Язык Java
и разработка Java-приложений

Владимир Юрьевич Романов,
Московский Государственный Университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
vromanov@cs.msu.su,
romanov.rvy@yandex.ru

Литература

1. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley
Java(TM) Programming Language, Java SE 8 Edition
https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

2. Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley
The JavaTM Virtual Machine Specification, Java SE 8 Edition
https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

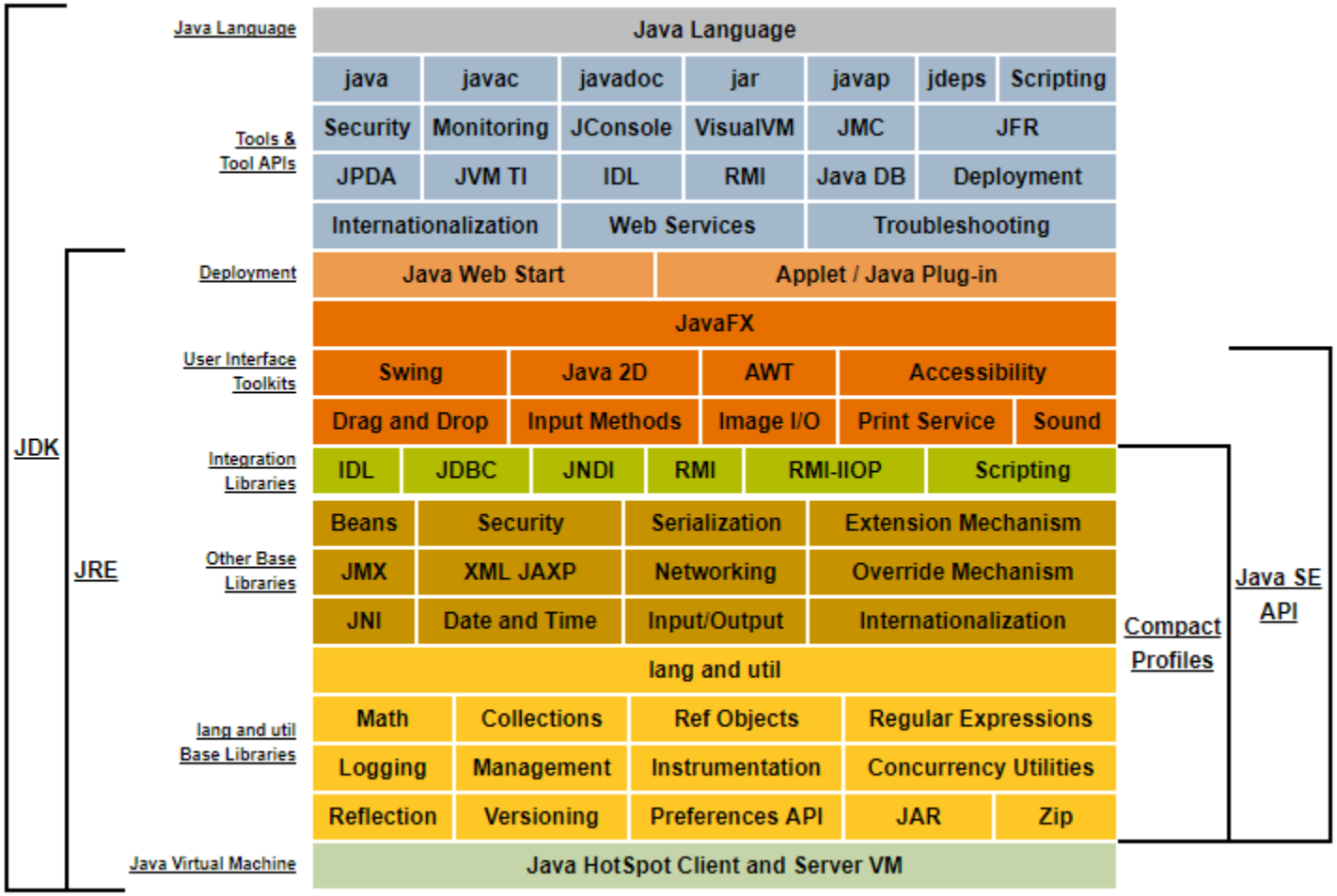
3. Eclipse web sites: **http://www.eclipse.org**

4. Oracle web site:
https://docs.oracle.com/javase/8/docs/

5. Брюс Эккель. Философия Java. 4-е издание.
Издательство «Питер». Петербург 2016.
ISBN 5-88782-105-1

Документация по Java

https://docs.oracle.com/javase/8/docs/



Цели курса

- Изучение языка Java перед изучением технологий основанных на языке Java
- Изучение среды разработки Eclipse распространяемой фирмой IBM в исходных текстах
- Подготовка к изучению курса: “Разработка объектно-ориентированных систем программирования в среде Eclipse”

О курсе “Разработка объектно-ориентированных систем программирования в среде Eclipse”

- Разработка расширений (plug-in) для среды Eclipse
- Разработка расширений среды для объектно-ориентированных систем программирования
- Разработка распознавателей для объектно-ориентированных языков программирования
- Разработка промежуточного представления компилятора с помощью Eclipse Modeling Framework

О курсе “Разработка объектно-ориентированных систем программирования в среде Eclipse”

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Визуализация промежуточного представления компилятора с помощью *Graphic Editing Framework*
- Моделирование программ с помощью реализованной в среде Eclipse *метамодели языка UML 2.0*
- Генерация кода для виртуальной машины Java
- Разработка в среде Eclipse отладчиков для языков программирования

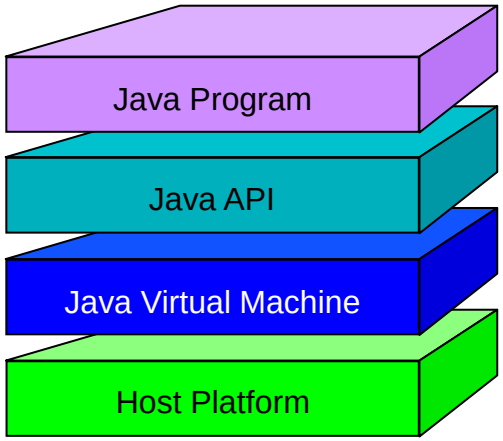
Язык программирования Java

Романов Владимир Юрьевич ©2025

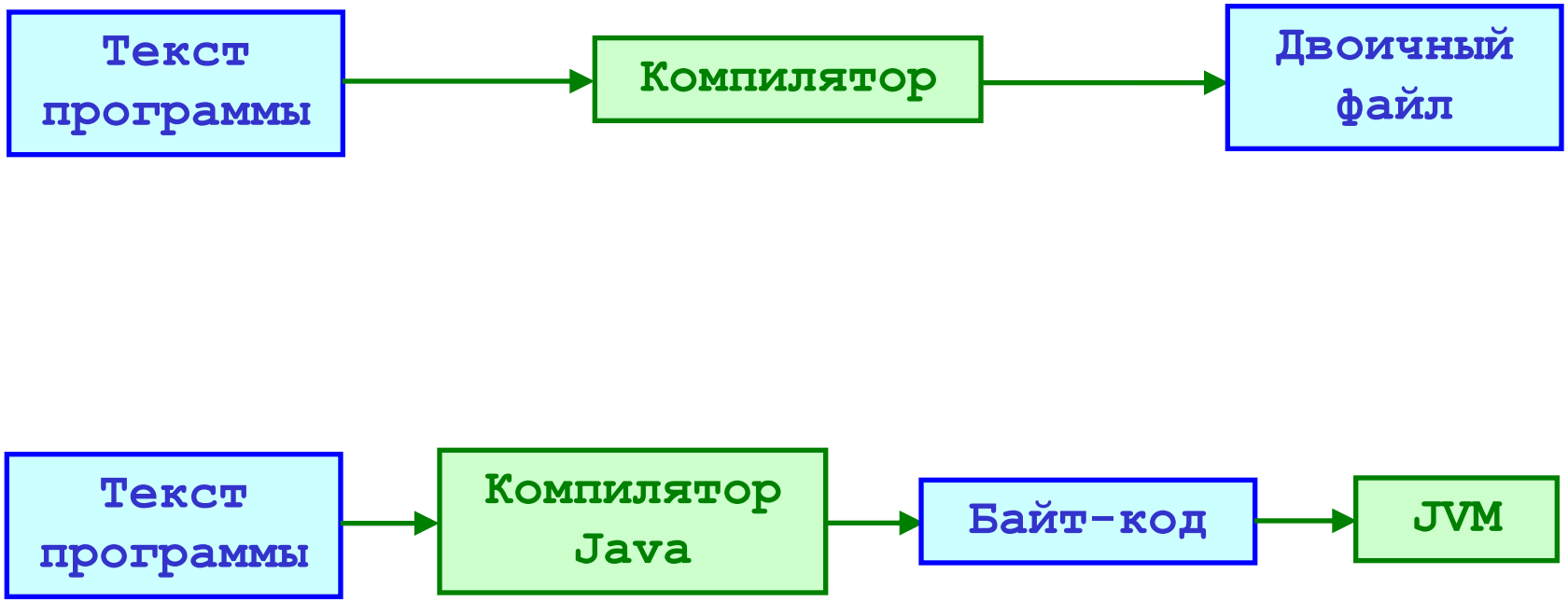
- Простота
- Язык высокого уровня
- Объектно-ориентированный
- Независим от архитектуры
- Переносим в исходных текстах и в двоичном коде
- Интерпретируемый и компилируемый
- Многопоточный

Платформа Java

- Платформа – окружение разработки где работает программа
- Java – платформа работает на разных операционных системах
- Java – платформа состоит из виртуальной машины Java и интерфейса приложений

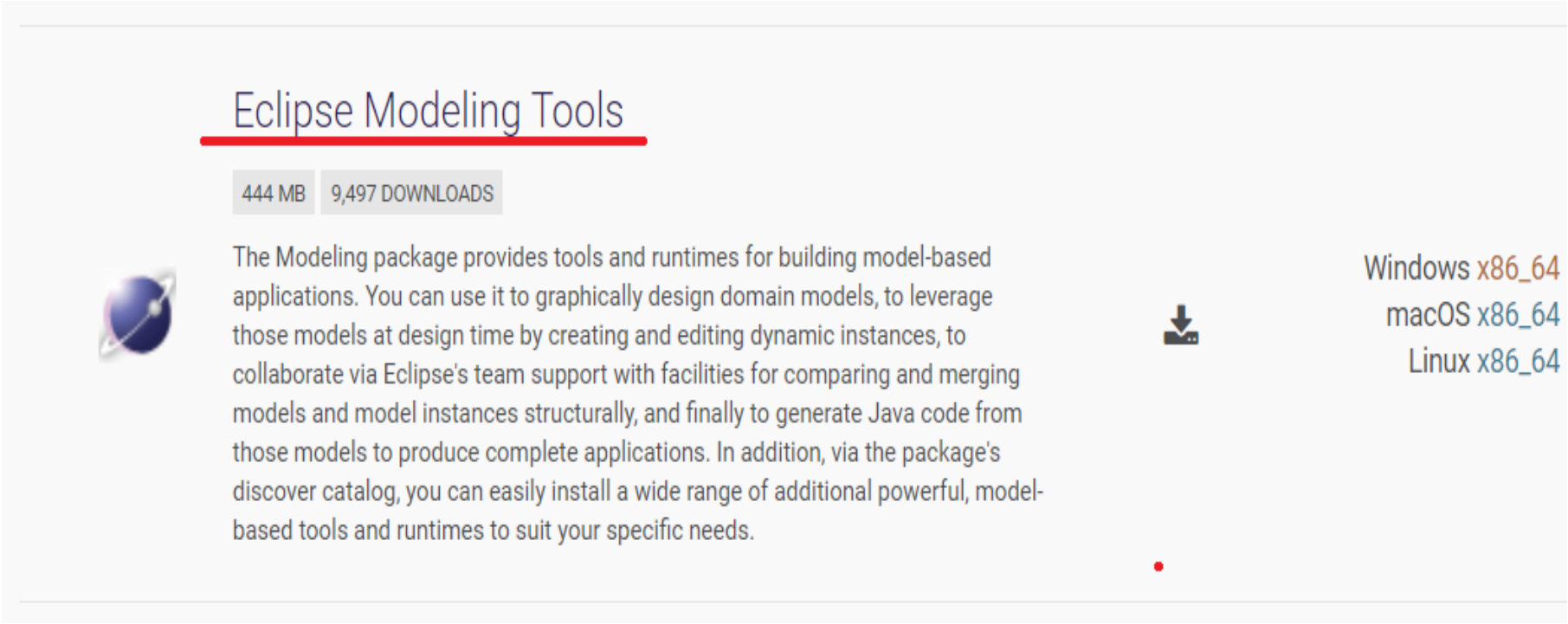


Компилятор и виртуальная машина Java



Установка Eclipse. Версии среды

- Среда Eclipse бесплатна
- Дистрибутив Eclipse загружается с сайта:
  https://www.eclipse.org/downloads/packages/release/2020-06/r



Установка Eclipse.
Eclipse Modeling Tools

МГУ им

Eclipse IDE for JavaScript and Web Developers
156 MB - Downloaded 29,419 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse IDE for Java and DSL Developers
309 MB - Downloaded 19,275 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse for Parallel Application Developers
227 MB - Downloaded 18,259 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse for RCP and RAP Developers
254 MB - Downloaded 13,260 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse Modeling Tools
392 MB - Downloaded 8,603 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse IDE for Java and Report Developers
315 MB - Downloaded 4,941 Times

