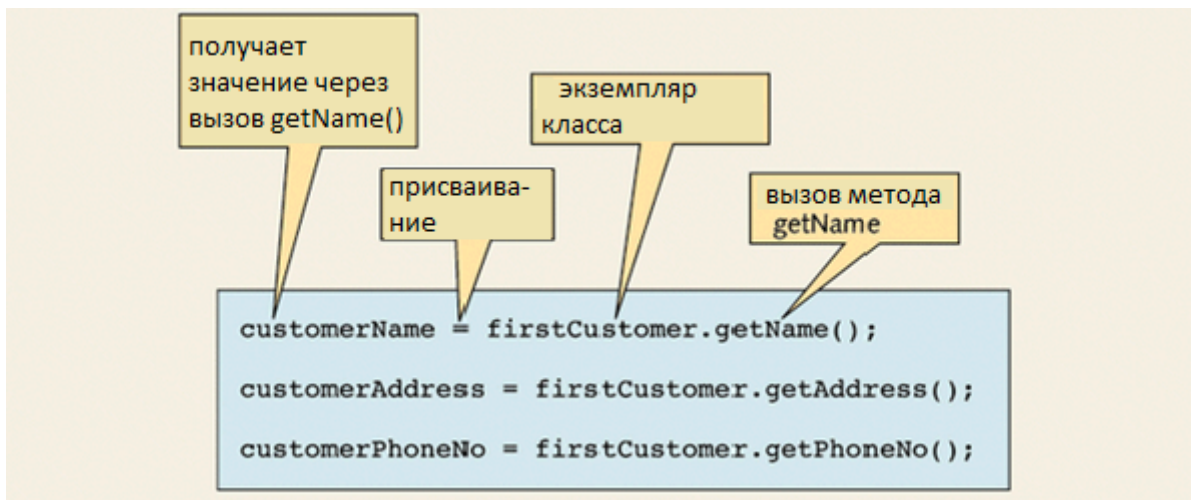
Для создания экземпляров класса необходимо:

- определить переменные – ссылки;
- специфицировать тип данных;
- Использовать оператор *new* для создания экземпляра класса.

Так как оператор *new* возвращает ссылку на объект.

Например:

```
Customer firstCustomer = new Customer();
```



5.7. Использование ссылочных переменных

Когда объявляются переменные примитивных типов, не объекты, то в памяти отводится место, где значения такого типа могут храниться. Например:

```
int myVar1;
int myVar2;
myVar1 = 25;
myVar2 = myVar1;
```

В результате выполнения фрагмента кода мы имеем две разных переменных в разных областях памяти, но с одинаковым значением. Работа с объектными ссылками происходит совсем по-другому.

Вот ещё пример. Допустим есть класс:

```
class Cup{
    private double volume;
    .....
    Cup(){} // \пустой конструктор
    Cup(double vol) // конструктор с параметрами
    {volume =vol;
    }
}
```

В программе программист написал такой фрагмент кода:

```
.....
Cup mycup;
Cup yourcup;
mycup = new Cup(0,25);
yourcup = mycup;
.....
```

Что произойдёт в результате выполнения этого фрагмента кода. Будет создан только один объект, так как мы только раз вызывали конструктор, но существуют две ссылки на объект, и обе этих ссылки относятся к этому объекту.

Если бы мы два раза вызвали конструктор, то создали бы два объекта. Например:

```
тусир = new Сир(0,25);
```

```
тусир = new Сир(0,5);
```

Но ссылка будет по-прежнему одна, так как имя у нас одно и то же, по правилам ссылка не может относиться к двум объектам одновременно, в результате, когда создается второй объект, то первый утрачивает ссылку.. когда такое случается, то объект все ещё существует в программе, но программист его уже использовать не может, так как программа потеряла над ним контроль.. В этом случае через некоторое время его подберёт Garbage Collector Java.

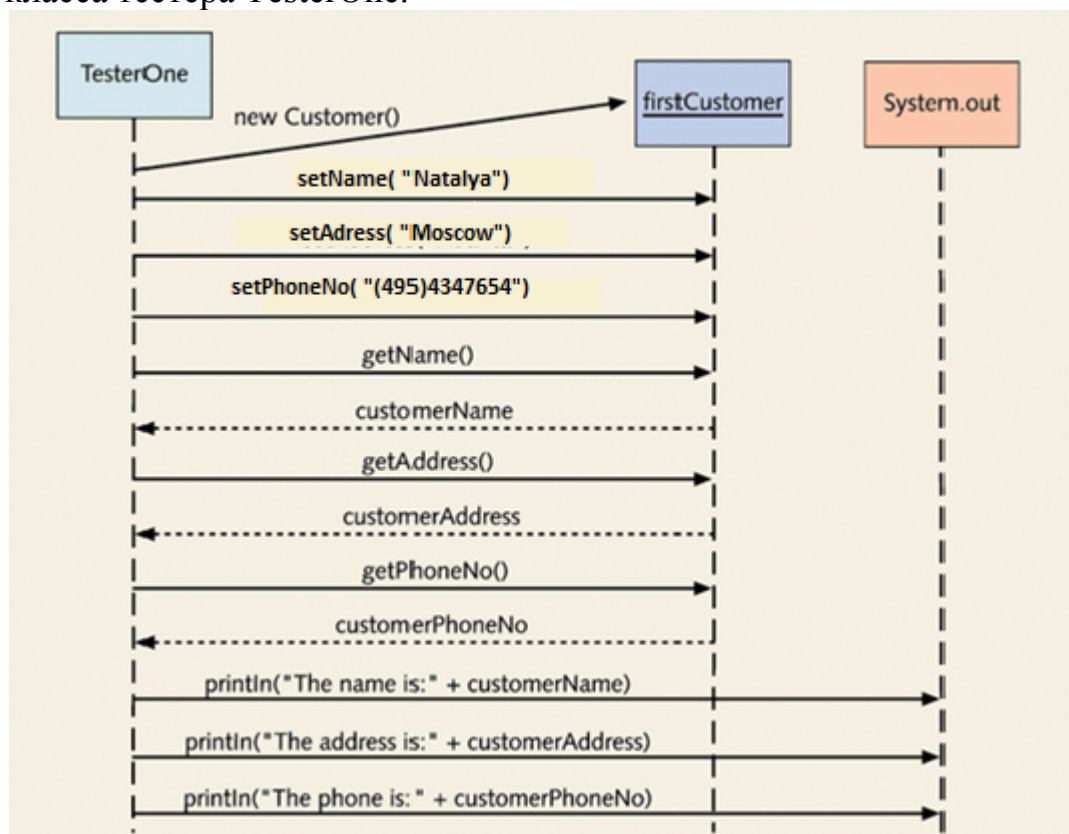
5.8. Написание класса-тестера

Для тестирования работы созданного класса используется подход, который получил название “клиент-сервер”. Тестирующий класс или по-другому класс-тестер используется, чтобы эмулировать работу “клиента”, который посылает сообщения “серверу”, чтобы протестировать собственные операции методами сервера.

Суть тестирования состоит в том, что в методе main, который создается экземпляр класса клиента и затем, вызываются его методы.

5.9. Тестирование пользовательских классов

На диаграмме последовательностей представлена схема работы класса-тестера TesterOne.



Результат работы класса тестера представлен ниже

```
The name is Natalya  
The sdress is Moscow  
The phone is (495)4347654
```

Смысл тестирования в том, что логика класса может быть проверена только программой которая им пользуется. Полная тестовая программа должна использовать все методы и конструкторы, определённые в классе, и выдавать на экран текстовое сообщение (аналог трассировки программы при процедурном программировании), которое позволит разработчику определить все ли правильно сделано.

5.10. Работа с несколькими экземплярами объектов типа класс

В процессе работы приходится, как правило, создавать несколько экземпляров типа класс. Но каждый раз при создании объекта вызывается конструктор.

Например, для нашего примера с чашками:

```
.....  
Cup cup;  
for( int count; count < 10; count++)  
    cup = new Cup();  
.....
```

Итак, с помощью конструктора и нехитрого цикла мы создали три объекта типа чашка. Объекты мало создать, их где то ещё надо хранить, о об этом поговорим поподробнее, когда будем обсуждать коллекции.

5.11. Создание и инициализация объектов типа класс

5.11.1. Написание методов–конструкторов

Методы - конструкторы, это специальные методы класса, которые позволяют конструировать объект типа класс и размещать его в памяти. Они автоматически создают экземпляры объектов типа класс с помощью вызова оператора new, столько раз, сколько раз вызывается сам конструктор.

Правила написания конструкторов таковы:

- Название конструктора совпадает с именем класса
- Они не возвращают значения (в отличие от методов, даже void)

5.11.2. Конструкторы по умолчанию

Java сама создает конструкторы по умолчанию, если программист не написал свой собственный конструктор в классе. Код для такого конструктора будет не виден, но тем не менее, его можно использовать.

Например:

```
public Customer() { }
```

Программист также может создать свой собственный пустой конструктор. Конструкторы как правило используются для инициализации полей данных класса, чтобы не получить о время выполнения программы некоторое непредвиденное значение. В нашем примере конструкторы должны инициализировать значения полей данных нашего класса Customer пустыми строками.

5.11.3. Конструкторы с параметрами

Кроме пустых конструкторов используются также конструкторы с параметрами или параметризованные конструкторы.

У этих конструкторов, в отличие от пустых конструкторов есть список параметров (как у метода) . В параметризованных конструкторах список параметров получает аргументы инициализации атрибутов экземпляра класса с помощью одного из способов:

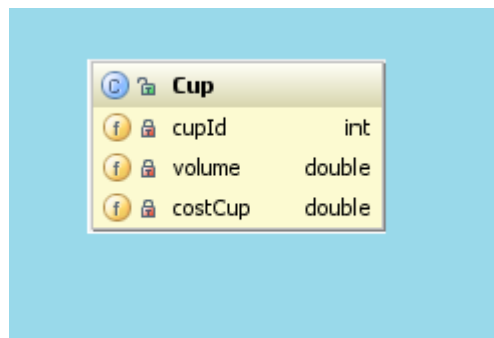
- Вызова стандартных методов класса – аксессоров или мутаторов¹ для каждого экземпляра класса;
- Прямой инициализации значения поле й данных экземпляра класса.

