МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Программирование на Джава

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень	бакалавриат
	(бакалавриат, магистратура, специалитет)
Форма обучения	очная
	(очная, очно-заочная, заочная)
Направления подготовки	09.03.04 «Программная инженерия
	(код(-ы) и наименование(-я))
Институт	информационных технологий (ИТ)
	(полное и краткое наименование)
Кафедра	инструментального и прикладного программного обеспечения (ИППО)
	(полное и краткое наименование кафедры, реализующей дисциплину (модуль))
Лектор	Зорина Наталья Валентиновна
	(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2019/20

(учебный год цифрами)

Проверено и согласовано «____» _____20___г. _____ *А.С. Зуев*

(подпись директора Института/Филиала с расшифровкой)

Москва 2019 г.

Содержание курса:

Лекция 1. Введение в язык программирования Java. Реализация принципов объектно-ориентированного программирования в Java

Лекция 2. Приемы разработки ООП программ на Java. Стандартные потоки ввода/вывода. Массивы в Java. Использование рекурсии в программах на Java.

Лекция 3. Реализация наследования в программах на Java. Абстрактные классы и интерфейсы. Реализация алгоритмов сортировок и поиска на Java.

Лекция 4. Работа со строками в Java. Обработка исключений на Java.

Лекция 5. Паттерны проектирования программ. Метапрограммирование и использование дженериков.

Лекция 6. Абстрактные типы данных и их реализация на Java. Поведенческие паттерны

Лекция 7. Стандартные классы Java Framework Collection и их использование в программах. Структурные паттерны.

Лекция 8. Пакеты java.lang, java.util. Потоки ввода/вывода в Java

Рекомендованная литература:

а) основная литература:

1. Объектно-ориентированный анализ и программирование [Электронный ресурс]: учебное пособие / Н. В. Зорина. — М.: РТУ МИРЭА, 2019. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/3240
2. Вязовик, Н.А. Программирование на Java: учебное пособие / Н.А. Вязовик. — 2-е изд. — Москва: ИНТУИТ, 2016. — 603 с. — Текст: электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/100405 (дата обращения: 18.11.2019). — Режим доступа: для авториз. пользователей

б) дополнительная литература:

1. Объектно-ориентированное программирование [Электронный ресурс]: конспект лекций / Н. В. Зорина. — М.: РТУ МИРЭА, 2019. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/3357
2. Васильев Алексей Николаевич Java. Объектно-ориентированное программирование: Базовый курс по объектно-ориентированному программированию: для магистров и бакалавров / А. Н. Васильев. — СПб.: Питер, 2014. — 397 с.: ил. — (Учебное пособие). — Библиогр.: с. 377 (11 назв.)

в) учебно-методические пособия:

1. Объектно-ориентированное программирование на Java [Электронный ресурс]: практикум / Н. В. Зорина. — М.: РТУ МИРЭА, 2019. — Электрон. опт. диск (ISO) https://library.mirea.ru/share/3185
2. Объектно-ориентированное программирование на Java [Электронный ресурс]: метод. рекомендации / Н. В. Зорина [и др.]. — М.: РТУ МИРЭА, 2018. — Электрон. опт. диск (ISO)

г) современные профессиональные базы данных и информационные справочные системы:

1. http://www.oracle.com/technetwork/java/index.html - Технология Java
2. http://www.ibm.com/developerworks/ru/edu/j-intserv/index.html
3. http://www.ibm.com/developerworks/ru/edu/ws-jax/index.html
4. https://docs.oracle.com/javase/tutorial/

Лекция 1. Введение в программирование на Java. Основные сведения о языке программирования.

Введение.

В этой лекции перечислены основные сведения о языке программирования Java: история создания, особенности языка, его лексика и синтаксис, основные типы данных, встроенные в язык, так называемые примитивные типы, основные операции над ними, операторы управляющие ходом программы, структура и организация программы. является кроссплатформенным, разрабатывался под девизом “сделано однажды, работает всегда”. Он отлично подходит для разработки Web-ориентированных приложений. Кроме того, язык Java является чрезвычайно популярным среди разработчиков.

Мотивация к изучению курса.

Язык Java долгие годы остается наиболее популярным языком у разработчиков объектно-ориентированным языком программирования. Этот язык программирования изначально был разработан компанией Sun Microsystems и представляет собой «чисто» объектно-ориентированный язык. То есть на этом языке программирования нельзя написать программу, которая не содержит хотя бы один класс. Если на языке программирования C++, который тоже поддерживает принципы объектно-ориентированного программирования можно написать программу в процедурном стиле, то на языке Java этого сделать не получится. Синтаксис языка Java очень похож синтаксис языка C++ и обеспечивает довольно низкий уровень вхождения. То есть если вы ранее программировали на языке C++, то вам легко будет научиться программировать на Java. Язык Java является кроссплатформенным, и разрабатывался под девизом «сделано единожды, работает везде». Он отлично подходит для разработки Web-ориентированных приложений.

Почему этот язык необходимо изучать?

Как упоминалось выше язык Java является чрезвычайно популярным среди разработчиков. По информации источника TIOBE, который измеряет популярность языков программирования среди разработчиков этот язык в течении ряда занимает

первое место (см. рисунок 1.1 ниже).

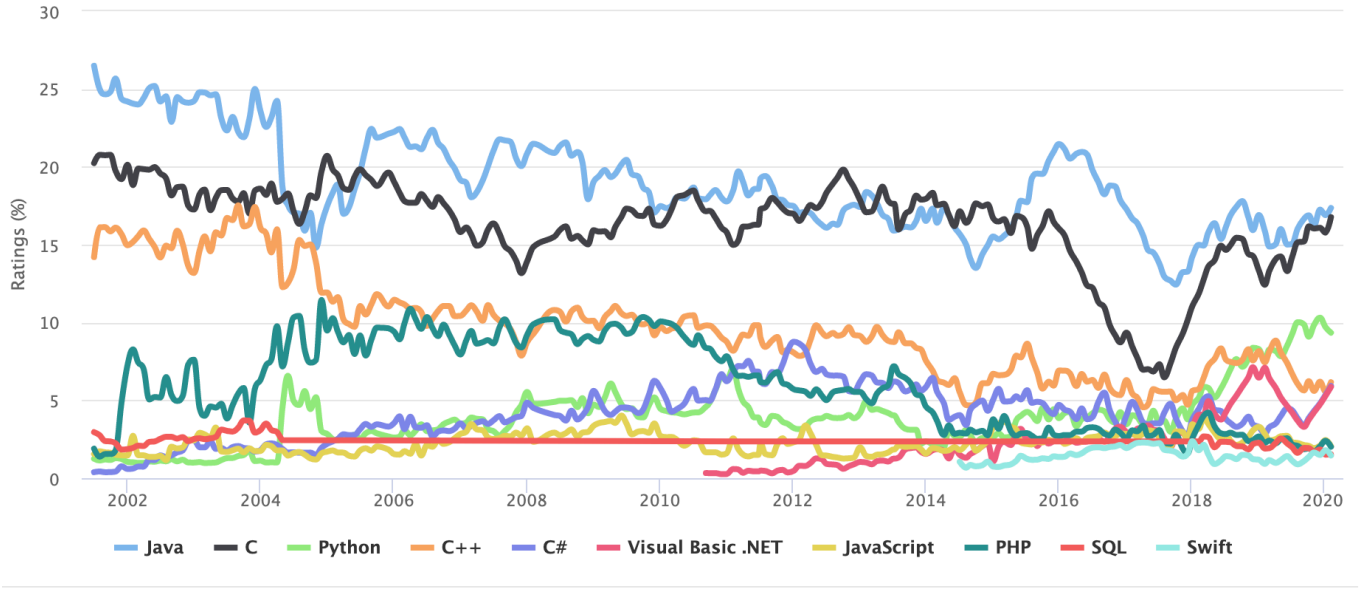


Рисунок 1.1 – Диаграмма популярности языков программирования (источник https://www.tiobe.com) IT

Как видно из рисунка 1.1 язык Java на начало 2020 года занимает первое место по среди других языков и по полярности сравним, пожалуй, только с языком Си. Этот язык позволяет писать как десктопные так и мобильные приложения и не-заменим в разработке «Энтерпрайз», серверная часть или бэкенд пишется часто именно на нем. Кроме того, именно изучение языка Java позволит вам понять, как реализуются принципы объектно-ориентированного программирования в программах. После изучения Java вам будет легко взяться за мобильную разработку и писать клиент-серверные приложения.

Общие сведения о языке программирования Java.

Приступая к изучению нового языка, всегда полезно поинтересоваться, какие исходные данные могут обрабатываться средствами этого языка, в каком виде их можно задавать, и какие стандартные средства обработки этих данных заложены в язык. Это довольно скучное занятие, поскольку в каждом развитом языке программирования множество типов данных и еще больше правил их использования.

Однако несоблюдение этих правил приводит к появлению скрытых ошибок, обнаружить которые иногда бывает очень трудно. Ну что же, в каждом ремесле приходится сначала «играть гаммы», и мы не можем от этого уйти при изучении любого нового языка.

Все правила языка Java исчерпывающе изложены в его спецификации, сокращенно называемой JLS. Иногда, чтобы понять, как выполняется та или иная конструкция языка Java, приходится обращаться к спецификации, но, к счастью, это бывает редко, правила языка Java достаточно просты и естественны.

Введение в платформу Java и краткая история создания.

Язык Java представляет из себя объектно-ориентированный язык программирования, со строгой типизацией. Этот язык является одновременно компилируемым и интерпретируемым, в отличие от языка C/C++, который является компилируемым или, например языка Бейсик, который является и интерпретируемым. За счет этого достигается кроссплатформенность приложений на языке программирования Java. Язык стал черезвычайно популярным и нашел свою нишу в связи именно с этой его чертой. Поскольку его развитие пришлось на эпоху становления и бурного развития сети Интернет, он рекламировался как универсальный язык программирования, приложения на котором будут работать на любой платформе, для которой есть виртуальная машина Java. Популярный слоган, под которым распространялся язык: “Write once, run anywere”. В переводе с английского это означает – «написано однажды, работает везде».

Точкой отсчета в истории создания языка является 1991 год, перечислим кратко основные вехи в истории языка:

- проекта 1991 “Green Project”;
- Oak language;
- Sun Microsystems;
- Java language;
- “Write once, run anywhere”;
- Oracle (с 27/01/2010).

В течении времени, с момента создания язык менялся и дополнялся, появлялись новые черты и возможности:

– 1996 – JDK 1.0 (JLS, JVM, JDK);

– 1997 – JDK 1.1 (JIT, JavaBeans, JDBC, RMI);

– 1998 – JDK 1.2 (изменения языка, policy/permission, JFC, …);

– 1999 – разделение развития на платформы

1. Java 2 Platform, Standard Edition (J2SE, JavaSE);
2. Java 2 Platform, Enterprise Edition (J2EE, JavaEE);
3. Java 2 Platform, Micro Edition (J2ME, JavaME);

– 2000 – JDK 1.3 (HotSpot (JIT) в составе JVM, …);

– 2002 – JDK 1.4 (новое API);

– 2004 – JDK 1.5 (изменения языка);

– 2006 – JDK 1.6 (скриптовые языки, работа с базами данных…);

– 2011 – JDK 1.7 (изменения языка…).

Перечислим особенности языка Java:

– строгая типизация;

– кроссплатформенный (из-за байта кода).

– объектно- ориентированный.

– встроенная модель безопасности (можно писать многопоточные приложения);

– ориентирован на разработку интернет-приложений, можно писать распределенные приложения.

– компилируемый и интерпретируемый;

– легко научиться программировать.

Преимущества использования языка Java:

– Встроенный сборщик мусора.

– Обнаружение ошибок на этапе компиляции.

– Встроенная обработка ошибок (exceptions handling).

– Переносимость кода (write once, run everywhere).

– Поддержка многозадачности на уровне языка.

– Динамическая загрузка классов (по необходимости).

– Поддержка работы с высокоуровневыми сетевыми протоколами.

В данный момент существ уют *три разные платформы* для программирования на языке Java, так называемые Java Editions:

– Java Stadard Edition (J2SE) – используется для разработки самостоятельных приложений или апплетов, так называемая Core Java;

– Java Enterprise Edition (J2EE) – используется для создания приложений на серверной стороне (в терминах приложений с клиент-серверной архитектурой), содержит пакеты для работы с Java Servlets, Java Server Pages (JSP), JDBC и т.д;

– Java Micro Edition (J2ME) – используется для разработки самостоятельных приложений на мобильных устройствах.

Мы с вами на протяжении всего курса будем изучать именно Java Standard Edition.

Работа с памятью в Java:

– Выделять память физически не требуется (нет работы с адресной арифметикой), также отсутствуют указатели.

– Освобождение памяти происходит автоматически с помощью встроенного сборщика мусора.

– Сборщик мусора (garbage collector) автоматически проверяет область памяти, где живут объекты Java – Java Heap (куча) – и уничтожает их, если они стали не нужны программе.

– Алгоритм работы сборщика мусора зависит от конкретной платформы – а значит, конкретной JVM.

Основные термины и инструментарий разработчика.

Кроссплатформенность означает, что существуют средства разработки для большинства аппаратных платформ.

– Виртуальная машина Java (Java Virtual Machine, JVM) гарантирует единообразие интерфейса с операционной системой.

– Переносимость: «Write once, run everywhere». Поставляется с исчерпывающей библиотекой классов JDK (Java Development Kit).

– JRE (Java Runtime Environment) – среда, позволяющая запустить программу, написанную на языке Java.

Java Virtual Machine (JVM):

– Осуществляет поддержку конкретной аппаратной платформы.

– Работает с аппаратно-независимым байт-кодом, полученным на этапе компиляции исходного кода в байт-код.

– Байт-код может быть запущен на любом компьютере (win/mac/unix), на котором установлена JVM.

– Программная реализация JVM содержится в составе Java Runtime Environment (JRE).

– JRE можно установить отдельно – а можно, в составе Java Development Kit (JDK).

– (http://www.oracle.com/technetwork/java/javase/downloads/index.html).

Компиляция и запуск программы на Java

Процесс компиляции и запуск проекта на языке Java представлен на рисунке 1.2.

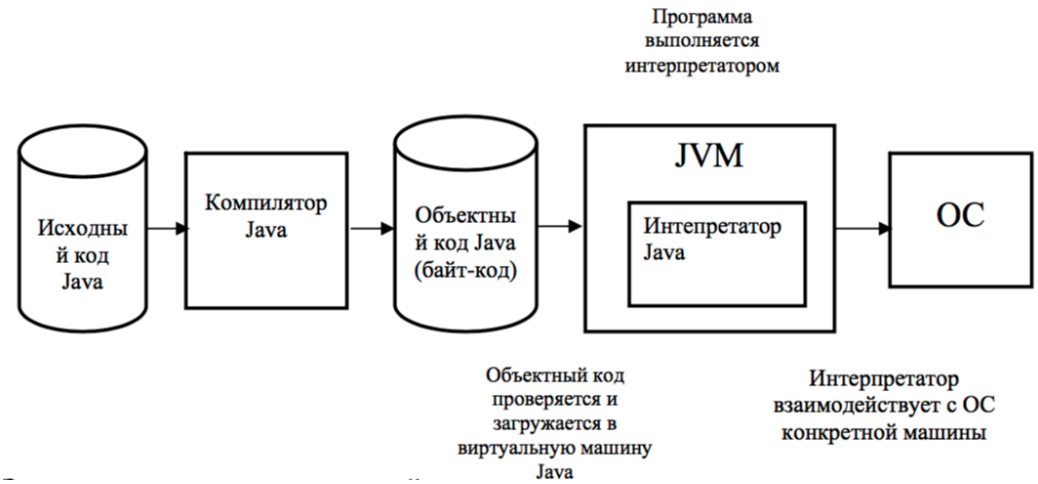


Рисунок 1.2 – Компиляция и запуск проекта

Java платформа:

1. Множество различных аппаратных систем:

– Intel x86, Sun SPARC, PowerPC и др.

2. Множество разных программных систем:

– MS Windows, Linux, Mac OS, Sun Solaris, и др.

3. Потребность в одинаковом функционале на различных платформах.
4. Java Virtual Machine (JVM), универсальность:

– Исходный код открыт с 1999 г.

Популярные интегрированные среды разработки (IDE) для Java.

Вы, конечно, можете писать программы на Java в обычном текстовом редакторе, например таком как vim или atom, или, например emacs, сохранить исходник с расширением .java, а затем откомпилировать его в командной строке, используя компилятор javac. Но большинство из вас привыкло работать с интегрированными средами разработки, которые включают редактор, компилятор и отладчик. Перечислим наиболее распространенные IDE для разработки программ на Java:

– NetBeans (www.**netbeans**.org);
– Eclipse (www.**eclipse**.org);
– IntelliJ IDEA (www.jetbrains.com/idea/).

Нужно отметить, что наиболее популярная среди разработчиков IDE для разработки программ на Java является все- таки IDE IntelliJ IDEA от компании Jet Brains. И сегодня IntelliJ IDEA — это наиболее интеллектуальная среда. Она по ходу написания и выполнения может проанализировать код, выявить ошибки и предложить подходящее для каждого конкретного случая решение. Интеллектуальность среды IDEA состоит в том, что она строит синтаксическое

дерево, только когда вы еще только набираете код. Проанализировав ссылки и те пути, по которым программа может быть выполнена, IDE предлагает варианты того, как код можно дополнить. IntelliJ IDEA является именно профессиональной средой разработки, поэтому как только вы пройдете стадию адаптации к такому ритму работы, то вам будет очень сложно вернуться на использованные ранее бесплатные программы. Поэтому советую вам выбрать именно этот инструмент для изучения программирования на этом языке, тем более что он доступен для разработки бесплатно, по академической лицензии.

Основные преимущества этой IDE:

- удобство и быстрота разработки (код пишется частично вами, а частично машинным интеллектом)
- интеллектуальность, разбор кода на лету;
- рефакторинг кода;
- множество доступных плагинов и фреймворков;
- красивый и интуитивный интерфейс

Для большинства веб-языков созданы свои отдельные среды, основанные также на IntelliJ IDEA. Среди них:

- PhpStorm — среда разработки на PHP, которая активно используется как для изучения языка, так и для профессиональной детальности;
- RubyMine — среда для взаимодействия с языком Ruby;
- PyCharm для Python;
- AppCode для Objective-C
- и многие другие.

Этапы программного решения задачи:

1. Создание модели, определение данных для предстоящей обработки
2. Разработка алгоритма: определение операций над данными и последовательности шагов по преобразованию текущего состояния модели в следующее

3. Формулировка модели и алгоритма на языке программирования

Разработка и запуск программы:

На рисунке 1.2 представлена схема запуска проекта на Java.

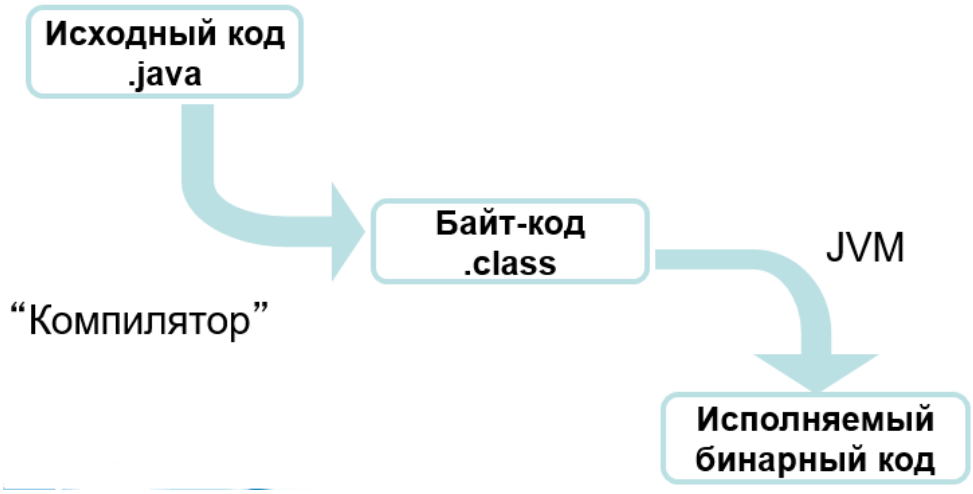


Рисунок 1.2 – Запуск проекта

Программа на языке Java – это обычный текстовый файл с расширением java. Файлы подаются на вход Java компилятора, который переводит их в специальное представление Java кода. 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 нем существует только восемь базовых или встроенных в язык типов данных. Остальные типы данных создаются самостоятельно программистами с использованием средств языка. Особенностью языка Java также является то, что здесь отсутствуют деструкторы, так-как есть встроенная сборка мусора (англ. Garbage collector). Программистам нет необходимости следить за утечками памяти.

Первая Java программа.

По давней традиции, восходящей своими корнями к языку Си, учебники по языкам программирования начинаются обычно начинаются с программы «Hello, World!». Мы с вами не будем нарушать эту традицию. В листинге 1.1 представлена эта программа на языке программирования Java в самом простом виде.

Листинг 1.1. Первая программа на языке Java;

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

Вот и все, всего пять строчек! Но даже на этом простом примере можно заметить целый ряд существенных особенностей языка Java.

– Всякая программа представляет собой один или несколько классов, в этом простейшем примере только *один класс (class)* и называется он *MyFirstClass* .

– Начало класса следует за служебным словом *class*, за которым следует сразу пишем имя класса, выбираемое произвольно, в данном случае *MyFirstClass*. Все, что содержится внутри классе, записывается в фигурных скобках и составляет тело класса *(class body)*.

В отличие от языка C++ мы видим, что точка входа в программу находится внутри класса, следовательно для того-чтобы написать самую простую программу результат выполнения которой всего лишь вывод на экран строчки текста нам пришлось создать класс.

Синтаксис языка.

На этой лекции мы изучим основные встроенные в язык базовые или примитивные типы данных, операции над ними, операторы для управления ходом программы, а также обсудим структуру программы на Java. Но начнем, по традиции, с простейшей программы.

Понятие класса и пакета. Абстрагирование и инкапсуляция. Поля данных и методы класса. Экземпляр объекта типа класс. Методы Геттеры и Сеттеры. Модификаторы доступа. Импортирование пакетов. Разрешение конфликтов имен. Класс в Java как тип данных, создаваемый программистом. UML нотация записи класса. Информация по сокрытию реализации и инкапсуляции. Ключевое слово this.

Конструкторы класса. Ключевое слово “this”

1.2 Объектно-ориентированное программирование, основные понятия

В результате развития подходов к созданию программ появилось объектно-ориентированное программирование. В процедурных языках программист выполняет действия над данными с помощью различных инструкций языка. В объектно-ориентируемых языках демонстрируется подход, при котором происходит объединение данных и методов их обработки.

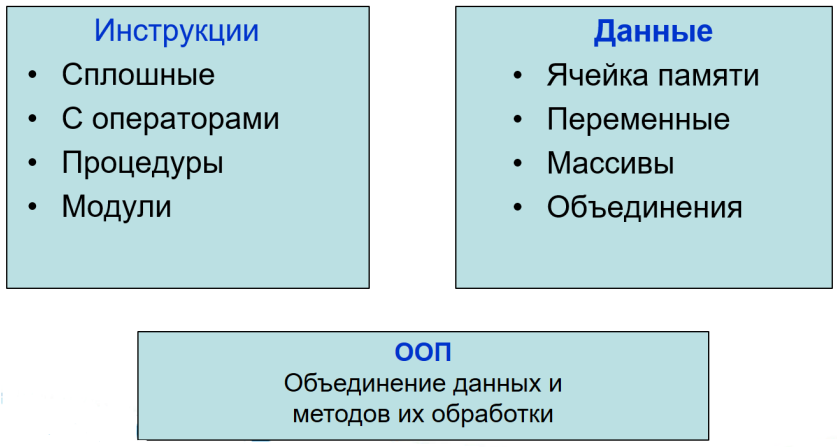


Рисунок 1.3 – Подходы к разработке программ

Объектно-ориентированный поход оперирует типами данных, создаваемыми программистами при решении прикладных задач программирования.

Стиль ООП. Объекты и классы.

Рисунок 1.4 – Объекты и классы.