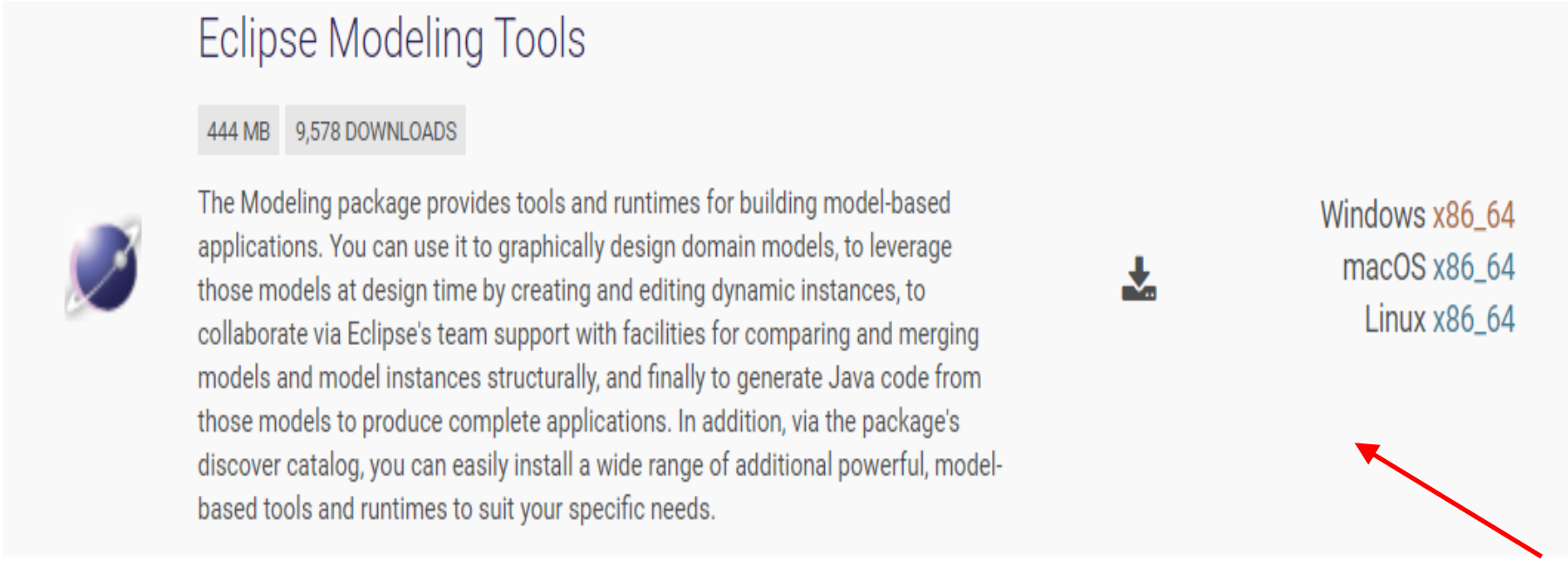
Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse for Testers
131 MB - Downloaded 4,115 Times

Windows **32-bit 64-bit**
Mac Cocoa **64-bit**
Linux **32-bit 64-bit**

Eclipse Modeling Tools

https://www.eclipse.org/downloads/packages/release/2020-06/r



IntelliJ IDEA Community Edition

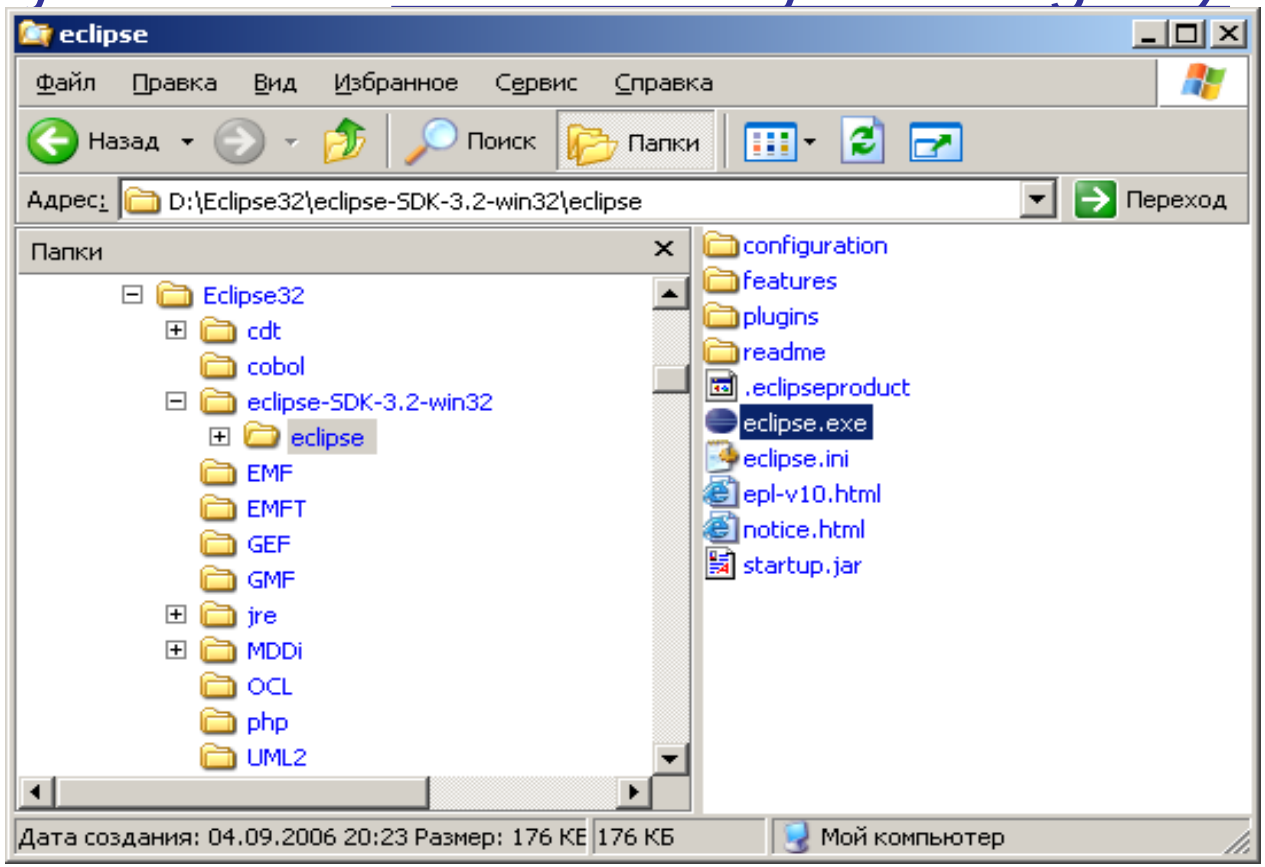
https://www.jetbrains.com/idea/download/#section=windows

Романов Владимир Юрьевич ©2025

Краткий обзор среды Eclipse для разработки на языке Java

Запуск среды Eclipse на исполнение

- Установка – просто распаковка Zip-архива. При установке не используется *registry*



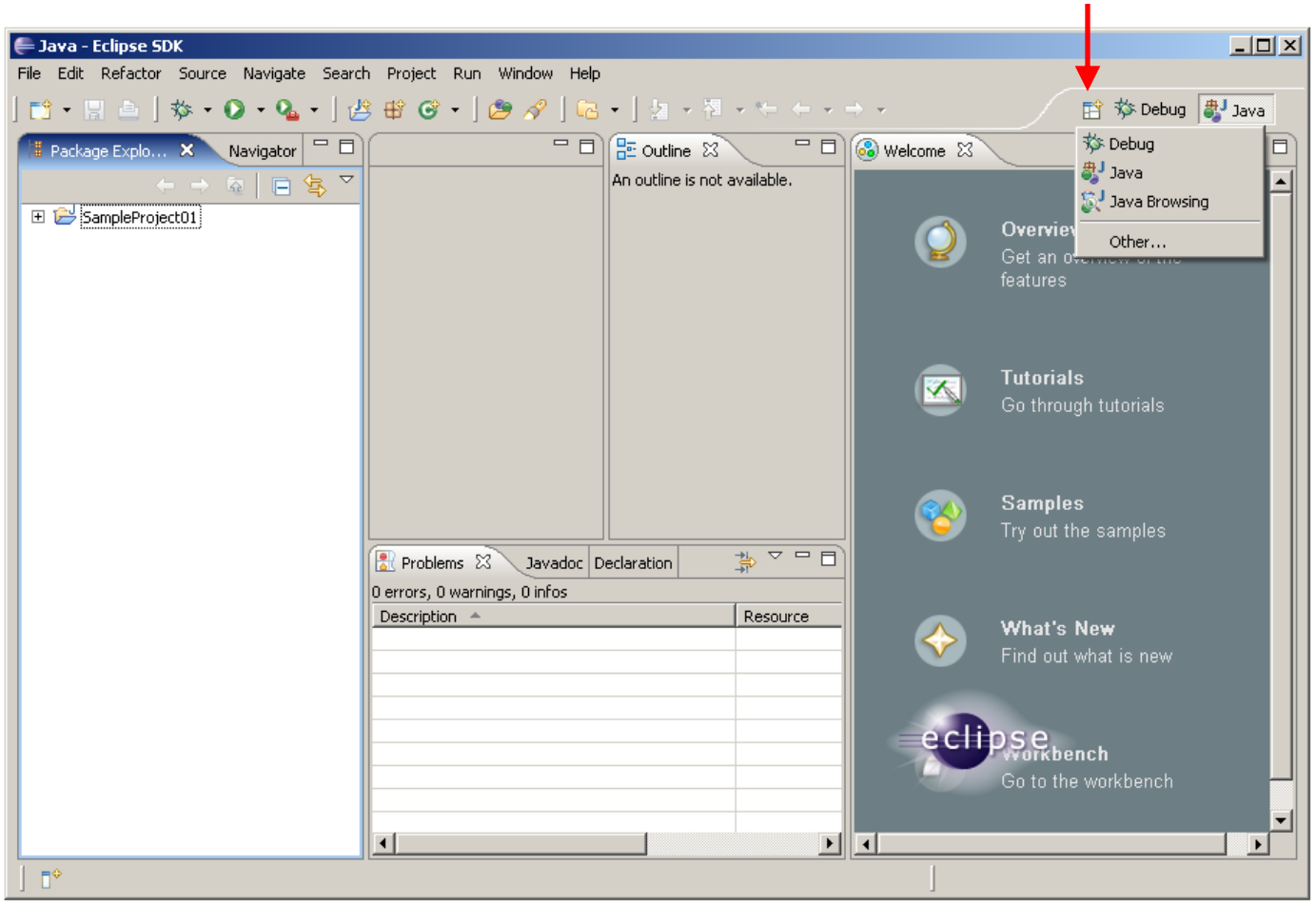
Обзор среды Eclipse

- Перспективы (Perspectives)
- Виды (Views)
- Редакторы (Editors)
- Работа с файлами
- Навигация по рабочему месту (Workbench)
- Консоль (Console)

Обзор среды Eclipse. Перспективы (Java)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

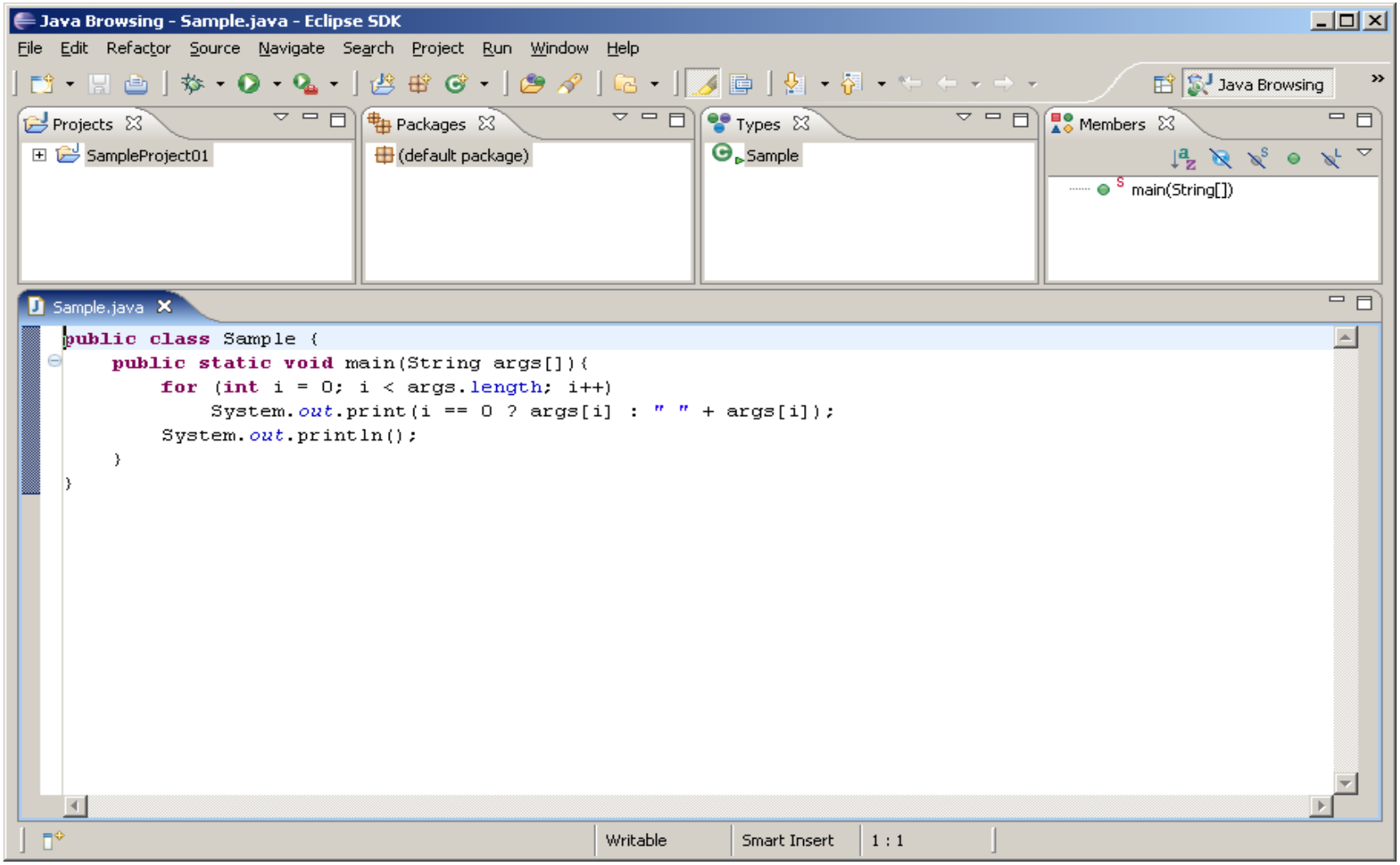
- На рабочем месте (workbench) может быть открыто несколько перспектив
- Различные перспективы могут быть выбраны из панели перспектив



Обзор среды Eclipse. Перспективы (Java Browsing)