Глава 6. Написание пользовательских методов

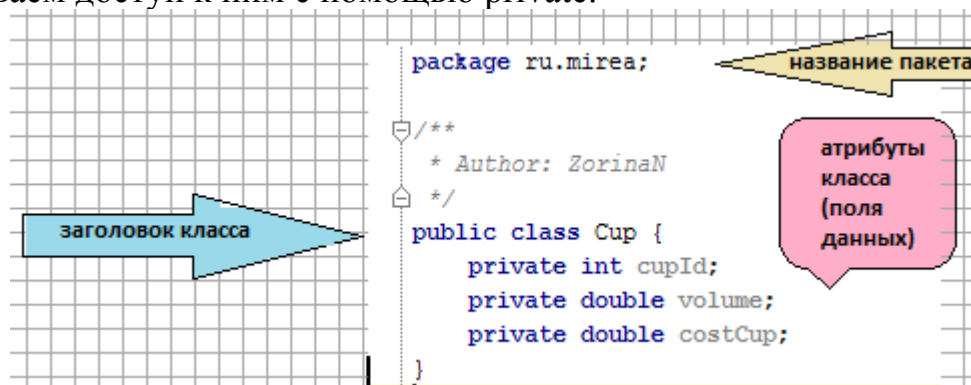
Итак, в этой главе поподробнее о том, как писать пользовательские методы. Для начала разберём пример, допустим, у нас есть некий склад, где хранится посуда в виде чашек. Нужно заметить, что данный пример, конечно, утрированный, но до наследования мы пока не добрались, а нам нужны объекты именно одного класса. Для начала нужно охарактеризовать объект предметной области *Чашка*, выделить существенные для нас характеристики объекта и записать их в виде атрибутов – полей соответствующего класса. У Чашки есть ёмкость, в виде литров, это значение дробное, например 0,25 литра, у чашки есть стоимость это рубли, и нам понадобится некий идентификатор, чтобы знать количество товара на складе, т.е. каждой чашке присвоим уникальный номер, это будет целое.

Вот так будет выглядеть диаграмма нашего класса Cup:

¹ Так называются в англоязычной литературе методы геттеры и сеттеры класса

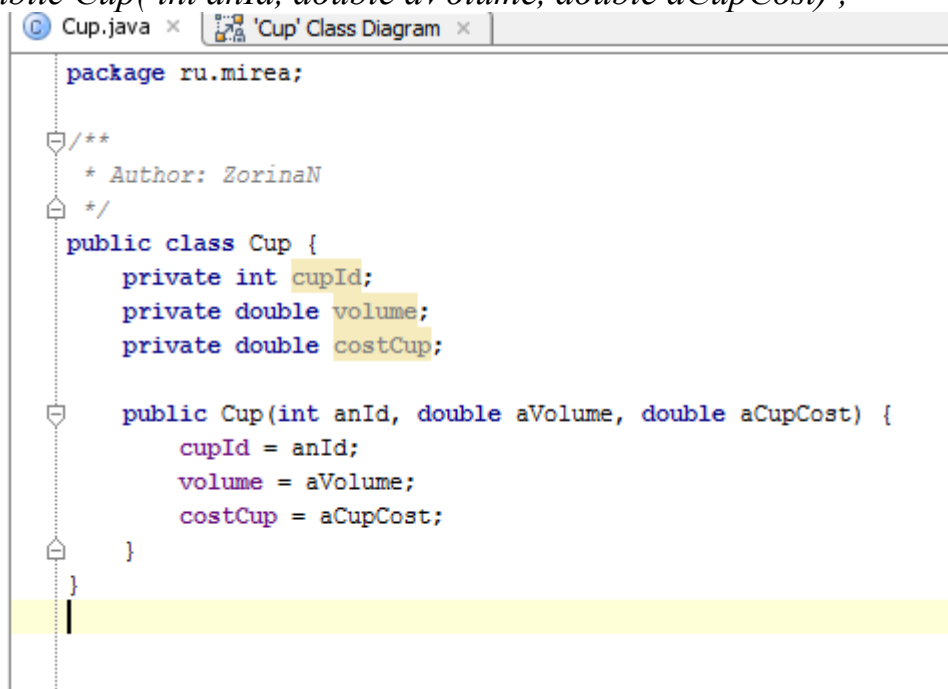


Теперь приступим к написанию кода на Java. Класс назовём Cup, все поля данных инкапсулированы внутри класса, соответственно ограничиваем доступ к ним с помощью `private`.

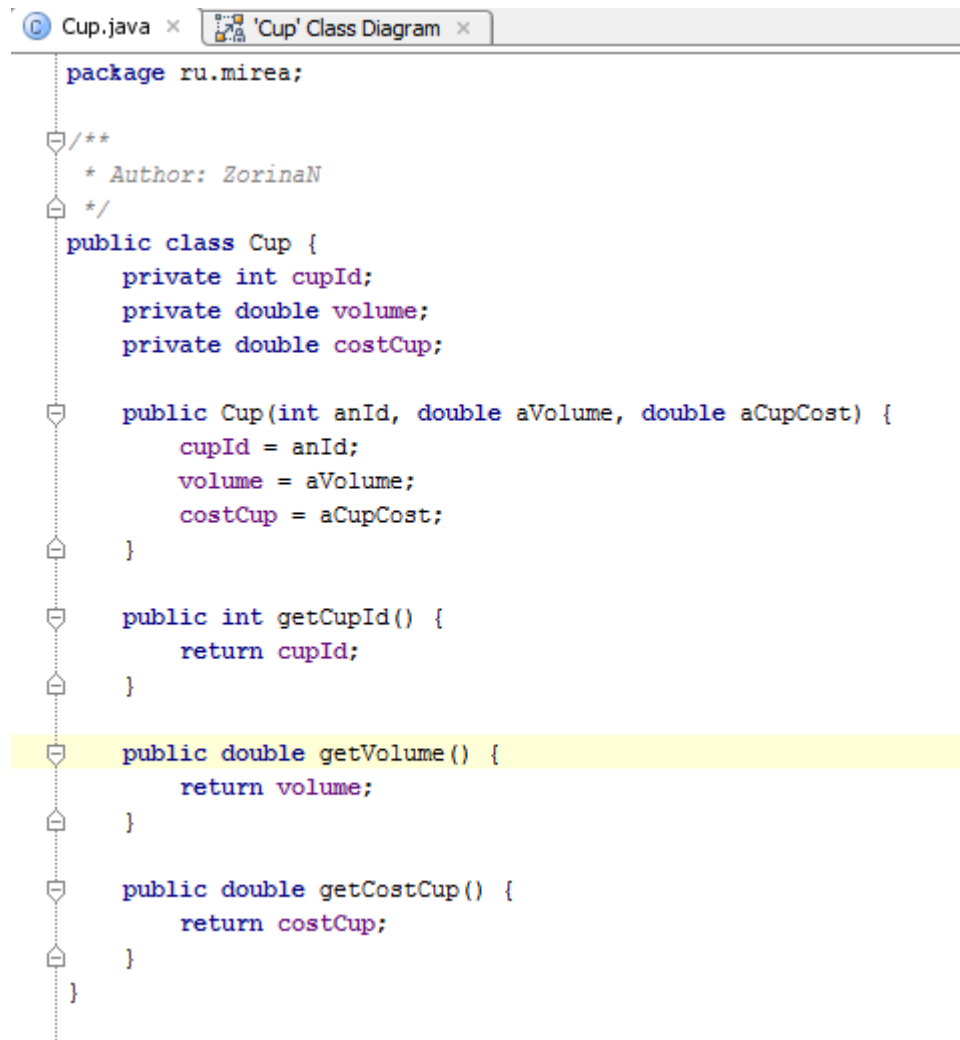


Нужно предусмотреть инициализацию полей данных, для этого добавим параметризованный конструктор.

public Cup(int anId, double aVolume, double aCupCost) ;



Теперь нужно написать стандартные методы класса, а именно методы аксессоры (конструктор) мы уже написали, так как мы закрыли поля данных с помощью private.



The screenshot shows an IDE with two tabs: 'Cup.java' and 'Cup' Class Diagram. The 'Cup.java' tab is active, displaying the following Java code:

```
package ru.mirea;

/**
 * Author: ZorinaN
 */
public class Cup {
    private int cupId;
    private double volume;
    private double costCup;

    public Cup(int anId, double aVolume, double aCupCost) {
        cupId = anId;
        volume = aVolume;
        costCup = aCupCost;
    }

    public int getCupId() {
        return cupId;
    }

    public double getVolume() {
        return volume;
    }

    public double getCostCup() {
        return costCup;
    }
}
```

The 'Cup' Class Diagram tab is also visible, showing a class diagram for the 'Cup' class.