Напомним основные принципы ООП:

– Инкапсуляция:

объединение данных и методов их обработки в одну сущность, приводящее к сокрытию реализации класса и отделению его внутреннего представления от внешнего.

– Полиморфизм:

способность объекта соответствовать во время выполнения двум или более возможным типам.

– Наследование:

отношение между классами, при котором один класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов.

Отношения между классами.

Классы в ООП программах могут находиться между собой в различных отношениях:

– Наследование:

Объекты дочернего класса наследуют свойства родительского класса.

– Ассоциация:

Объекты классов вступают во взаимодействие между собой.

– Агрегация:

Объекты разных классов образуют целое, оставаясь самостоятельными.

– Композиция:

Объекты одного класса входят в объекты другого, не обладая самостоятельностью.

– Класс-метакласс:

Экземплярами класса являются классы.

Достоинства ООП.

Основные достоинства, которые мы получаем6 программируя в стиле ООП:

– Упрощение разработки:

Разделение функциональности, локализация кода, инкапсуляция.

– Возможность создания расширяемых систем: Обработка разнородных структур данных, изменение поведения на этапе выполнения, работа с наследниками.

– Повторное использование кода.

– Легкость модернизации с сохранением совместимости.

Недостатки ООП.

Можно перечислить следующие недостатки ООП программ:

– Неэффективность на этапе выполнения.

– Неэффективность в смысле распределения памяти.

– Излишняя избыточность.

– Психологическая сложность проектирования.

– Техническая сложность проектирования и документирования и сопровождения.

Объектная модель языка Java.

На вершине иерархии наследования лежит класс Object. Это означает, что все остальные классы являются его потомками. Это означает, что когда вы создаете собственные классы, то вы неявным образом наследуетесь от этого класса. У этого класса есть методы, которые вы можете переопределять в своих классах. Есть две разновидности классов - просто классы и интерфейсы. Вы можете создавать свои собственные классы, используя ООП, например при помощи наследования.

Итак, объектный язык Java оперирует объектами:

– Все сущности в Java являются объектами, классами либо интерфейсами.

– Строгая реализация инкапсуляции.

– Реализовано одиночное наследование от класса и множественное от интерфейсов.

1.3 Пакеты в Java

Понятие о пакетах:

– Способ логической группировки классов.

– Комплект ПО, могущий распространяться независимо и применяться в сочетании с другими пакетами.

– Членами пакетов являются:

1. классы;
2. интерфейсы;
3. вложенные пакеты;
4. дополнительные файлы ресурсов.

Функциональность пакетов

– Позволяют группировать взаимосвязанные классы и интерфейсы в единое целое.

– Способствуют созданию пространств имен, позволяющих избежать конфликтов идентификаторов, относящихся к различным типам.

– Обеспечивают дополнительные средства защиты элементов кода.

– Формируют иерархическую систему.

Способы реализации и доступ к пакетам:

1. Пакеты могут быть реализованы:

– в виде структуры каталогов с файлами классов,

– в виде jar-архива.

2. Путь к используемым пакетам указывается:

– непосредственно при запуске JVM,

– через переменную окружения `CLASSPATH` (по умолчанию `CLASSPATH=""`).

1.4 Правила именования и лексика языка

Понятие имени:

Имена задаются посредством идентификаторов, указывают на компоненты программы. Имена бывают простые и составные. Пространства имен подразделяется на:

– пакеты

– типы

– поля

– методы

– локальные переменные и параметры

– метки

Понятие модуля компиляции:

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

– объявления пакета;

```
package mypackage;
```

– выражений импортирования;

```
import java.net.Socket;
import java.io.*;
```

– объявлений верхнего уровня – классов и интерфейсов.

Объявление пакета:

– первое выражение в модуле компиляции (например, для файла java/lang/Object.java);

– `package java.lang;`

– при отсутствии объявления пакета модуль компиляции принадлежит безымянному пакету (не имеет вложенных пакетов);

– пакет доступен, если доступен модуль компиляции с объявлением пакета;

– ограничение на доступ к пакетам нет.

Выражения импорта:

– доступ к типу из данного пакета – по простому имени типа;

– доступ к типу из других пакетов – по составному имени пакета и имени типа:

1. import-выражения упрощают доступ:
2. импорт одного типа (`import java.net.URL;`)
3. импорт пакета с типами (`import java.net.*;`)

– попытка импорта пакета, недоступного на момент компиляции, вызовет ошибку;

– дублирование импорта игнорируется;

– нельзя импортировать вложенный пакет;

– при импорте типов пакета вложенные пакеты не импортируются;

Алгоритм работы компилятора при анализе типов:

1. выражения, импортирующие типы;
2. другие объявленные типы;
3. выражения, импортирующие пакеты.

Если тип импортирован явно невозможны:

1. объявление нового типа с таким же именем;
2. доступ по простому имени к одноименному типу в текущем пакете.

Импорт пакета не мешает объявлять новые типы или обращаться к имеющимся типам текущего пакета по простые имена. Импорт конкретных типов дает возможность при прочтении кода сразу понять, какие внешние типы используются.

Объявление верхнего уровня:

Объявление в языке Java производится следующим образом, показанном на

рисунке 1.5.

```
package first;
class MyFirstClass {
}
interface MyFirstInterface {
}
```

Рисунок 1.5 – Объявление верхнего уровня

Существуют следующие правила объявления:

– область видимости типа – пакет;

– доступ к типу извне его пакета:

1. по составному имени,
2. через выражения импорта.

– разграничение (модификаторы) доступа;

– в модуле компиляции может быть максимум один `public` тип;

– имя публичного типа и имя модуля компиляции должны совпадать;

– другие не-`public` типы модуля должны использоваться только внутри текущего пакета;

– как правило, один модуль компиляции содержит один тип.

Конвенция кода Java (или правила именования):

– Пакеты:

`java.lang, javax.swing, ru.mtu.mirea.isbo`

– Типы:

`Student, ArrayIndexOutOfBoundException Cloneable, Runnable, Serializable`

– Поля:

`value, enabled, distanceFromShop`

– Методы:

`getValue, setValue, isEnabled, length, toString`

– Поля-константы:

`PI, SIZE_MIN, SIZE_MAX, SIZE_DEF`

– Локальные переменные.

Пример программы на Java представлен на рисунке 1.6.

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

Рисунок 1.6 – Пример программы на языке Java

Лексика языка Java состоит из следующих элементов:

– Структура исходного кода и его элементы.

– Типы данных.

– Описание классов:

1. Общая структура,
2. Поля,
3. Методы,
4. Конструкторы,
5. Блоки инициализации.

– Точка входа программы.

Кодировка

– Java ориентирован на Unicode;

– первые 128 символов почти идентичны набору ASCII;

– символы Unicode задаются с помощью escape-последовательностей \u262f, \uu2042, \uuu203d;

– Java чувствителен к регистру.

1.5 Структура исходного кода и его элементы

Исходный код разделяется на:

– Пробелы:

1. ASCII-символ SP, \u0020, дес. код 32,
2. ASCII-символ HT, \u0009, дес. код 9,
3. ASCII-символ FF, \u000c, дес. код 12,
4. ASCII-символ LF, символ новой строки,
5. ASCII-символ CR, возврат каретки,
6. символ CR, за которым сразу следует символ LF.

– Комментарии.

– Лексемы.

Комментарии в коде указываются по следующим правилам:

– //комментарий;

Символы после // и до конца текущей строки игнорируются.

– /*комментарий*/;

Все символы, заключенные между /* и */, игнорируются.

– /**комментарий*/;

Комментарии документирования.

Комментарии документирования (javadoc):

– начинаются с /**, заканчиваются */;

– в строках начальные символы * и пробелы перед ними игнорируются;

– допускают использование HTML-тэгов, кроме заголовков;

– специальные теги : @see, @param, @deprecated.

К лексемам языка относят:

– идентификаторы;

– служебные слова `class, public, const, goto`;

– литералы;

– разделители `{ } [ ] ( ) ; . , ;`

– операторы = > < ! ? : == && || .

Идентификаторы

– имена, задаваемые элементам языка для упрощения доступа к ним;

– можно записывать символами Unicode;

– состоят из букв и цифр, знаков _ и **$** ;

– не допускают совпадения со служебными словами, литералами `true, false, null`;

– длина имени не ограничена.

В языке Java, как и во всех языках, существуют свои служебные (ключевые) слова, которые приведены на рисунке 1.7.

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

Рисунок 1.7 – Служебные слова в Java

1.6 Типы данных

Ссылочные типы данных.

К ссылочным типам относятся типы классов (в т.ч. массивов) и интерфейсов. Переменная ссылочного типа способна содержать ссылку на объект, относящийся к этому типу. Ссылочным литералом является null. Данные типы предназначены для работы с объектами. Ссылочные переменные содержат в себе ссылки на объект, но не стоит их путать с указателями C++. Тип переменной определяет контракт доступа к объекту.

К ссылочным типам относятся типы классов (в т.ч. массивов) и интерфейсов. Переменная ссылочного типа способна содержать ссылку на объект, относящийся к этому типу. Ссылочным литералом является null.

Примитивные (простые) типы данных.

Предназначены для работы со значениями естественных, простых типов. Переменные содержат непосредственно значения. К примитивным типам относятся:

– `boolean` допускает хранение значений `true` или `false.`

– целочисленные типы:

1. `char` – 16-битовый символ Unicode,
2. `byte` – 8-битовое целое число со знаком,
3. `short` – 16-битовое целое число со знаком,
4. `int` – 32-битовое целое число со знаком,
5. `long` – 64-битовое целое число со знаком.

– Вещественные типы:

1. `float` – 32-битовое число с плавающей точкой (IEEE 754-1985),
2. `double` – 64-битовое число с плавающей точкой (IEEE 754-1985).

Литералы в Java бывают следующие:

– булевы `true false;`

– символьные `'a' '\n' '\\' '\377' '\u0064';`

– целочисленные `29 035 0x1D 0X1d 0xffffL:`

По умолчанию имеют тип `int` .

– числовые с плавающей запятой `1. .1 1e1 1e-4D 1e+5f:`

По умолчанию имеют тип `double`.

– строковые "Это строковый литерал" "" .

1.7 Описание классов, поля и методы

Класс может содержать:

– поля;

– методы;

– вложенные классы и интерфейсы.

Пример создания класса в языке Java приведен на рисунке 1.8.

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

  public static long nextID = 0;
}
```

Рисунок 1.8 – Пример класса в Java

Модификаторы объявления класса:

– `public` – признак общедоступности класса;

– `abstract` – признак абстрактности класса;

– `final` – завершенность класса (класс не допускает наследования);

– `strictfp` – повышенные требования к операциям с плавающей точкой.

Поля класса

Объявление поля осуществляется следующим способом:

```
    [модификаторы] <тип> {<имя> [=
<инициализирующее выражение>]};
    double sum = 2.5 + 3.7;
    public double val = sum + 2 * Math.sqrt(2)
```

Если поле явно не инициализируются, ему присваивается значение по умолчанию его типа (`0, false` или `null`)

Модификаторы полей бывают:

– `static` – поле статично (принадлежит контексту класса);

– `final` – поле не может изменять свое значение после инициализации;

– `transient` – поле не сериализуется[1] (влияет только на механизмы сериализации);

– `volatile` – усиливает требования к работе с полем в многопоточных программах.

Методы

Синтаксис объявления метода:

`[модификаторы] <тип> <сигнатура> [throws исключения] {<тело>}`

Пример объявления метода в языке Java приведен на рисунке 1.9.

```
class Primes {
  static int nextPrime(int current) {
    <Вычисление простого числа в теле метода>
  }
}
```

Рисунок 1.9 – Пример объявления метода

Модификаторы методов:

– `abstract` – абстрактность метода (тело при этом не описывается);

– `static` – статичность метода (метод принадлежит контексту класса);

– `final` – завершенность метода (метод не может быть переопределен при наследовании);

– `synchronized` – синхронизированность метода (особенности вызова метода в многопоточных приложениях);

– `native` – «нативность» метода (тело метода не описывается, при вызове вызывается метод из native-библиотеки);

– `strictfp` – повышенные требования к операциям с плавающей точкой.

[1] Это понятие раскрывается в лекции 9

Особенности методов

– Для нестатических методов вызов через ссылку на объект или в контексте объекта `reference.method();methodReturningReference().method();`

– Для статических методов вызов через имя типа, через ссылку на объект или в контексте класса `ClassName.staticMethod();reference.staticMethod();staticMethodReturningReference().method();`

– Наличие круглых скобок при вызове обязательно, т.к. они являются оператором вызова метода

На время выполнения метода управление передается в тело метода. Возвращается одно значение простого или объектного типа `return someValue`.

Аргументы передаются по значению, т.е. значения параметров копируются в стек. Для примитивных типов копируются сами значения, а для ссылочных типов копируется значение ссылки. Перегруженными являются методы с одинаковыми именами и различными сигнатурами.

Создание объектов в Java (пример создания объектов приведен на рисунке 1.10):

– создание ссылки и создание объекта – различные операции;

– используется оператор `new`, он возвращает ссылку на объект;

– после оператора указывается имя конструктора и его параметры.

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

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

Рисунок 1.10 – Пример создания объектов

1.8 Конструкторы

Конструкторы предназначены для формирования начального состояния объекта. Правила написания конструктора сходны с правилами написания методов, имя конструктора совпадает с именем класса. •Память для объекта выделяет оператор `new`. Для конструкторов разрешено использование только модификаторов доступа. При написании конструктор не имеет возвращаемого типа. Оператор возврата `return` прекращает выполнение текущего конструктора.

Конструкторы могут быть перегружены. Они также могут вызывать друг друга с помощью ключевого слова `this` в первой строке конструктора. Если в классе явно не описан ни один конструктор, автоматически создается т.н. конструктор по умолчанию, не имеющий параметров. Если в классе описан хотя бы один конструктор, то автоматически конструктор по умолчанию не создается. Также конструктором по умолчанию называют конструктор, не имеющий параметров. Пример создания конструктора приведен на рисунке 1.11.

```
class Body {
  public long idNum;
  public String name = "No Name";
  public Body orbits = null;

  private static long nextID = 0;

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

Рисунок 1.11 – Пример создания конструктора

Деструкторы

В ряде языков деструкторы выполняют действия, обратные действию конструкторов: освобождают память, занимаемую объектом, и «деинициализируют» объект (освобождают ресурсы, очищают связи, изменяют состояние связанных объектов). Если после вызова деструктора где-то осталась ссылка (указатель) на объект, ее использование приведет к возникновению ошибки. В Java деструкторов нет, вместо них применяется механизм автоматической сборки мусора.

Автоматическая сборка мусора

В случае нехватки памяти для создания очередного объекта виртуальная машина находит недостижимые объекты и удаляет их. Процесс сборки мусора можно инициировать принудительно. Для явного удаления объекта следует утратить все ссылки на этот объект и инициировать сбор мусора. Взаимодействие со сборщиком осуществляется через системные классы `java.lang.System` и `java.lang.Runtime`.

1.9Блоки инициализации

Если некоторые действия по инициализации должны выполняться в любом варианте создания объекта, удобнее использовать блоки инициализации. Тело блока

инициализации заключается в фигурные скобки и располагается на одном уровне с полями и методами. При создании объекта сначала выполняются инициализирующие выражения полей и блоки инициализации (в порядке их описания в теле класса), а потом тело конструктора (рисунок 1.12).

```
class Body {
  public long idNum;
  public String name = "No Name";
  public Body orbits = null;

  private static long nextID = 0;

  {
    idNum = nextID++;
  }

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

Рисунок 1.12 – Инициализация

Статическая инициализация.

Статический блок инициализации выполняет инициализацию контекста класса. Вызов статического блока инициализации происходит в процессе загрузки класса в виртуальную машину.

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

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

  //nextPrime() declaration etc.
}
```

Рисунок 1.13 – Статическая инициализация

Модификаторы доступа.

В Java есть несколько модификаторов доступа:

– `private` – доступ только в контексте класса;

– `(package, default, none)` – доступ для самого класса и классов в том же пакете;

– `protected` – доступ в пределах самого класса, классов-наследников и классов пакета;

– `public` – доступ есть всегда, когда доступен сам класс.

Лекция 2. Реализация принципов ООП в Java. Класс как тип данных. Массивы в Java. Использование рекурсии в программах на Java.

Типы данных в объектно-ориентированных языках, построенные на отношениях, агрегация и композиция. Понятие агрегации и композиции. UML нотация агрегации двух классов. Различия между композицией и агрегацией. Создание новых типов данных с помощью наследования. Массивы как объектные типы данных, программирование действий с массивами. Декомпозиция как основной подход для реализации ООП программ. Метод toString(). Константы (final)

Рекурсивные алгоритмы. Виды рекурсии и ее реализация в программах на Джаве. Рекурсия и исключения. Организация кода в Java программах вложенные и анонимные классы

Создание объектов.

Класс Circle и его экземпляры (рисунок 2.1):

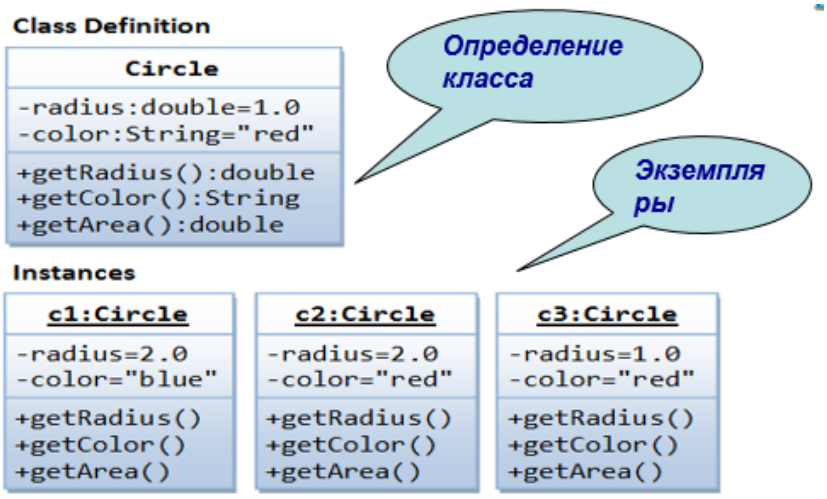


Рисунок 2.1 – Класс Circle

Конструкторы.

Конструкторы в Java (рисунок 2.2) очень похожи конструкторы C++. Вы можете перегружать конструкторы (так же, как и другие любые методы класса). Конструктор, который не берет параметров называется “пустым конструктором”.

Вы можете не иметь конструктора вовсе, в этом случае используется конструктор по умолчанию, как пустой конструктор. Конструктор может вызывать другой конструктор того же самого класса, используя служебное слово‘this’. Вызов другого конструктора можно сделать только как первую инструкцию вызова конструктора.

```
public class Circle {
  …..//поля класса, геттеры и сеттеры
  public Circle(){
    radius = 0;
    color  = "";
}
  public Circle( int r, String c){
    radius = r;
    color  = c;
}
  public Circle (int r) {
      radius  = r;

    }
}
…...
```

Рисунок 2.2 – Конструкторы Java

Создание объектов.

Обычно мы используем `new` оператор, чтобы создать объект:

```
Circle c1 = new Circle (10,”green”);
```

Такая запись вызывает конструктор `Circle()` с параметрами который является специальным методом, который создает и инициализирует объект. Создание объектов называется инстанцированием. Любой объект – экземпляр типа класс (instance of class).

Конструкторы класса Circle

Пример ниже показывает, как можно создать три объекта типа класс Circle с помощью различного вида конструкторов:

```
Circle c1 = new Circle();
Circle c2 = new Circle(2.0);
Circle c3 = new Circle(3.0, "red");
```

Создание объектов через вызов конструктора

```
Circle c1, c2, c3; //объявили три переменных типа круг
// сконструировать экземпляр объекта типа класс и
разместить его в памяти можно только через оператор new
```

```
    c1 = new Circle(); /*создали с помощью вызова
конструктора объект круг в памяти и инициализировали
переменную  c1 ссылкой на созданный объект*/
    c2 = new Circle(2.0); // используем второй конструктор
    c3 = new Circle(3.0, "red"); //третий конструктор
    //можно совместить объявление и  создание объектов
    Circle c4 = new Circle();
```

Вызов методов класса

Мы видели, объект — это экземпляр класса, и мы можем использовать оператор для получения доступа к компонентам объекта - “точка” (аналогично Си++), чтобы вызывать его методы (рисунок 2.3).

```
String title;

int count;

count = title.length()
```

Рисунок 2.3 – Вызов методов

Метод может возвращать значение, которое мы можем использовать, например, для присваивания значения в выражении.

Использование конструкторов.

Примеры использования конструкторов на рисунках 2.4 – 2.5.

```
public class Person {
    String name = ""; // поля можно инициализировать!
    Date birthDate = new Date();
    public Person() {} // пустой конструктор
    public Person(String name, Date birthDate) {
       this(name); // первый параметер
       this.birthDate = birthDate;
    }
    public Person(String name) {
       this.name = name;
    }
}
```

Рисунок 2.4 – Пример 1.

```
public class Person {
    String name = "";
    Date birthDate = new Date();
    public Person(String name, Date birthDate) {
       this.name = name;
       this.birthDate = birthDate;
    }
}
Person p; // OK
p = new Person(); // плохо – ошибка компиляции
```

Рисунок 2.5 – Пример 2.5

Инициализация полей класса.

Инициализация — это блок инструкций, который выполняется сразу же после создания полей и перед вызовом конструктора. Класс необязательно должен иметь инициализацию, хотя в действительности это не так (рисунок 2.6).

Пример:

```
public class Thingy {
    String s;
    //блок ниже представляет инициализацию
    {  s="Hello"; }
}
```

Рисунок 2.6 – Инициализация классов

Обычно инициализация выполняет более сложную работу.

Статическая инициализация — это блок инструкций, который выполняется, когда класс загружается самый первый раз. Статическая инициализация может быть полезной, чтобы производить однократную инициализацию статических полей данных (рисунок 2.7).

Пример:

```
public class Thingy {
    static String s;
    // блок ниже-статическая инициализация
  static {  s="Hello"; }
}
```

Рисунок 2.7 – Статическая инициализация

Ключевое слово – this.

В Java ‘this’ – это ссылка на сам объект (в C++ это указатель…). Служебное слово ‘this’ также используется, чтобы вызвать другой конструктор в том же класса– как, мы увидим позже!

```
public class Point {
    private int radius;
    private String color;
    public Point(int radius, String color) {
       this.radius = radius;
       this.color = color;
    }
}
```

Рисунок 2.8 – Использование ‘this’

2.3 Массивы в Java

Массивом называется множество однотипных объектов, объединенных одним именем и доступ к каждому объекту в этом множестве, осуществляется по порядковому номеру (индексу).

Массив в Java это объектный или ссылочный тип данных, у него есть некоторые поля и методы. Примеры на рисунках 2.9 – 2.10:

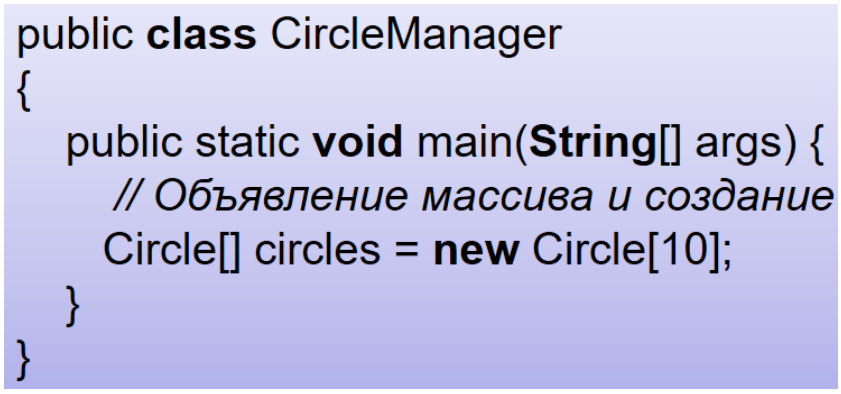


Рисунок 2.9 – Пример создания массива объектов

```
public class CircleManager  {
        public static void main(String[] args) {
      // Объявление массива и создание
         Circle[] circles = new Circle[10];
    // С помощью цикла изменяем переменную i и используем ее
     // для обращения к элементу массива
        for(int i=0; i<10; i++) {
        // Печатаем элемент массива
        System.out.println(circles[i]);
      }
   }
}
```

Рисунок 2.10 – Вывод массива объектов

Инициализация массива.