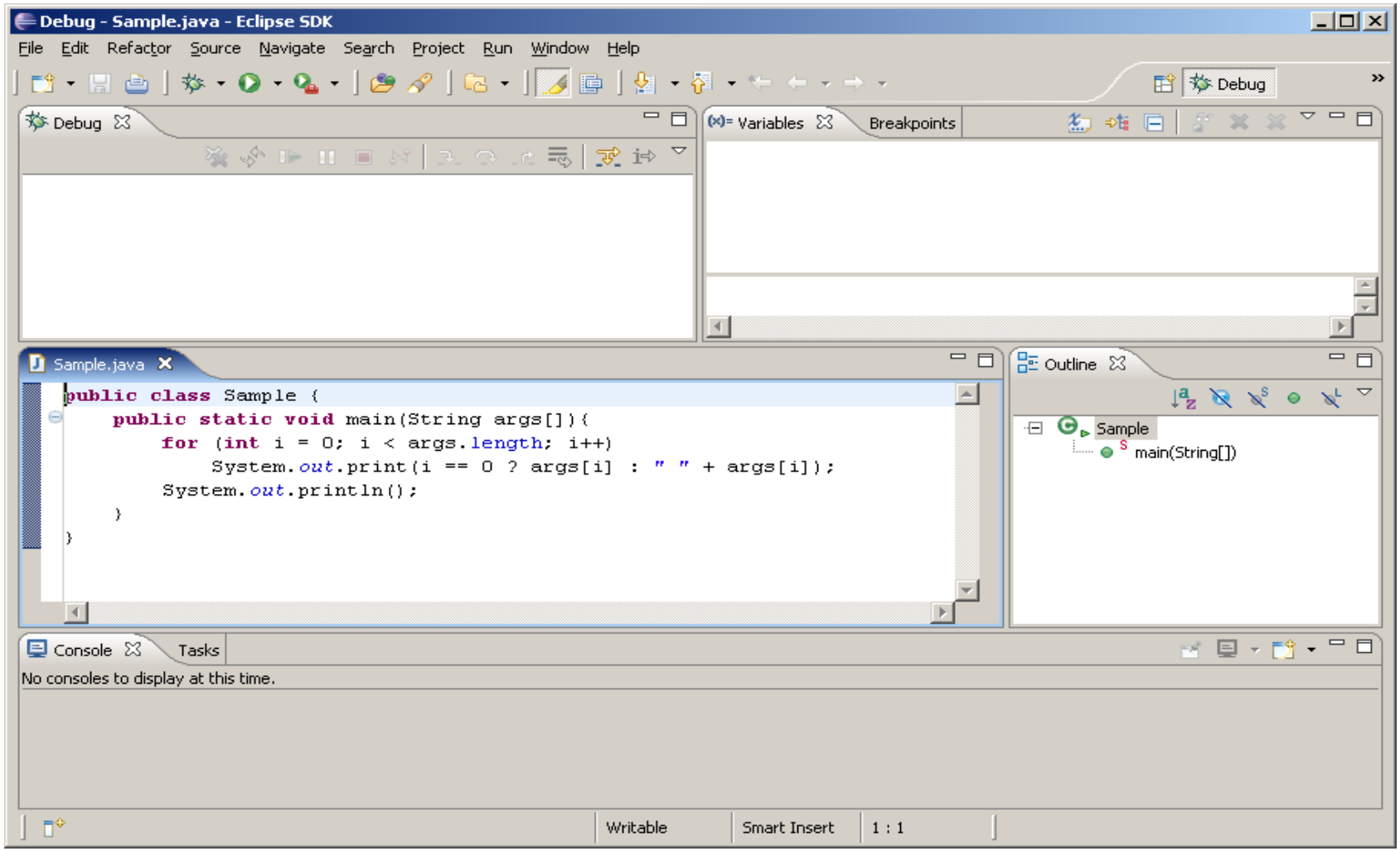
МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025



Обзор среды Eclipse. Перспективы (Debug)

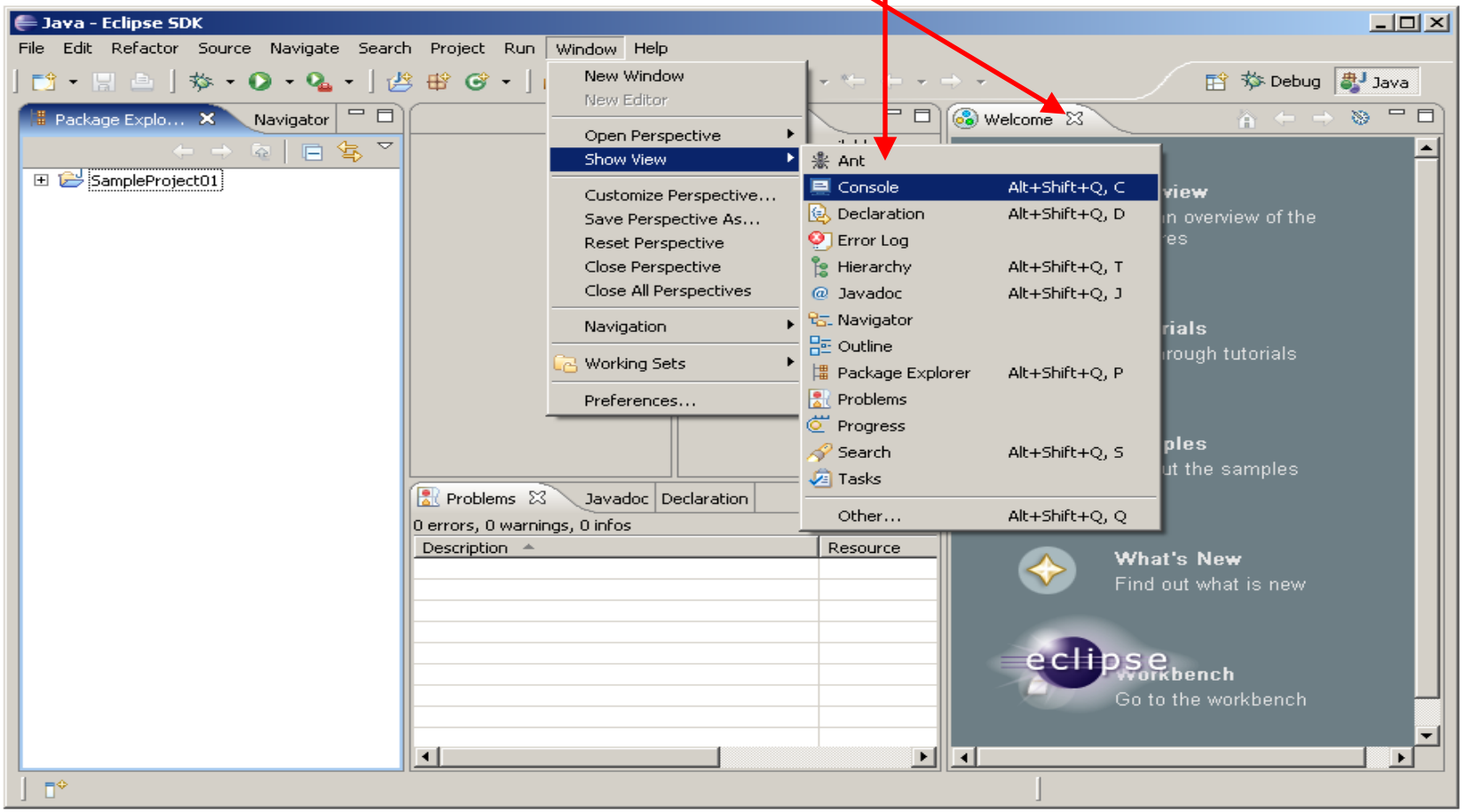
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Обзор среды Eclipse. Добавление видов к перспективам

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

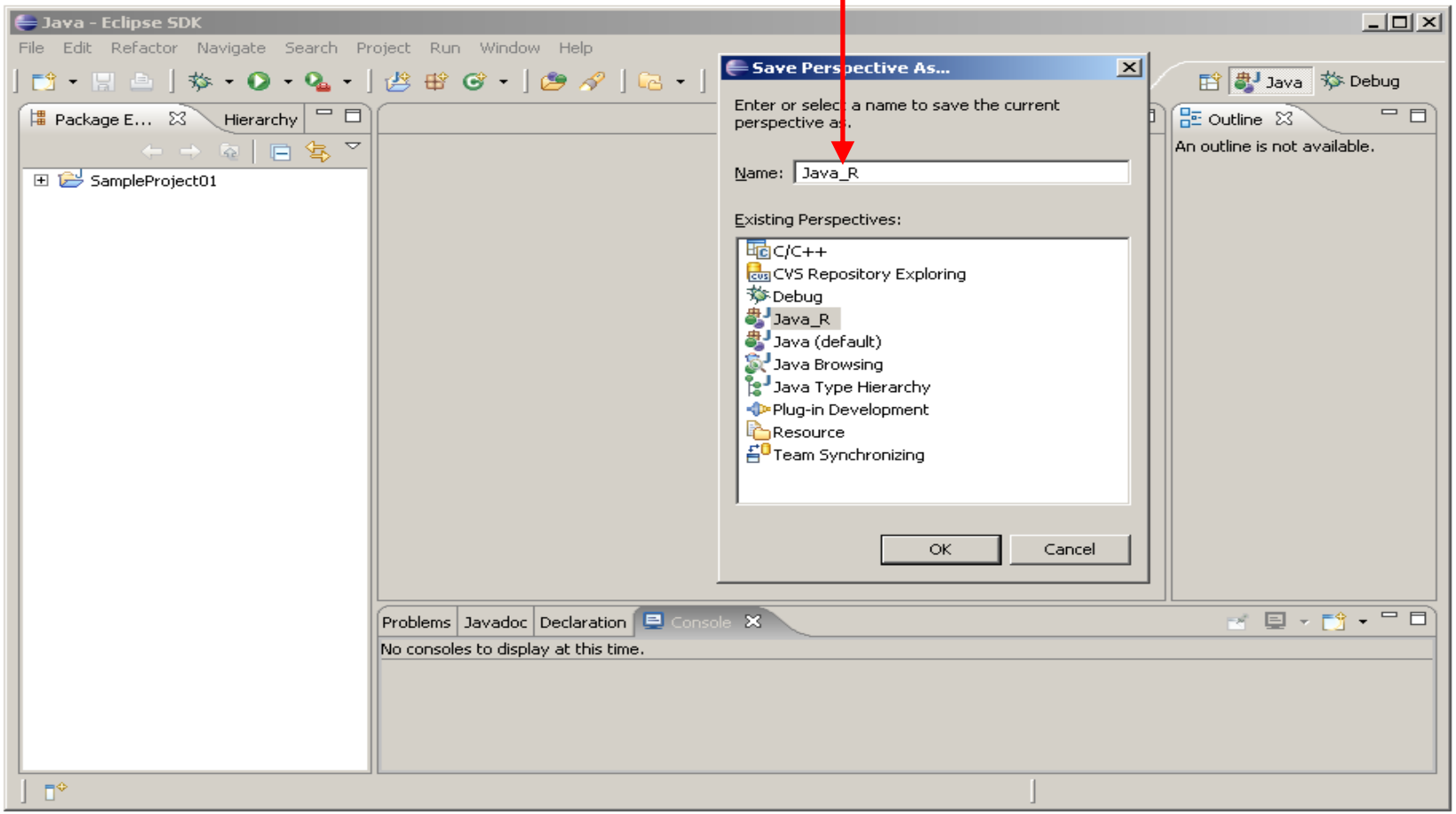
- Добавление к перспективе нового вида (Console)
- Удаление вида Welcome из перспективы



Обзор среды Eclipse. Сохранение перспективы

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Сохранение перспективы под новым именем



Обзор среды Eclipse. Виды

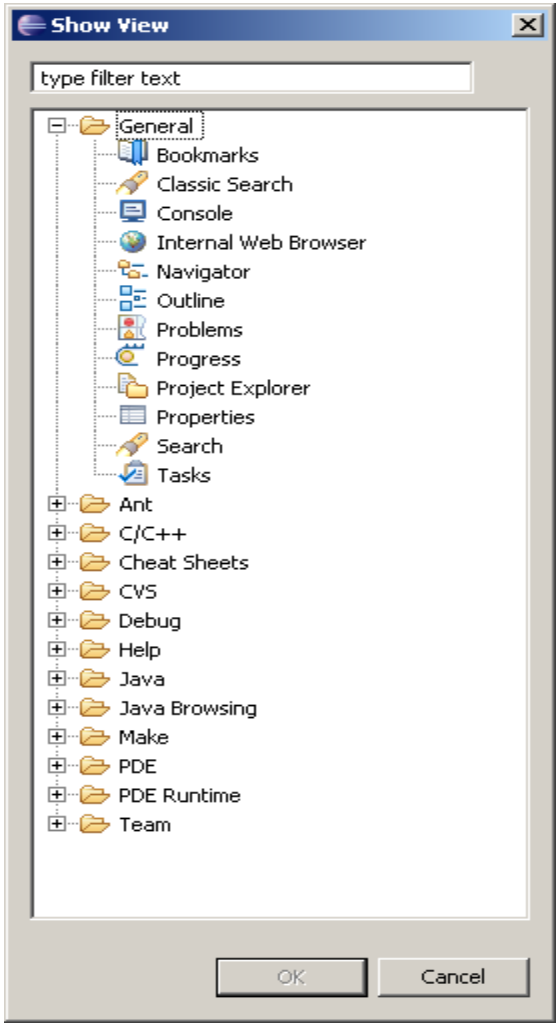
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