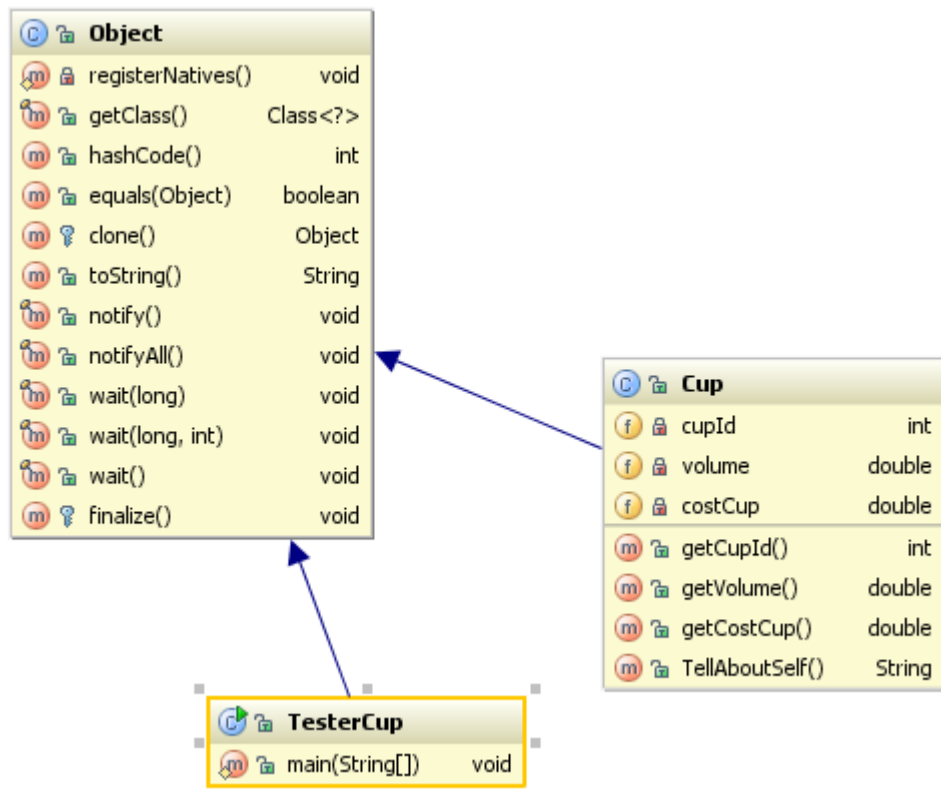
Теперь напишем класс тестер, чтобы проверить, как работает наш класс.

```
public String TellAboutSelf()
{
    String ss;
    ss = "my Id =" + cupId + "my volume = " + volume + "my cost = " + costCup;
    return ss;
}

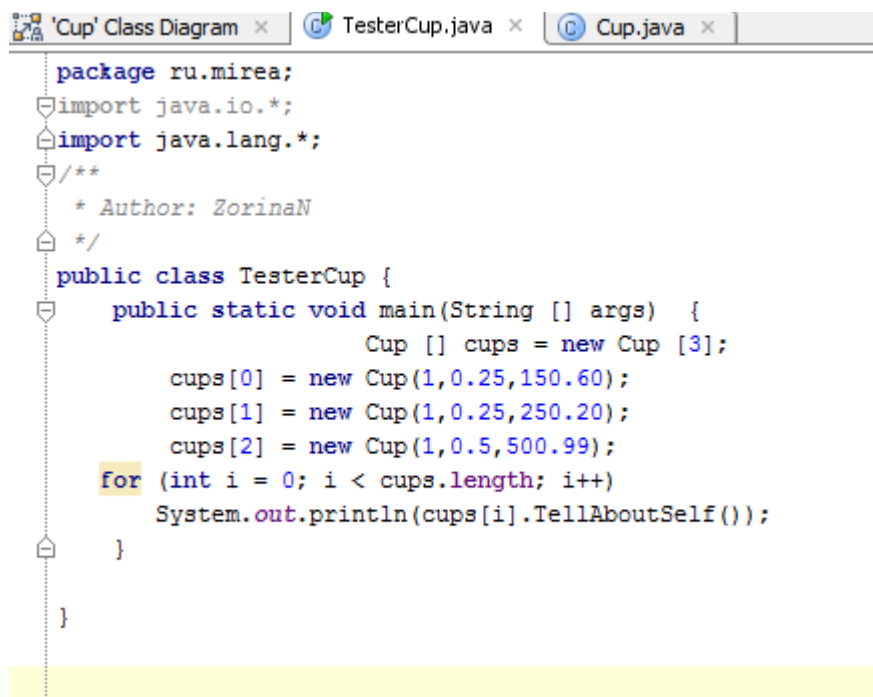
}
```

На диаграмме классов видно, что наше приложение состоит из двух классов, оба они наследуют класс Object, который находится на самой

вершине иерархии классов Java. Это класс по умолчанию предок любого класса и кроме того предоставляет в наше распоряжение свои методы.



Тестирующий класс **TesterCup** имеет только один метод – `main` и с помощью него запускает наше приложение. Каждый класс находится в отдельном файле с расширением `.java`. Классы могут работать вместе и имеют общую область видимости, так как находятся в одном пакете `ru.mirea`.



```
package ru.mirea;
import java.io.*;
import java.lang.*;
/**
 * Author: ZorinaN
 */
public class TesterCup {
    public static void main(String [] args) {
        Cup [] cups = new Cup [3];
        cups[0] = new Cup(1,0.25,150.60);
        cups[1] = new Cup(1,0.25,250.20);
        cups[2] = new Cup(1,0.5,500.99);
        for (int i = 0; i < cups.length; i++)
            System.out.println(cups[i].TellAboutSelf());
    }
}
```

Итак, подведём итог: стандартные методы класса пишутся, чтобы сохранять и возвращать значения:

- конструкторы;
- аксессоры (get-теры и set-теры);
- перегруженный метод `toString()` или так называемый `tellAboutSelf` метод. (Единственный метод, возвращающий строку, которая содержит все поля данных класса и их значения)

6.1. Пользовательских методы, их написание и использование

Пользовательские методы класса пишутся, для того чтобы реализовать какой-либо процесс обработки.

Порядок написания таков:

- Пишется заголовок метода (его описание), где определяется:
 - модификатор видимости (обычно *public*)
 - тип возвращаемого значения в соответствии с возвращаемым типом данных методом
 - список формальных параметров с указанием типа или только типы (в описании), если метод вызывается с параметрами
- пишется определение метода – реализация какого-либо процесса
- пишется тестирующий класс, для проверки работы метода

Напишем пользовательский метод для класса `Cup` из нашего примера. Метод будет устанавливать скидку в зависимости от товара (типа чашки).

На рисунке ниже мы добавили в диаграмму классов класса `Cup` метод `public double countTax()`.

Cup		
(m)	getCupId()	int
(m)	getVolume()	double
(m)	getCostCup()	double
(m)	TellAboutSelf()	String
(m)	countTax()	double

А вот определение метода countTax(), метод возвращает размер скидки, в зависимости от атрибута cupId.

```

    public double countTax() {
        double tax;
        switch(cupId) {
            case 0: tax = 0.05;
                    break;
            case 1: tax = 0.1;
                    break;
            case 2: tax = 0.25;
                    break;
            default : tax = 0;
        }
        return tax;
    }
}

```

Самое время протестировать новый пользовательский метод нашего класса. Для этого подправим код тела цикла for (). Название для переменной можно оставить tax, так как это локальная переменная.

```

for (int i = 0; i < cups.length; i++)
{
    double tax = cups[i].countTax();
    System.out.println("Cup's Id = " + cups[i].getCupId() + "it costs : " + cups[i].getCostCup()*tax );
}

```

Таким образом, мы узнаем стоимость каждой единицы товара с учётом скидки. Конечно решение не очень изящное, но вполне иллюстративное.

Вот, результат работы программы:

```

"C:\Program Files\Java\jdk1.6.0_22\bin\java" -Didea.launcher.port=75:
Cup's Id = 1it costs :15.06
Cup's Id = 1it costs :25.02
Cup's Id = 1it costs :50.0990000000000004

```


6.2. Форматирование числовых данных на экране

Для форматирования числовых данных используется класс **NumberFormat**, для работы с ним нужен пакет с названием **java.text**.

Его методы обеспечивают форматирование числовых данных с использованием специальных знаков - запятой, знака доллара и десятичной точки. Они также обеспечивают форматирование текущих форматов для различных стран (интернализация).

Два шага по форматированию данных:

- Вызвать метод *getCurrencyInstance()*, чтобы получить экземпляр класса *NumberFormat*

```
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance( );
```

- Вызвать форматирующий метод для полученного экземпляра класса

```
System.out.println( "Currency: " + currencyFormat.format( fee ) );
```

Для форматирования десятичных числовых данных используется класс **DecimalFormat**, для работы с ним нужен пакет с названием **java.text**.

Методы этого класса обеспечивают форматирование числовых данных, содержащих разделители в виде запятой и десятичной точки.

Два шага по форматированию данных:

- Создать экземпляр класса *DecimalFormat* , с использованием оператора *new* и *маски формата*

```
DecimalFormat decimalFormat = new DecimalFormat( "##,##0.00" );
```

- Вызвать форматирующий метод для полученного экземпляра класса

```
System.out.println( "Decimal: " + decimalFormat.format( tax ) );
```

Форматирование вывода

В Java как и в C используется форматирование вывода с использованием так называемых *escape* последовательностей. Эти последовательности могут состоять из одного или более специальных служебных символов. Суть их работы – управление выводом на экран. Отличительная особенность этих специальных символов, то, что они начинаются со знака обратный слэш (\) и не отображаются при выводе. Каждый эскейп символ имеет своё назначение, см Таблица 6.1.

Таблица 6.1. Некоторые эскейп символы

Эскейп символ	Обозначение
\n	новая строка
\r	возврат каретки
\'	апостроф
\"	двойная кавычка
\\	обратный слэш

\t	знак табуляции
----	----------------

6.3. Использование статических переменных и методов

Рассмотрим пример. Предположим, необходимо хранить количество всех людей (экземпляров класса *Human*, существующих в системе). Понятно, что общее число людей не является характеристикой какого-то одного человека, оно относится ко всему типу в целом. Отсюда появляется название "поле класса" в отличие от "поля объекта".

Объявление статических полей

Объявляются такие поля с помощью модификатора `static`:

```
class Human {
    public static int totalCount;
}
```

Обращение к статическому полю

Чтобы обратиться к такому полю, ссылка на объект не требуется, вполне достаточно имени класса:

```
Humans.totalCount++; // рождение еще одного человека
```

Для удобства позволяет обращаться к статическим полям и через ссылки:

```
Human h = new Human();
h.totalCount=100;
Рассмотрим ещё один пример:
Human h1 = new Human(), h2 = new Human();
Human.totalCount=5;
h1.totalCount++;
System.out.println(h2.totalCount);
```

все обращения к переменной `totalCount` приводят к одному единственному полю, и результатом работы такой программы будет 6. Это поле будет существовать в единственном экземпляре независимо от того, сколько объектов было порождено от этого класса, и были вообще создан хоть один объект.