Можно использовать присваивание элементам массивов значений через прямую инициализацию так же, как в Си, например:

```
int[] sample = {12, 56, 7, 34, 89, 43, 23, 9};
```

Или вот так:

```
Circle[] array = {new Circle(1, 1, "red"), new Circle(3, 4, "green"), new Circle(1, 3, "")};
```

Перебор элементов массива.

Для перебора элементов массива часто используется цикл `foreach(`, работа этой конструкции, аналогична работе с итераторами в C++:

```
public class ForEachExample {
public static void main(String[] args) {
int[] sample = {12, 56, 7, 34, 89, 43, 23, 9};
// выводим элементы в цикле foreach
for (int t : sample) {
```

```
  System.out.println(t); }
  } //end of main
}//end of class
```

Что здесь важно помнить? При каждом проходе цикла в переменной t последовательно будет храниться значение текущего элемента массива, в переменную t копируется значение из элемента массива.

Таким образом получается, что переменная t и переменная sample[0] — это разные переменные!

Пример:

```
public class ForEachExample {
public static void main(String[] args) {
int[] sample = new int[5];
System.out.println("До foreach");
// выводим элементы в цикле foreach - их значение 0
for (int t : sample) {
System.out.println(t);
}
for (int t : sample) { // Думаем, что происходит
инициализация
t= 99;  }
System.out.println("После foreach");
// выводим элементы в цикле foreach - снова 0
for (int t : sample) {
System.out.println(t);
}}}
```

Пример сортировки массива:

```
public class SortArray{
public static void main(String[] args) {
int[] sample = {12, 56, 7, 34, 89, 43, 23, 9};
// выставляем признак "обмена" переменных в true, чтобы
начать цикл
boolean changed = true;
// цикл длится до тех пор, пока при проверке массива ни
одного обмена не произошло
while (changed) {
// Надеемся, что обмена данных не будет
changed = false;
// Проходим по всему массиву
```

```
    for (int i = 0; i < sample.length - 1; i++) {
    // Если впереди стоящее число больше, чем следующее -
меняем их местами и выставляем признак, что был обмен
    if (sample[i] > sample[i + 1]) {
    // Производим обмен с использованием дополнительной
переменной
    int tmp = sample[i];
    sample[i + 1] = tmp;
    // Выставляем признак обмена в true
    changed = true;
    }}}
    // Выводим отсортрованный массив
    for (int i = 0; i < sample.length; i++) {
    System.out.println(sample[i]);
    }}}
```

Интерактивные программы — это программы, которые осуществляют взаимодействие с пользователем, например через консоль.

Для этого программам обычно требуется использование потоков ввода и вывода. Класс Scanner предоставляет удобные методы для чтения входных значений различных типов. Объект Scanner можно настроить, чтобы читать входные данные из различных источников, включая значения опечаток пользователей на клавиатуре. Ввод с клавиатуры представлен объектом System.in. Объект потокового ввода in имеет множество перегруженных методов для работы с различными типами данных.

2.4 Ввод/вывод данных

Следующая строка кода демонстрирует создание объекта Scanner, который считывает данные, вводимые пользователем с клавиатуры:

```
Scanner scan = new Scanner (System.in);
```

Оператор `new` создает объект `Scanner` .

Однажды созданный объект Scanner может быть использован для вызова различных методов ввода, таких как:

```
 answer = scan.nextLine();
```

Класс Scanner часть библиотеки java.util и должен быть импортирован в

программу, чтобы можно было им пользоваться. Метод nextLine считывает весь ввод до конца строки Детали библиотек и создания класса объектов обсуждаются далее.

```
EchoJava.java
import java.util.Scanner;
public class EchoJava
{
    public static void main (String[] args)
    {
        String message;
        Scanner scan = new Scanner (System.in);
        System.out.println ("Enter a line of text:");
        message = scan.nextLine();
        System.out.println ("You entered: \"" + message
+ "\"");
    }
  }}
```

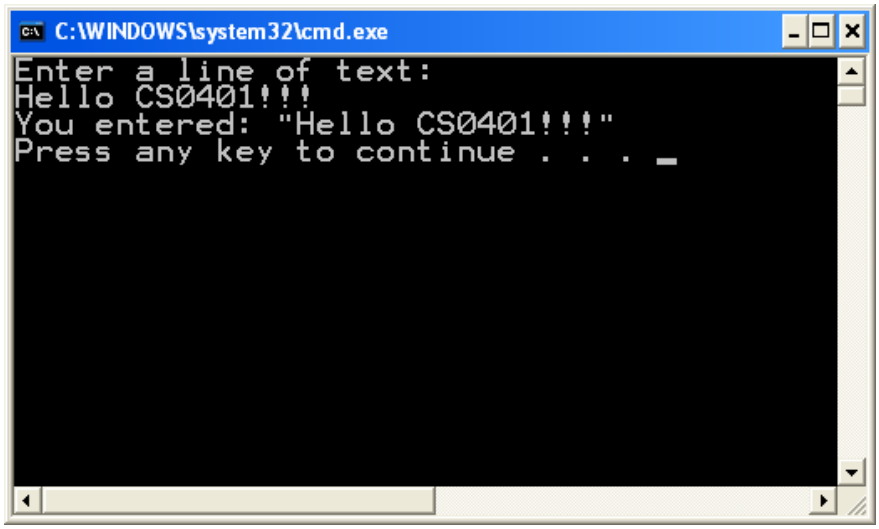


Рисунок 2.11 – Работа программы

Ввод отдельных символов

Если не указано иное, то пробелы пространство используется для разделения элементов (называемых токенов) входного потока. Включает в себя символы пробела, табуляции, символы новой строки.

Метод next класса Scanner читает следующий элемент на входе и возвращает его в виде строки. Такие методы как `nextInt` и `nextDouble` считывают данные конкретных типов.

```
FuelConsumption.java
import java.util.Scanner;
public class FuelConsumption{
```

```
        public static void main (String[] args) {
            int kilometres;
            double litres, kpl;
            Scanner scan = new Scanner (System.in);
            System.out.print ("Enter the number of
kilometres: ");
            kilometres = scan.nextInt();
            System.out.print ("Enter the litres of fuel
used: ");
            litres = scan.nextDouble();
            kpl = kilometres/litres;
            System.out.println ("Kilometres Per litr: " +
kpl);
        }
    }
```

Ввод /Вывод в Java программах.

```
  import java.util.Scanner; // импортируем класс
public class Main {
    public static void main(String[] args) {
// создаём объект класса Scanner
        Scanner sc = new Scanner(System.in);
        int i = 2;
        System.out.print("Введите целое число: ");
        if(sc.hasNextInt()) { // возвращает истину если с
потока ввода   можно считать целое число
          i = sc.nextInt(); // считывает целое число с
потока ввода и сохраняем в переменную
          System.out.println(i*2);
        } else {
          System.out.println("Вы ввели не целое число");
        }
    }
}
```

Метод `hasNextDouble()`, применённый к объекту класса `Scanner`, проверяет, можно ли считать с потока ввода вещественное число типа `double`, а метод `nextDouble()` — считывает его.

Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит).

Существуют методы `hasNext название_типа()` для различных

примитивных типов.

Вывод с помощью класса System, объекта out и методов этого объекта, например println() переопределены для различных типов.

Форматированный вывод.

Часто возникает необходимость в форматировании числовых значений так, чтобы их можно было при выводе представить должным образом. Стандартная библиотека Java классов содержит классы, которые обеспечивают возможности форматированного вывода.

Класс NumberFormat позволяет значениям формата, как валюты или проценты format values as currency or percentages.

Класс DecimalFormat позволяет вам форматировать значения, основываясь на шаблонах

Оба класса находятся в пакете java.text package. Метод getCurrencyInstance() возвращает объект NumberFormat. Метод getPercentInstance() возвращает объект NumberFormat для изображения знака процентов.

Каждый объект-форматировщик имеет метод, называемый format(), который возвращает строку с указанной информацией в соответствующем формате. Класс NumberFormat содержит статические методы, которые возвращают отформатированный объект. Листинг примера (рисунок 2.22) представлен ниже.

```
PurchaseOrder.java
import java.util.Scanner;
import java.text.NumberFormat;
public class PurchaseOrder
{
   public static void main (String[] args)
   {
      final double TAX_RATE = 0.05;  // 5% налог с
продаж
      int quantity;
      double subtotal, tax, totalCost, unitPrice;

      Scanner scan = new Scanner (System.in);
```

```
            NumberFormat fmt1 =
NumberFormat.getCurrencyInstance();
            NumberFormat fmt2 =
NumberFormat.getPercentInstance();
            System.out.print ("Enter the quantity: ");
            quantity = scan.nextInt();
               System.out.print ("Enter the unit price: ");
            unitPrice = scan.nextDouble();
            subtotal = quantity * unitPrice;
            tax = subtotal * TAX_RATE;
            totalCost = subtotal + tax;
            // Вывод на печать с соответствующим
форматированием
            System.out.println ( "Subtotal: " +
fmt1.format(subtotal));
            System.out.println ("Tax: " + fmt1.format(tax)+ "
at "
                                    + fmt2.format(TAX_RATE));
            System.out.println ( "Total: " +
fmt1.format(totalCost));
        }
    }
```

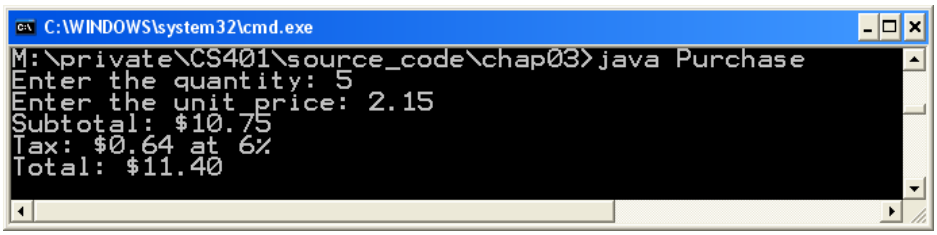


Рисунок 2.12 – Пример форматированного вывода

2.5 Форматированный вывод

Класс `DecimalFormat` может быть использован для форматирования различными способами значений с плавающей запятой. Например, можно указать, что при выводе число должно быть сокращено до трех знаков после запятой. Конструктор `DecimalFormat()` принимает строку, которая представляет собой шаблон для форматированного вывода числа. Для того-чтобы создать объект `DecimalFormat`:

```
DecimalFormat formatter = new DecimalFormat(pattern);
```

Pattern должен быть строкой, которая содержит требуемый шаблон (pattern),

например: “0.00” будет отображать две цифры после запятой. Вы можете изменить шаблон (pattern) с помощью:

```
applyPattern(pattern)
```

Для форматирования чисел используется метод `format()`. Например:

```
System.out.println(formatter.format(15.026));
```

2.6 Перечисляемые типы

Java позволяет определить перечисляемый тип, который затем можно использовать для объявления переменных. Определение перечисляемого типа задает все возможные значения, которые может принимать переменная этого типа. Следующее объявление создает перечисляемый тип, который называется Season.

```
enum Season {WINTER, SPRING, SUMMER, FALL};
```

В перечислении может быть любое количество значений. Представляет собой список именованных констант, и определяет новый тип данных. Для создания перечисления служит ключевое слово enum.

Объект перечислимого типа может принимать лишь значения, содержащиеся в списке. Перечисления удобно использовать, когда требуется определить ряд значений, обозначающих совокупность элементов. Например, с помощью перечисления можно представить набор кодов состояния (успешное завершение, ошибка, необходимость повторной попытки).

Пример перечисления видов транспортных средств:

```
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

Идентификаторы `CAR`, `TRUCK` и так далее называются константами перечислимого типа. Каждый из них автоматически неявно объявляется как открытый (public), статический (static) член перечисления `Transport`.

Тип этих констант соответствует типу перечисления (в данном случае — `Transport`). В терминологии Java подобные константы называются

самотипизированными (приставка “само” означает, что в качестве типа константы принимается тип перечисления).

После того, как тип определен, переменная этого типа может быть объявлена

```
Season time;
```

а потом ей можно присвоить значение

```
time = Season.FALL;
```

Значения указываются через имя типа. Перечисляемые типы являются безопасными типами - вы не можете присвоить им любое значение, кроме тех, которые уже перечислены. Определив перечисление, можно создать переменную данного типа. Но несмотря на то, что перечисление определяется как тип класса, получить экземпляр объекта типа enum с помощью оператора new нельзя.

Переменная перечислимого типа создается подобно переменной простого типа. Например, для объявления переменной tp упомянутого выше перечислимого типа `Transport` служит следующее выражение:

```
Transport tp;
```

Переменная tp относится к типу Transport, и поэтому ей можно присваивать только те значения, которые определены в данном перечислении. Например, в следующей строке кода переменной tp присваивается значение `AIRPLANE`:

```
tp = Transport.AIRPLANE;
```

Для проверки равенства констант перечислимого типа служит оператор сравнения =. Например:

```
if(tp == Transport.TRAIN) // ...
```

Также можно использовать `switch()`. Применение перечисления для управления оператором switch:

```
switch(tp) {
case CAR:
    // ...
case TRUCK:
    // ...
```

Порядковые значения

Внутри перечисления каждое значение перечисляемого типа хранится как целое число, называемое его порядковое значение. Первое значение перечисляемого

типа имеет порядковое значение равное нулю, второе - двум, и так далее.

Тем не менее, вы не можете присвоить числовое значение данным перечисляемого типа, даже если оно соответствует действительным порядковым значением, заданным в перечислении.

Объявление перечисляемого типа представляет собой особый тип класса, а каждая переменная этого типа является объектом.

Метод `ordinal()` возвращает порядковое значение объекта в перечислении. Метод `name()` возвращает имя идентификатора, соответствующего значению объекта в перечислении.

2.7 Оболочки классов

Пакет java.lang содержит классы-оболочки, соответствуют каждому типу-примитиву (рисунок 2.23):

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

Рисунок 2.13 – Оболочки типов

Следующее объявление создает объект Integer, который представляет собой целое число 40, как объект:

```
Integer age = new Integer(40);
```

Объект класса-оболочки, может быть использован в любой ситуации, когда использование примитивного типа будет недостаточно. Например, некоторые объекты служат в качестве контейнеров других объектов. Примитивные значения не могут быть сохранены в таких контейнерах, но объекты-оболочки могут.

Классы-оболочки также содержат статические методы, которые помогают

управлять ассоциированными с ними типами. Так, например, класс Integer содержит метод, чтобы преобразовать целое число, которое хранится в String в значение типа int:

```
num = Integer.parseInt(str);
```

Классы-оболочки также содержат полезные и нужные константы. Так, например, у класса Integer есть константы: `MIN_VALUE` и `MAX_VALUE` которые содержат самое маленькое и самое большое значения типа int.

Автоупаковка.

Автоупаковка – это автоматическое преобразование примитивного значения к соответствующему объекту обертки

```
Integer obj;
int num = 42;
obj = num;
```

Присваивание создает соответствующий объект Integer. Обратное преобразование (называется распаковка) и. также происходит автоматически, по мере необходимости.

2.8 Инкапсуляция в Java.

Мы можем взять один из двух представлений объекта:

– внутреннее, это когда детали переменных и методов класса, который определяет его;

– внешние – сервисы или описание того, что объект содержит и как объект взаимодействует с остальной частью системы.

С внешней точки зрения, объект представляет собой инкапсулированную сущность, предоставляя набор конкретных услуг по взаимодействию с ним. Эти услуги определяют интерфейс объекта. Один объект (так называемый клиент) может использовать другой объект с помощью услуг, которые он предоставляет.

Клиент объекта может потребовать его услуги (вызывать его методы), но он не должен знать о том, как эти услуги осуществляются. Мы должны сделать трудным, если не вообще невозможным для клиента доступ к переменным объекта напрямую. То есть, объект должен быть самоуправляемым.

Инкапсулированный объект (рисунок 2.14) можно рассматривать как черный ящик - его внутренняя работа скрыта от клиента. Клиент вызывает методы интерфейса объекта, которые управляют данными экземпляра класса.

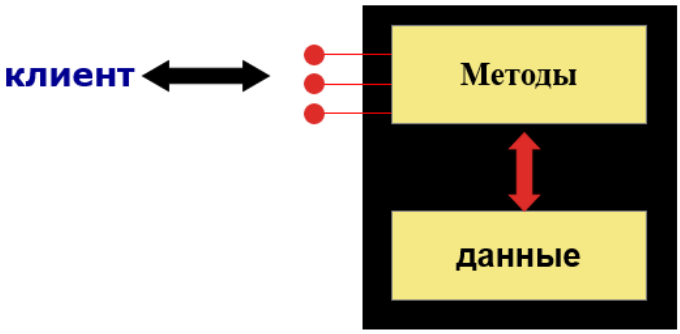


Рисунок 2.14– Инкапсулируемый объект

Модификаторы доступа (видимости).

В Java, мы выполняем инкапсуляцию с помощью соответствующего использования модификаторов видимости (рисунок 2.15). Модификатор в Java зарезервированное слово, которое определяет конкретные характеристики метода или данных. Есть различные виды модификаторов.

Данные и методы класса, объявленные с `public` доступны в любом месте программы. Данные и методы класса, объявленные с `private` доступны внутри класса. Все, что объявлено без модификатора видимости по умолчанию имеет видимость и может быть доступно из любого класса в том же пакете.

Публичные переменные нарушают инкапсуляцию, потому что они позволяют клиенту "достичь данных" и изменять значения напрямую. Поэтому переменные экземпляра не должны быть объявлены общедоступными. Вполне допустимо, чтобы дать константам общественную видимость, что позволяет использовать их вне класса.

Public константы не нарушают инкапсуляцию, потому что, хотя клиент может получить доступ к ним, но зато не может изменить значение. Методы, которые предоставляют услуги объекта и объявляются с помощью `public`, так что они могут быть вызваны клиентами. Общедоступные методы (`public`) также называются сервисными (обслуживающими) методами. Метод создан просто для оказания

помощи сервисному методу называется поддерживающим методом. Так как поддерживающий метод не предназначен для вызова клиентом, то он не должен быть объявлен (`public`) общедоступным.

Рисунок 2.15 – Модификаторы доступа или видимости

Модификатор final.

Мы используем модификатор final, например, чтобы определить константы. Три вида применения модификатора final представлены на рисунке 3.3.

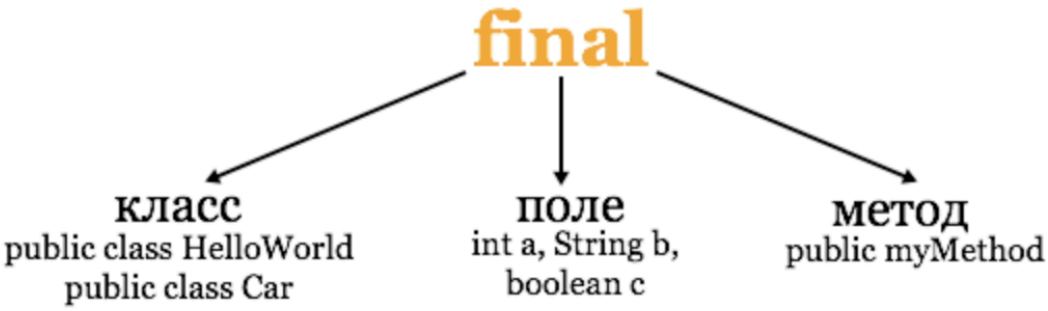


Рисунок 3.3 – Применение модификаторов

Стандартные методы класса геттеры и сеттеры.

Поскольку данные экземпляра является private, то класс, как правило, предоставляет услуги по доступу и изменения значения данных для своих клиентов. Методы геттеры возвращают текущее значение переменной. Методы сеттеры изменяют значение переменной. Названия методов геттеров и сеттеров(мутаторов) выглядят как getX и setX, где X это название поля данных.

Ограничения для сеттеров.

Использование сеттеров дает классу разработчику возможность ограничить возможности клиентов класса по изменению состояния объекта. Сеттеры (их еще называют мутаторами) часто сконструированы таким образом, что значения переменных могут быть установлены только в пределах определенных границ.

Например, `setFaceValue` это сеттер класса Die, который должен ограничить значение допустимого диапазона (от 1 до MAX).

2.9 Метод toString() и его использование в программах.

Метод `toString()` это метод класса Object, который лежит на вершине иерархии классов java. Создавая собственные классы, мы неявным образом наследуемся от класса Object. Для всех пользовательских классов необходимо переопределить метод `toString()`.

Метод `toString()` возвращает значение символьной строки, для того чтобы тем самым представить данный объект. Он вызывается автоматически, когда объект конкатенируется со строкой или когда он передается внутрь метода `println()`.

Пример класс Account.

Давайте рассмотрим еще один пример, демонстрирующий детали реализации классов и методов. Мы будем представлять сущность - банковский счет классом по имени Account. Его состояние может включать в себя номер счета, текущий баланс, и имя владельца.

Поведение счета (или услуги), включают в себя депозиты (зачисление денег на счет) и снятие денег.

Программа Тестер (Driver).

Для каждого примера мы будем писать класс тестер, в нем будем тестировать создаваемые нами классы. Программа тестер управляет использованием других, более интересных частей программы.

Программы Тестеры часто используются для тестирования частей

программного обеспечения. Класс Transactions содержит метод main(), который обеспечивает использование класса Account, осуществляет сервис.

Пример класса с банковским счетом:

```
    package ru.mirea.java.lecture3;
import java.text.NumberFormat;
public class Account {
    private final double RATE = 0.035;
     //начисляемые проценты 3.5%
    private long acctNumber;
    private double balance;
    private String name;
    public Account (String owner, long account, double
initial)  {
            name = owner;
            acctNumber = account;
            balance = initial;
        } public double deposit (double amount)   {
        balance = balance + amount;
        return balance;}
public double withdraw (double amount, double fee)  {
        balance = balance - amount - fee;
        return balance;}
    public double addInterest ()
        balance += (balance * RATE)
        return balance
    } public double getBalance ()   {
        return balance;}
    public String toString ()   {
            NumberFormat fmt =
NumberFormat.getCurrencyInstance();
            return (acctNumber + "\t" + name + "\t" +
fmt.format(balance));
        }}

    package ru.mirea.java.lecture3;
    public class TestAccount {
        public static void main (String[] args) {
            Account acct1 = new Account ("Ilon Musk", 72354,
102.56);
            Account acct2 = new Account ("Unkle Scroodge",
69713, 40.00);
            Account acct3 = new Account ("Mac Dak", 93757,
759.32);
```

Лекция 3. Реализация наследования в программах на Java. Абстрактные классы и интерфейсы. Реализация алгоритмов сортировок и поиска на Java

3.1 Реализация наследования в Java

В Java, мы используем слово - расширяет extends для наследования:

```
public class CheckingAccount
  extends BankAccount {
```

Объекты нового класса получают:

– все поля (состояния) и поведение (методы) родительского класса;

– конструкторы и статические методы/поля не наследуются;

– по умолчанию, родит. класс для всех Object.

Сила Java в том, что только один родительский класс (“единичное наследование”).

Иерархия наследования

Наследование – это цепочка классов, несколько уровней иерархии (рисунок 3.1)

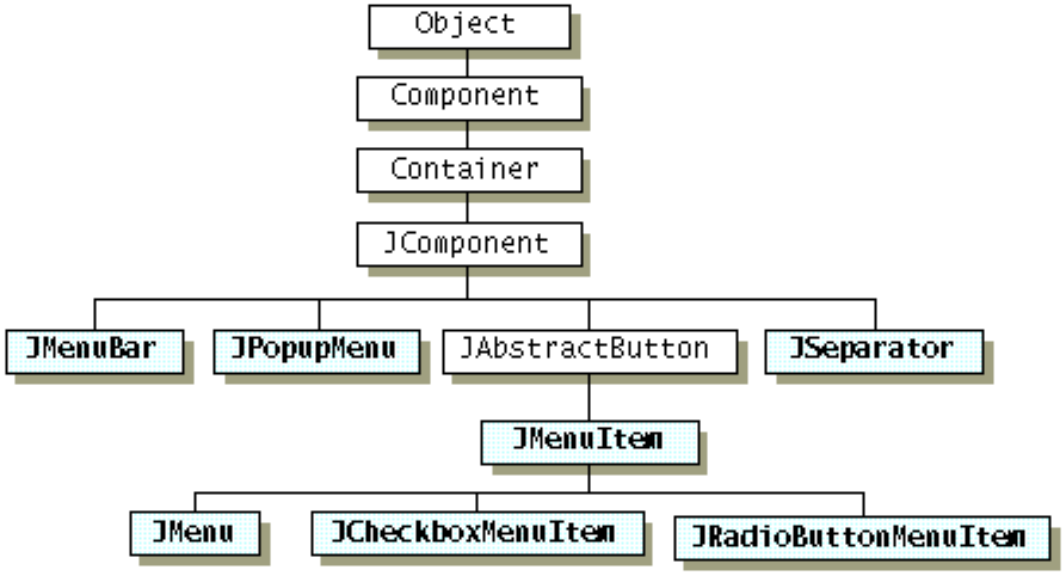


Рисунок 3.1 – Иерархия наследования

Отношение "Has-a”

Отношение "Has-a”: когда один объект содержит другой, как поле:

```
public class BankAccountManager {
      private List<Account> accounts;
```

```
  // ...
}
```

Например объект BankAccountManager имеет или содержит (“has-a”) List внутри себя, и таким образом использует список счетов клиентов в качестве поля класса.

Отношение “Is-a”

Отношение является (“Is-a”) представляет реализацию множество возможностей. Реализуется через интерфейсы и наследование.