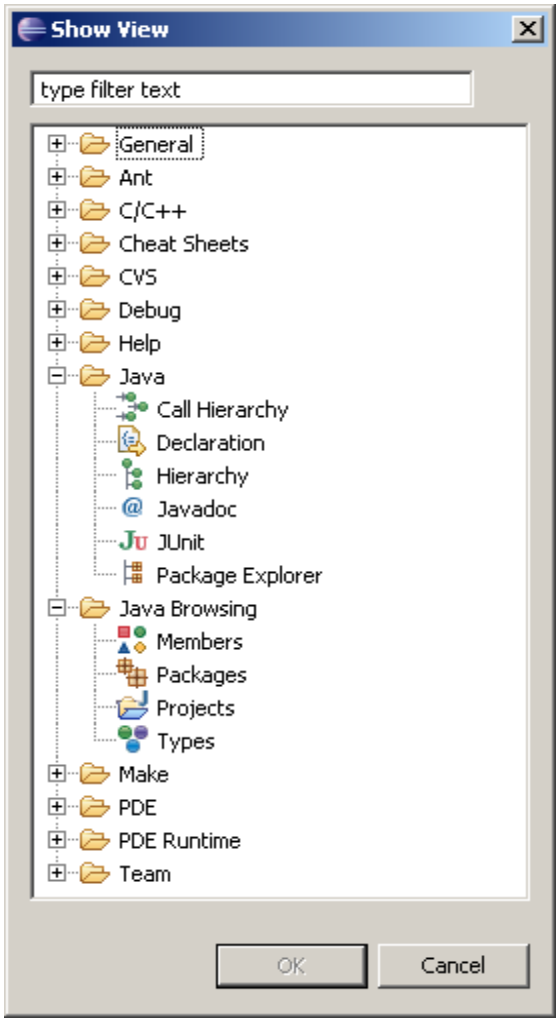
Java Perspective

- Declaration
- Javadoc
- Outline
- Package Explorer
- Problems
- Welcome

Debug Perspective

- Console
- Debug
- Outline
- Tasks
- Variables
- Welcome



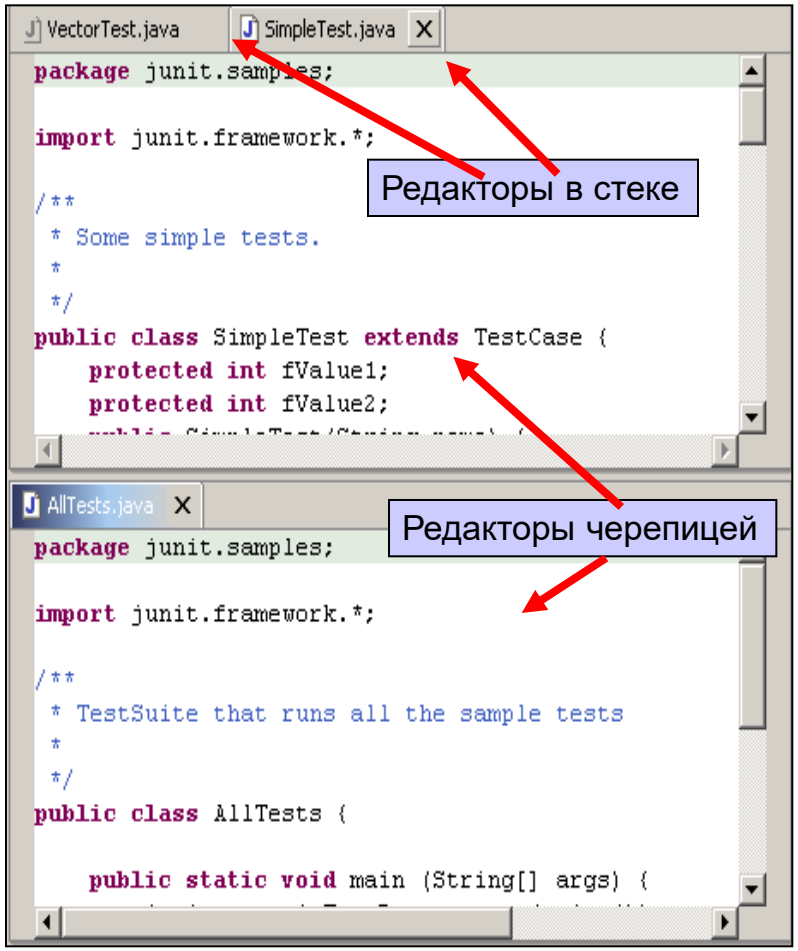


Обзор среды Eclipse. Редакторы

МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025

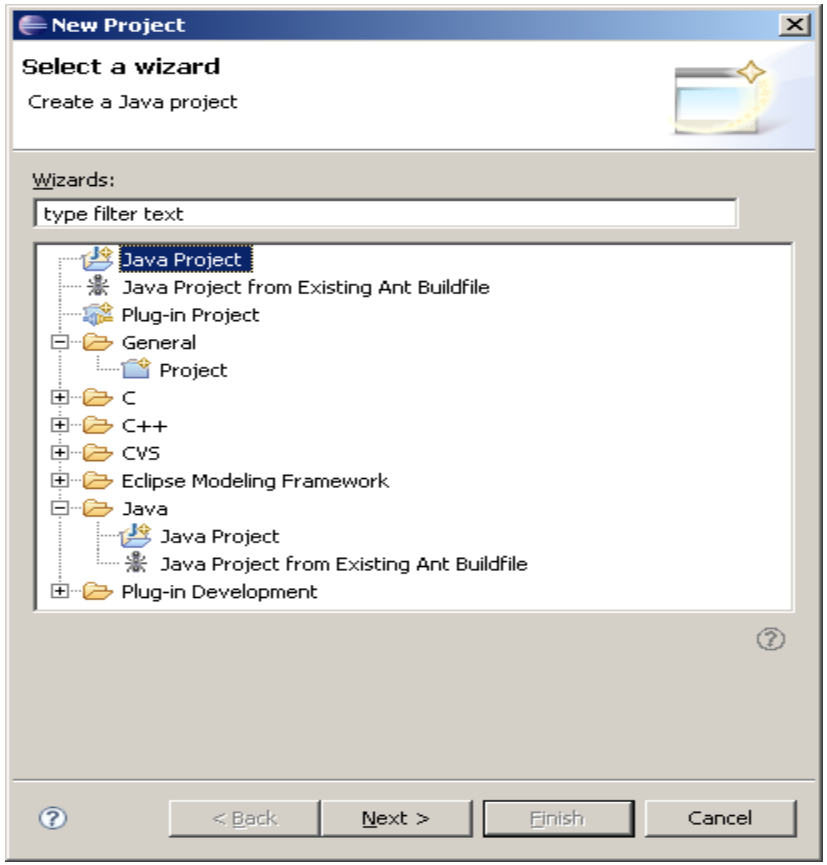
- Редакторы могут быть открыты для большинства ресурсов
- Несколько редакторов может быть открыто в стеке
- Если содержимое редактора было модифицировано, но не сохранено, то перед именем файла *
- Если редактор активен, то в меню и панели инструментов содержатся операции применимые к активному редактору

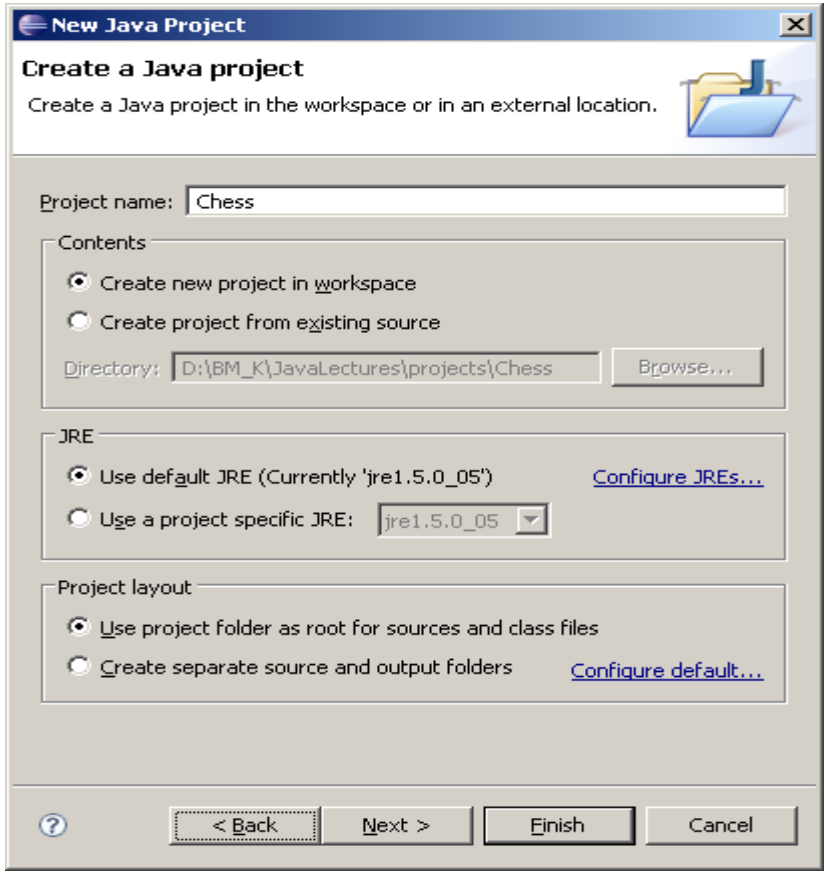


Обзор среды Eclipse. Создание нового проекта

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- File | New | Project



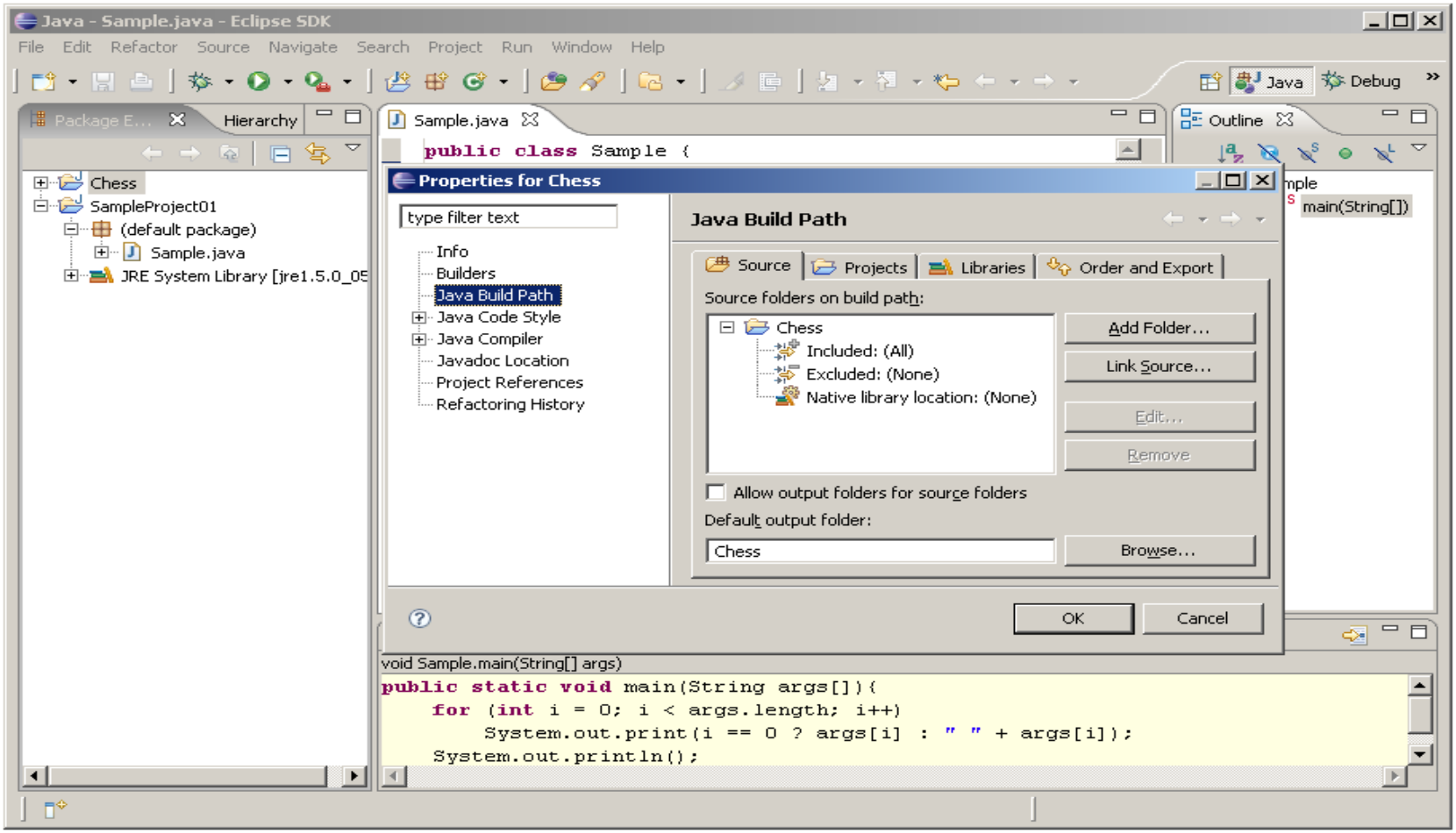


Обзор среды Eclipse. Свойства

МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025

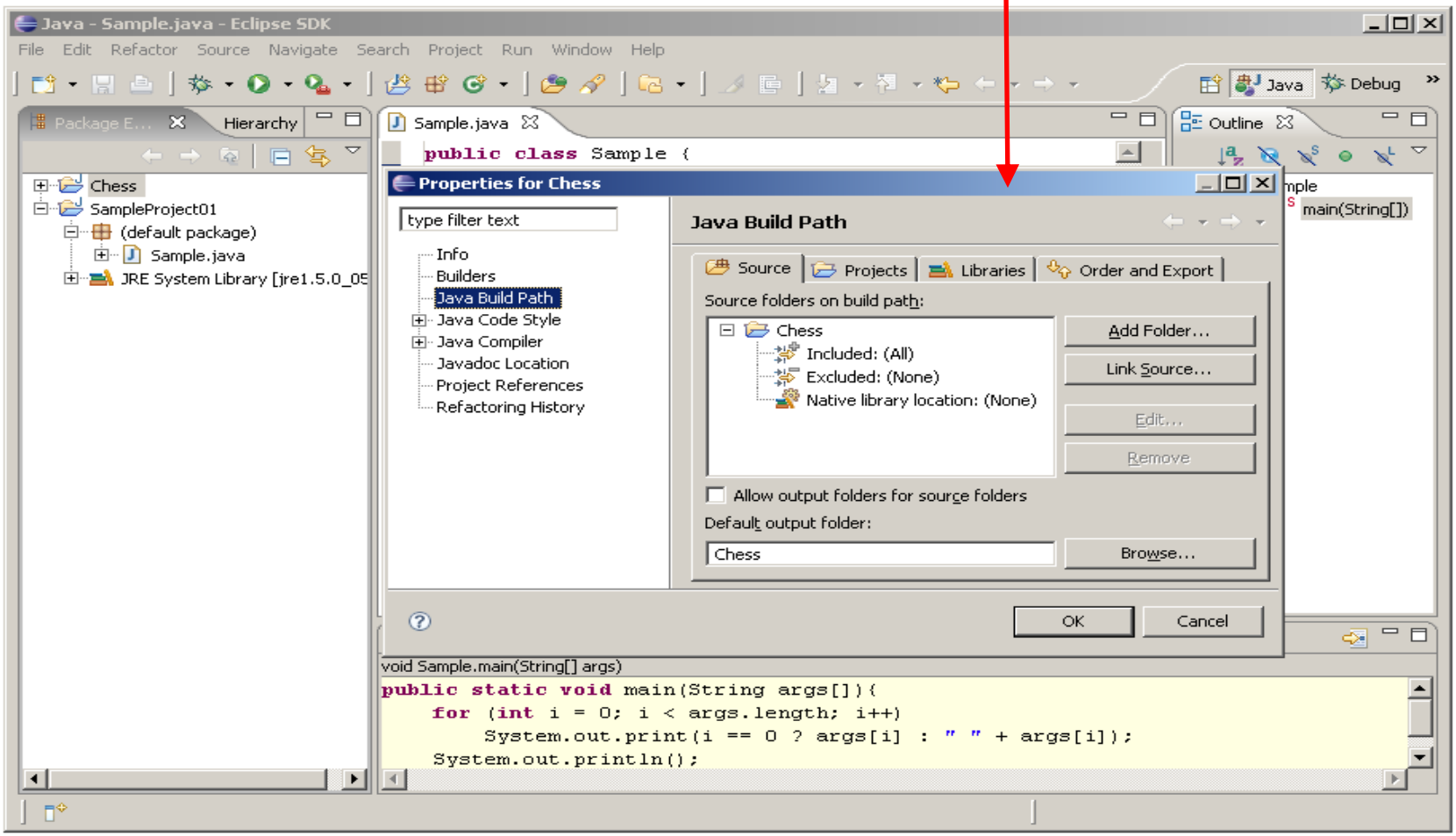
Все ресурсы имеют свойства.