Объявление статических методов

```
class Human {
    private static int totalCount;
    public static int getTotalCount() {
        return totalCount;
    }
}
```

Для **вызова статического метода** не требуется ссылки на объект.

```
Human.getTotalCount();
```

Пример обращения через ссылку:

```
Human h=null;
h.getTotalCount(); // Два эквивалентных обращения к
```

Human.getTotalCount(); // одному и тому же методу

обращения через ссылку позволяют, но принимается во внимание только тип ссылки. Хотя приведенный пример технически корректен, все же использование *ссылки на объект* для обращения к *статическим* полям и методам не рекомендуется, поскольку запутывает код

Кроме полей и методов статическими могут быть инициализаторы

```
class Human {  
    static {  
        System.out.println("Class loaded");  
    }  
}
```

Они также называются инициализаторами класса в отличие от инициализаторов объекта, рассматриваемых ранее. Их код выполняется один раз во время загрузки класса в память виртуальной машины. Их запись начинается с модификатора `static`:

```
class Test {  
    static int a;  
    static {  
        a=5;  
        // b=7; // Нельзя использовать до объявления!  
    }  
    static int b=a;  
}
```

Если объявление статического поля совмещается с его инициализацией, то поле инициализируется также однократно при загрузке класса. На объявление и применение статических полей накладываются аналогичные ограничения, что и для динамических тоже. Нельзя использовать поле в инициализаторах других полей или в инициализаторах класса до того, как это поле объявлено.

Это правило распространяется только на обращения к полям по простому имени. Если использовать составное имя, то обращаться к полю можно будет раньше (выше в тексте программы), чем оно будет объявлено:

```
class Test {  
    static int b=Test.a;  
    static int a=3;  
    static {  
        System.out.println("a="+a+", b="+b);  
    }  
}
```

Если класс будет загружен в систему, на консоли появится текст:
`a=3, b=0`

Видно, что поле `b` при инициализации получило значение по

умолчанию поля `a`, т.е. 0. Затем полю `a` было присвоено значение 3.

Статические поля и методы `final`

- Статические поля также могут быть объявлены как `final`, что означает, что они должны быть проинициализированы строго один раз, и затем уже больше не менять своего значения.
- Аналогично, статические методы могут быть объявлены как `final`, что означает, что их нельзя перекрывать в классах-наследниках.

Статические и динамические контексты полей

Нужно помнить о том что:

- для инициализации статических полей можно пользоваться статическими методами и нельзя обращаться к динамическим. Для этого вводят специальные понятия - статический и динамический контексты;
- к статическому контексту относят статические методы, статические инициализаторы, инициализаторы статических полей;
- все остальные части кода имеют динамический контекст;
- при выполнении кода в динамическом контексте всегда есть объект, с которым идет работа в данный момент.

Например, для динамического метода это будет объект, у которого он был вызван, и так далее.

Особенности использования статического контекста

Напротив, со статическим контекстом ассоциированных объектов нет. Например, как уже указывалось, стартовый метод `main()` вызывается в тот момент, когда ни один объект еще не создан. При обращении к статическому методу, например, `MyClass.staticMethod()`, также ни одного экземпляра `MyClass` может не быть. Обращаться к статическим методам класса `Math` можно, а создавать его экземпляры нельзя. А раз нет ассоциированных объектов, то и пользоваться динамическими конструкциями нельзя. Можно только ссылаться на статические поля и вызывать статические методы.

Пример обращения к объектам:

```
class Test {  
    public void process() {  
    }  
    public static void main(String s[]) {  
        // process(); - ошибка! у какого объекта его вызывать?  
        Test test = new Test();  
        test.process(); // так правильно  
    }  
}
```

Можно обращаться к объектам через ссылки на них, полученные в результате вызова конструктора или в качестве аргумента метода и т.п.

6.4. Написание перегруженных методов

Переопределение метода по-другому называется перегрузкой метода.

Сигнатура метода состоит из: названия метода, список параметров метода, не включая типа возвращаемого значения.

Java идентифицирует метод посредством его имени и сигнатуры.

Перегруженные методы это методы класса, которые имеют одинаковое название, но разные сигнатуры.

Нужно отметить, что перегруженные методы представляют проявления полиморфизма – одного из основных принципов ООП.

Допустим мы разрабатываем класс для работы с массивом чисел, и нам нужен пользовательский метод, который будет находить минимальное число. Для работы этого метода с целыми числами и с вещественными числами нужен один и тот же алгоритм, а вот возвращать метод будет разные типы, так как в одном случае `min` элемент целое число, а во втором дробное. Можно иметь два пользовательских метода, с одинаковым названием и одинаковым алгоритмом, но сигнатуры у методов будут разные. Такие методы называются перегруженными.

Перегруженными могут быть также конструкторы, так как один класс может иметь несколько конструкторов, имя у них будет одинаковое, это название класса, а вот сигнатуры разные.

Проявление такого полиморфизма называют слабым. Сильный полиморфизм требует виртуальных функций. Но об этом попозже.

Перепишем наш предыдущий пример:

```
public class Cup {  
    private static final double DEFAULT_VOLUME = 0.25;  
    private static final double DEFAULT_CUP_COST = 150.60;  
    public Cup( int anId ) {  
        this( anId, DEFAULT_VOLUME, DEFAULT_CUP_COST );  
    }  
    ...  
}
```

Вывод: любой метод может быть перегружен.

6.5. Проверка данных и генерирование исключений

Обработка исключений, это еще одна из причин по которой программист на JAVA может не только гордиться языком но и быть спокойным, по сравнению с программистом на C/C++. В JAVA можно перехватить совершенно любое исключение (за исключением падения виртуальной машины, но это уже другая история), а это значит, что программа становится еще немножко надежнее, стабильнее и адекватнее.

В программирование на C/C++ по Windows есть такое понятие, как SEH. Но оно к сожалению не всегда работает с конструкциями C++, а

разыменование нуля оно и в Африке - разыменование нуля. Все упало, все пропало. В “линуксах” дело обстоит немного похуже, видать сказалось, долгое поверье о избранности UNIX программистов. Но век великих «Нео» и «Морфиусов» прошёл и простые смертные стали касаться «комиссарского» тела. И что они увидели? Да практически никакой обработки исключений, не считая конечно сигналов (но это не спасет от «Segmentation fault»). Но это была прелюдия.

Все исключения JAVA, также как и обычные смертные классы наследуют Object, исключения наследуют базовый класс Exception. Для отслеживания исключений используется конструкция try catch (finally).

```
public static void main(String[] args) {
    try {
        double g = 10 % Integer.parseInt("0");
        System.out.println("Result: " + g);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

В этом примере мы немного обманули компилятор, конструкцией *Integer.parseInt*, чтобы можно было “свалиться” в runtime. Но не в этом суть. В конструкции *try catch* мы сможем “поймать” любую ошибку, что и произойдет, если мы соберем и запустим наш пример. Деление на ноль – банальная ошибка, но трудноуловимая в большом коде, если конечно везде не натывать меток и через каждые 2 строки не делать проверок на значение делителя. Вывод данного кода будет таков.

```
java.lang.ArithmeticException: / by zero
at ru.mirea.lecture.exceptions.ExceptionsSampleBase.main(ExceptionsSampleBase.java:7)
```

И что мы видим, о чудо, “умная” JAVA нам даже номер строки, где произошёл вызов ошибки написала. Это называется *Stack trace* (стек вызовов). Даже если не “ловить” ошибки, а они неизбежны, то JAVA все равно выдаст «лист» сделанных вызовов от начала main. Тем же самым собственно и занимается метод, с говорящим именем *printStackTrace*. Если же нам необходимо обрабатывать определенного рода ошибки, а не все, то мы смело можем записать конструкцию такого вида:

```
public static void main(String[] args) {
    try {
        double g = 10 % Integer.parseInt("0");
        System.out.println("Result: " + g);
    } catch (ArithmeticException ex) {
        System.out.println("Произошла ошибка: " + ex.getMessage());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Количеством catch, нас никто не ограничивает. А что делать, если мы хотим, например, уверенно в любой ситуации сделать какое-нибудь действие? Нам поможет конструкция finally.

```
public static void main(String[] args) {
    try {
        double g = 10 % Integer.parseInt("0");
        System.out.println("Result: " + g);
    } catch (ArithmeticException ex) {
        System.out.println("Произошла ошибка: " + ex.getMessage());
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        System.out.println("Эта конструкция выполнится при любой ситуации!");
    }
}
```

Эта инструкция хороша. Но применять ее нужно с толком. Например:

```
public static boolean illegalResult() {
    try {
        return true;
    } finally {
        return false;
    }
}
```

Как Вы думаете, что вернет этот метод, истину или ложь? По идее, должен вернуть истину, но на самом деле, вернет ложь. Так что — надо быть осторожнее.

Реализовывая какой-нибудь метод, вы можете с определенной долей уверенности сказать, что в нем может произойти ошибка, какая-нибудь конкретная или неизвестная. И если Вы хотите оповестить об этом другого программиста или просто, чтобы самим не забыть, для этого нужна такая конструкция:

```
public static boolean exception(String ex)
    throws Exception {
    return ex.isEmpty();
}
```

Замкнув определение метода ключевым словом throw и класс исключения, вы тем самым заставите вызывающего этот метод, обернуть вызов в ловушку исключений. Иначе, умный компилятор выдаст ошибку.