```
public class CheckingAccount
  extends BankAccount {
  // ...
}
```

Объект CheckingAccount является (“is-a”) BankAccount. Таким образом, он может делать то, что BankAccount может делать:

– он может быть заменен везде, где необходимо BankAccount;

– Переменная типа BankAccount может ссылаться на объект CheckingAccount.

Код, который не компилируется

Переменная `CheckingAccount` не может ссылаться `BankAccount` (не каждый `BankAccount` "is-a" `CheckingAccount`)

```
CheckingAccount c = new BankAccount();.
```

Не может вызвать метод `CheckingAccount` для переменной типа `BankAccount` (может только использовать поведение `BankAccount`)

```
BankAccount b = new CheckingAccount(0.10);
b.applyInterest();
```

Не может переносить поведение счета на переменную `Object`

```
Object o = new CheckingAccount(0.06);
System.out.println(o.getBalance());
o.applyInterest();
```

Принцип ООП - наследование.

Свойства системы, позволяющие описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом. Общие свойства и методы объектов можно вынести в класс-«родитель». Все «дети»-наследники автоматически получают их.

Пример: класс домашнее животное:

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

Пример: класс кот.

```
public class Cat extends Pet {
    private Date lastMeowDate;
    public Cat() { this("Unnamed"); }
    public Cat(String name) { super(name); }
    public Cat(String name, int age) { super(name, age);
}
    @Override
    public String getType() { return "Кот"; }
    public void meow() {
        System.out.println("Meow!");
        lastMeowDate = new Date();
    }
}
```

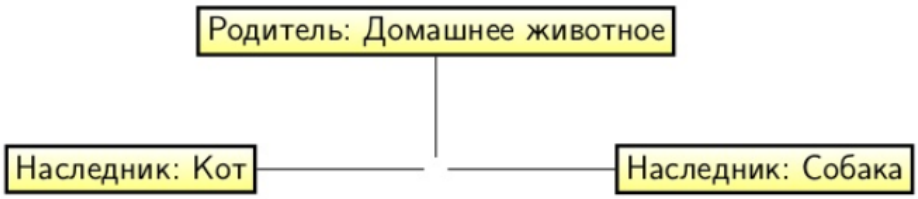


Рисунок 3.2 – Схема наследования

super() означает ссылку на базовый класс, которую можно использовать в дочерних классах. Overriding – переписывание (переделывание, переопределение) в классе-потомке уже существующего метода класса-родителя.

3.2 Аннотирование в Java.

Специальная форма синтаксических метаданных, которая может быть добавлена в исходный код. Аннотации используются для анализа кода, компиляции или выполнения. Аннотируемы пакеты, классы, методы, переменные и параметры.

Например: @Override - проверяет, переопределён ли метод. Вызывает ошибку компиляции, если метод не найден в родительском классе;

UML диаграмма отношения классов – наследование (рисунок 3.3).

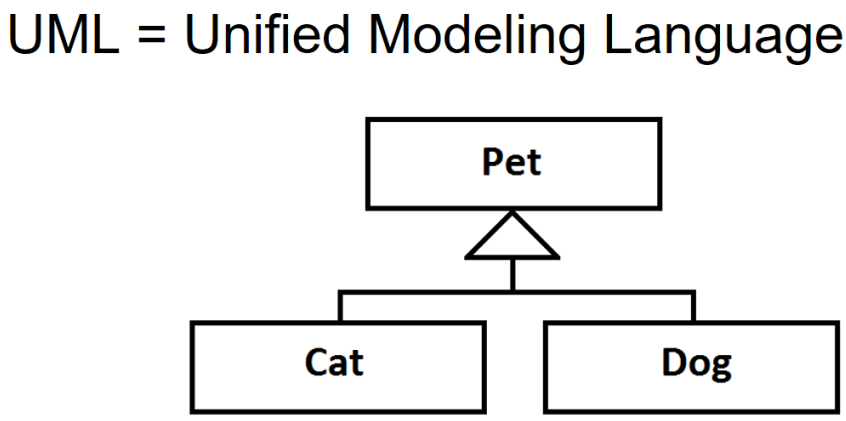


Рисунок 3.3 – Отношение классов

3.3Компоненты и контейнеры

Все примеры программ, которые мы рассмотрели с вами до этого момента были на основе текстового вывода. Они называются приложениями командной строки, которые взаимодействуют с пользователем с помощью простых текстовых подсказок.

Давайте рассмотрим некоторые Java-приложения, в которых используются

графические компоненты. Эти компоненты будут служить основой для программ, которые представляют настоящие графические интерфейсы пользователя (GUIs).

Компоненты GUI

Компонент GUI это объект, который представляет собой элементы представленные на экране, такие как кнопки или текстовые поля и т.п. Связанные с GUI классы определяются в первую очередь в пакете java.awt и в javax.swing packages. Классы, относящиеся к пакету Abstract Windowing Toolkit (AWT) были первоначальным пакетом для GUI в Java.

Пакет Java, который называется Swing появился позже и обеспечивает дополнительные и более универсальные компоненты. Оба пакета необходимы для создания GUI программ Java

Контейнеры GUI.

GUI контейнер представляет собой компонент, который используется для хранения и организации работы с другими компонентами.

Frame (фрейм) является контейнером, который используется для отображения основанного на GUI Java приложения. Элемент фрейм отображается как отдельное окно с заголовком - он может быть перемещен и изменен на экране по мере необходимости.

Элемент panel является контейнером, который не может быть отображен в одиночку, но используется для организации других компонентов. Панель должна быть добавлена к другому контейнеру, который будет отображаться.

GUI контейнеры можно классифицировать как тяжеловесные или легковесные. Тяжеловесные контейнеры это те, которые находятся под управлением базовой операционной системы.

Легковесные контейнеры это те, которые находятся под управлением самих Java программ. Время от времени это различие важно. Фрейм является тяжеловесным контейнером, а панель представляет собой легковесный контейнер.

Элемент Label

Элемент *label* это GUI компонент, который отображает строку текста (обычно название), называют еще этикеткой. Этикетки обычно используются для

отображения информации или чтобы идентифицировать другие компоненты в интерфейсе.

Давайте посмотрим на программу (рисунок 3.7), которая организует две этикетки на одной панели и отображает эту панель во фрейме. Вообще-то эта программа не является интерактивной, но зато фрейм можно перемещать и изменять его размер.

```
import java.awt.*;
import javax.swing.*;
public class Authority
{
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Authority");
      frame.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
      JPanel primary = new JPanel();
      primary.setBackground (Color.yellow);
      primary.setPreferredSize (new Dimension(250, 75));
      JLabel label1 = new JLabel ("Question
authority,");
      JLabel label2 = new JLabel ("but raise your hand
first.");
      primary.add (label1);
      primary.add (label2);
      frame.getContentPane().add(primary);
      frame.pack();ф
      frame.setVisible(true);
   }
}
```

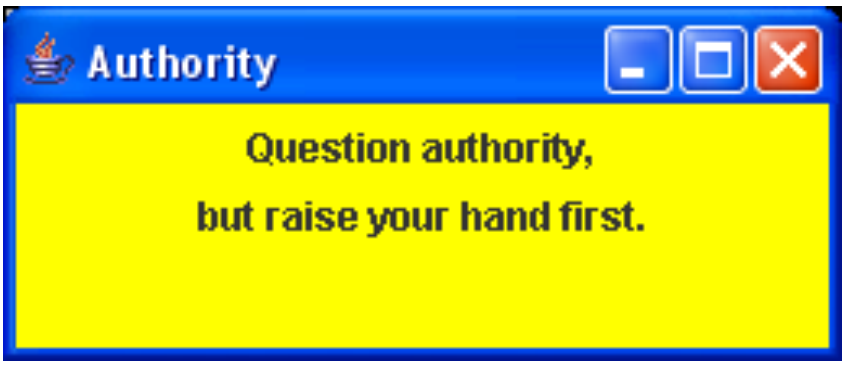


Рисунок 3.7 – Работа программы

Вложенные панели.

Контейнеры, которые содержат другие компоненты образуют иерархию некую защитную оболочку интерфейса. Эта иерархия может быть довольно-таки сложной, для создания желаемого визуального эффекта. В следующем примере две панели вложены внутрь третьей панели - обратите внимание, какое влияние это имеет и как изменился размер рамки.

```
import java.awt.*;
import javax.swing.*;
public class NestedPanels
{
  public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Nested Panels");
      frame.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
      // Set up first subpanel
      JPanel subPanel1 = new JPanel();
      subPanel1.setPreferredSize (new Dimension(150,
100));
      subPanel1.setBackground (Color.green);
      JLabel label1 = new JLabel ("One");
      subPanel1.add (label1);
// Set up second subpanel
      JPanel subPanel2 = new JPanel();
      subPanel2.setPreferredSize (new Dimension(150,
100));
      subPanel2.setBackground (Color.red);
      JLabel label2 = new JLabel ("Two");
      subPanel2.add (label2);
      // Set up primary panel
      JPanel primary = new JPanel();
      primary.setBackground (Color.blue);
      primary.add (subPanel1);
      primary.add (subPanel2);
      frame.getContentPane().add(primary);
      frame.pack();
      frame.setVisible(true);
   }
}
```

3.4 Работа с графикой.

Изображения часто используют в программах с графическим интерфейсом. В Java можно использовать различные форматы изображений как JPEG, так и GIF и PNG. Как мы вскоре увидим, объект Jlabel может быть использован для отображения строки текста. Его также можно использовать для отображения изображения. То есть, Label может состоять из текста, или изображения, или из того и другого одновременно

Класс ImageIcon используется для представления изображения, которое хранится в Label. Положение текста по отношению к изображению можно задать в явном виде. Также можно установить выравнивание текста и изображения в Label.

```
    package ru.mirea.java.lecture3;
    import java.awt.*;
    import javax.swing.*;
    public class LabelDemo {
            public static void main (String[] args)   {
                JFrame frame = new JFrame ("Label Demo");
                frame.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
                ImageIcon icon = new ImageIcon
("/Users/natalazorina/Downloads/FreeBSD.png");
    JLabel label1, label2, label3;
    label1 = new JLabel ("FreeBS Left", icon,
SwingConstants.CENTER);
    label2 = new JLabel ("FreeBSD Right", icon,
SwingConstants.CENTER);
    label2.setHorizontalTextPosition (SwingConstants.LEFT);
    label2.setVerticalTextPosition (SwingConstants.BOTTOM);
    label3 = new JLabel ("FreeBSD Above", icon,
SwingConstants.CENTER);
    abel3.setHorizontalTextPosition (SwingConstants.CENTER);
    label3.setVerticalTextPosition (SwingConstants.BOTTOM);
    JPanel panel = new JPanel();
            panel.setBackground (Color.cyan);
            panel.setPreferredSize (new Dimension (200,
250));
            panel.add (label1);
            panel.add (label2);
            panel.add (label3);
            frame.getContentPane().add(panel);
            frame.pack();
```

```
            frame.setVisible(true);
        }}
```

Графические объекты.

Некоторые объекты содержат информацию, которая определяет, каким образом объект должен быть представлен визуально. Большинство компонентов GUI являются графическими объектами. Мы можем управлять рисованием объектов. Давайте посмотрим на некоторые другие примеры графических объектов.

Рисование графических примитивов

java 2D™ API обеспечивает несколько классов, которые определяют общие геометрические объекты, такие как точки, строки, кривые, и прямоугольники. Эти классы геометрии являются частью java.awt.geom[2] пакета.

Метод paintComponent().

У каждого графического объекта swing есть метод paintComponent(), он автоматически вызывается при прорисовке компонента.

```
public void paintComponent(Graphics g) {
   ...
Graphics2D g2= (Graphics2D) g;
... }
```

Пример:

```
public class MyPanel extends JPanel {
private String text;

public void someAction() {
text = "Привет!";
repaint();
}
@Override
public void paint(Graphics g) {
g.drawString(text, 10, 10);
}
}
```

[2] Пакет java.awt.geom источник https://docs.oracle.com/javase/7/docs/api/java/awt/geom/package-frame.html

3.5Полиморфизм в Java.

Полиморфизм является одним из принципов объектно-ориентированного программирования, и позволяет нам создавать универсальные конструкции программного обеспечения:

– использование наследования для создания полиморфных ссылки;

– использование интерфейсов для создания полиморфных ссылок;

– использование полиморфизма для реализации алгоритмов сортировки и поиска.

Связывание

Рассмотрим строчку кода:

```
obj.doIt();
```

В тот самый момент, когда происходит вызов метода, то происходит его связывание с определением в классе. Если это связывание происходит во время компиляции, то во время вызова метода всегда будет работать именно эта строка, всегда, когда происходит вызов Тем не менее, Java позволяет выполнить отсрочку связывания- выполнить связывание только во время выполнения - это называется динамическое связывание или позднее связывания. Позднее связывание обеспечивает гибкость в разработке и осуществлении программ.

Термин полиморфизм в буквальном смысле переводится как "имеющий много форм". Существуют разные проявления полиморфизма:

– полиморфная ссылка является переменной, которая может относиться к различным типам объектов в разные моменты времени;

– метод вызываемый с помощью полиморфной ссылки может изменяться от одного вызова к другому;

Запомните! Все ссылки на объекты в Java являются потенциально полиморфными.

3.6Интерфейсы в Java

Библиотека стандартных классов Java содержит много полезных интерфейсов. Интерфейс `Comparable` одержит один абстрактный метод, называемый

compareTo, которая используется для сравненивать два объекта. Мы обсудим его compareTo при обсуждении класса String.

Класс String реализует Comparable, то дает нам возможность поставить строки в лексикографическом порядке.

Интерфейс Comparable.

Любой класс может реализовать Comparable чтобы обеспечить механизм для сравнения объектов этого типа:

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

Значение, возвращаемое compareTo должно быть отрицательным если obj1 меньше чем obj2, если они равны, и положительно, если obj1 больше чем obj2. Когда программист проектирует класс, который реализует интерфейс Comparable, то он должен следовать этому намерения.

Интерфейс Iterator.

Итератор создается формально, реализовав интерфейс Iterator, который содержит три метода. Метод hasNext возвращает логический результат - истинно, если есть элементы, которые остались для обработки.

Метод next метод возвращает следующий объект в итерации.

Метод remove удаляет объект, который совсем недавно, возвратил next

Реализуя интерфейс Iterator, а класс формально устанавливает, что объекты этого типа являются итераторы. Программист должен решить, как наилучшим образом реализовать функции итератора. После того, как появилась версия для цикла for-each можно использовать для обработки элементов с помощью итераторов.

Вы могли бы написать класс, который реализует определенные методы (такой как compareTo) без формальной реализации интерфейса (Comparable). Тем не менее, формально, установление взаимосвязи между классом и интерфейсом позволяет, которые позволяет Java установить связи с объектом в некоторых отношениях

Интерфейсы являются одним из ключевых аспектов объектно-

ориентированного проектирования в Java.

Предположим, что мы создали следующую ссылочную переменную:

```
Occupation job;
```

Java позволяет этой ссылке указывать на объект `Occupation`, или на любой объект любого совместимого типа (в одной иерархии). Эта совместимость может быть установлена с помощью наследования или с помощью интерфейсов. Внимательное использование полиморфных ссылок может привести к элегантной, надежной конструкции программного обеспечения.

3.7Использование полиморфных ссылок для сортировки и поиска

Ссылка на объект может ссылаться только на объект своего же класса, или на объект любого класса, связанного с ним наследованием. Например, если класс `Holiday` используется для создания класса под названием `Christmas`, то ссылка на объект `Holiday` можно использовать, чтобы ссылаться на объект `Christmas`.

Ассоциация с дочерним объектом родительской ссылки считается расширяющим тип преобразованием (`upcasting`), и может быть выполнена с помощью простого присваивания.

Назначение родительскому объекту дочерней ссылки также является допустимым (`downcast`), но считается сужающим преобразованием и должно быть выполнено с использованием операции преобразования типов. Расширяющее преобразование является наиболее полезным. Тип объекта, на который ссылаются, а не тип ссылки, определяет, какой метод будет вызывается.

Предположим класс `Holiday` имеет метод, называемый `celebrate`, и класс `Christmas` переопределяет его.Теперь рассмотрим следующий вызов:

```
day.celebrate();
```

Если `day` ссылается на объект `Holiday`, он вызывает версию `celebrate` для `Holiday`; если он ссылается на объект `Christmas`, то вызывается версия `Christmas`. Имя интерфейса `Speaker` может быть использовано как тип

ссылочной переменной объекта `current`. Ссылка `current` ссылка может быть использована, чтобы ссылаться на объект любого класса, который реализует интерфейс `Speaker`. Версия метода `speak()` что следующая строка вызывает зависимость от типа объекта, который является ссылкой на `current`.

```
current.speak();
```

Предположим, что два класса, `Philosopher` и `Student`, оба реализуют интерфейс `Speaker`, обеспечивая предоставление различных версий метода `speak()`. В следующем коде, первый вызов `speak` вызывает одну версию, а второй вызывает другую версию:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Student();
guest.speak();
```

Реализация алгоритмов сортировок в Java программах.

Сортировка является процесс упорядочивания списка элементов (организация в определенном порядке). Процесс сортировки основан на упорядочивании конкретных значений, например:

– сортировка списка результатов экзаменов баллов в порядке возрастания результата;

– сортировка списка людей в алфавитном порядке по фамилии.

Есть много алгоритмов для сортировки списка элементов, которые различаются по эффективности. Мы рассмотрим два конкретных алгоритма:

– сортировка выбором;

– сортировка вставками.

Сортировка выбором.

Подход – сортировка выбором:

– выбрать значение и поместить его на его окончательное место в списке;

– повторить для всех остальных значений элементов;

Более детально:

– найти наименьшее значение в списке элементов;

– поменять его со значением в первого элемента;

– найти следующее наименьшее значение в списке;

– поменять его со значением второго элемента;

– повторять, пока все значения не будут находятся на своих местах.

Пример приведен на рисунке 5.4:

```
original:          3   9   6   1   2
smallest is 1:     1   9   6   3   2
smallest is 2:     1   2   6   3   9
smallest is 3:     1   2   3   6   9
smallest is 6:     1   2   3   6   9
```

Рисунок 5.4 – Алгоритм сортировки выбором

Каждый раз, наименьшее оставшееся значение найдено, то происходит обмен с элементом на "следующей" позиции.

Обработка алгоритма выбора сортировки включает в себя обмен значениями двух элементов. Для выполнения операции обмена требуется три оператора присваивания и переменная для временного хранения значения:

```
temp = first;
first = second;
second = temp;
```

Методу сортировки все-равно, что именно он будет сортировать, ему только необходимо иметь возможность вызвать метод `compareTo()`. Это обеспечивается использованием интерфейса `Comparable` как типа параметра. Кроме того, таким образом каждый класс “для себя” решает, что означает для одного объекта, быть меньше, чем другой.

```
public class Contact implements Comparable{
   private String firstName, lastName, phone;
   public Contact (String first, String last, String
telephone) {
      firstName = first;
      lastName = last;
```

```
            phone = telephone;
        }
        public String toString () { return lastName + ", " +
firstName + "\t" + phone;}
        public boolean equals (Object other) {
            return
(lastName.equals(((Contact)other).getLastName()) &&
firstName.equals(((Contact)other).getFirstName()));
        }
    public String getFirstName () {return firstName;  }
    public String getLastName () {return lastName;}
    public int compareTo (Object other)
        {
            int result;
            String otherFirst =
((Contact)other).getFirstName();
            String otherLast = ((Contact)other).getLastName();
            if (lastName.equals(otherLast))
               result = firstName.compareTo(otherFirst);
            else
               result = lastName.compareTo(otherLast);
            return result;
        }
    }
```

Полиморфизм при программировании сортировок.

Напомним, что класс, который реализует интерфейс `Comparable` определяет метод `compareTo()`, чтобы определить относительный порядок своих объектов. Мы можем использовать полиморфизм, чтобы разработать обобщенную сортировку для любого набора `Comparable` объектов. Метод сортировки принимает в качестве параметра массив `Comparable` объектов. Таким образом, один метод может быть использован для сортировки любых объектов, например, People (людей), Books (книг), или любой каких-либо других объектов.

Сортировка вставками.

Работа метода сортировки:

– выбрать любой элемент и вставить его в надлежащее место в отсортированный подсписок;

– повторять, до тех пор, пока все элементы не будут вставлены.

Более детально:

– рассматриваем первый элемент списка как отсортированный подсписок (то есть первый элемент списка);

– вставим второй элемент в отсортированный подсписок, сдвигая первый элемент по мере необходимости, чтобы освободить место для вставки нового элемента;

– вставим третий элемент в отсортированный подсписок (из двух элементов), сдвигая элементы по мере необходимости;

– повторяем до тех пор, пока все значения не будут вставлены на свои соответствующие позиции.

```
    //вставками
    public static void insertionSort (Comparable[] list) {
          for (int index = 1; index < list.length; index++)
{
             Comparable key = list[index];
             int position = index;
             //  Shift larger values to the right
             while (position > 0 &&
key.compareTo(list[position-1]) < 0) {
                list[position] = list[position-1];
                position--;
             }

             list[position] = key;
          }
       }
    }
```

Пример сортировки вставками на рисунке 5.5:

```
первоначально:    3   9   6   1   2
     insert 9:    3   9   6   1   2
     insert 6:    3   6   9   1   2
     insert 1:    1   3   6   9   2
     insert 2:    1   2   3   6   9
```

Рисунок 5.5 – Алгоритм сортировки вставками

Сравнение сортировок.

Алгоритмы сортировки выбора и вставки аналогичны по эффективности. Они оба имеют внешние циклы, которые сканируют все элементы и внутренние циклы, которые сравнивают значение внешнего цикла почти со всеми значениями в списке.

Приблизительно n2 число сравнений будут сделаны для сортировки списка размера n. Поэтому мы говорим, что эти виды сортировок имеют порядок n2. Другие виды сортировок являются более эффективными: порядок n log2 n.

Алгоритмы поиска в Java программах

Поиск является процессом нахождения целевого элемента в пределах группы элементов, которая называется пул поиск. Целевой или искомый элемент может находится в поисковом пуле, а может там отсутствовать. В любом случае, мы хотим, чтобы эффективно выполнять поиск, сводя к минимуму количество сравнений.

Давайте рассмотрим поиск на двух классических подходах поиска: линейный поиск и бинарный поиск. Точно так же, как мы это делали с сортировкой, мы будем осуществлять поиск с полиморфными параметрами `Comparable`.

Алгоритм линейного поиска.