Обзор среды Eclipse. Свойства

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

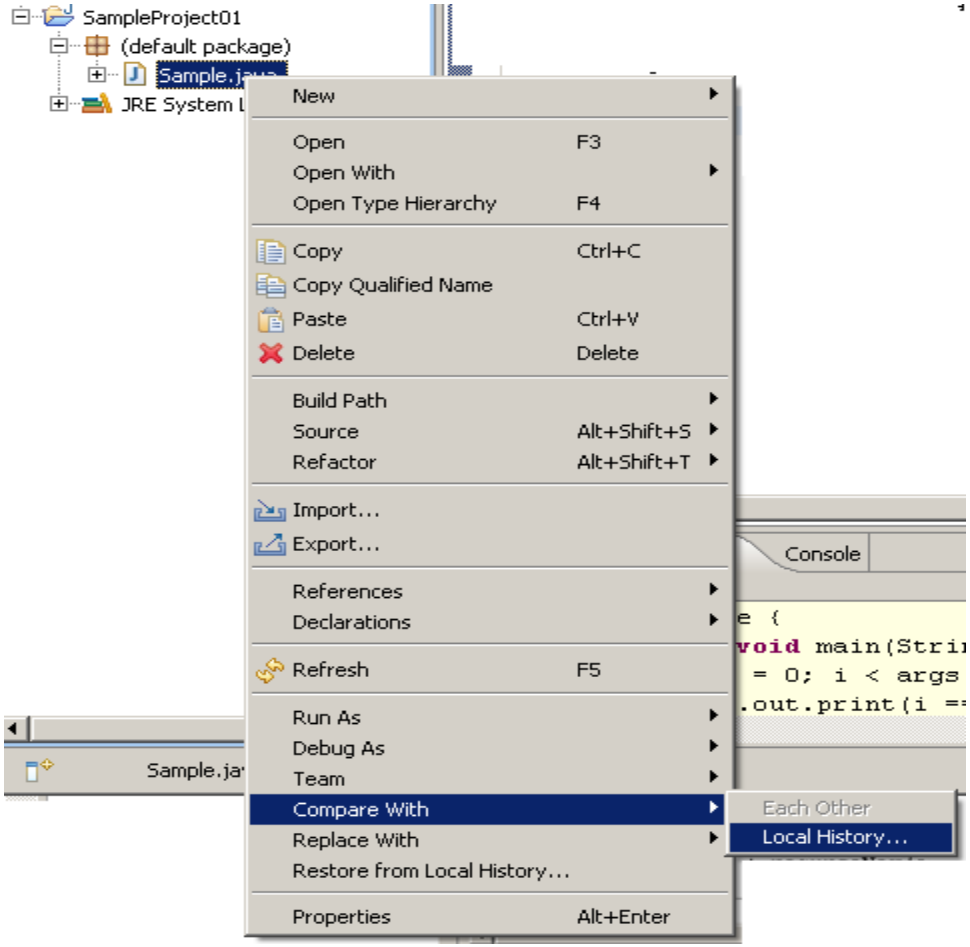
Все ресурсы имеют свойства. Свойства проекта *Chess*:



Обзор среды Eclipse. Локальная история (1)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

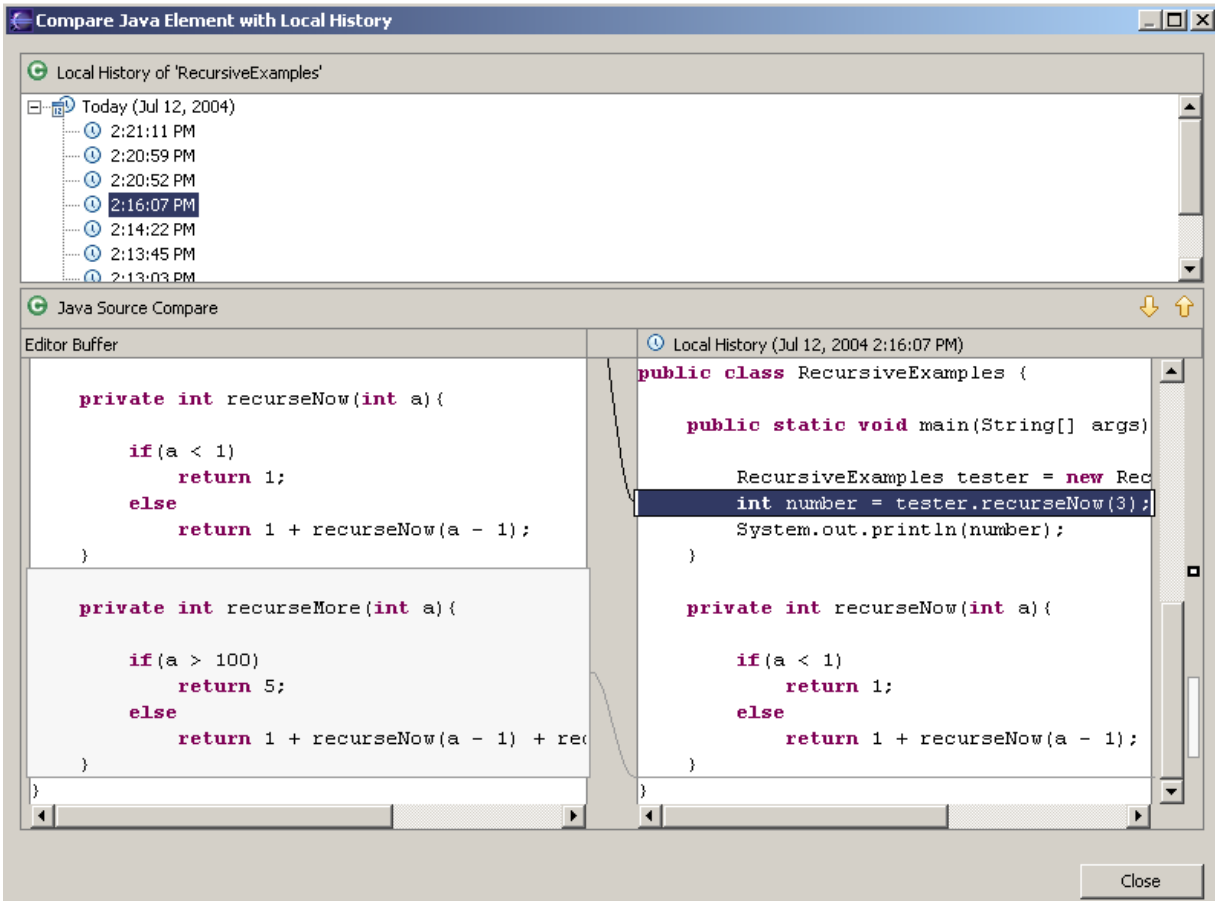
Для каждого файла хранится *локальная история* файла



Обзор среды Eclipse. Локальная история (2)

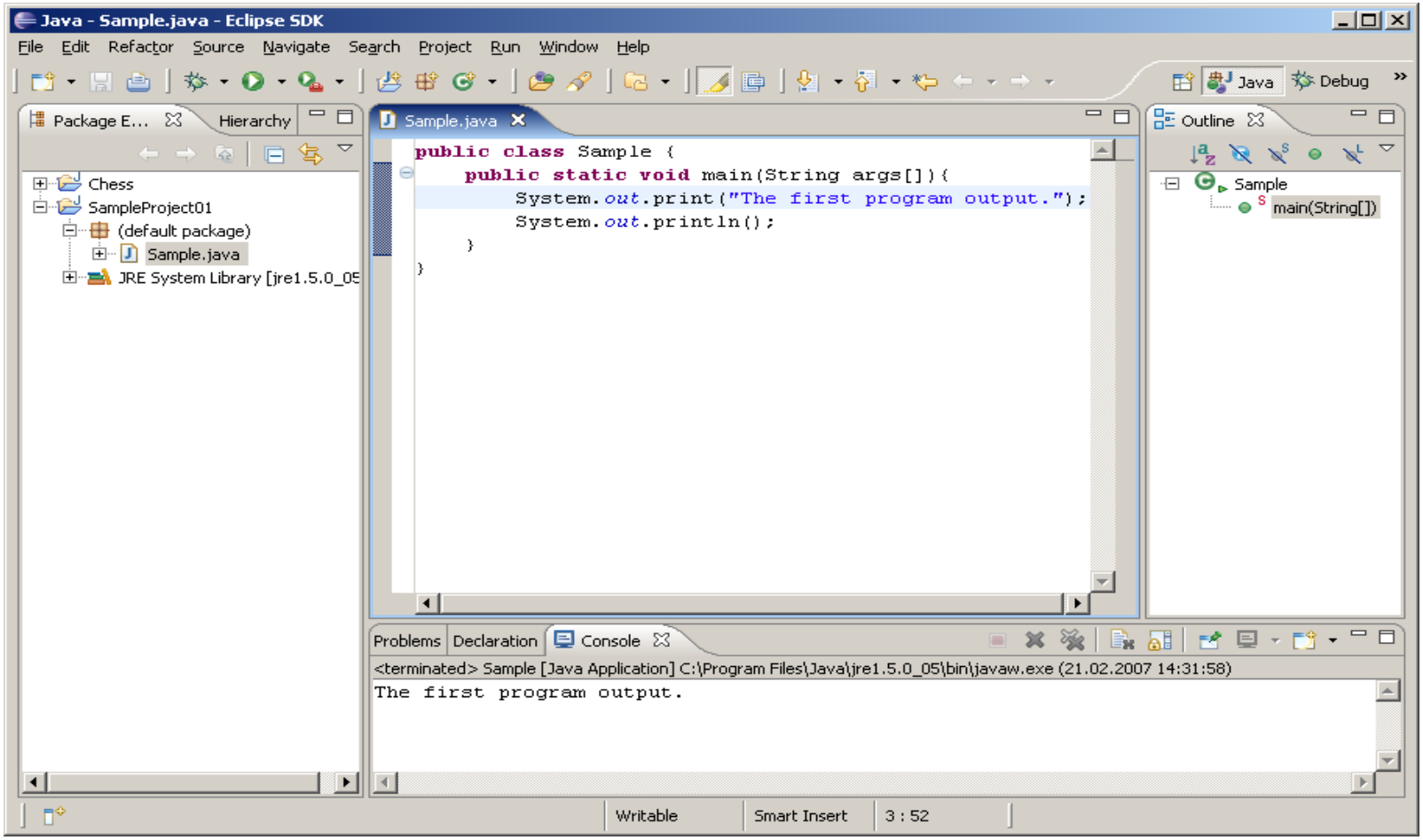
МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Каждое использование команды *Save* сохраняет редакцию файла
- Редакции файлов можно сравнить



Вывод программы на консоль

Романов Владимир Юрьевич ©2025



Романов Владимир Юрьевич ©2025

Введение в язык Java

Классы Java и их синтаксис
Члены класса

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Классы инкапсулируют атрибуты (поля) и поведение (методы)

- Поля и методы являются *членами* класса

- Члены класса могут принадлежать всем экземплярам класса. В этом случае поля и методы помечаются ключевым словом *static*

- Члены класса могут принадлежать конкретным экземплярам класса. В этом случае они называются полями и методами экземпляров класса

- В одном файле с расширением *.java не может быть более одного публичного класса

Классы Java и их синтаксис. Члены классов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Классы инкапсулируют атрибуты (поля) и поведение (методы)

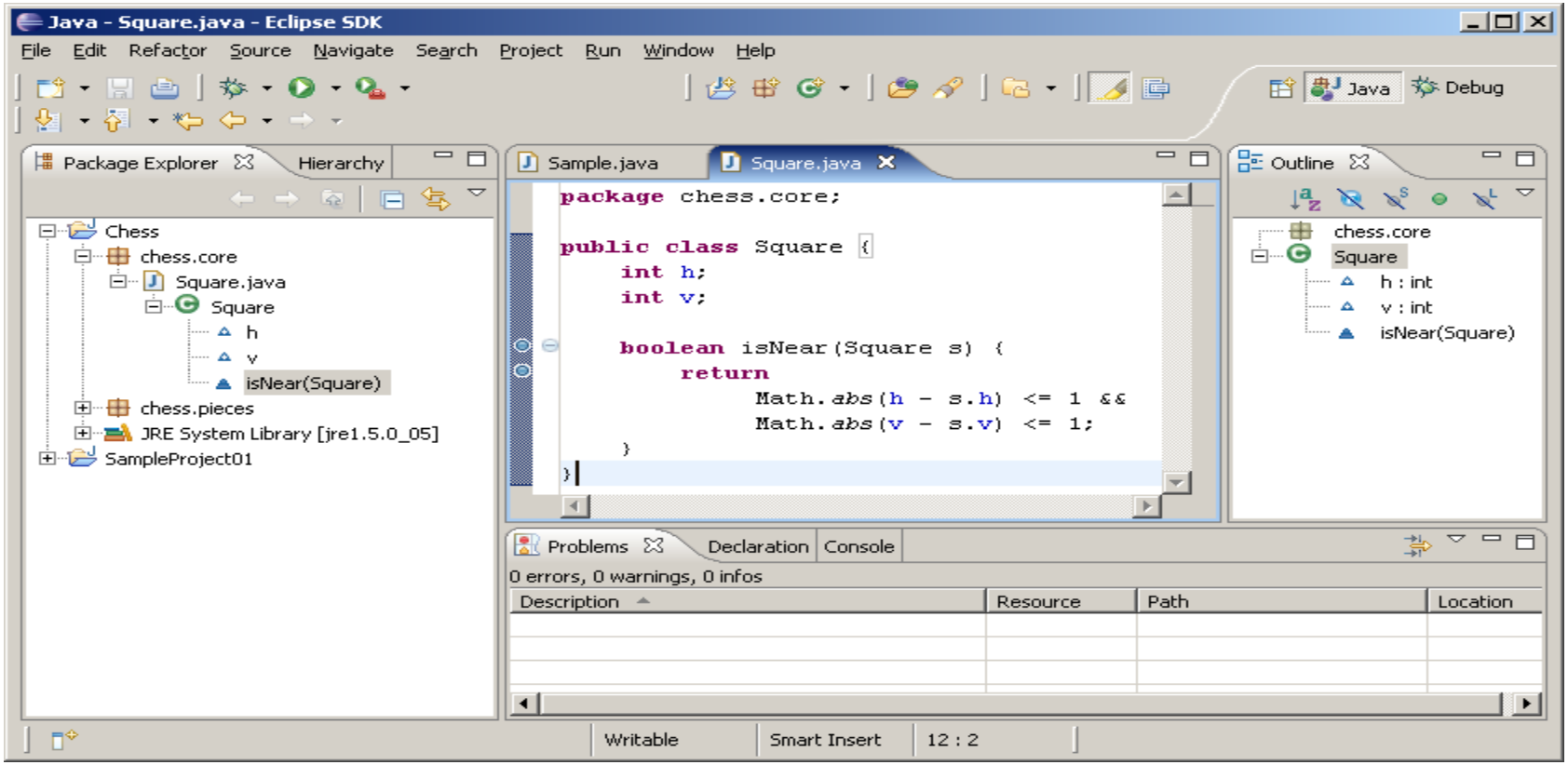
- Атрибуты и методы являются *членами* класса

```
class Square {
      int h;
      int v;
      boolean isNear(Square s) {
         return
            Math.abs(h - s.h) <= 1 &&
            Math.abs(v - s.v) <= 1;
      }
}
```