Глава 7. Подклассы и наследование

7.1. Создание иерархий классов, суперкласс и подкласс

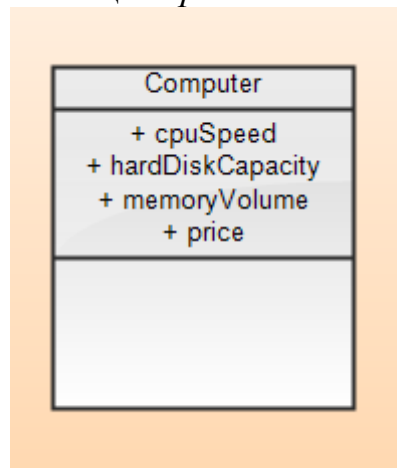
Наследование — один из основных принципов разработки ООП. Все классы в Java располагаются в иерархическом порядке. Непосредственно над каждым классом находится другой класс, который по отношению к нему называется родительским или *суперклассом*. Класс может иметь

больше одного класса ниже себя в иерархии. Класс, следующий сразу за некоторым классом, называется дочерним классом или **потомком**, а остальные классы ниже в иерархии называются **подклассами**. На самой вершине иерархии в Java стоит класс **Object**. О нём чуть-чуть попозже.

7.2. Использование ключевого слова языка Java - extends для реализации наследования с пользовательскими классами

Рассмотрим пример, нам нужно описать объект компьютер, с точки зрения потенциальных покупателей.

В объекте компьютер, что интересует потенциальных покупателей? Пожалуй, в первую очередь технические характеристики и конечно, цена. На диаграмме класса Computer мы указали следующие атрибуты: скорость ЦП *cpuSpeed*, емкость винчестера *hardDiskCapacity*, объем оперативной памяти *memoryVolume*, и конечно цена *price*.

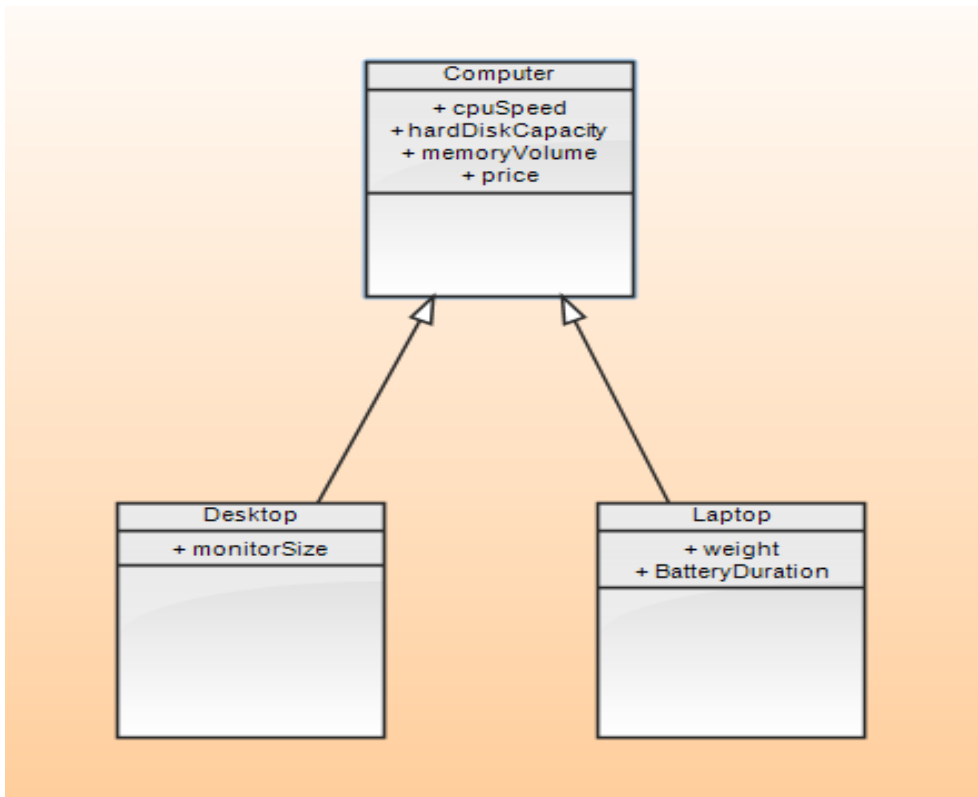


Наш класс включает четыре атрибута, нам нужен конструктор, который вызывает 4 стандартных метода set-тера, а всего стандартных методов аксессоров будет 8 (нам понадобятся ещё 4 get-тера).

Для того чтобы протестировать созданный нами класс, понадобится класс тестер.

Покупатель может быть заинтересован в покупке разных видов компьютеров, например одного клиента интересует настольный компьютер, а другого ноутбук. Оба эти вида компьютеров являются компьютерами, т.е. обладают всеми характеристиками как у разработанного нами класса, но в тоже время у каждого из них есть дополнительные характеристики, которые нужно учесть. Можно конечно спроектировать класс настольный компьютер и к перечисленным выше атрибутам добавить размер экрана, а для описания ноутбука спроектировать другой класс и добавить вес и минимум времени работы от батареи. Поскольку большинство атрибутов точно такие же как у нашего класса Computer, то можно использовать наследование, а именно при создании нового класса расширить существующий за счёт введения новых

атрибутов, то есть унаследовать от родителя часть характеристик и создать два класса потомка.



Класс потомок объявляется так

```
class имя_класса_родителя extends имя_класса_потомка {
```

атрибуты класса потомка

методы класса потомка

```
}
```

Для нашего примера код будет выглядеть так:

```
public class Desktop extends Computer {
```

```
    private Integer monitorSize;
```

```
    .....
}
```

```
public class Laptop extends Computer {
```

```
    private Double weight;
```

```
    private Integer batteryDuration;
```

```
    .....
}
```

Т.е. используя ключевое слово *extends*, мы создали два новых класса. Общий суперкласс включает атрибуты и методы, которые разделяются специализированными подклассами.

Результат, получаемый от наследования:

- Экземпляры подкласса наследуют атрибуты и методы суперкласса;
- Включение дополнительных атрибутов и методов, которые определяются в подклассе.

Чтобы наследовать:

- Нужно знать, как написан конструктор и сигнатуры методов;
- Не нужно иметь доступ к источнику кода и знать, как класс написан (его реализацию).

7.3. Использование конструкторов суперклассов при создании конструкторов подклассов

Конструкторы отличаются от обычных методов и в эти отличия проявляются и при использовании полиморфизма. Конструктор базового класса всегда вызывается при конструировании объекта производного класса. Вызов конструкторов происходит автоматически вверх по цепочке наследования. Так как производный класс обычно не имеет доступа к атрибутам суперкласса, которые обычно объявляются как *private*, поэтому только конструктор базового класса обладает всеми правами доступа, чтобы инициализировать атрибуты суперкласса, которые потомок наследует. Если вы явно не вызываете конструктор базового класса в теле подкласса, то компилятор подставит конструктор по умолчанию (обсуждался в предыдущей главе).

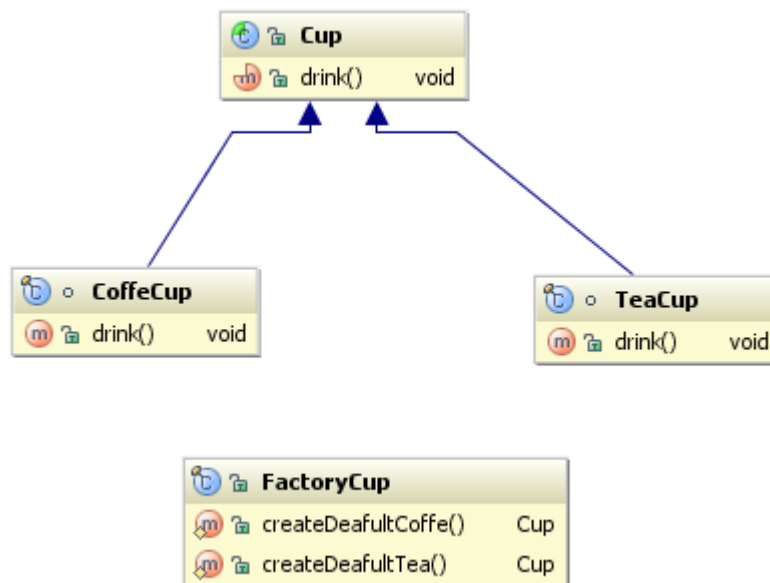
7.4. Использование ключевых слов *abstract* и *final* в суперклассах Java.

Абстрактными могут быть объявлены как классы, так и методы. В первом случае, если класс объявляется абстрактным, то в нем выделяются некие общие характеристики для объекта предметной области. Фактически такой класс создается только для конструирования других классов, т.е. для наследования, поэтому он и называется абстрактным. Поэтому нельзя создать экземпляры объектов типа абстрактный класс. Назначение такого класса объединить атрибуты и операции, которые будут общими у субклассов.

Для объявления абстрактного класса используется ключевое слово *abstract*. Абстрактный класс содержит абстрактные методы

```
public abstract Название_класса{  
    методы_класса;  
}
```

В нашем примере на UML диаграмме видно, что класс Cup наследуется двумя другими классами CoffeCup и TeaCup.