Линейный или последовательный поиск начинается на с одного конца списка просматриваемых элементов, и все элементы по очереди проверяется на искомый элемент. Поиск заканчивается в случае, если найден искомый элемент или достигаем конца списка.

Алгоритм бинарного поиска.

Бинарный (двоичный) поиск принимает список элементов и помещает их в отсортированный пул поиска. Это исключает большую часть поискового пула с одним сравнением. Двоичное первый исследует средний элемент списка - если она соответствует цели, поиск окончен. Если этого не произойдет, только половина из оставшихся элементов нужно искать. Так как они сортируются, цель может быть только в одной половине другой.

Процесс продолжается путем сравнения среднего элемента с оставшимися кандидатами на сравнение. Каждое сравнение исключает приблизительно половину оставшихся данных. В конце концов, либо искомая цель найдена, или данные для

поиска исчерпаны.

```
    if (found != null)
            System.out.println ("Found: " + found);
         else
            System.out.println ("The contact was not
found.");
         System.out.println ();
         Sorting.selectionSort(friends);
         test = new Contact ("Mario", "Guzman", "");
         found = (Contact) Searching.binarySearch(friends,
test);
         if (found != null)
            System.out.println ("Found: " + found);
         else
            System.out.println ("The contact was not
found.");
       } }
```

Java Framework Collection

Класс `ArrayList` часть пакета java.util. Как и массив, он может хранить список значений и ссылаться на каждый из них, используя числовой индекс (порядковый номер элемента). Тем не менее, вы не можете использовать синтаксис квадратных скобок для объектов `ArrayList`.

В `ArrayList` число объектов увеличивается или уменьшается по мере необходимости, корректировка мощности происходит по мере необходимости.

Элементы могут быть вставлены или удалены с помощью одного лишь вызова соответствующего метода. Когда элемент вставлен, другие элементы "отодвигаются в сторону", чтобы освободить место тем, которые мы добавили. Точно так же, когда элемент удаляется, то список “сжимается”. В Индексы элементов вносятся соответствующие коррективы

`ArrayList` хранит ссылки объектов класса `Object`, что позволяет ему хранить любой вид объекта. Мы также можем определить объект `ArrayList` принимал конкретный тип объекта. Следующее объявление создает объект `ArrayList` который хранит только объекты `Family`.

```
ArrayList<Student> group = new ArrayList<Student>
```

В этом примере используются generics, которые мы обсудим позже.

```
import java.util.ArrayList;
public class Insects{
public static void main (String[] args) {
      ArrayList box = new ArrayList();
      box.add (”fly");
      box.add (”moskito");
      box.add (”spider");
      System.out.println (box);
      int location = box.indexOf (”fly");
      box.remove (location);
      System.out.println (box);
      System.out.println ("At index 1: " + box.get(1));
      box.add (2, ”spider");
      System.out.println (box);
      System.out.println (” quantity in box: " +
box.size());
       for(String box : box){
            System.out.println(box);
         }
       if(box.contains(”fly")){
            System.out.println(«В коллекции есть fly");
        }
  }
}
```

Эффективность ArrayList

Класс ArrayList реализован с использованием базового массива. Массивом объектов можно манипулировать, причем индексы остаются непрерывными при добавлении или удалении элементов. Если элементы добавляются и удаляются из конца списка, эта обработка будет довольно эффективной. Но, как только элементы вставляются и удаляются в начале или в середине списка, то остальные элементы сразу же сдвигаются.

Пример реализации коллекции ArrayList для хранения объектов разных типов:

```
public interface Product {/* ... */}
public class Picture implements Product { /* ... */ }
public class Shoe implements Product { /* ... */ }
public class Book implements Product { /* ... */ }
```

```
public class Toy implements Product { /* ... */ }

List< Product > products = new ArrayList< Product >();

products.add(new Picture());
products.add(new Shoe());
products.add(new Book());
products.add(new Toy());
```

Здесь, наверное, логичнее будет использовать абстрактный класс `Product` и наследоваться от него, чем делать имплементация интерфейса. Product может содержать методы общие для всех потомков, а вообще - коллекция более гибкий инструмент для такого рода данных, чем примитивный массив.

Алгоритм для копирования связанный список

```
public Node copyList (Node p)
{
   Node q;
   q = null;
   if (p != null)
          {
    q = new Node();
    q.data = p.data;
    q.link = copyList(p.link);
       }
   return q;
}
```

Лекция 4. Работа со строками в Java

Кратко о классе String

В Java строки представляют собой неизменяемую последовательность символов Unicode. В отличие от представления в C / C ++, где строка является просто массивом типа char, любая Java, строка является объектом класса java.lang. Однако Java строка, представляет собой в отличие от других используемых классов особый класс, который обладает довольно специфичными характеристиками. Отличия класса строк от обычных классов:

- Java строка представляет из себя строку литералов (текст), помещенных в двойные кавычки, например:

"Hello , World! ". Вы можете присвоить последовательность строковых литералов непосредственно переменной типа String, вместо того чтобы вызывать конструктор для создания экземпляра класса String.

- Оператор '+' является перегруженным, для объектов типа String, и всегда используется, чтобы объединить две строки операндов. В данном контексте мы говорим об операции конкатенации или сложения строк. Хотя '+' не работает как оператор сложения для любых других объектов, кроме строк, например, таких как Point и Circle.
- Строка является неизменяемой, то есть, символьной константой. Это значит, что ее содержание не может быть изменено после ее (строки как объекта) создания. Например, метод toUpperCase () – преобразования к верхнему регистру создает и возвращает новую строку вместо изменения содержания существующей строки.

Наиболее часто используемые методы класса String приведены ниже. Обратитесь к API JDK для того чтобы ознакомиться с полным списком возможностей класса String в java.lang.String.

```
//длина
int length()        // возвращает длину String
boolean isEmpty()   // то же самое thisString.length == 0
```

```
// сравнение
boolean equals(String another) // НЕЛЬЗЯ использовать'==' или'!=' для сравнения объектов String в Java
boolean equalsIgnoreCase(String another)
int compareTo(String another) // возвращает 0 если эта строка совпадает с another;
                              // <0 если лексикографически меньше another; or >0
int compareToIgnoreCase(String another)
boolean startsWith(String another)
boolean startsWith(String another, int fromIndex) // поиск начинается с fromIndex
boolean endsWith(String another)

// поиск & индесирование
int indexOf(String search)
int indexOf(String search, int fromIndex)
int indexOf(int character)
int indexOf(int character, int fromIndex)     // поиск вперед от fromIndex
int lastIndexOf(String search)
int lastIndexOf(String search, int fromIndex) // поиск назад от fromIndex
int lastIndexOf(int character)
int lastIndexOf(int character, int fromIndex)

// выделение  char или части строки из String (подстрока)
char charAt(int index)              // позиция от 0 до  (длина строки-1)
String substring(int fromIndex)
String substring(int fromIndex, int endIndex)  // exclude endIndex

// создается новый String или char[] из исходного (Strings не изменяются!)
String toLowerCase() //преобразование к нижнему регистру
String toUpperCase() //преобразование к верхнему регистру
String trim()           // создается новый String с помощью удаления пробелов спереди и сзади
String replace(char oldChar, char newChar) // создание нового String со старым  oldChar перемещается посредством буфера newChar
String concat(String another)                 // то же самое какthisString + другое
char[] toCharArray()                      // создается char[]
```

```
из string
void getChars(int srcBegin, int srcEnd, char[] dst, int
dstBegin)  // копируется в массив назначения dst char[]

// статические методы для преобразования примитивов в String
static String ValueOf(type arg)  // тип может быть примитивный
или char[]

// статические методы дают форматированный String используя
спецификаторы форматирования
static String format(String formattingString, Object... args)
// так же как printf()

// регулярные выражения (JDK 1.4)
boolean matches(String regexe)
String replaceAll(String regexe, String replacement)
String replaceAll(String regexe, String replacement)
String[] split(String regexe)             // разделяет String
используя regexe как разделитель,
                                          //       возвращает
массив String
String[] split(String regexe, int count)   // для подсчета
количества раз только (count)
```

Статический метод String.format() (JDK 1.5)

Статический метод String.format() (введен в JDK 1.5) может быть использован для получения форматированного вывода, таким же образом как это делается в языке Си с использованием функции printf() и спецификаторов вывода для различных типов данных. Метод format() делает то же самое, что функция printf(). Например:

```
String.format("%.1f", 1.234);   // возвращает String "1.2"
```

Удобно использовать String.format(), если вам нужно получить простую отформатированную строку для некоторых целей (например, для использования в методе ToString()). Для сложных строк, нужно использовать StringBuffer / StringBuilder с Formatter. Если вам просто нужно отправить простую отформатированную строку на консоль, то просто воспользуйтесь методом System.out.printf(), например:

System.out.printf("%.1f", 1.234);

Пример использования методов lastIndexOf() и substring() в пользовательском классе Filename

```
1   public class Filename {
2       private String fullPath;
3       private char pathSeparator,
4                    extensionSeparator;
5       public Filename(String str, char sep, char ext) {
6           fullPath = str;
7           pathSeparator = sep;
8           extensionSeparator = ext;
9       }
10      public String extension() {
11          int dot = fullPath.lastIndexOf(extensionSeparator);
12          return fullPath.substring(dot + 1);
13      }
14       // получение имени файла без расширения
15      public String filename() {
16          int dot = fullPath.lastIndexOf(extensionSeparator);
17          int sep = fullPath.lastIndexOf(pathSeparator);
18          return fullPath.substring(sep + 1, dot);
19      }
20      public String path() {
```

```
21          int sep = fullPath.lastIndexOf(pathSeparator);
22          return fullPath.substring(0, sep);
23      }
```

Теперь рассмотрим программу, которая использует класс Filename:

```
1   public class FilenameTester {
2       public static void main(String[] args) {
3           final String FPATH = "/home/user/index.html";
4           Filename myHomePage = new Filename(FPATH, '/', '.');
5           System.out.println("Extension = " + myHomePage.extens
6           System.out.println("Filename = " + myHomePage.filenar
7           System.out.println("Path = " + myHomePage.path());
8       }
9   }
```

```
Программа выведет:
Extension = html
Filename = index
Path = /home/user
```

Особенности класса String

Строки получили специальное значение в Java , потому что они часто используются в любой программе. Следовательно, эффективность работы с ними (с точки зрения вычислений и хранения) имеет решающее значение.

Разработчики Java все-таки решили сохранить примитивные типы в объектно-ориентированном языке, вместо того, чтобы сделать вообще все в виде объектов. Нужно сказать, что сделано это в первую очередь, для того чтобы повысить производительность языка. Ведь примитивы хранятся в стеке вызовов, и следовательно, требуют меньше пространства для хранения, и ими легче управлять. С другой стороны, объекты хранятся в области памяти, которую используют

программы, и которая называется “куча” (heap), а этот механизм требуют сложного управления памятью и потребляет гораздо больше места для хранения.

По соображениям производительности, класс String в Java разработан, так, чтобы быть чем-то промежуточным между примитивными типами данных и типами данных типа класс. Как уже было отмечено выше специальные характеристики типа String включают в себя:

- '+' оператор, который выполняет сложение примитивных типов данных (таких, как int и double), и перегружен, чтобы работать на объектах String. Операция '+' выполняет конкатенацию двух операндов типа String.
- Java не поддерживает механизма перегрузки операций по разработке программного обеспечения. В языке, который поддерживает перегрузку операций, например C ++, вы можете превратить оператор '+' (с помощью перегрузки) в оператор для выполнения сложения или вообще вычитания, например двух матриц, кстати это будет примером плохого кода. В Java оператор '+' является единственным оператором, который внутренне перегружен, чтобы поддержать конкатенацию (сложение) строк в Java. Нужно принять к сведению, что '+' не работает на любых других произвольных объектах, помимо строк, например, таких как рассмотренные нами ранее классы Point или Circle.

Теперь, собственно о строках. Существуют несколько способов создания строк. Строка String может быть получена одним из способов:

- непосредственно из присвоения строкового литерала ссылке типа String – таким же способом как примитивные типы данных;
- или с помощью оператора new и конструктора класса String, аналогично вызову конструктора любого другого класса. Тем не менее, этот способ не часто используется и использовать его не рекомендуется.

```
Для примера:
String str1 = "Java is Hot"; // неявный вызов конструктора
через присваивание строкового литерала
String str2 = new String("I'm cool");
// явный вызов конструктора через new
```

```
char[] array = { 'h', 'e', 'l', 'l', 'o', '.' };
//еще один способ создания строки
```

В первом случае str1 объявлена как ссылка типа String и инициализируется строкой " Java is Hot " . Во второй строрке, str2 объявлена как ссылка на строку и инициализируется с помощью вызова оператора new и конструктора, который инициализирует ее значением “ I'm cool” .

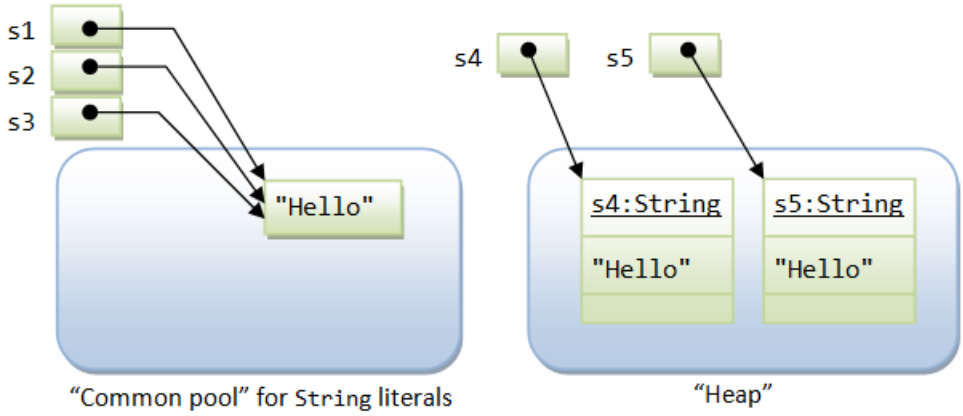
- Строковые литералы хранятся в общем пуле. Это облегчает совместное использование памяти для строк с тем же содержанием в целях сохранения памяти. Объекты строк , выделенные с помощью оператора new хранятся в куче (heap), а там нет разделяемого хранилища для того же самого контента (содержания).

Строковые литералы и Объекты типа String

Как уже упоминалось, есть два способа создания строк: неявное создание путем присвоения строкового литерала переменной или явного создания объекта String, через вызов оператора new и вызов конструктора. Например:

```
String s1 = "Hello";                // String литерал
String s2 = "Hello";                // String литерал
String s3 = s1;                    // одинаковые ссылки
String s4 = new String("Hello");   // String объект
String s5 = new String("Hello");   // String объект
```

Java предоставляет специальный механизм для хранения последовательностей символьных литералов (строк), так называемый общий пул строк. Если две последовательности литералов (строки) имеют одинаковое содержание, то они разделяют общее пространство для хранения внутри общего пула. Такой подход принят для того чтобы сохранить место для хранения часто используемых строк. С другой стороны, объекты типа String (строки), созданные с помощью оператора new и конструктора хранятся в куче.



Каждый объект String в куче имеет свое собственное место для хранения, как и любой другой объект. Там нет обмена хранения в куче, даже если два объекта Строковые имеют то же содержание.

А в куче нет разделяемого пространства для хранения двух объектов, даже если эти два объекта являются объектами типа String и имеют одинаковое содержание.

Вы можете использовать метод equals() класса String для сравнения содержимого двух строк. Вы можете использовать оператор сравнения на равенство '==', чтобы сравнить ссылки (или указатели) двух объектов. Изучите следующие коды:

```
s1 == s1;          // true, одинаковые ссылки
s1 == s2;          // true, s1 and s1 разделяют общий пул
s1 == s3;           // true, s3 получает то же самое значение
что ссылка s1
s1.equals(s3);     // true, одинаковое содержимое
s1 == s4;          // false, различные ссылки
s1.equals(s4);     // true, одинаковое содержимое
s4 == s5;          // false, различные ссылки в куче
s4.equals(s5);     // true, одинаковое содержимое
```

Важные замечания

- В приведенном выше примере, используется оператор отношения для того чтобы проверить на равенство '==' ссылки двух объектов String. Это сделано, чтобы показать различия между строковыми последовательностями литералов, которые используют совместное пространство для хранения в

общем пуле строк и объектов String, созданных в куче. Это логическая ошибка в использовании выражения (str1 == str2) в программе, чтобы сравнить содержимое двух объектов типа String.

- Строка может быть создана непосредственно путем присваивания последовательности литералов (строки), которая разделяет общий пул строк. Не рекомендуется использовать оператор new для создания объектов String в куче.

String является неизменяемым

С тех самых пор, когда в языке Java появились возможности по использованию разделяемого пространства для хранения строк с одинаковым содержанием в виде строкового пула, String в Java стали неизменяемыми. То есть, как только строка создается (как объект в памяти программы), ее содержание не может быть изменено никаким образом (по аналогии с Си – строковые литералы — это символьные константы). В противном случае, если этого не сделать, другие ссылки String разделяющие ту же самую ячейку памяти будут зависеть от изменений, которые могут быть непредсказуемыми и, следовательно, является нежелательными.

Такой метод, как например, toUpperCase () казалось-бы может изменить содержимое объекта String.

Хотя на самом деле, создается совершенно новый объект String и возвращается как раз он в точку вызова. Исходный объект-строка будет впоследствии удален сборщиком мусора (Garbage-collected), как только не окажется больше ссылок, которые ссылаются на него.

Вот поэтому-то объект типа String и считается неизменяемым объектом, вследствие этого, считается не эффективным использовать тип String, например, в том случае, если вам нужно часто модифицировать строку (так вы в таком случае будете создавать много новых объектов типа String, которые каждый раз будут занимать новые места для хранения). Например,

```
// неэффективный код
```

```
String str = "Hello";
for (int i = 1; i < 1000; ++i) {
   str = str + i;
}
```

Классы StringBuffer и StringBuilder

Как объяснялось выше, строки String являются неизменяемыми, поэтому строковые литералы с таким контентом хранятся в пуле строк. Изменение содержимого одной строки непосредственно может вызвать нежелательные побочные эффекты и может повлиять на другие строки, использующие ту же память.

JDK предоставляет два класса для поддержки возможностей по изменению строк: это классы StringBuffer и StringBuilder (входят в основной пакет java.lang).

Объекты StringBuffer или StringBuilder так же, как и любые другие обычные объекты, которые хранятся в куче, а не совместно в общем пуле, и, следовательно, могут быть изменены , не вызывая нехороших побочных эффектов на другие объекты .

Класс StringBuilder как класс был введён в JDK 1.5. Это то же самое , как использование класса StringBuffer, за исключением того, что StringBuilder не синхронизирован по многопоточных операций. Тем не менее, для программы в виде одного потока или нити управления, использование класса StringBuilder, без накладных расходов на синхронизацию, является более эффективным.

- **Использоание java.lang.StringBuffer**

Прочитайте спецификацию API JDK для использования java.lang.StringBuffer.

Методы класса:

```
// конструкторы
StringBuffer()                          // инициализация пустым
anStringBuffer
StringBuffer(int  size)              //  определяет  размер  при
инициализации
StringBuffer(String s)      //инициализируетя  содержимым   s
// Length
```

```
int length()

// Методы для конструирования содержимого
StringBuffer append(type arg)  // тип может бытьпримитивным,
char[], String, StringBuffer, и т.д.
StringBuffer insert(int offset, arg)

// Методы для манипуляции содержимым
StringBuffer delete(int start, int end)
StringBuffer deleteCharAt(int index)
void setLength(int newSize)
void setCharAt(int index, char newChar)
StringBuffer replace(int start, int end, String s)
StringBuffer reverse()

// Методы для выделения целого/части содержимого
char charAt(int index)
String substring(int start)
String substring(int start, int end)
String toString()

// Методы для поиска
int indexOf(String searchKey)
int indexOf(String searchKey, int fromIndex)
int lastIndexOf(String searchKey)

int lastIndexOf(String searchKey, int fromIndex)
```

Обратите внимание, что объект класса StringBuffer является обычным объектом в прямом понимании этого слова. Вам нужно будет использовать конструктор для создания объектов типа класс StringBuffer (вместо назначения в строку буквальном). Кроме того, оператор '+' не применяется к объектам, в том числе и к объектам StringBuffer . Вы должны будете использовать такой метод, как append() или insert() чтобы манипулировать StringBuffer .

Чтобы создать строку из частей, более эффективно использовать класс StringBuffer (для многопоточных программ) или StringBuilder (для однопоточных), вместо конкатенации строк. Например,

```
// Создадим строку типа YYYY-MM-DD HH:MM:SS
```

```
int year = 2010, month = 10, day = 10;
int hour = 10, minute = 10, second = 10;
String dateStr = new StringBuilder().append(year).append("-
").append(month).append("").append(day).append("").append(ho
ur)
.apend(":").append(minute).append(":").append(second).toStri
ng();
System.out.println(dateStr);

// StringBuilder более эффективный конкатенация String
String anotherDataStr = year + "-" + month + "-" + day + " "
+ hour + ":" + minute + ":" + second;
System.out.println(anotherDataStr);
```

Компилятор JDK , по сути, использует оба класса как String, так и StringBuffer для обработки конкатенации через операцию сложения строк '+'. Для примера,

```
String msg = "a" + "b" + "c";
```

будут скомпилированы в следующий код для повышения эффективности:

```
String msg = new
StringBuffer().append("a").append("b").append("c").toString(
);
```

В этом процессе создаются, промежуточный объект StringBuffer и возвращаемый объект String.

Использование java.lang.StringBuilder (JDK 1.5)

Программа ниже демонстрирует разные способы инвертирования длинных строк. Сравниваются три способа работы со строками: как с объектами класса String, так и с помощью StringBuffer и StringBuilder с использованием метода reverse(). Для измерения времени выполнения различных участков кода в примере используется метод

```
public static native long nanoTime();
```

Возвращает текущее значение наиболее точное время системных часов (таймера), в наносекундах.

Этот метод можно использовать только для измерения затраченного времени на выполнение операций и никак не связан с системным временем и текущим мировым временем.

Возвращаемое значение представлено в виде наносекунд, с момента фиксации, на любой произвольный момент времени. Например, чтобы определить, сколько времени занимает некоторый код, чтобы выполнить

```
     /*
     *   long startTime = System.nanoTime();
     *   // ... the code being measured ...
     *   long estimatedTime = System.nanoTime() - startTime;
     *
     *
     * @return The current value of the system timer, in
nanoseconds.
     * @since 1.5
     */
```

```
1 // Reversing a long String via a String vs. a StringBuffer
2 public class StringsBenchMark {
3    public static void main(String[] args) {
4       long beginTime, elapsedTime;
5
6       // Build a long string
7       String str = "";
8       int size = 16536;
9       char ch = 'a';
10      beginTime = System.nanoTime();   // Reference time in nano
11      for (int count = 0; count < size; ++count) {
12         str += ch;
13         ++ch;
14         if (ch > 'z') {
15            ch = 'a';
16         }
17      }
18      elapsedTime = System.nanoTime() - beginTime;
19      System.out.println("Elapsed Time is " + elapsedTime/1000 +
20
21      // Reverse a String by building another String character-b
22      String strReverse = "";
23      beginTime = System.nanoTime();
24      for (int pos = str.length() - 1; pos >= 0 ; pos--) {
```

```
25          strReverse += str.charAt(pos);   // Concatenate
26       }
27       elapsedTime = System.nanoTime() - beginTime;
28       System.out.println("Elapsed Time is " + elapsedTime/1000 +
29
30       // Reverse a String via an empty StringBuffer by appending
31       beginTime = System.nanoTime();
32       StringBuffer sBufferReverse = new StringBuffer(size);
33       for (int pos = str.length() - 1; pos >= 0 ; pos--) {
34          sBufferReverse.append(str.charAt(pos));      // append
35       }
36       elapsedTime = System.nanoTime() - beginTime;
37       System.out.println("Elapsed Time is " + elapsedTime/1000 +
38
39       // Reverse a String by creating a StringBuffer with the gi
40       beginTime = System.nanoTime();
41       StringBuffer sBufferReverseMethod = new StringBuffer(str);
42       sBufferReverseMethod.reverse();     // use reverse() metho
43       elapsedTime = System.nanoTime() - beginTime;
44       System.out.println("Elapsed Time is " + elapsedTime/1000 +
45
46       // Reverse a String via an empty StringBuilder by appendir
47       beginTime = System.nanoTime();
48       StringBuilder sBuilderReverse = new StringBuilder(size);
49       for (int pos = str.length() - 1; pos >= 0 ; pos--) {
50          sBuilderReverse.append(str.charAt(pos));
51       }
52       elapsedTime = System.nanoTime() - beginTime;
53       System.out.println("Elapsed Time is " + elapsedTime/1000 +
54
55       // Reverse a String by creating a StringBuilder with the g
56       beginTime = System.nanoTime();
57       StringBuffer sBuilderReverseMethod = new StringBuffer(str)
58       sBuilderReverseMethod.reverse();
59       elapsedTime = System.nanoTime() - beginTime;
60       System.out.println("Elapsed Time is " + elapsedTime/1000 +
61    }
62 }
```

```
Elapsed Time is 332100 usec (Build String)

Elapsed Time is 346639 usec (Using String to reverse)
Elapsed Time is 2028 usec   (Using StringBuffer to reverse)
Elapsed Time is 847 usec     (Using StringBuffer's reverse()
method)
Elapsed Time is 1092 usec   (Using StringBuilder to reverse)
```

```
Elapsed Time is 836 usec    (Using StringBuidler's reverse()
method)
```

Обратите внимание, что StringBuilder в 2 раза быстрее, чем StringBuffer, и в 300 раз быстрее, чем String. Метод reverse() работает быстрее всего, и занимает примерно одинаковое время как для StringBuilder, так и для StringBuffer.