Классы Java и их синтаксис. Члены классов. Eclipse

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Классы инкапсулируют атрибуты (поля) и поведение (методы)
- Поля и методы являются *членами* класса



Классы Java и их синтаксис.
Объявление полей класса и экземпляра

• Поля класса могут принадлежать *всем* экземплярам класса. В этом случае поля помечаются ключевым словом *static*.

```
class Point {
      public int x, y;

      static public final
      Point ORIGIN = new Point(0,0);

      public Point(int newX, int newX)
            { x = newX; y = newX; }

      public Point(Point p)
            { x = p.x; y = p.y; }
}
```

Поле экземпляра класса

Поле всех экземпляров класса

Классы Java и их синтаксис. Использование полей класса

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Для доступа к статическому полю (класса) используется имя класса, для доступа к полю экземпляра - имя экземпляра.

```
        // Использование поля класса.
        Point p1 = Point.ORIGIN;

        // Создание экземпляра класса.
        Point p2 = new Point(3, 4);
        Point p3 = new Point(5, 6);

        // Использование поля экземпляра.
        p2.y = 100;
        p3.y = p2.y;
}
```

Классы Java и их синтаксис.
Объявление методов класса и экземпляра

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Методы класса принадлежат всем экземплярам класса.

В этом случае методы помечаются ключевым словом *static*

```
class Point {
      static
      public int distanse(Point p1, Point p2) {
          int dx = p1.x - p2.x;
          int dy = p1.y - p2.y;
          return Math.sqrt(dx * dx + dy * dy);
      }
      public int distanse(Point p) {
          int dx = p.x - x;
          int dy = p.y - y;
          return Math.sqrt(dx * dx + dy * dy);
      }
}
```

Классы Java и их синтаксис. Использование методов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Для доступа к статическому методу (класса) используется имя класса, для доступа к методу экземпляра - имя экземпляра класса.

```
        Point p1 = new Point(3, 4);
        Point p2 = new Point(5, 6);

        // Использование метода класса.
        int d1 = Point.distanse(p1, p2);

        // Использование метода экземпляра.
        int d2 = p2.distanse(p1);
}
```

Классы Java и их синтаксис. Область видимости статических методов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- В статических методах класса можно использовать только статические поля класса и вызывать только статические методы класса.

```
class Test {
        public int p;

        public void process() {
        }
        static public void main(String s[]) {
                p = 1; // Ошибка
                process(); // Ошибка

                Test test = new Test();
                test.p = 1;
                test.process();
        }
}
```

Классы Java и их синтаксис. Наследование классов

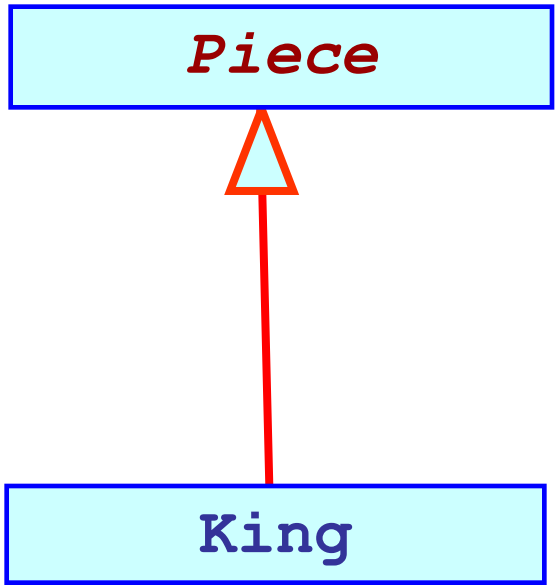
- Классы могут быть независимы друг от друга

- Классы могут быть связаны *отношением наследования*. (суперкласс/подкласс).

```
public class King extends Piece {
      void move(Square s) {
      }
}
```

Изображение наследования классов Java на UML-диаграммах

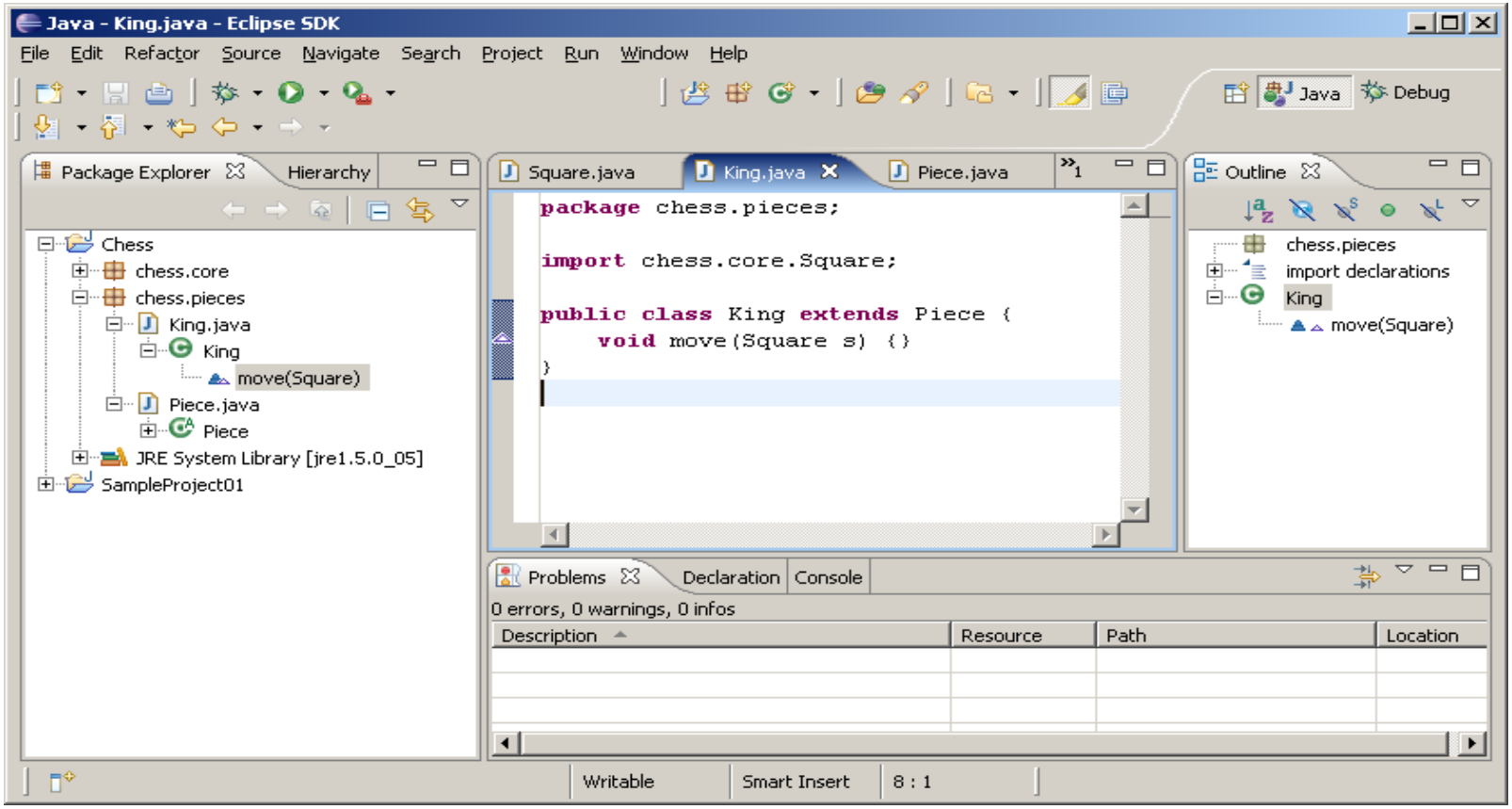

```
public class King extends Piece {
      void move(Square s) {
      }
}
```



Классы Java и их синтаксис. Наследование классов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Классы могут быть связаны отношением наследования (суперкласс/подкласс).



Интерфейсы в Java и их синтаксис. Объявление интерфейсов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Интерфейсы описывают методы, которые должны быть реализованы в классах
- Могут описывать неизменяемые (**final**) переменные
- Метафора – что должен уметь делать класс

```
// Интерфейс для классов представляющих
// шахматные ходы
public interface Move {
    // Передвинуть фигуры на доске.
    void doMove();

    // Вернуть фигуры в исходное состояние.
    void undoMove();
}
```

Интерфейсы в Java и их синтаксис. Реализация интерфейсов классами

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

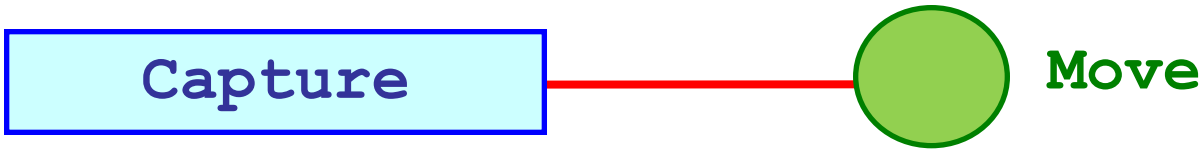
- Классы реализуют методы интерфейса
- Абстрактные классы могут реализовать не все методы интерфейса. Остальные реализуют потомки класса.

```
// Шахматный ход – рокировка.
public class Castling implements Move {
    void doMove() {
        // Передвинуть короля и ладью
    }

    void undoMove() {
        // Вернуть короля и ладью
    }
}
```

Интерфейсы в Java и их синтаксис. Реализация интерфейсов классами

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
// Шахматный ход – захват чужой фигуры
public class Capture implements Move {
    void doMove() {
        // Убрать чужую фигуру.
        // Передвинуть на ее место свою.
    }
    void undoMove() {
       // Передвинуть свою фигуру на прежнее место.
       // Поставить чужую фигуру.
    }
}
```



Интерфейсы в Java и их синтаксис. Не изменяемые поля интерфейса.

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
public interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
```

```
int color = BaseColors.RED;
```

Интерфейсы в Java и их синтаксис.
Статические методы интерфейсов в Java 8.

Романов Владимир Юрьевич ©2025

```
// Фигура ходит по диагонали.
public interface DiagonalPiece {

  static boolean isDiagonalPined(Square target) {
     // Может ли фигура пойти на поле target?
     // Откроется ли королю шах если диагональная
     // фигура пойдет на клетку target
  }

}
```

Интерфейсы в Java и их синтаксис.
Статические методы интерфейсов в Java 8.

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

```
// Фигура ходит по вертикали или горизонтали.
public interface LinePiece {
  static
  boolean isVerticalPined(Square target) {
     // Может ли фигура пойти на поле target?
     // Откроется ли королю шах если
     // фигура пойдет на клетку target
  }

  static
  boolean isHorizontalPined(Square target) {
     // Может ли фигура пойти на поле target?
     // Откроется ли королю шах если
     // фигура пойдет на клетку target
  }
}
```

Интерфейсы в Java и их синтаксис
Использование статических методов в Java 8


```
public class Queen extends Piece
    implements DiagonalPiece, LinePiece
{
    boolean isCorrectMove(Square s) {
        if (isHorizontalPined(s))
            return false;
        if (isVerticalPined(s))
            return false;
        if (isDiagonalPined(s))
            return false;
        // …
    }
    // …
}
```

Интерфейсы в Java и их синтаксис.
Использование статических методов в Java 8


```
public class Bishop extends Piece
    implements DiagonalPiece
{
    void isCorrectMove(Square s) {
       if (isDiagonalPined(s))
           return false;
       // …
    }
    // …
}
```

Интерфейсы в Java и их синтаксис
Использование статических методов в Java 8


```
public class Rook extends Piece
    implements LinePiece
{
 boolean isCorrectMove(Square s) {
      if (isHorizontalPined(s))
          return false;
      if (isVerticalPined(s))
          return false;
      // …
    }
    // …
}
```

Интерфейсы в Java и их синтаксис
Расширение интерфейсов


```
// Интерфейс для классов представляющих
// шахматные ходы
public interface Move {
    void doMove();
    void undoMove();

    Board getBoard(); // Ошибка!!!
}
```

Классы в уже оттранслированных библиотеках не смогут реализовать новый метод интерфейса