Мы его сделали абстрактным, и добавили в тело метод drink(), который является общим для двух subclasses. Метод сделали абстрактным, поэтому класс-потомок должен его переопределить.

```
package ru.mirea.cup;

public abstract class Cup {
    public abstract void drink();
}
```

Метод drink() выделен в абстрактный класс, так как можно пить как из кофейной чашки, так и из чайной, т.е. операция общая для двух объектов.

```
final class TeaCup
    extends Cup {
    public void drink() {
        System.out.println("I am drink tea.");
    }
}
```

Вот так мы его переопределили в subclasse Teacup.

```
final class CoffeCup extends Cup {
    public void drink() {
        System.out.println("Iam drink coffe.");
    }
}
```

А вот так в subclasse CoffeCup

```

package ru.mirea.cup;

public final class FactoryCup {
    public static Cup createDeafultCoffe() {
        return new CoffeCup();
    }

    public static Cup createDeafultTea() {
        return new TeaCup();
    }
}

```

Кроме того мы использовали ключевое слово ***final***, что значит окончательный и изменению не подлежит. Это значит, что от классов `TeaCup` и `CoffeCup` нельзя наследовать, т.е. эти типы данных окончательные и изменению не подлежат. Если класс объявлен `final`, то поля данных

Для того чтобы эффективно производить чашки мы использовали паттерн проектирования «фабрика». И создали отдельный специальный фабричный класс, который используется для создания объектов – чашек как кофейных, так и чайных.

```

package ru.mirea.cup;

public final class FactoryCup {
    public static Cup createDeafultCoffe() {
        return new CoffeCup();
    }

    public static Cup createDeafultTea() {
        return new TeaCup();
    }
}

```

Для запуска приложения используется отдельный класс `Main`, который содержит точку входа в программу - функцию `main`. В этом файле импортируются оба пакета, в том числе и пакет *`ru.mirea.cup`*, поэтому написанные нами классы видны

```

package ru.mirea;

import ru.mirea.cup.Cup;
import ru.mirea.cup.FactoryCup;

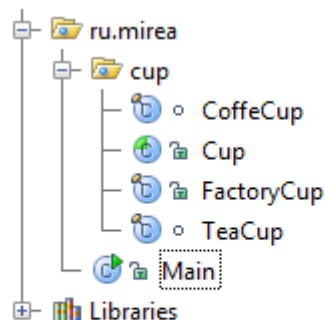
public class Main {

    private static final Cup[] CUPS = new Cup[]{
        FactoryCup.createDeafultCoffe(),
        FactoryCup.createDeafultTea(),
        FactoryCup.createDeafultCoffe()
    };

    public static void main(String[] args) {
        for (Cup CUP : CUPS) {
            CUP.drink();
        }
    }
}

```

Дерево проекта, на котором хорошо видна структура файлов.



7.5. Перегрузка методов суперкласса в подклассах

В чем еще главная прелесть наследования, так это то, что subclass наследует все открытые и защищенные методы и переменные базового класса. Он также может переопределять их.

```

public long рассчитатьМассу() {
    return super.рассчитатьМассу() + 10;
}

```

Мы совсем забыли массу самого ящика, т.е. если тара – воздушный шар – то это еще ничего, а если тара – свинцовый ящик, как тогда быть? Все очень просто, мы переопределяем метод рассчитать Массу в классе Ящик и прибавляем его свинцовую массу. Не правда ли легко? Не разрушая нашу стройную абстрактную систему мы решили сложную для C++

задачу (т.к. если Вы знаете в C++ можно переопределять только виртуальные методы, а если исходных кодов нет, то соответственно нет возможности поправить код – все, конец, надо будет писать ложную обертку и так далее по тексту, много шума из ничего).

7.6. Полиморфизм и дочерние классы в Java

Термин «полиморфизм» происходит от двух греческих корней «поли» и «форма», то есть много форм. Можно создать ссылку суперкласса на объект подкласса. Любая такая ссылка может относиться как к объектам своего класса, так и к объектам подклассов. Тип ссылки не указывает, на какой объект она ссылается. Предположим, что каждый класс в иерархии наследования каждый класс переопределяет метод. Теперь попробуем ответить на вопрос, если метод вызывается ссылкой суперкласса, то какую версию метода выполнит система. Какая именно версия метода будет вызвана зависит от ссылки, т.е. от фактического типа объекта на который указывает ссылка во время выполнения (а не тот который описан в коде). Если метод вызывается через ссылку суперкласса на объект подкласса, то будет выполняться версия метода, определенный в подклассе. Если же метод не был переопределён в подклассе, то вызывается соответственно унаследованный метод.

7.7. Последствия использования защищенного и закрытого доступа в суперклассах

Закрытый доступ (private):

- не существует других объектов, которые могут прямо прочитать или модифицировать значения атрибутов (значения инкапсулированы)
 - для этого обязательно нужно использовать методы-аксессоры, чтобы получить доступ к приватным переменным (private) суперкласса
- Также ограничены возможности subclasses для получения прямого доступа к приватным (private) переменным
 - Нужно использовать методы аксессоры.

Защищённый доступ (protected):

- Субкласс имеет прямой доступ к приватным переменным суперкласса
 - Другие классы в том же самом пакете имеют прямой доступ так же как он
- Можно также использовать методы
- При использовании нужно быть внимательным

Замечание

- Конечно можно помочь исправить ситуацию с помощью использования открытого доступа *public* к атрибутам, но тогда грубо нарушается принцип ООП об инкапсуляции данных и сокрытии информации

Глава 8. Абстрагирование, полиморфизм и интерфейсы

8.1. Использование абстрактных методов для перегрузки методов суперкласса в подклассе

Рассмотрим пример из некоторой предметной области, например представим, что нам поставлена задача - рассчитать перевозку грузов. В качестве тары решено использовать ящики.

```
public static abstract class Предмет {  
  
    public abstract long масса();  
}
```

Первым делом мы выделим предмет, который можно положить в тару, и исследуем его характеристики. Естественно у предмета есть своя собственная масса (габариты и форму не учитываем), поэтому метод масса мы сделали абстрактным, чтобы наследник этого объекта обязан был его переопределить.

```

public static abstract class Тара {

    private final Collection<Предмет> предметы = new Vector<Предмет>();

    public void добавитьПредмет(Предмет предмет) {
        предметы.add(предмет);
    }

    public long рассчитатьМассу() {
        long результат = 0;
        for (Предмет предмет : предметы) {
            результат += предмет.масса();
        }
        return результат;
    }

    public abstract long длина();

    public abstract long высота();

    public abstract long ширина();

    public long рассчитатьОбъем() {
        return длина() * высота() * ширина();
    }
}

```

Дальше определяем объект Тара. Опять же у каждой тары есть свои собственные параметры, поэтому мы их выделили, как абстрактные. Но в тоже время у каждой тары есть что-то общее – это в нее можно положить несколько предметов, рассчитать массу тары и рассчитать объем. Поэтому мы сразу же реализовываем эти методы.

```

public static abstract class Фрукт
    extends Предмет {

    public abstract Color цвет();
}

```

В качестве предмета, который мы будем описывать в классе мы взяли фрукты. У каждого фрукта есть свой уникальный цвет и таким образом мы можем расширить класс Предмет. Фрукты можно взвешивать, следовательно для каждого фрукта понадобится такая характеристика как масса. Таким образом, Фрукт является Предметом.


```

public static class Апельсин
    extends Фрукт {

    private final Color _цвет;
    private final long _масса;

    public Апельсин(Color _цвет, long _масса) {
        this._цвет = _цвет;
        this._масса = _масса;
    }

    public Color цвет() {
        return _цвет;
    }

    public long масса() {
        return _масса;
    }
}

public static class Груша
    extends Фрукт {

    private final Color _цвет;
    private final long _масса;

    public Груша(Color _цвет, long _масса) {
        this._цвет = _цвет;
        this._масса = _масса;
    }

    public Color цвет() {
        return _цвет;
    }

    public long масса() {
        return _масса;
    }
}

```

Для примера реализуем два фрукта, Апельсин и Груша. Все они фрукты, имеют свой цвет и массу. Т.е. наследуя Фрукт, Апельсин как бы говорит, что он таким и является, тем самым реализуя метод цвет, но в тоже время Фрукт является предметом, поэтому и Апельсин является предметом, тем самым реализуя метод масса. Подытожив вышесказанное: можно смело сказать, что каждый фрукт является предметом, каждый апельсин является фруктом и значит, каждый апельсин является предметом, но не наоборот. Тоже происходит и с грушей.

Что нам это дает? А дает это нам независимость. Т.е. мы можем работать с предметом, не зная, “что он фрукт” и в тоже время работать с фруктом,

не зная, “что это апельсин”. Из этого вывод – хорошая абстракция – это хорошо, но в меру, можно не переборщить и запутаться в классах.

Перейдем к непосредственной работе. Рассмотрим пример:

```
Тара тара = new Ящик(100, 100, 100);
Фрукт фрукт0 = new Апельсин(КРАСНЫЙ, 10);
Фрукт фрукт1 = new Груша(ЗЕЛЕНый, 10);
Фрукт фрукт2 = new Апельсин(ЗЕЛЕНый, 11);
тара.добавитьПредмет(фрукт0);
тара.добавитьПредмет(фрукт1);
тара.добавитьПредмет(фрукт2);
System.out.println("Масса тары: " + тара.рассчитатьМассу());
System.out.println("Объем тары: " + тара.рассчитатьОбъем());
```

Создаем тару объектом Ящик, с определенными габаритами. Берем тройку фруктов и добавляем их в тару. Потом можем смело узнать массу тары, ее объем и т.д. Мы не привязаны ни к определенному предмету, ни к определенной таре, ведь предметом могут быть гвозди, а тарой – урна. Нам без разницы, с чем работать.

8.2. Полиморфизм и интерфейсы

Поскольку в Java нет множественного наследования классов, а иногда бывает нужно создать тип, который представляет собой класс с характеристиками двух других разных типов, то для этого используются интерфейсы. То есть можно наследовать несколько интерфейсов. Вообще из самых важных принципов объектно-ориентированного проектирования программ подразумевает программирование на уровне интерфейсов, а не на уровне реализации класса. Такие программы легко сопровождать, и вносить в них изменения.

Существуют два способа, чтобы переопределить метод:

- Добавить абстрактный метод в суперкласс, который subclasses будут переопределять;
- Определить интерфейс в реализации класса.

Интерфейс это специальный компонент объектной модели Java, который определяет абстрактные методы и константы, которые должны включаться в реализацию классов.