Пример выполнения программы:

```
Elapsed Time is 332100 usec (Build String)
Elapsed Time is 346639 usec (Using String to reverse)
Elapsed Time is 2028 usec   (Using StringBuffer to reverse)
Elapsed Time is 847 usec    (Using StringBuffer's reverse()
method)
Elapsed Time is 1092 usec   (Using StringBuilder to reverse)
Elapsed Time is 836 usec    (Using StringBuidler's reverse()
method)
```

Механизм исключительных ситуаций

Механизм исключительных ситуаций в Java поддерживается пятью ключевыми словами:

- try
- catch
- finally
- throw
- throws

В Java всего около 50 ключевых слов, и пять из них связано с исключениями: try, catch, finally, throw, throws. Из них catch, throw и throws применяются к экземплярам класса, причём работают они только с Throwable и его наследниками.

На рисунке 1 представлена иерархия классов исключений, используемая в Java.

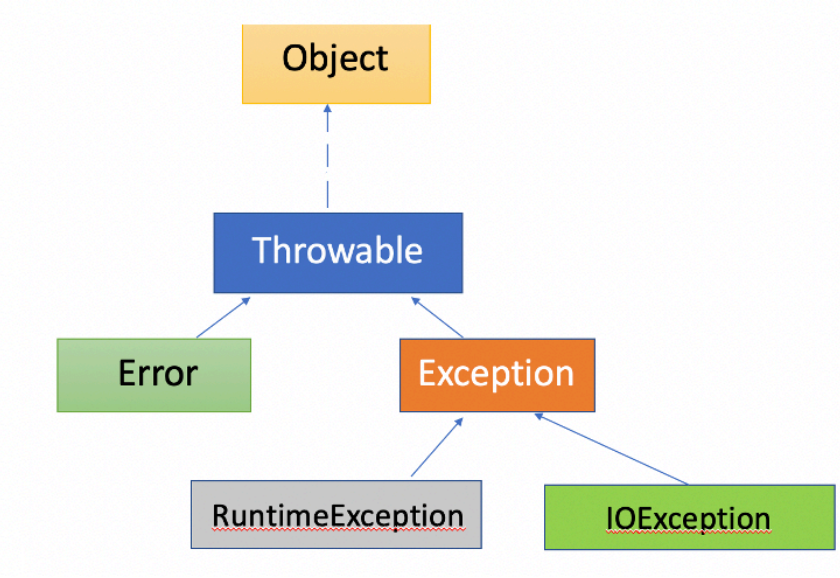


Рисунок 1 иерархия классов исключений

Наиболее популярные исключений в Java представлены в таблице 1.

Таблица 1. Классы исключений в Java

№ пп	Класс исключения	Класс предок/тип
Требования	ArithmeticException	RuntimeException
	NegativeArraySizeException	RuntimeException
	ArrayIndexOutOfBoundsException	RuntimeException
	NoSuchElementException	RuntimeException
	ArrayStoreException	RuntimeException
	NotSerializableException	Exception
	AssertionError	Error
	NullPointerException	RuntimeException
	ClassCastException	RuntimeException
	NumberFormatException	RuntimeException

	ClassNotFoundException	Exception
	OutOfMemoryError	Error
	CloneNotSupportedException	Exception
	SecurityException	RuntimeException
	ConcurrentModificationException	RuntimeException
	StackOverflowError	Error
	EOFException	Exception
	StringIndexOutOfBoundsException	RuntimeException
	FileNotFoundException	Exception
	ThreadDeath	Error
	IllegalArgumentException	RuntimeException
	UnsupportedEncodingException	Exception
	InterruptedException	Exception
	UnsupportedOperationException	RuntimeException

То, что исключения являются объектами важно по двум причинам:

1) они образуют иерархию с корнем java.lang.Throwable (java.lang.Object — предок java.lang.Throwable, но Object —это не исключение!)

2) они могут иметь поля и методы

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу класса родителя перехватывает исключения для экземпляров объектов как родительского класса, так или его наследников(т.е. экземпляры непосредственно самого родительского класса или любого его потомка).

Пример:

```
public class App {

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

```
        try {
           System.err.print(" 0");
           if (true) {throw new RuntimeException();}
           System.err.print(" 1");
        } catch (Exception e) {    // catch по Exception ПЕРЕХВАТЫВАЕТ RuntimeException
             System.err.print(" 2");
          }
        System.err.println(" 3");
      }// end main
}
```

Результат работы программы:

>> 0 2 3

Лекция 5. Паттерны проектирования программ. Метапрограммирование и использование дженериков

5.1 Паттерны проектирования программ

Что такое шаблоны проектирования?

"Каждый паттерн описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что этим ключом можно пользоваться при решении самых разнообразных задач". Кристофер Александр.

Шаблоны проектирования (паттерн, pattern) — это эффективные способы решения характерных задач проектирования, в частности проектирования компьютерных программ.

Паттерн не является законченным образцом проекта, который может быть прямо преобразован в код, скорее это описание или образец для того, как решить задачу, таким образом, чтобы это можно было использовать в различных ситуациях.

История появления шаблонов проектирования

В этом же году Эрих Гамма заканчивает свою докторскую работу и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) публикует книгу Design Patterns — Elements of Reusable Object-Oriented Software.

В этой книге описаны 23 шаблона проектирования. Также команда авторов этой книги известна общественности под названием «Банда четырёх» (англ. Gang of Four, часто сокращается до «GoF»). Именно эта книга стала причиной роста популярности шаблонов проектирования.

Список Паттернов:

- Порождающие паттерны:

1. Abstract Factory (Абстрактная Фабрика),
2. Builder (Строитель),
3. Factory Method (Фабричный Метод),
4. Prototype (Прототип),

5. Singleton (Одиночка),

– Структурные паттерны:

1. Adapter (Адаптер),
2. Bridge (Мост),
3. Composite (Компоновщик),
4. Decorator (Декоратор),
5. Facade (Фасад),
6. Flyweight (Приспособленец),
7. Proxy (Заместитель),

– Поведенческие паттерны:

1. Chain of Responsibility (Цепочка Обязанностей),
2. Command (Команда),
3. Interpreter (Интерпретатор),
4. Iterator (Итератор),
5. Mediator (Посредник),
6. Memento (Хранитель),
7. Observer (Наблюдатель),
8. State (Состояние),
9. Strategy (Стратегия),
10. Template Method (Шаблонный Метод),
11. Visitor (Посетитель).

Для чего их использовать

Программные системы, построенные с использованием паттернов (шаблонов) удобно сопровождать.

Как фигурист понимая, что он плохо исполняет определенный элемент танца, концентрируется на отработке именно этого элемента, при этом не повторяя те элементы, которые он исполняет хорошо, так и проектировщик программных систем, концентрируется на определенном (проблемном) элементе системы (если программная система построена с использованием шаблонов проектирования).

Формат описания паттернов проектирования

При рассмотрении паттернов проектирования используется единый формат описания. Описание каждого шаблона состоит из следующих разделов:

– Название:

Название паттерна (на Русском языке) отражающее его назначение. Также известен как альтернативное название паттерна (если такое название имеется).

– Классификация:

Классификация паттернов производится:

1. по цели (порождающий, структурный или поведенческий);
2. по применимости (к объектам и/или к классам).

– Частота использования (рисунок 5.1):

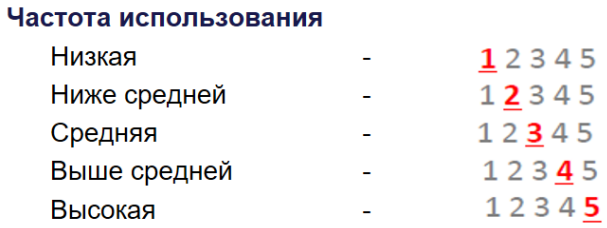


Рисунок 5.1 – Частота использования паттернов

Формат описания паттернов проектирования:

– Назначение.

Краткое описание назначения паттерна и задачи проектирования, решаемые с его использованием.

– Введение.

Описание паттерна с использованием метафор, позволяющих лучше понять идею, лежащую в основе паттерна, в общем виде охарактеризовать специфические аспекты использования паттерна проводя ассоциации с другими знакомыми процессами, для формирования ясного представления механизма работы паттерна.

– Структура паттерна на языке UML.

Графическое представление паттерна с использованием диаграмм классов языка UML. На диаграммах показаны основные участники (классы) и связи отношений между участниками.

– Структура паттерна на языке Java.

5.2 Программная реализация паттерна с использованием языка Java

– Участники.

Имена участников (классы, которые входят в состав паттерна) и описание их назначения.

– Отношения между участниками.

Описание отношений (взаимодействий) между участниками (классами и/или объектами).

– Мотивация.

Определение потребности в использовании паттерна. Рассмотрение способов применения паттерна.

– Применимость паттерна.

Рекомендации по применению паттерна.

– Результаты.

Особенности и варианты использования паттерна. Результаты применения.

– Реализация.

Описание вариантов и способов реализации паттерна.

Проблема которую нужно решить - пример использования паттерна

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик. В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь,

объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод. Схема нашей программы приведена на рисунке 5.2.

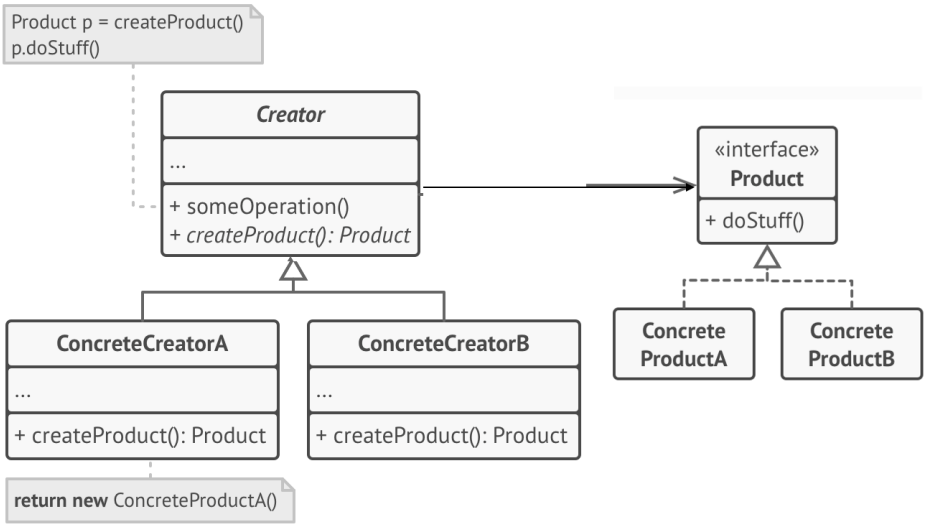


Рисунок 5.2 – Программа с применением паттерна

Участники Паттерна

– Product:

Определяет интерфейс объектов, которые создает фабричный метод.

– ConcreteProduct:

Реализует интерфейс продукта

– Creator:

Объявляет фабричный метод, который возвращает объект типа product. Может содержать стандартную реализацию фабричного метода. Создатель полагается на свои подклассы, чтобы определить фабричный метод, чтобы он возвращал экземпляр соответствующего конкретного продукта.

– ConcreteCreator:

Переопределяет фабричный метод для возврата экземпляра ConcreteProduct.

Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты. Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

5.3 Фабричный метод - factory method (Создание классов)

Основное по этому паттерну:

– определите интерфейс для создания объектов, но пусть подклассы сами решают, какие объекты создавать;

– Factory Method позволяет классу отложить создание экземпляров для подклассов.

Мотивация к использованию:

– Фреймворк использует абстрактные классы для определения и поддержания отношений между объектами в иерархии.

– Фреймворк также должен создавать объекты (он должен создавать экземпляры классов, но знает только об абстрактных классах – экземпляры которых он которые не может создавать.

– Фабричный метод инкапсулирует знания о том, какой подкласс создавать, - выводит эти знания за рамки.

Когда использовать

Используйте шаблон Factory Method, когда:

– вы заранее не можете предвидеть в классе сколько классов объектов на его основе будет создано;

– класс хочет, чтобы его подклассы указывали объекты, которые он создает;

– классы делегируют ответственность одному из нескольких вспомогательных подклассов, и вы хотите локализовать сведения о том, какой вспомогательный подкласс является делегатом;

– когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код;

– фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует;

– благодаря этому, код производства можно расширять, не трогая основной;

– так, для того чтобы, добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

Мотивация

– Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.

– Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?

– Решением будет дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить.

– Например, вы используете готовый UI-фреймворк для своего приложения. Но вот беда — требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс RoundButton. Но как сказать главному классу фреймворка UIFramework, чтобы он теперь создавал круглые кнопки, вместо стандартных?

Пример

Автозавод выпускает различные автомобили (объекты в виде машин).

В оригинале:

– разные классы реализуют интерфейс автомобиль;

– непосредственно создавать объекты автомобилей;

– нужно изменить клиента, чтобы сменить авто.

С использованием паттерна:

– используйте класс carFactory class, чтобы создавать объекты- машины;

– можно менять автомобиль, посредством изменения carFactory.

Пример

```
class 350Z implements Car;         // fast car
class Ram implements Car;     // truck
class Accord implements Car;              // family car
Car fast = new 350Z();             // returns fast car
public class carFactory {
   public static Car create(String type) {
     if (type.equals("fast"))     return new 350Z();
     if (type.equals("truck"))         return new Ram();
     else if (type.equals(“family”)   return new
Accord();
      }
}
Car fast = carFactory.create("fast"); // returns fast
car
```

5.4 Паттерн абстрактная фабрика.

Техника использования объекта-фабрики для порождения объектов-продуктов, была положена в основу всех порождающих паттернов. Методы, принадлежащие объекту-фабрике, которые порождают и возвращают объекты-продукты, принято называть фабричными-методами (или виртуальными конструкторами). Схема представлена на рисунке 5.3.

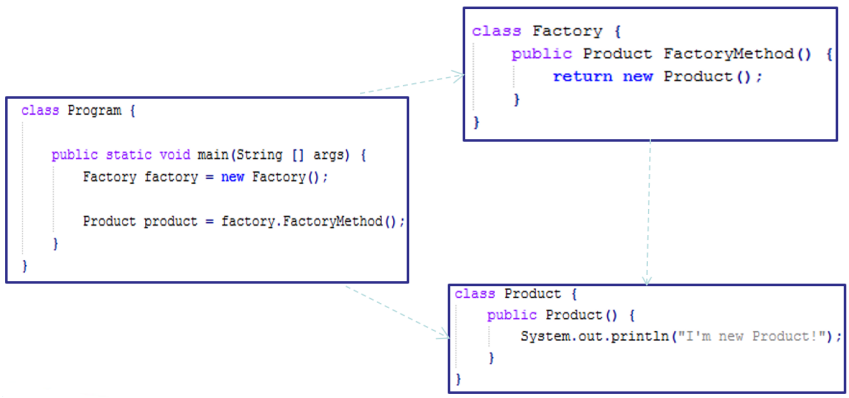


Рисунок 5.3 – Пример использования паттерна

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Арт-деко, Викторианском и Модерне.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

5.5 Дженерики или обобщенные типы в Java.

Пример 1 – Определение обобщенных типов:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
public interface Map<K,V> {
    V put(K key, V value);
}
```

Пример 2 –определение (своего собственного) типа дженерика:

```
public class GenericClass<T> {
    private T obj;
    public void setObj(T t) {obj = t;}
    public T getObj() {return obj;}
    public void print() {
        System.out.println(obj);
    }
}
Main:
GenericClass<Integer> g = new GenericClass<Integer>();
g.setObj(5); // auto-boxing
int i = g.getObj(); // auto-unboxing
g.print();
```

Пример

```
class Crossover {}
class Garage {
    private Crossover parkedAuto;
    public Garage(Crossover parkedAuto) {
        this.parkedAuto = parkedAuto;
    }
    public Crossover getParkedAuto(){ return
parkedAuto;}
}
class Man {
    public static void main(String[] args) {
        Garage garage = new Garage (new Crossover());
    }
}
```

Проблемы

Обычные классы и методы работаю с конкретными типами (примитивы или классы). Что делать, если нужно работать с разнообразными типами?

Пример иерархии типов

```
class Auto {}
class Crossover extends Auto{}
class Hatchback extends Auto{}
class Garage {
    private Auto parkedAuto;
    public Garage(Auto parkedAuto) { //конструктор
        this.parkedAuto = parkedAuto;
    }
    public Auto getParkedAuto(){ return parkedAuto;}
}
class Main {
    public static void main(String[] args) {
        Garage garage = new Garage (new Crossover());
        Garage anotherGarage = new Garage (new
Hatchback());
    }
}
```

Увеличение гибкости программ за счет наследования.

Код работает с объектами по ссылке на базовый тип. Этот код может быть использован для работы с производными типами. Достигаемая гибкость работает только в пределах одной иерархии (наследования).

```
interface Wheeled{} //общий интерфейс
class Crossover implements Wheeled{}
class Hatchback implements Wheeled{}
class Trailer   implements Wheeled{}
class Garage {
   private Wheeled parkedThing;
   public Garage(Wheeled parkedThing){this.parkedThing =
parkedThing;}
   public Wheeled getParkedThing(){ return parkedThing;}
}
class Main {
   public static void main(String[] args) {
```

```
            Garage garage = new Garage (new Crossover());
            Garage anotherGarage = new Garage (new
Trailer());
        }
    }
```

Увеличение гибкости за счет использования интерфейсов.

Преимущества, которые мы получаем:

– код работает с объектами по интерфейсной ссылке;

– этот код может быть использован для работы с любыми объектами, реализующими этот интерфейс;

– более гибкий механизм (по сравнению с наследованием), но требует реализации интерфейса.

```
    interface Wheeled{}
    class Crossover implements Wheeled{}
    class Hatchback implements Wheeled{}
    class Trailer   implements Wheeled{}
    class PotatoesBag {};
    class Garage {
       private Object keepingThing;
       public Garage(Object keepingThing){
          this.keepingThing = keepingThing;
       }
       public Object getKeepingThing(){ return keepingThing;
}
    }
    class Man {
       public static void main(String[] args) {
            Garage garage = new Garage (new PotatoesBag());
            Object o = garage.getKeepingThing();
            PotatoesBag bag;
            if (o instanceof PotatoesBag) bag = (PotatoesBag)
o;
       }
    }
```

Дженерики или обобщенные типы (generics).

– позволяют указать «условный тип», с которым работает код;

– используются, когда один и тот же код применим к большому числу типов;

– позволяют повысить безопасность, за счет контроля типов на стадии компиляции программы, а не выполнения;

– в основном используются контейнерами. Все интерфейсы и классы коллекций сделали параметризированными.

```
    interface Wheeled{}
    class Crossover implements Wheeled{}
    class Hatchback implements Wheeled{}
    class Trailer   implements Wheeled{}
    class PotatoesBag {};
    class Garage<T> {
       private T keepingThing;
       public Garage(T keepingThing){ this.keepingThing =
keepingThing; }
       public T getKeepingThing(){ return keepingThing; }
    }
    class Man {
       public static void main(String[] args) {
           Garage<Wheeled> garage = new Garage<Wheeled> (new
Crossover());
           Garage<PotatoesSack> anotherGarage =
               new Garage<PotatoesBag> (new PotatoesBag());
           PotatoesBag bag =
anotherGarage.getKeepingThing();
       }
    }
```

Особенности использования дженериков.

– код работает только с тем типом, который указан в угловых скобках;

– позволяют создавать более компактный код, чем при использовании ссылок типа `Object`;

– обеспечивают автоматическую проверку и приведение типов;

– позволяют создавать хороший повторно используемый код.

Особенности параметризованных типов.

Использовать примитивные типы в качестве параметрических типов нельзя.

Если одинаковые настраиваемые типы имеют различные аргументы (типы-параметры), то это различные типы.

```
//несовместимые, различные типы
Garage<Wheeled> garage;
Garage<Crossover> anotherGarage;
```

Статические компоненты класса не могут использовать его типы-параметры. Также настраиваемый класс не может расширять класс Throwable.

Стирание типа.

В обобщенном коде информация о параметрах-типах обобщения стирается (это значит, что при использовании обобщения любая конкретная информация о типе теряется). Тип известен только на стадии компиляции (во время статической проверки типов). После этого каждый обобщенный тип стирается, то есть заменяется необобщенным верхним ограничением. Компилятор обеспечивает внутреннюю целостность использования типов, контролируя их на «входе» и «выходе», самостоятельно выполняя приведения типов.

Миграционная совместимость.

Обобщения не были частью языка с самого начала, и появились только в версии 1.5. Стирание необходимо, чтобы можно было использовать обобщенный клиентский код с необобщенными библиотеками и наоборот.

Расплата за стирание

Операции, для выполнения которых нужно точно знать типы времени выполнения, работать не будут.

- Приведение типов: `(T) var;`
- Операция `instanceof`: `varue instaceof T;`
- Операция new `T var = new T();`

  `T[] array = new T[size]`
- Создание массива конкретного типа:

```
Type<Integer> arr = new Type<Integer>[10];
```

Применение параметризованных типов в классах

```
    class Name <T,E> {
        private T t;
        private E e;
        public Name(T t, E e) { this.t = t; this.e = e; }
        public T getT() { return t; }
        public void setE(E e) { this.e = e;}
    }


    class test {
     public static void main(String[] args){
       Name<String, Integer> obj1 = new Name<String,
Integer>("string", 3);
        String a = obj1.getA();
        obj1.setB(10);
        Name obj2 = new Name(3, new ArrayList(3));
        Integer i = (Integer)obj2.getA();
         }
    }
```

Особенности параметризованных типов.

Тип можно указывать не только для классов и интерфейсов, но и методов и конструкторов (не зависимо от того, параметризирован класс и интерфейс или нет. В этом случае тип указывается перед возвращаемым значением. При использовании одинаковых идентификаторов типа у класса и у метода или конструктора, тип последних скрывает тип класса (интерфейса).