Интерфейсы в Java и их синтаксис

Использование умалчиваемых методов в Java 8

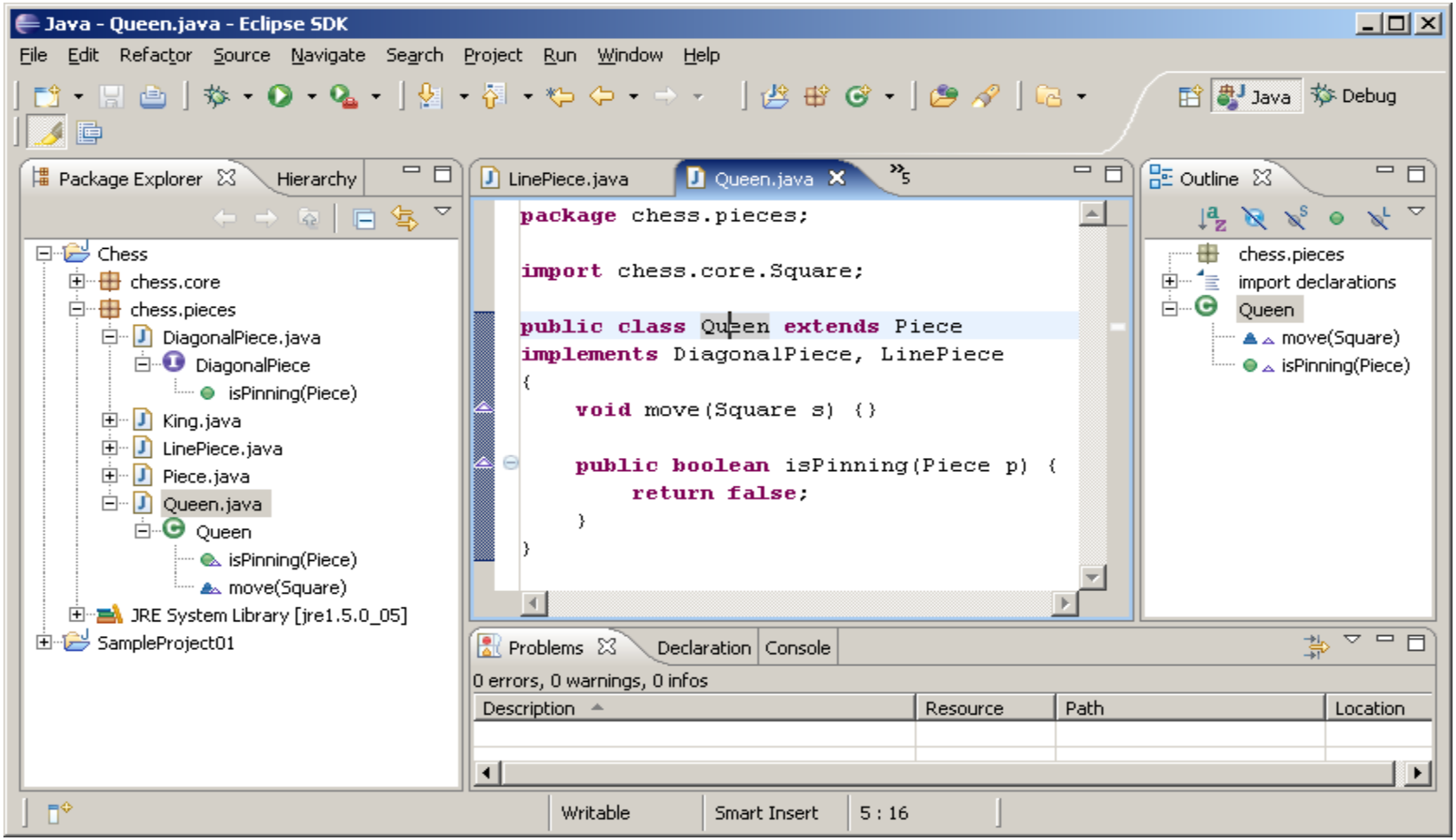

```
// Интерфейс для классов представляющих
// шахматные ходы
public interface Move {
    void doMove();
    void undoMove();

    default Board getBoard() {
       // …
    }
}
```

Для классов в уже оттранслированных библиотеках используется умалчиваемая реализация метода интерфейса

Классы Java и их синтаксис. Реализация интерфейсов. Eclipse

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Классы могут реализовывать один и более *интерфейсов*



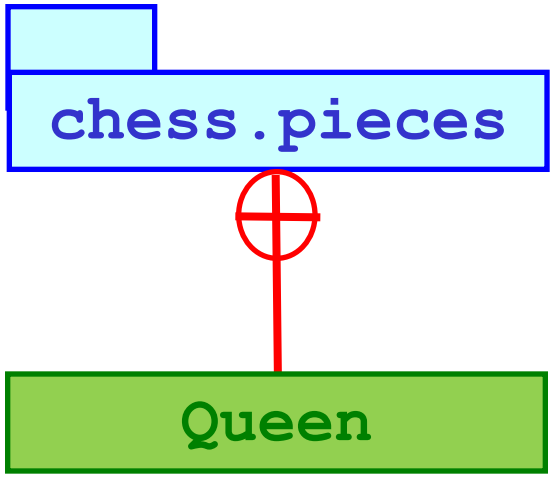
Классы Java и их синтаксис. Группирование классов в пакеты

- Логически связанные классы группируются в *пакеты*

```
package chess.pieces;

import chess.core.Square;

public class Queen extends Piece
        implements DiagonalPiece, LinePiece
{
      void move(Square s) {}

      public boolean isPinning(Piece p) {
            return false;
      }
}
```

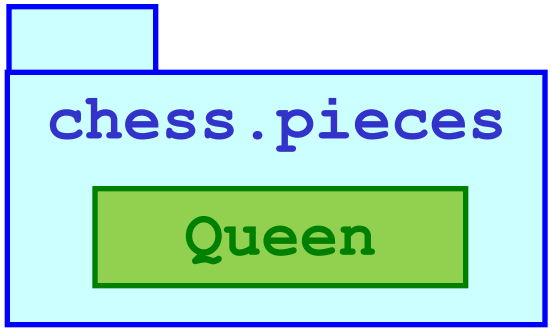
Изображение пакетов на UML-диаграммах


```
package chess.pieces;

public class Queen extends Piece
implements DiagonalPiece, LinePiece
{
}
```



Изображение пакетов на UML-диаграммах

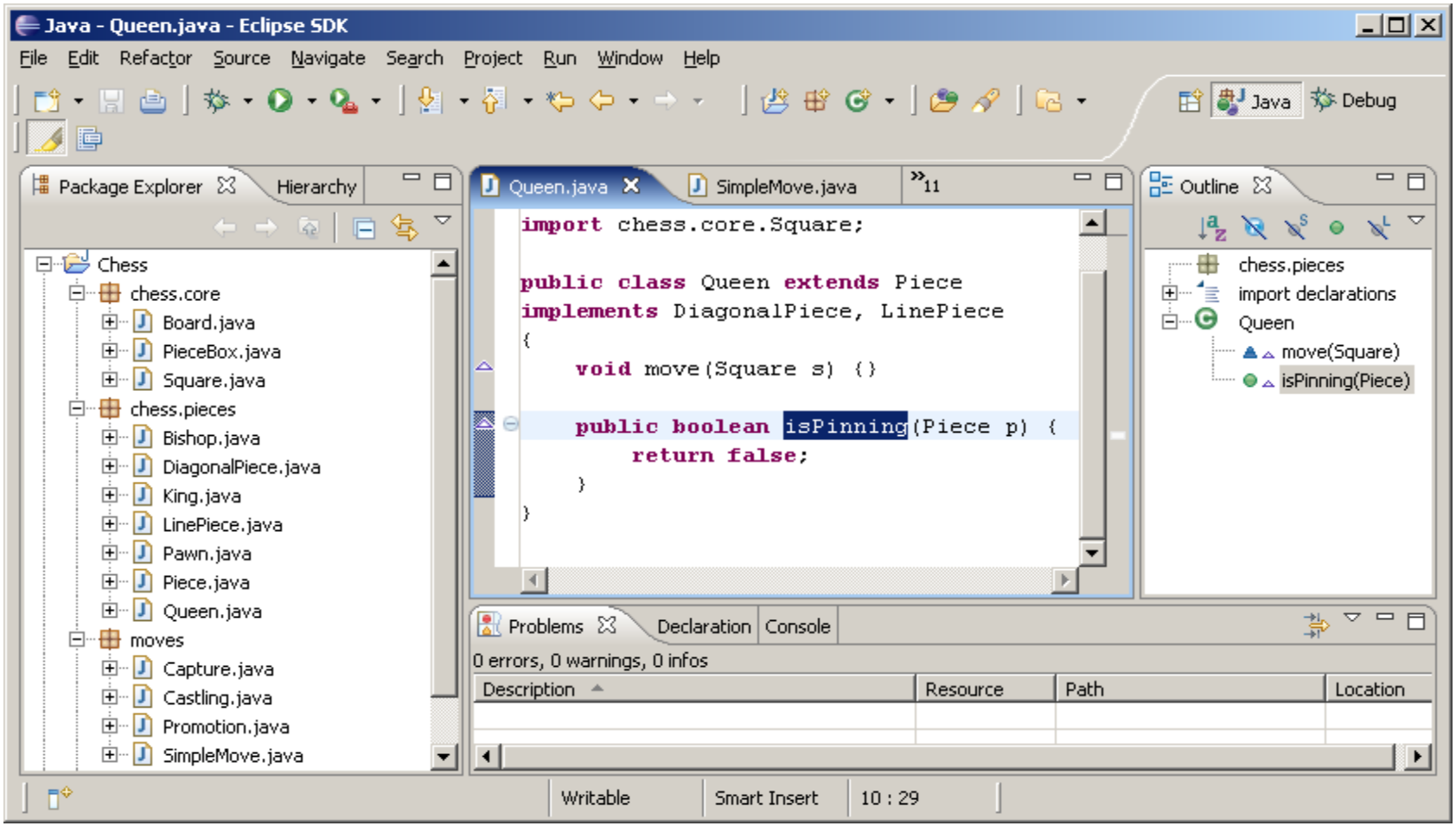

```
package chess.pieces;

public class Queen extends Piece
implements DiagonalPiece, LinePiece
{
}
```



Классы Java и их синтаксис. Группирование классов в пакеты

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Логически связанные классы группируются в пакеты



Классы Java и их синтаксис. Модификаторы классов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Модификатор *public*

Класс доступен для всех других классов. Отсутствие модификатора означает доступность класса только классам внутри пакета содержащего данный класс.

- Модификатор *private*

Этот модификатор допустим только для классов, которые *вложены (nested)* в другие классы

- Модификатор *abstract*

Запрет на создание экземпляров класса

- Модификатор *final*

Запрет на создание подклассов данного класса

Изображение модификаторов класса на UML

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
abstract public class Piece
{
}
```

Конструкторы классов

- *Конструктор* – метод который создает экземпляр класса. Имя конструктора совпадает с именем класса. Допустимы несколько конструкторов с различными параметрами

- Конструктор используется для инициализации объектов

- Тело класса содержит по меньшей мере один конструктор

- Конструктор возвращает указатель на созданный объект. Оператор ***return*** в конструкторе отсутствует

- Для создания *экземпляров* класса используется ключевое слово **new** с именем конструктора

```
King piece = new King();
```

Умалчиваемые конструкторы классов

• Умалчиваемый конструктор не имеет аргументов.
Предоставляется по умолчанию платформой Java если нет ни одного явно определенного конструктора

• При определении хотя бы одного явного конструктора необходимо объявить явно и умалчиваемый конструктор

```
class King {}
King piece = new King();
```

```
class King {
        King(Square s) {}
}
King piece = new King(); // Ошибка компиляции!!!
```

```
class King {
        King() {}
        King(Square s) {}
}
King piece = new King(); // Ошибки компиляции теперь нет
```

Цепочка конструкторов

- *Цепочка конструкторов* - вызов внутри класса одного конструктора другим конструктором. Цепочки используются для разделения общего кода между конструкторами

- Вызов цепочки конструкторов:

this(список аргументов)

```
class Piece {
boolean isWhite;
Square square;

Piece(Square square, boolean isWhite)
     { this.square = square; this.isWhite = isWhite; }
Piece(Square square)
     { this(square, true); }
Piece()
     { this(null, true); }
}
```

Конструкторы суперкласса

- Объекты суперкласса строятся до построения объекта-подкласса

- Для инициализации членов суперкласса используется вызов:
  - **super(**список-аргументов)

- Первой строкой конструктора могут быть:
  - **super(**список-аргументов)
  - **this(**список-аргументов)

- В одном конструкторе нельзя использовать одновременно и **super** и **this**

- Компилятор предоставляет *неявный вызов конструктора* **super()** для всех конструкторов подклассов

Освобождение памяти объектов

- В Java отсутствует понятие *деструктора* для неиспользуемых объектов

- Освобождение памяти выполняется автоматически виртуальной машиной Java

- Сборщик мусора освобождает память объектов, на которые нет ссылок

- Связь между объектом и ссылкой на объект уничтожается при задании нового значения ссылке на объект

```
objectReference = null;
```

- Объект без ссылок – кандидат на освобождение при сборке мусора

Сборка мусора

- Можно явно запросить сборку мусора:

```
System.gc()
```

- Метод объекта *finalize* будет выполняться непосредственно перед сборкой мусора.