Основная концепция интерфейса:

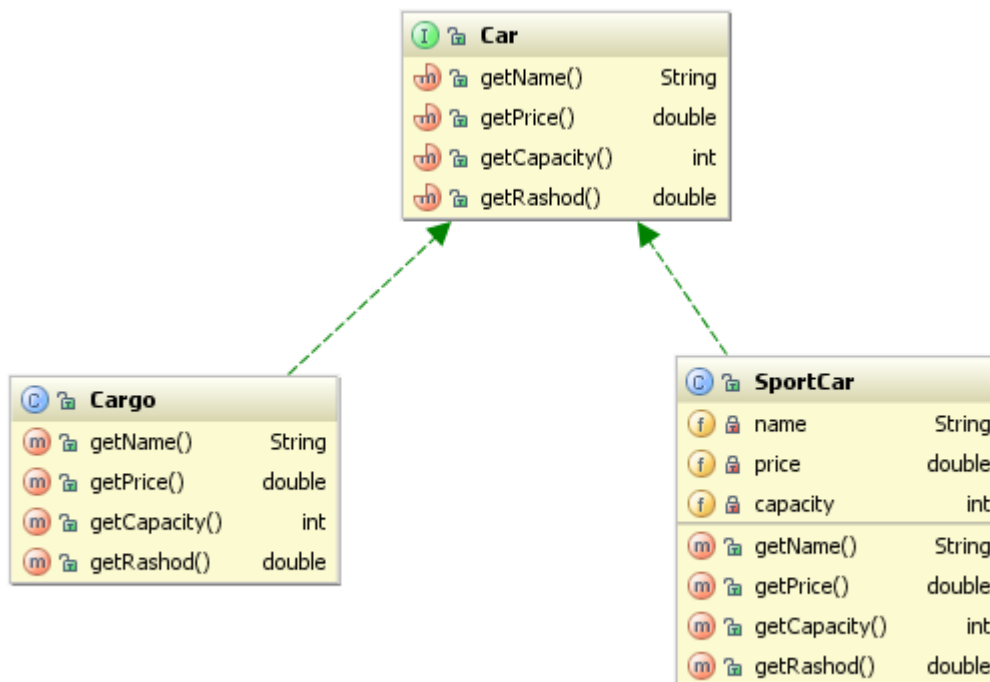
- Знание того как использовать что-либо означает знание как общаться с этим что-то;
- Методы объекта тип класс, которые могут отвечать на сообщения, могут быть определены как интерфейс класса;
- для объекта типа класс нужно определить набор методов, которые будут определять интерфейс, и в уже в классе реализовать его.

8.2.1. Создание и использование интерфейсов

Для того чтобы создать интерфейс, его необходимо для начала объявить. Это похоже на объявление класса.

Сначала создается заголовок с использованием ключевого слова *interface* вместо слова *class*, например:

```
public interface Название_интерфейса{  
    абстрактные_методы_интерфейса;  
}
```



Абстрактные методы не имеют реализации, их реализация помещается в классе, который наследует интерфейс. Говорят, что класс реализует некоторый интерфейс. Допустим нам нужно написать приложение для фирмы, которая занимается прокатом автомобилей. В прокат сдаются два вида автомобилей – легковые и грузовики. При проектировании предметной области были выделены общие методы взаимодействия и спроектирован интерфейс **Car**. В данном случае мы последовали основному принципу проектирования объектно-ориентированных программ, программировать через интерфейсы. В нашем примере, который представлен на UML диаграмме выше по тексту интерфейс **Car** имеет две реализации **Cargo** и **SportCar**.

Для создания реализации используется ключевое слово *implements*:

Приведённый ниже код демонстрирует что у нас есть три отдельных файла в одном пакете. Говорят, что *public class Cargo* реализует интерфейс **Car**.

```
package mirea.ru;
```

```
public interface Car {
```

```
String getName();

double getPrice();

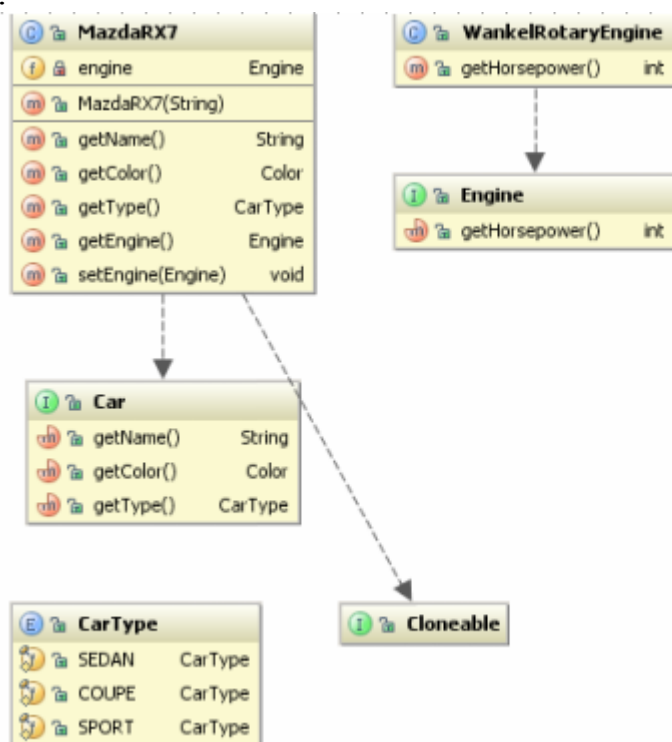
int getCapacity();

double getRashod();
}
package mirea.ru;

public class SportCar implements Car {
    private String name;
    private double price;
    private int capacity;

    public String getName() {
        return null;
    }
    public double getPrice() {
        return 0;
    }
    public int getCapacity() {
        return 0;
    }
    public double getRashod() {
        return 0;
    }
}
package mirea.ru;
public class Cargo implements car {
    public String getName() {
        return null;
    }
    public double getPrice() {
        return 0;
    }
    public int getCapacity() {
        return 0;
    }
    public double getRashod() {
        return 0;
    }
}
```

Классы в Java могут реализовывать больше чем один интерфейс, интерфейсы могут также включать абстрактные `final` переменные (константы).



8.2.2. Реализация интерфейсов классами через реализацию методов

Иногда бывает необходимо только *объявить* методы, которые должны поддерживаться объектом, без их конкретной реализации. До тех пор пока поведение объектов удовлетворяет некоторым критериям, подробности реализации методов оказываются несущественными. Например, если вы хотите узнать, входит ли значение в то или иное множество, конкретный способ хранения этих объектов вас не интересует. Методы должны одинаково хорошо работать со связным списком, хеш-таблицей или любой другой структурой данных.

Java использует так называемый **интерфейс** — некоторое подобие класса, где методы лишь объявляются, но не определяются. Разработчик интерфейса решает, какие методы должны поддерживаться в классах, *реализующих* данный интерфейс, и что эти методы должны делать. Приведем пример интерфейса `Lookup`:

```

interface Lookup {
    /** Вернуть значение, ассоциированное с именем, или
    null, если такого значения не окажется */
    Object.find(String name);
}

```

В интерфейсе `Lookup` объявляется всего один метод `find`, который получает значение (имя) типа `String` и возвращает значение, ассоциированное с данным именем, или `null`, если такого значения не найдется. Для объявленного метода не предоставляется никакой конкретной реализации — она полностью возлагается на класс, в котором реализуется данный интерфейс. Во фрагменте программы, где используются ссылки на объекты `Lookup` (объекты, реализующие интерфейс `Lookup`), можно вызвать метод `find` и получить ожидаемый результат независимо от конкретного типа объекта:

```
void processValues(String[] names, Lookup table) {  
    for (int i = 0; i < names.length; i++) {  
        Object value = table.find(names[i]);  
        if (value != null)  
            processValue(names[i], value);  
    }  
}
```

Класс может реализовать произвольное количество интерфейсов. В следующем примере приводится реализация интерфейса `Lookup` для простого массива (мы не стали реализовывать методы для добавления или удаления элементов):

```
class SimpleLookup implements Lookup {  
    private String[] Names;  
    private Object[] Values;  
    public Object find(String name) {  
        for (int i = 0; i < Names.length; i++) {  
            if (Names[i].equals(name))  
                return Values[i];  
        }  
        return null; }}  

```

Интерфейсы, подобно классам, могут расширяться посредством ключевого слова *extends*. Интерфейс может расширить один или несколько других интерфейсов, добавить к ним новые константы и методы, которые должны быть реализованы в классе, реализующем расширенный интерфейс.

Пакеты

Конфликты имен становятся источником серьезных проблем при разработке повторно используемого кода. Как бы тщательно вы ни подбирали имена для своих классов и методов, кто-нибудь может использовать это же имя для других целей. При использовании простых названий проблема лишь усугубляется — такие имена с большей вероятностью будут задействованы кем-либо еще, кто также захочет пользоваться простыми словами. Такие имена, как *set*, *get*, *clear* и т. д., встречаются очень часто, и конфликты при их использовании оказываются практически неизбежными.

Во многих языках программирования предлагается стандартное решение — использование —префикса пакета перед каждым именем класса, типа, глобальной функции и так далее. Соглашения о префиксах создают *контекст имен (naming context)*, который предотвращает конфликты имен одного пакета с именами другого. Обычно такие префиксы имеют длину в несколько символов и являются сокращением названия пакета — например, *Xt* для —X Toolkit или *WIN32* для 32-разрядного Windows API.

Если программа состоит всего из нескольких пакетов, вероятность конфликтов префиксов невелика. Однако, поскольку префиксы являются сокращениями, при увеличении числа пакетов вероятность конфликта имен повышается.

В Java принято более формальное понятие пакета, в котором типы и субпакеты выступают в качестве членов. Пакеты являются именованными и могут импортироваться. Имена пакетов имеют иерархическую структуру, их компоненты разделяются точками. При использовании компонента пакета необходимо либо ввести его полное имя, либо *импортировать* пакет — целиком или частично. Иерархическая структура имен пакетов позволяет работать с более длинными именами. Кроме того, это дает возможность избежать конфликтов имен — если в двух пакетах имеются классы с одинаковыми именами, можно применить для их вызова форму имени, в которую включается имя пакета.