Простые параметризованные типы в методах и конструкторах

```
    interface GIN <T> {
        <T> void meth1 (T t);  //Тип-параметр метода
скрывает тип интерфейса
        <E> E meth2 (E e);     //У метода свой тип-параметр
        T meth3 (T t);         //Метод использует тип
интерфейса
    }
    class GCL<T> implements GIN<T> {
        T t;
        public <T> GCL(T t){    }        //Конструктор
скрывает тип класса
        public <E> GCL (E e, T t) {    } //У конструктора
свой тип и он
                        //использует тип класса
        @Override
```

```
        public <T1> void meth1(T1 t1) { } //реализация
метода без путаницы                          //с именами типов-
параметров
        @Override
        public <E> E meth2(E e) {return e; }
        @Override
        public T meth3(T t) {return null;}
    }
```

Если информация о типе стирается, как вызвать определенный метод у объекта имеющего тип – параметр?

```
class GenericSpeaker<T> {
   T source;
   public void giveVoice() {
      //source.say(); ошибка компиляции
   }
}
```

Ограниченные типы

```
interface Speaker {
   void say()
}
class Cat implements Speaker{
   public void say() { System.out.println("maaauuuu"); }
}
class GenericSpeaker<T extends Speaker> {
   T source;
   public void giveVoice() {
      source.say();
   }}
```

Ограничение типа позволяет использовать у ссылок методы и поля, доступные в типе-ограничителе. Типы, не наследующие от указанного, не могут быть использованы при создании объектов.

```
Class Name<T extends ClassName &
             Interface1 & Interface2 & … > {...}
```

Как имя типа может быть указан 1 класс и множество интерфейсов, разделенных &. Как имя типа может быть указан ранее введенный параметр.

```
class GenericClass<T extends Comparable<T>> {...}
```

Маски и их использование.

Что делать при передаче экземпляров параметризованных типов в методы, т.е. как писать сигнатуру? Для этого используется маска <?>, обозначающая произвольный тип-параметр.

```
class Generic<T> {
  ...
  boolean compare(Generic<?> o) {
    return o.getObj() == obj;
}
```

Запись: `boolean compare(Generic<?> o) {…}`

означает, что в качестве параметра методу `compare` может быть передан тип `Generic`, параметризированный любым (а не конкретным, как если бы было `Generic<T>`) типом.

```
isEqual = ccompare(Generic<Runnable> o);
isEqual = compare(Generic<StringBuilder> o);
isEqual = compare(Generic<Calendar> o);
isEqual = compare(Generic<K> o);
ompare(Generic<Integer> o);
isEqual =
```

Маски позволяют задать отношение между параметризированными типами, а не типами-параметрами.

```
//пусть есть иерархия (рисунок 7.2)
class Fruit {}
class Apple extends Fruit{}
class Citrus extends Fruit{}
class Orange extends Citrus{}
```

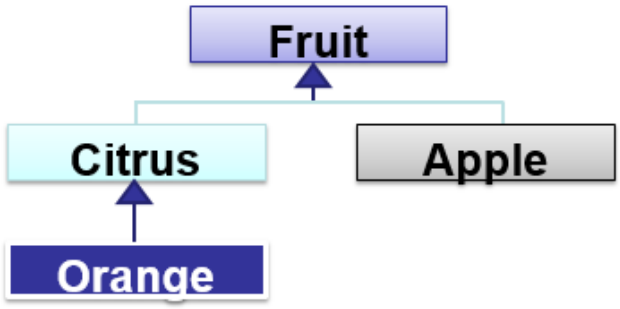


Рисунок 7.2 – Иерархия классов

```
    List<?> fruits = new ArrayList<Fruit>();
    //Все методы add не работают с ошибкой компиляции
    //add(capture<?>) в List не может быть применен к
<Тип_добавляемого_объекта>
    fruits.add(new Object());
fruits.add(new Fruit());
fruits.add(new Apple());
fruits.add(new Citrus());
fruits.add(new Orange());

//Этот метод работает
    Object o = fruits.get(0);
//Остальные методы get не работают, потому что get
возвращает Object и нужно приводить объект у нужному типу
    Fruit f = fruits.get(0);
Apple a = fruits.get(0);
Citrus c = fruits.get(0);
Orange or = fruits.get(0);
    \\Метод компилируется
    public static List<?> getList() {
    List<Fruit> fruits = new ArrayList<Fruit>();
    fruits.add(new Apple());
    return fruits;
}


    List<?> wFruits1 = getList();  \\Работает
List<Fruit> wFruits2 = getList();    \\Ошибка компиляции
List<Apple> wFruits3 = getList();  \\Ошибка компиляции


    public static void addToList(List<?> fruits) {
        \\add не работают все
    fruits.add(new Fruit());
```

```
    fruits.add(new Apple());
    fruits.add(new Citrus());
    fruits.add(new Orange());
        Object o = fruits.get(0); \\работает
        \\остальные get не работают
        Fruit f = fruits.get(0);
    Citrus c = fruits.get(0);
    Orange or = fruits.get(0);
}


    //все вызовы не вызывают ошибки компиляции
    addToList(new ArrayList<Fruit>());
    addToList(new ArrayList<Apple>());
    addToList(new ArrayList<Citrus>());
    addToList(new ArrayList());
```

Ограничение сверху.

Определяет отношение восходящего преобразования:

```
<? extends super>
```

Некоторый конкретный тип, параметр которого наследуется от `super`.

```
    List<? extends Fruit> fruits = new ArrayList<Fruit>();
    //Все методы add не работают с ошибкой компиляции
    fruits.add(new Object());
    fruits.add(new Fruit());
    fruits.add(new Apple());
    fruits.add(new Citrus());
    fruits.add(new Orange());

    //Этот метод работает
    Object o = fruits.get(0);
//Остальные методы get не работают, потому что get
возвращает Object и нужно приводить объект у нужному типу
    Fruit f = fruits.get(0);
    Apple a = fruits.get(0);
    Citrus c = fruits.get(0);
    Orange or = fruits.get(0);
```

Маски

```
    \\Метод компилируется
    public static List<?> getList() {
        List<Fruit> fruits = new ArrayList<Fruit>();
```

```
        fruits.add(new Apple());
        return fruits;
    }

    List<?> wFruits1 = getList();  \\Работает
    List<Fruit> wFruits2 = getList();\\Ошибка компиляции
    List<Apple> wFruits3 = getList();\\Ошибка компиляции

    public static void addToList(List<?> fruits) {
        \\add не работают все
        fruits.add(new Fruit());
        fruits.add(new Apple());
        fruits.add(new Citrus());
        fruits.add(new Orange());
        Object o = fruits.get(0); \\работает
        \\остальные get не работают
        Fruit f = fruits.get(0);
        Citrus c = fruits.get(0);
        Orange or = fruits.get(0);
    }

    //все вызовы не вызывают ошибки компиляции
    addToList(new ArrayList<Fruit>());
    addToList(new ArrayList<Apple>());
    addToList(new ArrayList<Citrus>());
    addToList(new ArrayList());
```

Ограничение снизу.

Определяет ограничение супертипа:

```
<? super sub>
```

Некоторый конкретный тип, параметр которого является суперклассом для `sub`.

Имеется:

```
List<Integer> list = new ArrayList<Integer>();
Map<Integer, Comparable<String>> m =
    new HashMap<Integer,Comparable<String>>();
// Зачем два раза писать одно и то же???
```

Необходимо:

```
List<Integer> list = new ArrayList<>();
Map<Integer, Comparable<String>> m =
    new HashMap<>();
```

Вывод типа (type inference) (Java 1.7)

Можно использовать, если компилятор из контекста может понять, какие типы нужны.

<> – «алмазная запись» (diamond notation)

В основном используется при создании объектов в ходе инициализации переменных. При создании объектов нельзя путать отсутствие указания типа (будет обобщенный тип) и оператор алмаз (будет вывод типа). Из-за механизма стирания в параметризованных типах могут возникать непроверяемые компилятором приведения.

Лекция 6. Абстрактные типы данных и их реализация на Java. Поведенческие паттерны

6.1 Использование регулярных выражений в Java.

Регулярные выражения — это инструмент, который задает шаблон для строк. Если у программиста стоит задача обработать большой набор строк и отыскать в нем нужную или проверить соответствует ли входящая строка определенному правилу оптимальное решение - регулярные выражения. Ниже, на рисунке 6.1, приведен пример использования регулярных выражений.

Pattern	Strings matching the pattern
'Alex%'	**Alex** **Alex**andr **Alex**ander **Alex**andra ....
'%x%'	Ma**x** Ma**x**im Alexandr ...
'%a'	Olg**a** Helen**a** Ir**a** ...

Рисунок 6.1 – Пример использования регулярных выражений

```
    import java.util.regex.Pattern;
    public class RegexExample {
    public static void isPasswordOK(String password) {
if(Pattern.matches("admin", password)) {
        System.out.println("Hi admin");
       }
    }
    public static void main(String[] args) {
       isPasswordOK("admin"); }
    }
```

Пакет `java.util.regex` включает три важных класса: `Pattern`, `Matcher`, `PatternSyntaxException` и интерфейс `MatchResult`.

Синтаксис регулярных выражений

Интерфейс `MatchResult` — результат операции сравнения. Класс `Matcher` — механизм, который выполняет операции сопоставления последовательности символов путем интерпретации шаблона

Класс `Pattern` — скомпилированное представление регулярного выражения. У класса `Pattern` есть метод `compile()`, который возвращает `Pattern`, соответствующий регулярному выражению. Метод `matches` — сравнивает выражение с набором символов и возвращает `true`, `false` в зависимости от того совпали строки или нет. Например, проверка пароля, которую мы делали через метод `equals` может быть реализована более элегантно с помощью метода `matches`.

Класс Pattern

Объект `Pattern` представляет собой скомпилированное представление регулярного выражения. Класс `Pattern` не предоставляет общедоступных конструкторов. Чтобы создать шаблон, вы должны сначала вызвать один из его открытых статических методов `compile()`, который затем вернет объект `Pattern`. Эти методы принимают регулярное выражение в качестве первого аргумента.

Класс Matcher

Объект `Matcher` – это механизм, который интерпретирует шаблон и выполняет операции сопоставления с входной строкой. Как и класс `Pattern`, `Matcher` не определяет общедоступных конструкторов. Вы получаете объект `Matcher`, вызывая метод `matcher()` для объекта `Pattern`.

Класс PatternSyntaxException

Объект PatternSyntaxException является непроверяемым исключением, которое указывает на синтаксическую ошибку в образце регулярного выражения.

Правила написания:

– . — точка — это соответствие любому символу;

– ^строка — находит регулярное выражение, которое должно совпадать в начале строки;

– строка$ — выражение, которое должно совпадать в конце строки;

– [абв] — только буквы а или б или в;

– [абв][яю] — только буквы а или б или в, за которыми следуют я или ю;

– [^abc] — когда символ каретки появляется в качестве первого символа в квадратных скобках, он отрицает шаблон. Этот шаблон соответствует любому символу, кроме a или b или c.

Метасимволы:

– \d — любая цифра — равнозначно [0-9];

– \D — только не цифра — равнозначно [^0-9];

– \s — символ пробела;

– \w — символ слова — равнозначно [a-zA-Z_0-9].

Квантификаторы

– * — символ звездочки означает от ноля до бесконечности;

– + — символ может встречаться от одного или несколько раз, сокращенно {1,};

– ? — встречается ни разу или один раз, знак вопроса это сокращение для {0,1};

– {X} — символ встречается X раз;

– {X,Y} — символ встречается от X до Y раз.

6.2 Обработка событий.

События Мыши

• События, связанные с мышью разделены на события мыши (рисунок 6.2) и события движения мыши (рисунок 6.3).

Нажатие Мыши	***кнопка мыши нажата***
Мышь отпущена	***Кнопка мыши отпущена***
Клик на мышь	***кнопка мыши нажата и освобожден без перемещения мыши между этими событиями***
Мышь вошла	***указатель мыши перемещается на (над) компонент***
Мышь вышла	***указатель мыши перемещается за пределы компонента***

Рисунок 6.2 – События мыши

Движение Мыши	мышь перемещается
Перетаскивание мышью	мышь перемещается, в то время пока нажата кнопка мыши

Рисунок 6.3 – События движения мыши

Слушатели событий мыши создаются при помощи интерфейсов `MouseListener` и `MouseMotionListener`. Объект `MouseEvent` передается в соответствующий метод, когда происходит событие мыши.

Для данной программы, мы можем обработать только одно или два события мыши. Для того, чтобы полностью реализовать интерфейс слушателя, пустые методы должны быть предусмотрены для всех неиспользуемых событий

Пример: Dots.java и DotsPanel.java

```
import javax.swing.JFrame;
public class Dots {
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Dots");
      frame.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().add (new DotsPanel());
      frame.pack();
      frame.setVisible(true);
   }
}

import javax.swing.*;
```

```
    import java.awt.*;
    import java.awt.event.*;
    import java.util.*;
    public class DotsPanel extends JPanel{
       private final int WIDTH = 300, HEIGHT = 200;
       private final int RADIUS = 6;
       private ArrayList pointList;
       private int count;
    //Устанавливаем эту панель, чтобы слушать события мыши.
    public DotsPanel() {
            pointList = new ArrayList();
          count = 0;
           addMouseListener (new DotsListener());
          setBackground (Color.black);
          setPreferredSize (new Dimension(WIDTH, HEIGHT));
       }

    //рисуем все точки, которые хранятся в списке
    public void paintComponent (Graphics page)  {
          super.paintComponent(page);
          page.setColor (Color.green);
          // создаем  итератор для ArrayList точек
    Iterator pointIterator = pointList.iterator();
          while (pointIterator.hasNext())  {
             Point drawPoint = (Point) pointIterator.next();
             page.fillOval (drawPoint.x - RADIUS,
drawPoint.y - RADIUS,
                            RADIUS * 2, RADIUS * 2);
          }
          page.drawString ("Count: " + count, 5, 15);
       }

    //класс слушателя событий мыши.
//*************************************************************
*******
       private class DotsListener implements MouseListener {
    //  добавляет текущую точку в список точек и
перерисовываем
          // При каждом нажатии кнопки мыши.
    public void mousePressed (MouseEvent event)  {
             pointList.add (event.getPoint());
             count++;
             repaint();
    //  Обеспечить пустые определения для неиспользуемых
методов событий.
```

```
        public void mouseClicked (MouseEvent event) {}
        public void mouseReleased (MouseEvent event) {}
        public void mouseEntered (MouseEvent event) {}
        public void mouseExited (MouseEvent event) {}
    }
}
```

Полиморфизм играет важную роль в развитии графического пользовательского интерфейса Java.

Как мы уже видели, мы устанавливаем связь между компонентом и слушателя:

```
JButton button = new JButton();
button.addActionListener(new MyListener());
```

Заметьте, что метод `addActionListener` принимает объект `MyListener`в качестве параметра. На самом деле, мы можем передать методу `addActionListener` любой объект, который реализует интерфейс `ActionListener`.

Код метода `addActionListener` принимает параметр `ActionListener` (интерфейс). Из-за полиморфизма, любой объект, который реализует этот интерфейс совместим с параметром ссылочной переменной.

Компонент может вызывать метод `actionPerformed` из-за связи между классом слушателем и интерфейсом. Расширение класса адаптера для создания слушателя представляет собой такую же ситуацию; класс адаптера уже реализует соответствующий интерфейс.

Диалоговые окна

Напомним, что диалоговое окно — это небольшое окно, которое "всплывает", чтобы взаимодействовать с пользователем в течение краткого время, для достижения конкретной цели.

Класс `JOptionPane` позволяет легко создавать диалоговые окна для представления информации, подтверждающей какие-то действия, или чтобы принимать некоторые значения ввода.

Давайте теперь рассмотрим использование диалоговых окон на двух других классах, которые позволяют нам создавать специализированные диалоговые окна.

Лекция 7. Стандартные классы Java Framework Collection и их использование в программах. Структурные паттерны

7.1 Стандартная библиотека Java Framework Collection и ее использование.

Коллекции в Java являются контейнерами объектов, которые благодаря полиморфизму может содержать объекты любого класса, производного от `Object` (который на самом деле и есть любой класс).

Какие бывают Коллекции?

Существуют два главных интерфейса (рисунок 7.1) для всех типов коллекций в Java:

– Collection<E>;

– Map<K,V>.

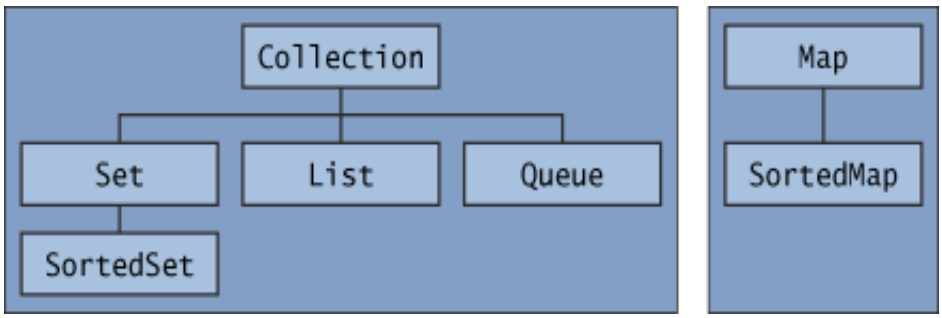


Рисунок 7.1 – Типы коллекций

Другие Коллекции.

– `Guava` (Google Collections Library) - библиотека добавляет несколько полезных реализаций структур данных, таких как мультимножество, мультиотображение и двунаправленное отображение. Улучшена эффективность.

– `Trove library` - реализация коллекций, позволяющая хранить примитивы (в Java Collections Framework примитивы хранить нельзя, только оберточные типы), что позволяет повысить эффективность работы.

– PCJ (Primitive Collections for Java) - так же, как и `Trove` предназначены для примитивных типов, что позволит повысить эффективность.

– Наконец, вы сами можете написать собственную коллекцию (тот же связной список). Но данный подход не рекомендуется.

Рекомендуется для начала необходимо освоить базовые коллекции Java которыми пользуются чаще всего. А также некоторые сторонние библиотеки реализуют интерфейсы Java Collections Framework (пример Guava http://guava-libraries.googlecode.com/svn/tags/release05/javadoc/overview-tree.html). То есть знание иерархии классов базовых коллекций позволит более быстро освоить сторонние библиотеки.

7.2 Базовые интерфейсы Java Framework Collection.

В библиотеке коллекций Java существует два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций:

– Collection - коллекция содержит набор объектов (элементов);

– Map -описывает коллекцию, состоящую из пар "ключ — значение".

Хоть фреймворк называется Java Collections Framework, но интерфейс map и его реализации входят в фреймворк тоже. Интерфейсы Collection и Map являются базовыми, но они не есть единственными. Их расширяют другие интерфейсы, добавляющие дополнительный функционал.

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

Map - описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (тар). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array). Никак НЕ относится к интерфейсу Collection и является самостоятельным.

Интерфейс Collection расширяют три базовых интерфейса:

– List;

– Set;

– Queue.

Рассмотрим, зачем нужен каждый:

List - Представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

Set - описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).

Queue - очередь. Это коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Класс java.util.LinkedList.

Используется, если вы хотите пройти по списку и посмотреть / изменить элементы в списке, вы используете объект итератора для этого списка (так же, как с другими коллекциями, которые реализуют Iterable интерфейс; все они имеют метод iterator (), который возвращает итератор).

7.3 Итераторы в Java

Итератор имеет доступ к элементам некоторой коллекции и может работать с ней в определенном порядке. Методы интерфейса Iterator<E> (над коллекцией или набором элементов типа E):

- `boolean hasNext() // говорит, что существуют ли какие-либо элементы слева, которые мы можем увидеть;`
- `E next() // возвращает следующий элемент;`
- `void remove() //удаляет текущий элемент из коллекции.`

Цели изучения интерфейсов.

При программировании на Java важно уметь использовать интерфейсы, поэтому нам необходимо их изучать. Для чего это нужно делать? Для того чтобы:

– узнать, что входит в состав интерфейса List;

– чтобы понять, как написать массив на основе реализации интерфейса List;

– изучить разницу между одно-, двусвязными, и циклическими связанными структурами данных в виде списков;

– чтобы узнать, как реализовать интерфейс `List`, используя связанный список;

– для того, чтобы понять, что такое интерфейс итератора.

Интерфейс `List` и класс `ArrayList`.

Массив является индексируемой структурой: можно выбрать все элементы в произвольном порядке, используя значение индекса. К элементам можно получить последовательный доступ с помощью цикла, который увеличивает индекс.

Вы не можете:

– увеличивать или уменьшать длину;

– добавить элемент в указанную позицию без перемещения других элементов, чтобы освободить место;

– удалить элемент в заданном положении без смещения других элементов, чтобы заполнить возникший пробел.

В интерфейс `List` включены следующие операции:

– поиск заданного элемента;

– добавление элемента в любой конец;

– удаление элемента с любого конца;

– обход структуры списка без использования индекса.

Не все классы выполняют перечисленные операции с одинаковой степенью эффективности. Массив предоставляет возможность хранить данные примитивных типов, тогда как в списке из объектов типа класс хранятся ссылки на объекты (этому способствует Autoboxing).

На рисунке 7.2 ниже приведена зависимость Java Collection. Красным здесь выделены интерфейсы, зеленым – абстрактные классы, а синим готовые реализации.

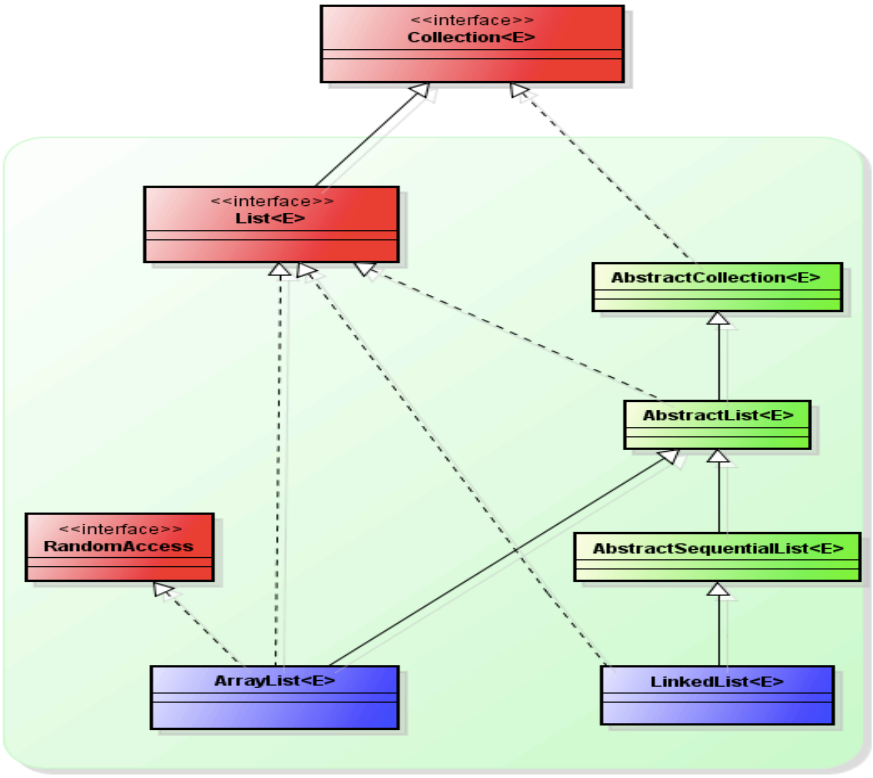


Рисунок 7.2 – Java Collection

`ArrayList` - самая часто используемая коллекция. `ArrayList` инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Так как `ArrayList` использует массив, то время доступа к элементу по индексу минимально (в отличии от `LinkedList`).

При удалении произвольного элемента из списка, все элементы, находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.

`LinkedList` - Двусвязный список.

Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка.

Доступ к произвольному элементу осуществляется за линейное время (но

доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, `LinkedList` в абсолютных величинах проигрывает `ArrayList` и по потребляемой памяти, и по скорости выполнения операций.

Класс `ArrayList`.

Довольно-таки простой класс, который реализует интерфейс List. Имеет улучшение по сравнению с массивами. Используется, если программист хочет добавить новые элементы в конец списка, но по-прежнему нуждается в возможности доступа к элементам, сохраненным в списке в произвольном порядке.

`ArrayList` является несихронизируемым динамически расширяемым массивом с эффективным доступом по индексу.

Пример:

```
ArrayList<Integer> arr =
    new ArrayList<Integer>(10/*initialCapacity*/);
arr.add(7);
```

`ArrayList` на самом деле не является списком (хотя реализует интерфейс `List`) Если вам нужен список, то используйте класс `LinkedList`.

Класс Vector.

`Vector` это и синхронизируемый динамически расширяемый массив массив с эффективным доступом по индексу.

Пример:

```
Vector<Integer> vec =
    new Vector<Integer>(10/*initialCapacity*/);
vec.add(7);
```

`Vector` является старым контейнером (Java 1.0) и сейчас гораздо реже используется, заменяется в основном `ArrayList` (Java 1.2), который не синхронизируется.

7.4 Использование родовых коллекций или Generic Collections.

Функция языка, **представленная**, начиная с версии Java 5.0 называется родовыми коллекциями (или дженериками).

Дженерики позволяют определить коллекцию, содержащую ссылки на объекты определенного типа:

```
List<String> myList = new ArrayList<String>();
```

Где `myList` является списком объектов `String`, где строка является параметрическим типом. В `myList` можно хранить только ссылки на объекты типа `String`, и все извлекаемые элементы тоже имеют тип `String`. Параметрические типы аналогичны параметрам методов. На рисунке 7.3 приведена спецификация класса `ArrayList`.

TABLE 4.1
Methods of Class `java.util.ArrayList<E>`

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `ArrayList`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `ArrayList`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `ArrayList`.
`public E remove(int index)`	Returns and removes the item at position `index` and shifts the items that follow it to fill the vacated space.

Рисунок 7.3 – Спецификация класса ArrayList

Использование `ArrayList`.

`ArrayList` дает дополнительные возможности сверх того, что обеспечивают обычные массивы. Сочетание Автоупаковки (Autoboxing) с Generic Collection (коллекциями дженериков) дает вам возможность хранить и извлекать примитивные типы данных при работе с `ArrayList`.

Односвязные списки и двусвязные списки. Реализация на Java.

`ArrayList`: методы добавления и удаления работают за линейное время, потому что они требуют, чтобы цикл сдвига элементов в массиве подстилающей.

`LinkedList` преодолевает это ограничение, предоставляя возможность добавлять или удалять элементы в любом месте в списке за постоянное время.

Каждый элемент списка (узел) содержит информационные поля и ссылку на следующий узел, и необязательно, ссылку на предыдущий узел.

Узел списка.

Узел содержит поля данных и одну или несколько ссылок. Ссылка является ссылкой на следующий узел. Узел обычно определяется внутри другого класса, что делает его внутренним классом (`inner`) для контейнера.

Схема односвязного списка на рисунке 7.4 – 7.6.

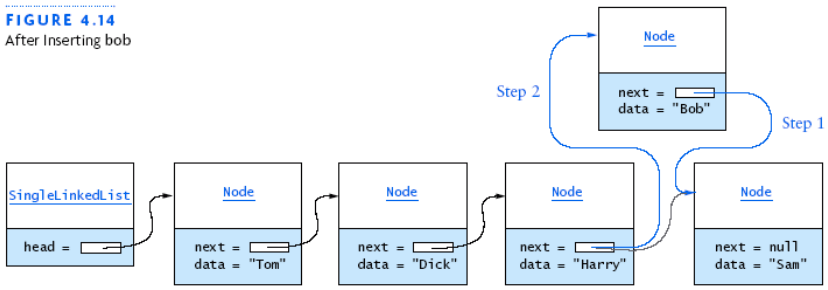


Рисунок 7.4 – Single-Linked Lists

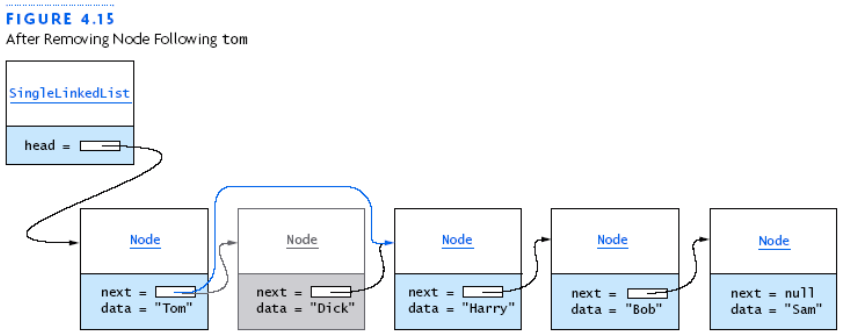


Рисунок 7.5 – Single-Linked Lists

DoubleLinkedList (двусвязный список).

Ограничения одно-связанный список включают в себя:

– вставка в начало списка O (1);

– вставка на другие позиции O(n) где n размер списка.

Можно вставить узел только после ссылочного узла. Можно удалить узел, только если у нас есть ссылка на его узел-предшественник. Может проходить по списку только в прямом направлении. Эти ограничения удаляются путем добавления ссылки в каждом узле к предыдущему узлу (двусвязный список, рисунок 7.8).

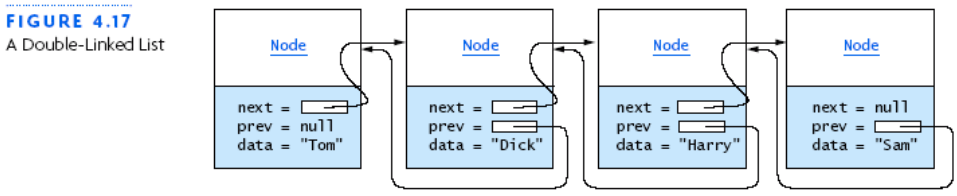


Рисунок 7.6 – Double-Linked Lists

Циклические списки.

Циклический связанный список (рисунок 7.7): связать последний узел двусвязного списка с первым узлом и первый с последним.

Преимущество: может двигаться в прямом или обратном направлении по списку, даже после того, как вы прошли последний или первый узел. Можно посетить все элементы списка из любой начальной точки. Никогда нельзя выйти за пределы списка (за последний элемент). Неудобство: бесконечный цикл.

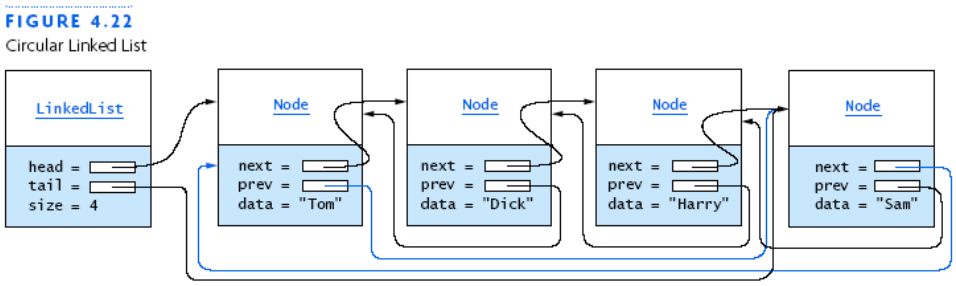


Рисунок 7.7 – Циклический список

Класс `LinkedList<E>`:

– часть Java API;

– реализует интерфейс `List<E>` с использованием двусвязного списка (double-linked list).

Методы данного класса указаны на рисунке 7.8.

TABLE 4.2
Selected Methods of the java.util.LinkedList<E> Class

Method	Behavior
public void add(int index, E obj)	Inserts object obj into the list at position index.
public void addFirst(E obj)	Inserts object obj as the first element of the list.
public void addLast(E obj)	Adds object obj to the end of the list.
public E get(int index)	Returns the item at position index.
public E getFirst()	Gets the first element in the list. Throws NoSuchElementException if list is empty.
public E getLast()	Gets the last element in the list. Throws NoSuchElementException if list is empty.
public boolean remove(E obj)	Removes the first occurrence of object obj from the list. Returns **true** if the list contained object obj; otherwise, returns **false**.
public int size()	Returns the number of objects contained in the list.

Рисунок 7.8 – Методы класса LinkedList<E>

Интерфейс Iterator<E>.

Интерфейс итератора `Iterator` составляет часть пакета API java.util. В интерфейсе `List` объявлен метод итератора, который возвращает объект итератора, который будет выполнять итерацию по элементам этого списка. `Iterator` (рисунок 7.9) не относится к какому-либо элементу и не указывает на конкретный узел в данный момент времени, но точки между узлами.

TABLE 4.3
The java.util.Iterator<E> Interface

Method	Behavior
boolean hasNext()	Returns **true** if the next method returns a value.
E next()	Returns the next element. If there are no more elements, throws the NoSuchElementException.
void remove()	Removes the last element returned by the next method.

Рисунок 7.9 – Методы интерфейса Iterator<E>

Интерфейс ListIterator<E>.

Ограничения `Iterator`:

– можно только пройти списка в прямом направлении;

– обеспечивает только метод удаления.

Необходимо заранее реализовывать итератор, используя свой собственный цикл, если исходное положение не в начале списка. `ListIterator<E>` (рисунок

7.10) является расширением `Iterator<E>` для преодоления ограничений. Можно представить Итератор как закладку, которая позиционируется между элементами связанного списка.

TABLE 4.4
The java.util.ListIterator<E> Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If method `previous` is called after add, the newly inserted object will be returned.
`boolean hasNext()`	Returns **`true`** if `next` will not throw an exception.
`boolean hasPrevious()`	Returns **`true`** if `previous` will not throw an exception.
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown.
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned.
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown.
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, –1 is returned.
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.

Рисунок 7.10 – Методы интерфейса ListIterator<E>

Сравнение итераторов Iterator и ListIterator.

`ListIterator` является подинтерфейсом интерфейса `Iterator`; классы, которые реализуют `ListIterator` обеспечивают все возможности и кроме того интерфейс итератора требует меньшего количества методов и может использоваться для итерации по более общим структурам данных, но только в одном направлении.

Для `Iterator` требуется интерфейс `Collection`, в то время как для `ListIterator` требуется только интерфейс `List`.

Различия между итератором ListIterator и индексом.

У `ListIterator` есть методы `nextIndex` и `previousIndex`, которые возвращают значения индексов, связанные с объектами, которые будут возвращаться при вызове методов `next` или `previous`. У класса `LinkedList` есть метод `ListIterator (int` индекс). Возвращает `ListIterator`, чей вызов `next`

возвращает элемент на следующей позиции индекса.

Сравнение производительности ArrayList и LinkedList.

LinkedList обеспечивает лучшую производительность в следующих случаях:

– при операциях добавления/удаления элементов в начале списка по индексу;

– при операциях добавления/удаления элементов внутри списка по итератору (при условии, что этот итератор был каким-то образом получен заранее);

– при доступе к первому/последнему элементу списка (getFirst() / getLast() / removeFirst() / removeLast() и т.д.). Во многих остальных случаях ArrayList показывает более высокую производительность.

LinkedList надо что-то вставить в середину в какую-то позицию X, то он будет прямо с начала по ссылкам искать эту позицию. С LinkedList надо использовать, к примеру, ListIterator, который умеет вставлять в середину списка по индексу (поэтому в LinkedList вставка на самом деле будет за мизерное постоянное время).

Пример:

```
    ListIterator<Integer> iter =
list.listIterator(inputIndex);
    iter.add(new Integer(123));
```

Поэтому пока ArrayList будет перемещать в памяти элементы для вставки, LinkedList давно уже будет сидеть и спокойно курить в сторонке.

Например, такая ситуация, в ArrayList и в LinkedList объемом 1 000 000 элементов надо сделать 100 вставок в середину по индексу 500 000. Как себя будет вести ArrayList: он 100 раз будет перемещать правую половину массива!

А LinkedList доберется за 1 раз до индекса 500 000 и сделает 100 "мгновенных" вставок. Вывод: выигрыш у LinkedList будет огромный!

P.S. Конечно, можно придраться, что мол давайте сделаем массив на 100 нужных для вставки элементов и за 1 вставку засунем в середину того ArrayList.

Но вот на практике так не всегда везет и приходится делать 100 вставок!

Для чего нужен Iterable?

Для обеспечения возможности работы ваших классов или объектов в выражении `for/in` необходимо реализовать интерфейс `Iterable`. Есть два основных сценария:

– расширить существующий класс коллекции, который уже реализует интерфейс `Iterable` (и, следовательно, уже поддерживает `for/in`);

– управлять итерацией вручную, определив свою собственную реализацию `Iterable`.

Интерфейс Iterable.

Интерфейс требует, чтобы каждый класс, который его реализует реализовывал метод – итератор. Интерфейс `Collection` расширяет интерфейс `Iterable`, таким образом, что все классы, которые реализуют интерфейс `List` (подинтерфейс `Collection`) обязаны представить реализацию итератора в виде метода.

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

Удобно использовать для своих собственных классов коллекций использовать цикл `for/in`. Чтобы сделать это, ваш класс должен реализовывать интерфейс `java.lang.Iterable <E>`. Простой пример:

```
public class MyCollection<E> implements Iterable<E>{
    public Iterator<E> iterator() {
        return new MyIterator<E>();
    }
}
```

Соответствующая реализация скелета класса MyIterator:

```
public class MyIterator <T> implements Iterator<T> {
    public boolean hasNext() {
        //реализация.
    }
   public T next() {
```

```
            //реализация...;
        }
        public void remove() {
            //реализация...если поддерживается.
        }}

    public static void main(String[] args) {
        MyCollection<String> stringCollection = new
MyCollection<String>();
        for(String string : stringCollection){
           }
    }
```

7.5 Иерархия классов Collection.

Как `ArrayList`, так и `LinkedList` представляют собой коллекцию объектов, на которые можно ссылаться с помощью индекса. Интерфейс `Collection` определяет подмножество методов, указанных в интерфейсе `List`. Иерархия Collection представлена на рисунке 7.13.

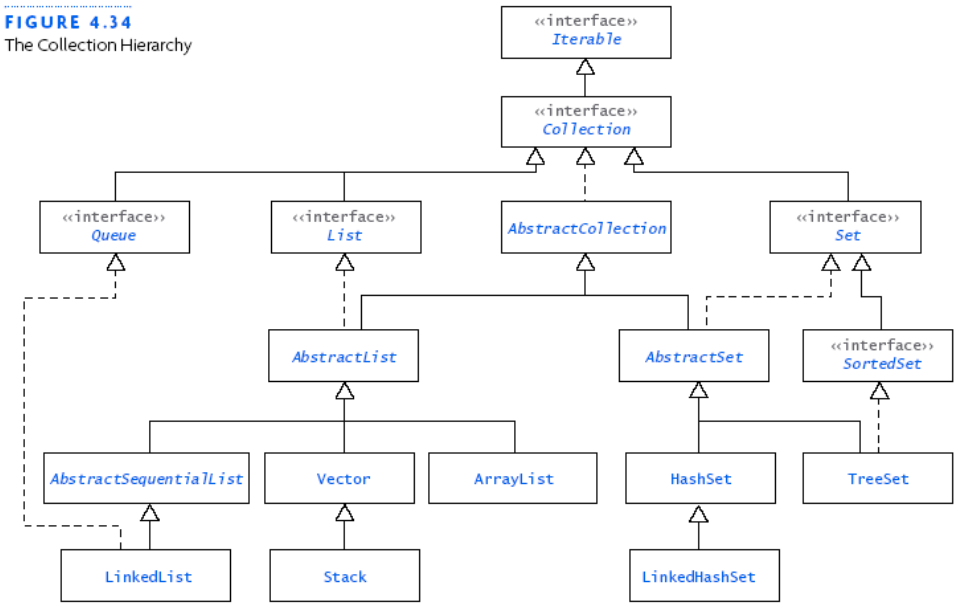


Рисунок 7.13 – Иерархия Collection

Общие характеристики коллекций

Интерфейс `Collection` определяет набор общих методов. Основные функции включают в себя:

– коллекции растут по мере необходимости;

– коллекции хранят ссылки на объекты;

– коллекции имеют по крайней мере два конструктора.

Реализации интерфейса Queue

`PriorityQueue` (рисунок 7.14) - единственная прямая реализация интерфейса `Queue` (не считая `LinkedList`, который больше является списком, чем очередью). Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс `Comparable`), либо с помощью интерфейса `Comparator`, полученному в конструкторе.

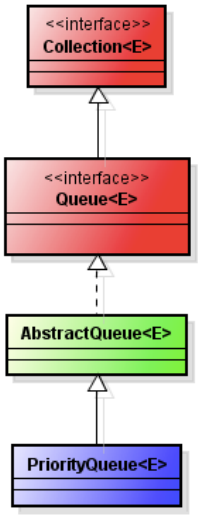


Рисунок 7.14 – Реализации интерфейса Queue

Лекция 8. Пакеты java.lang, java.util. Потоки ввода/вывода в Java.

В Java есть много классов, доступных для работы с датой/временем.

Класс Date

Класс Date изначально предоставлял набор функций для работы с датой - для получения текущего года, месяца и т.д. однако сейчас все эти методы не рекомендованы к использованию и практически всю функциональность для этого предоставляет класс Calendar. Класс Date так же определен в пакете java.sql поэтому желательно указывать полностью квалифицированное имя класса Date.

Существует несколько конструкторов класса Date однако рекомендовано к использованию два

Date() и Date(long date) второй конструктор использует в качестве параметра значение типа long который указывает на количество миллисекунд прошедшее с 1 Января 1970, 00:00:00 по Гринвичу. Первый конструктор создает дату использует текущее время и дату (т.е. время выполнения конструктора). Фактически это эквивалентно второму варианту new Date(System.currentTimeMillis); Можно уже после создания экземпляра класса Date использовать метод setTime(long time), для того, что бы задать текущее время.

Для сравнения дат служат методы after(Date date), before(Date date) которые возвращают булевское значение в зависимости от того выполнено условие или нет. Метод compareTo(Date anotherDate) возвращает значение типа int которое равно -1 если дата меньше сравниваемой, 1 если больше и 0 если даты равны. Метод toString() представляет строковое представление даты, однако для форматирования даты в виде строк рекомендуется пользоваться классом SimpleDateFormat определенном в пакте java.text

Классы Calendar и GregorianCalendar

Более развитые средства для работы с датами представляет класс Calendar. Calendar является абстрактным классом. Для различных платформ реализуются

конкретные подклассы календаря. На данный момент существует реализация Грегорианского календаря - GregorianCalendar. Экземпляр этого класса получается вызовом статического метода getInstance(), который возвращает экземпляр класса Gregorian. Подклассы класса Calendar должны интерпретировать объект Date по-разному. В будущем предполагается реализовать так же лунный календарь, используемый в некоторых странах. Calendar обеспечивает набор методов позволяющих манипулировать различными "частями" даты, т.е. получать и устанавливать дни, месяцы, недели и т.д. Если при задании параметров календаря упущены некоторые параметры, то для них будут использованы значения по умолчанию для начала отсчета. т.е.YEAR = 1970, MONTH = JANUARY, DATE = 1 и т.д.

Для считывания, установки манипуляции различных "частей" даты используются методы get(int filed), set(int filed, int value), add(int field, int amount), roll(int field, inr amount), переменная типа int с именем filed указывает на номер поля с которым нужно произвести операцию. Для удобства все эти поля определены в Calendar, как статические константы типа int. Рассмотрим подробнее порядок выполнения перечисленных методов.

Метод set(int field,int value)

Как уже отмечалось ранее данный метод производит установку какого - либо поля даты. На самом деле после вызова этого метода, немедленного пересчета даты не производится. Пересчет даты будет осуществлен только после вызова методов get(), getTime() или TimeInMillis(). Т.о. последовательная установка нескольких полей, не вызовет не нужных вычислений. Помимо этого, появляется еще один интересный эффект. Рассмотрим следующий пример. Предположим, что дата установлена на последний день августа. Необходимо перевести ее на последний день сентября. Если внутреннее представление даты изменялось бы после вызова метода set, то при последовательной установке полей мы получили бы вот такой эффект.

```
public class Test {
```

```
public Test() {
}
public static void main(String[] args) {
SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM
dd HH:mm:ss");
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR,2002);
cal.set(Calendar.MONTH,Calendar.AUGUST);
cal.set(Calendar.DAY_OF_MONTH,31);          "
System.out.println(" Initialy set date:    +
sdf.format(cal.getTime()));
cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
System.out.println(" Date with month changed
: " +
sdf.format(cal.getTime()));
cal.set(Calendar.DAY_OF_MONTH,30);
System.out.println(" Date with day changed "
:                                          +
sdf.format(cal.getTime()));
}
}
```

Вывод программы:

```
                    2002 August 31
Initialy set date:  22:57:47
Date with month changed : 2002 October
01 22:57:47
Date with day       2002 October 30
changed :           22:57:47
```

Еще пример листинга:

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTest {
   public static void main(String[] args) {
      Date now = new Date();
      System.out.println("toString(): " + now);  //
dow mon dd hh:mm:ss zzz yyyy

      // SimpleDateFormat может использоваться для
управления форматом отображения даты/времени:
```

```
      //   E (день недели): 3E or fewer (в текстовом
формате xxx), >3E (в полном текстовом формате)
      //   M (месяц): M (in number), MM ( в числовом
виде, впереди ноль)
      //              3M: (в текстовом формате xxx),
>3M: (в полном текстовом формате)
      //   h (часы): h, hh (with leading zero)
      //   m (минуты)
      //   s (секунды)
   //   a (AM/PM)
      //   H (часы 0 до 23)
      //   z (временная зона)
      SimpleDateFormat dateFormatter = new
SimpleDateFormat("E, y-M-d 'at' h:m:s a z");
      System.out.println("Format 1:   " +
dateFormatter.format(now));

      dateFormatter = new SimpleDateFormat("E
yyyy.MM.dd 'at' hh:mm:ss a zzz");
      System.out.println("Format 2:   " +
dateFormatter.format(now));

      dateFormatter = new SimpleDateFormat("EEEE, MMMM
d, yyyy");
      System.out.println("Format 3:   " +
dateFormatter.format(now));
```

Вывод программы

```
toString(): Sat Sep 25 21:27:01 SGT 2010
Format 1:   Sat, 10-9-25 at 9:27:1 PM SGT
Format 2:   Sat 2010.09.25 at 09:27:01 PM SGT
Format 3:   Saturday, September 25, 2010
```

Вывод

Класса Date будет достаточно, если вам просто нужна простая отметка времени. Вы можете использовать SimpleDateFormat для управления форматом отображения даты /времени. Используйте класс java.util.Calendar, если вам нужно извлечь год, месяц, день, час, минуту и секунду или манипулировать этими полями (например, 7 дней спустя, 3 недели назад).

Используйте java.text.DateFormat для форматирования даты (от даты до текста) и разбора строки даты (от текста к дате). SimpleDateForamt является подклассом

DateFormat.

Date является устаревшим классом, который не поддерживает интернационализацию. Calendar и DateFormat поддерживают локалзацию (вам нужно учитывать локализацию только в том случае, если ваша программа будет работать во нескольких странах одновременно).

Классы java.util.Calendar и java.util.GregorianCalendar

Теперь рассмотрим программу, которая использует класс Filename:

```
// Get the year, month, day, hour, minute, second
import java.util.Calendar;
public class GetYMDHMS {
   public static void main(String[] args) {
      Calendar cal = Calendar.getInstance();
      // You cannot use Date class to extract
individual Date fields
      int year = cal.get(Calendar.YEAR);
      int month = cal.get(Calendar.MONTH);      // 0
to 11
      int day = cal.get(Calendar.DAY_OF_MONTH);
      int hour = cal.get(Calendar.HOUR_OF_DAY);
      int minute = cal.get(Calendar.MINUTE);
      int second = cal.get(Calendar.SECOND);
   // Pad with zero
      System.out.printf("Now is %4d/%02d/%02d
%02d:%02d:%02d\n",
          year, month+1, day, hour, minute, second);
   }
}
```

Измерение времени

Любые приложения (такие как игры и анимация) требуют хорошего контроля времени. Java предоставляет эти статические методы в классе System. Метод System.currentTimeMillis() возвращает текущее время в миллисекундах с 1 января 1970 г. 00:00:00 по Гринвичу (известное как «эпоха») в длинном формате.

```
//Измерение прошедшего времени
long startTime = System.currentTimeMillis();
// измерение выполнения кода
  .......
```

```
long estimatedTime = System.currentTimeMillis() - startTime;
```

Метод System.nanoTime (): возвращает текущее значение наиболее точного доступного системного таймера, в наносекундах, в течение длительного времени. Введенный с JDK 1.5. метод nanoTime () предназначен для измерения относительного временного интервала вместо предоставления абсолютного времени.