Приведем пример метода, в котором полные имена используются для вывода текущей даты и времени с помощью вспомогательного класса Java с именем *Date*:

```
class Date1 {  
public static void main(String[] args) {  
java.util.Date now = new java.util.Date();
```

```
System.out.println(now);  
}  
}
```

Теперь сравните этот пример с другим, в котором для объявления типа `Date` используется ключевое слово `import`:

```
import java.util.Date;  
class Date2 {  
    public static void main(String[] args) {  
        Date now = new Date();  
        System.out.println(now);  
    }  
}
```

Пакеты Java не до конца разрешают проблему конфликтов имен. Два различных проекта *могут* присвоить своим пакетам одинаковые имена. Эта проблема решается только за счет использования общепринятых соглашений об именах. По наиболее распространенному из таких соглашений в качестве префикса имени пакета используется перевернутое имя домена организации в Internet. Например, если фирма Acme Corporation содержит в Internet домен с именем `acme.com`, то разработанные ей пакеты будут иметь имена типа `COM.acme.package`.

Точки, разделяющие компоненты имени пакета, иногда могут привести к недоразумениям, поскольку те же самые точки используются при вызове методов и доступе к полям в ссылках на объекты. Возникает вопрос — что же именно импортируется? Новички часто пытаются импортировать объект `System.out`, чтобы не вводить его имя перед каждым вызовом `println`. Такой вариант не проходит, поскольку `System` является классом, а `out` — его статическим полем, тип которого поддерживается методом `println`.

С другой стороны, `java.util` является пакетом, так что допускается импортирование `java.util.Date` (или `java.util.*`, если вы хотите импортировать все содержимое пакета). Если у вас возникают проблемы с импортированием чего-либо, остановитесь и убедитесь в том, что вы импортируете тип.

Классы Java всегда объединяются в пакеты. Имя пакета задается в начале файла:

```
package com.sun.games;  
class Card  
{  
    // ...  
}  
// ...
```

Если имя пакета не было указано в объявлении `package`, класс становится частью *безымянного пакета*. Хотя это вполне подходит для приложения (или апплета), которое используется отдельно от другого кода,

все классы, которые предназначаются для использования в библиотеках, должны включаться в именованные пакеты.

8.3. Создание и использование пользовательских исключений, для обеспечения детализированной информации, посредством расширения класса исключений.

Что делать, если в программе произошла ошибка? Во многих языках о ней свидетельствуют необычные значения кодов возврата — например, `-1`. Программисты нередко не проверяют свои программы на наличие исключительных состояний, так как они полагают, что ошибок — быть не должно. С другой стороны, поиск опасных мест и восстановление нормальной работы даже в прямолинейно построенной программе может затемнить ее логику до такой степени, что все происходящее в ней станет совершенно непонятным. Такая простейшая задача, как считывание файла в память, требует около семи строк в программе. Обработка ошибок и вывод сообщений о них увеличивает код до 40 строк. Суть программы теряется в проверках как иголка в стоге сена — это, конечно же, нежелательно.

При обработке ошибок в Java используются *проверяемые исключения* (*checked exceptions*). Исключение заставляет программиста предпринять какие-то действия при возникновении ошибки. Исключительные ситуации в программе обнаруживаются при их возникновении, а не позже, когда необработанная ошибка приведет к множеству проблем.

Метод, в котором обнаруживается ошибка, *возбуждает* (*throw*) исключение. Оно может быть *перехвачено* (*catch*) кодом, находящимся дальше в стеке вызова — благодаря этому первый фрагмент может обработать исключение и продолжить выполнение программы. Неперехваченные исключения передаются стандартному обработчику Java, который может сообщить о возникновении исключительной ситуации и завершить работу потока в программе.

Исключения в Java являются объектами — у них имеется тип, методы и данные. Представление исключения в виде объекта оказывается полезным, поскольку объект-исключение может обладать данными или методами (или и тем и другим), которые позволят справиться с конкретной ситуацией. Объекты-исключения обычно порождаются от класса `Exception`, в котором содержится строковое поле для описания ошибки. Java требует, чтобы все исключения были расширениями класса с именем `Throwable`.

Основная парадигма работы с исключениями Java заключена в последовательности `try-catch-finally`. Сначала программа пытается (`try`) что-то сделать; если при этом возникает исключение, она его перехватывает (`catch`); и наконец (`finally`), программа предпринимает некоторые итоговые действия в стандартном коде или в коде обработчика исключения — в зависимости от того, что произошло.

Ниже приводится метод `averageOf`, который возвращает среднее арифметическое двух элементов массива. Если какой-либо из индексов

выходит за пределы массива, программа запускает исключение, в котором сообщает об ошибке. Прежде всего, следует определить новый тип исключения `IllegalAverageException` для вывода сообщения об ошибке. Затем необходимо указать, что метод `averageOf` возбуждает это исключение, при помощи ключевого слова `throws`:

```
class IllegalAverageException extends Exception {  
}  
class MyUtilities {  
public double averageOf(double[] vals, int i, int j)  
throws IllegalAverageException  
{  
try {  
return (vals[i] + vals[j]) / 2;  
} catch (IndexOutOfBoundsException e) {  
throw new IllegalAverageException();  
}  
}  
}
```

Если при определении среднего арифметического оба индекса `i` и `j` оказываются в пределах границ массива, вычисление происходит успешно и метод возвращает полученное значение. Однако, если хотя бы один из индексов выходит за границы массива, возбуждается исключение `IndexOutOfBoundsException` и выполняется соответствующий оператор `catch`. Он создает и возбуждает новое исключение `IllegalAverageException` — в сущности, общее исключение нарушения границ массива превращается в конкретное исключение, более точно описывающее истинную причину. Методы, находящиеся дальше в стеке выполнения, могут перехватить новое исключение и должным образом прореагировать на него.

Если выполнение метода может привести к возникновению проверяемых исключений, последние должны быть объявлены после ключевого слова `throws`, как показано на примере метода `averageOf`. Если не считать исключений `RuntimeException` и `Error`, а также подклассов этих типов исключений, которые могут возбуждаться в любом месте программы, метод возбуждает лишь объявленные в нем исключения — как прямо, посредством оператора `throw`, так и косвенно, вызовом других методов, возбуждающих исключения.

Объявление исключений, которые могут возбуждаться в методе, позволяет компилятору убедиться, что метод возбуждает только эти исключения и никакие другие. Подобная проверка предотвращает ошибки в тех случаях, когда метод должен обрабатывать исключения от других методов, однако не делает этого. Кроме того, метод, вызывающий ваш метод, может быть уверен, что это не приведет к возникновению неожиданных исключений. Именно поэтому исключения, которые должны быть объявлены

после ключевого слова `throws`, называются *проверяемыми исключениями*. Исключения, являющиеся расширениями `RuntimeException` и `Error`, не нуждаются в объявлении и проверке; они называются *непроверяемыми исключениями*.

8.4. Базовый класс `Object` и его методы

Как уже говорилось, все классы Java наследуют от класса `Object`, который стоит во главе иерархии наследования. У класса `Object` целый набор полезных методов, которые программист может использовать в дочерних классах. Класс `Object` позволяет определить базовую функциональность для любого другого производного класса, которую программист естественно можно расширить в подклассах.

Все построенные иерархии классов имеют класс `Object` как общего родителя, т.е. общий суперкласс.

Другими словами, все пользовательские классы расширяют класс `Object`.

Таблица 8.1. Методы класса `Object`.

Название метода	Сигнатура	Замечания
<code>clone()</code>	<code>protected Object clone()</code>	Создает и возвращает копию объекта
<code>equals</code>	<code>public Boolean equals(Object obj)</code>	Проверяет равны ли объекты
<code>finalize()</code>	<code>protected void finalize()</code>	Вызывается через сборщик мусора для любого объекта, когда сборщик мусора определяет, что на объект нет больше никаких ссылок
<code>getClass()</code>	<code>public final Class getClass()</code>	Вызывается классом времени выполнения объекта
<code>hashCode()</code>	<code>public int hashCode()</code>	Возвращает значение хэш кода для конкретного объекта
<code>toString</code>	<code>public String toString()</code>	Возвращает строковое представление объекта

СОДЕРЖАНИЕ

Глава 1. Объектно-ориентированная разработка, концепции и преимущества	3
Глава 2. Основы программирования на Java	35
Глава 3. Базовые классы Java	57
Глава 4. Основные понятия и концепция объектно-ориентированного анализа и объектно-ориентированного дизайна	63
Глава 5. Использование Java для создания пользовательских классов	65
Глава 6. Написание пользовательских методов	75
Глава 7. Подклассы и наследование	87
Глава 8. Абстрагирование, полиморфизм и интерфейсы	95
СОДЕРЖАНИЕ	108
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	109

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Н.В. Зорина, Курс лекций по Объектно-ориентированному программированию на Java, МИРЭА, 2016
2. Буч, Г. Язык UML. Руководство пользователя. [/ Г. Буч, Д. Рамбо, И. Якобсон. — Электрон. дан. — М. : ДМК Пресс, 2008. — 496 с.
3. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. — СПб.: Символ-Плюс, 1999
4. Вендров А.М. Проектирование программного обеспечения экономических информационных систем: Учебник. — М.: Финансы и статистика, 2005
5. Коберн А. Быстрая разработка программного обеспечения. — М.: ЛОРИ, 2002
6. Соммервилл И. Инженерия программного обеспечения (6-е изд.), м.: Вильямс, 2002.- 624 с., ил.
7. Schach S.R.: Object-Oriented and Classical Software Engineering (5 ed.) McGraw-Hill, 2001, 744 pp.
8. Зыков С.В. Проектирование корпоративных порталов.— М.: МФТИ, 2005.— 258 с.
9. П.Ноутон, Г.Шилдт - "Java 2. Наиболее полное руководство"
10. Head First Java. Second Edition. O'Reilly. ISBN 0596009208.
11. Bruce Eckel "Thinkng in Java" она же "Философия Java".
12. Joshua Bloch "Effective Java. Second Edition". Addison-Wesley; ISBN 978-0-321-35668-0