```
@Override
protected void finalize() {
    // TODO Здесь освободить захваченные ресурсы
}
```

Используется при:

- освобождении памяти выделенной с помощью ***native***-методов
- закрытии открытых объектом файлов перед тем, как память объекта будет освобождена.

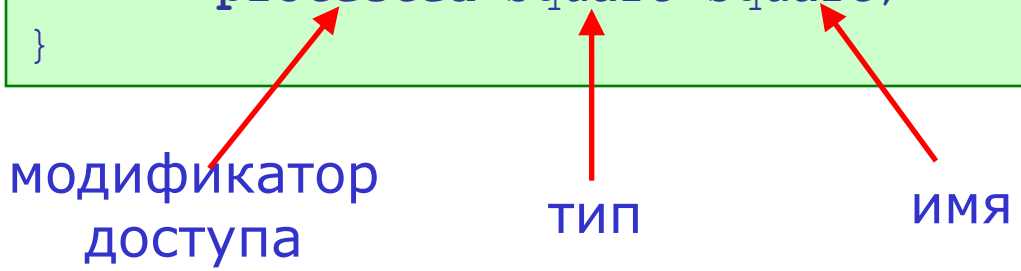
Поля классов

- *Поля*
  - Часть определения классов
  - Состояние объекта хранится в полях
  - Каждый экземпляр получает собственную копию переменных экземпляра

- В месте объявления поля могут быть *инициализированы*

- Если поля не инициализируются явно, то используются *умалчиваемые значения*

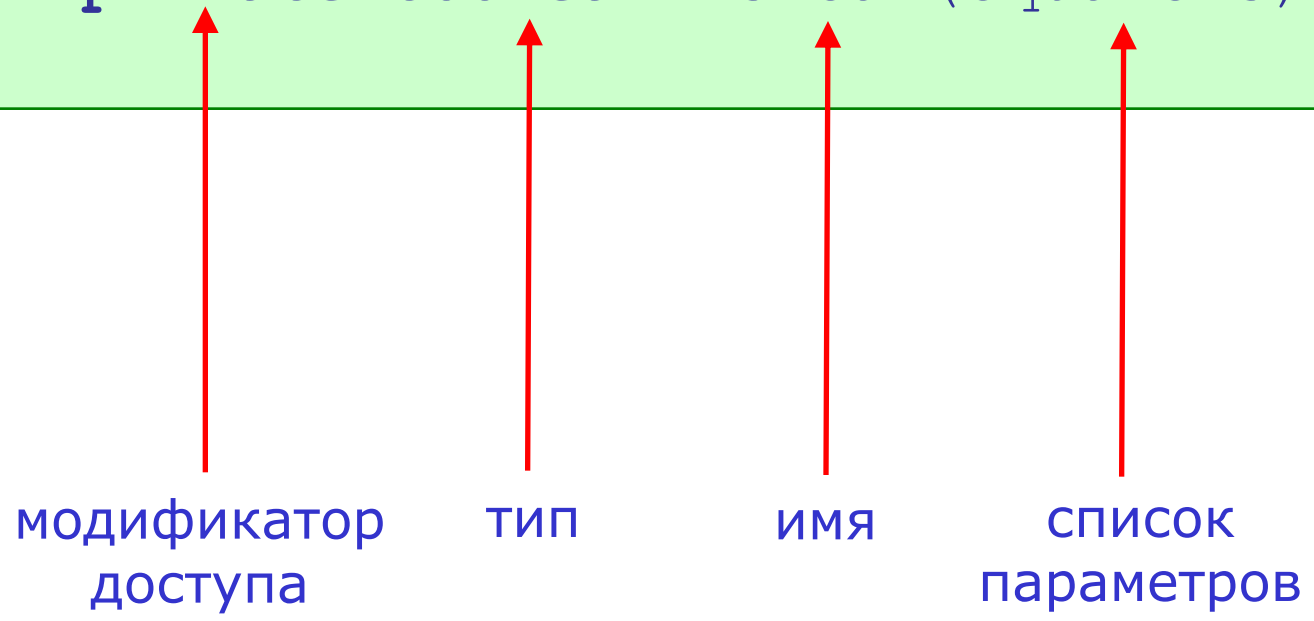
```
public class Piece {
        public boolean isWhite;
        protected Square square;
}
```



Методы

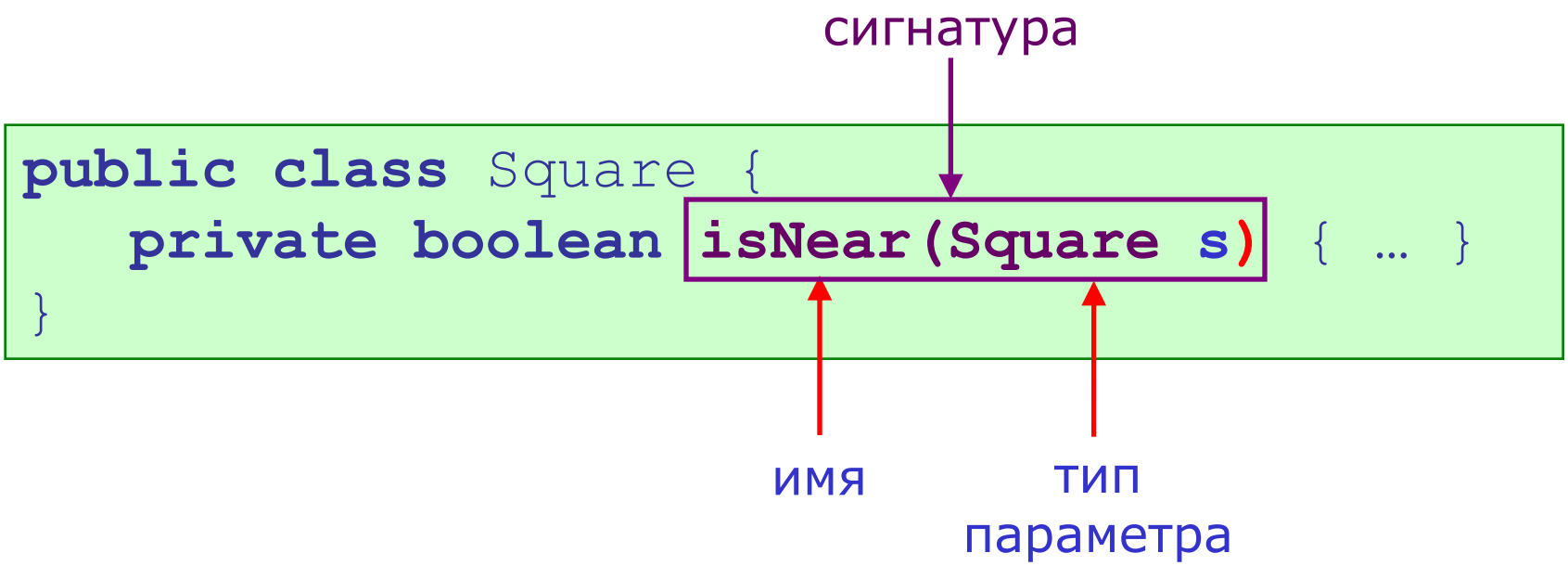
- *Методы* определяют
  - Как объект отвечает на сообщение
  - Поведение класса
  - Все методы принадлежат классу

```
public class Square {
   private boolean isNear(Square s) { … }
}
```



Сигнатура метода

- Класс может иметь множество методов с одинаковыми именами
- Каждый метод должен иметь другую сигнатуру
- Сигнатура – количество аргументов и типы аргументов



Параметры метода

- Аргументы (параметры) пересылаются:
  - примитивные типы по значению
  - ссылки на объекты для ссылочных типов

- Примитивные значения не могут быть модифицированы при пересылке в качестве аргументов

```
public void method1() {
    int a = 0;
    System.out.println(a); // вывод 0
    method2(a);
    System.out.println(a); // вывод 0
}

void method2(int a){
    a = a + 1;
}
```

Вызов метода и возврат из метода

- И для вызова метода используется оператор точка
  - для методов экземпляра и для методов класса
  - если вызываемый метод в том же классе, то оператор точка не требуется

Вызов через оператор точка метода класса

```
public class King {
    boolean wasCastling() { … }

    boolean isCorrectMove(Square newSquare) {
       Square s = King.oppositeKing().square;
       return !s.isNear(newSquare) && !wasCastling();
    }
}
```

Вызов через оператор точка метода экземпляра

метод определен в том же классе

Перекрытие метода

- Метод суперкласса перекрывается методом подкласса с той же сигнатурой

Перекрытый метод

```
public class Piece {
   boolean isCorrectMove(Square newSquare) {…}
}

public class King extends Piece {
   boolean isCorrectMove(Square newSquare) {
       if (!super.isCorrectMove(newSquare))
           return false;

       Square s = King.oppositeKing().square;
       return !s.isNear(square) && !wasCastling();
    }
}
```

Перекрывающий метод

Вызов перекрытого метода

Метод *Main*

- Приложение не может быть выполнено без хотя бы одного класса с методом *main*

- Виртуальная машина Java загружает класс и начинает выполнение с метода *main*

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

- public – метод может быть вызван любым объектом
  - **public**
- static – нет необходимости сначала создавать объект

- void – этот метод ничего не возвращает

Инкапсуляция

- Объект – содержит данные и действия, которые можно выполнить над данными.

- Принцип сокрытия информации – объект знает о себе все, другие объекты запрашивают информацию об этом объекте.

- Все объекты отличаются друг от друга и программа – это обмен сообщениями между объектами

- Для скрытия информации используется модификатор доступа *private*

Статические члены

- Статические поля и методы принадлежат классу

- При изменении статического значения одним из объектов данного класса изменяется значение для всех объектов данного класса

- Статические методы и поля могут быть доступны без создания экземпляров класса

Конечные (*final*) члены класса

- Поле с модификатором *final* не может быть модифицировано. Это аналог констант в языке Java

- Константы связанные с классом обычно для простоты доступа объявляются с модификаторами *static final*

- Общепринято константы записывать большими буквами

```
public class Piece {

    static final public  KING  = 1;
    static final public  QUEEN = 2;
    static final public  PAWN  = 3;
}
```

Абстрактные классы

- Невозможно создать экземпляр абстрактного класса. Предполагается, что они будут суперклассами для других классов

- Методы с модификатором *abstract* не имеют реализации

- Если класс имеет абстрактные методы, то он должен быть объявлен абстрактным

```
abstract public class Piece {
   boolean isCorrectMove(Square s) { … }
   abstract void makeMove(Square s);
}
```

Пакеты классов

- Классы могут быть сгруппированы в пакеты

```
package chess.pieces;

abstract public class Piece {
   boolean isCorrectMove(Square s) { … }
   abstract void makeMove(Square s);
}
```

- Различные пакеты могут иметь классы с одинаковыми именами

- По соглашению имена пакетов задаются в нижнем регистре

Видимость классов

- Для ссылки на классы в том же пакете можно использовать только имя класса

- Для ссылки на классы из других пакетов необходимо использовать полностью квалифицированное имя класса

```
package chess.movies;

public class Castling extends Move {
    void doMove(Square s) {
        if (chess.pieces.King.wasCasling())
            return;
    }
}
```

Импорт классов

- Предложение *import* используется для того, что бы сделать классы непосредственно доступными

```
package chess.movies;

import game.core.*;
import chess.pieces.King;
import chess.pieces.Rook;

public class Castling extends Move {
    void doMove(Square s) {
        if (King.wasCasling()) return;
        if (Rook.wasMoved())   return;
    }
}
```

Импорт классов

- Предложение *import* используется для того, что бы сделать классы непосредственно доступными

```
package chess.movies;

import chess;

public class Castling extends Move {
    void doMove(Square s) {
        if (pieces.King.wasCasling()) return;
        if (pieces.Rook.wasMoved())   return;
    }
}
```

Импорт статических методов классов в Java8

- Пакет java.lang импортируется по умолчанию.

```
package samples;

//import java.lang.Math;

public class StaticImportSample {

    public static void main(String[] args) {
        Math.sin(Math.PI);
    }

}
```

Импорт статических методов классов в Java8

- Предложение ***import*** используется для импорта статической функции ***sin*** и статического поля **PI** класса **Math**.

```
package samples;

import static java.lang.Math.sin;
import static java.lang.Math.PI;

public class StaticImportSample {

   public static void main(String[] args) {
      sin(PI);
   }

}
```

Импорт статических методов классов в Java8

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Предложение *import* используется для импорта **всех** статических методов и полей класса **Math**.

```
package samples;

import static java.lang.Math.*;

public class StaticImportSample {

    public static void main(String[] args) {
        sin(PI);
    }

}
```

Пакеты ядра Java (1)

- java.lang

- неявно импортируется во все пакеты
- предоставляет фундаментальные классы языка программирования Java (Object, String, StringBuffer, …)

- java.util

- библиотека классов-коллекций
- модель для программирования событий
- классы для работы с датами и временем
- классы для локализации программ на различных национальных языках

- java.io

- работа через потоки ввода/вывода
- сериализация
- работа с файловой системой

Пакеты ядра Java (2)

- java.math

- стандартные математические функции
- работа с большими целыми числами BigInteger
- работа с большими вещественными числами BigDecimal

- java.sql

- классы для анализа структуры реляционной базы данных
- классы для выполнения запросов к базе данных на языке SQL

- java.text

классы и интерфейсы для обработки текста, дат, чисел способом независимым от национальных языков

Наследование в языке Java

- Понимание наследования полей и методов
- Знакомство с иерархией классов
- Как подклассы специализируют классы
- Как выполняется поиск метода
- Как создаются и используются подклассы
- Понимание полиморфизма
- Рефакторинг в иерархии наследования

Иерархии классов

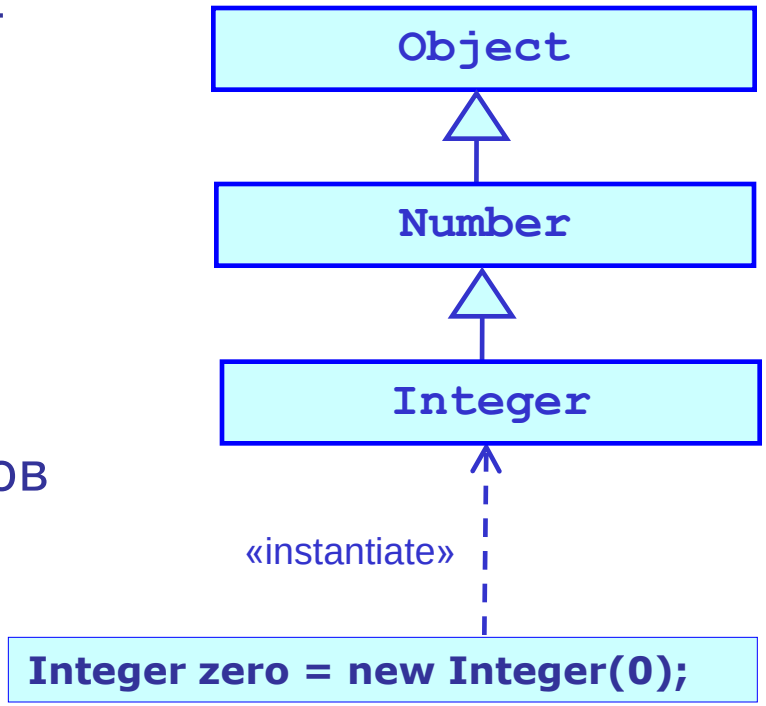
Романов Владимир Юрьевич ©2025

- Каждый объект принадлежит классу (является экземпляром класса)

- Каждый класс (кроме класса Object) имеет суперкласс

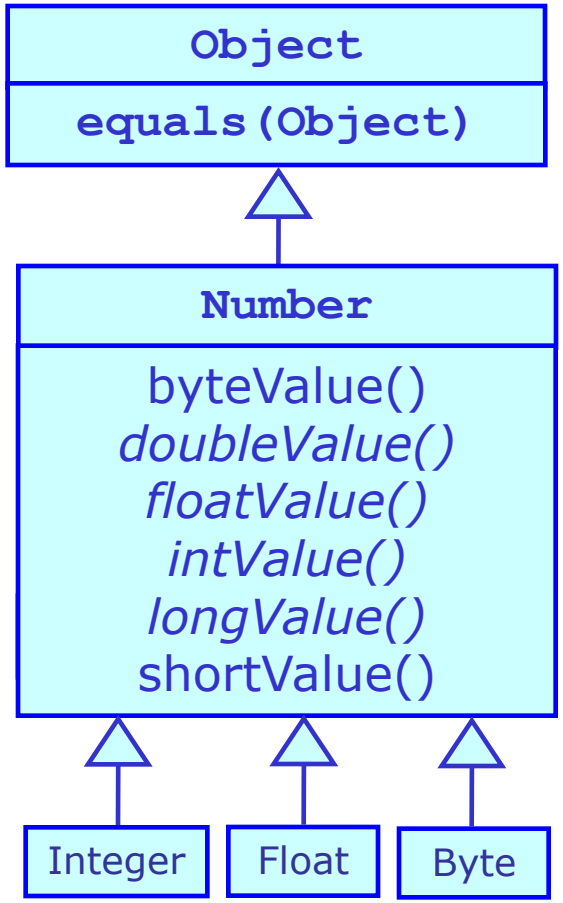
- Корень всей иерархии классов – класс ***Object***.

- При определении нового класса



Специализация и обобщение

- Подкласс есть специализация его суперкласса
- Суперкласс есть обобщение его подклассов.
- Общее состояние и поведение подкласса перемещается в суперкласс и становится доступным всем подклассам

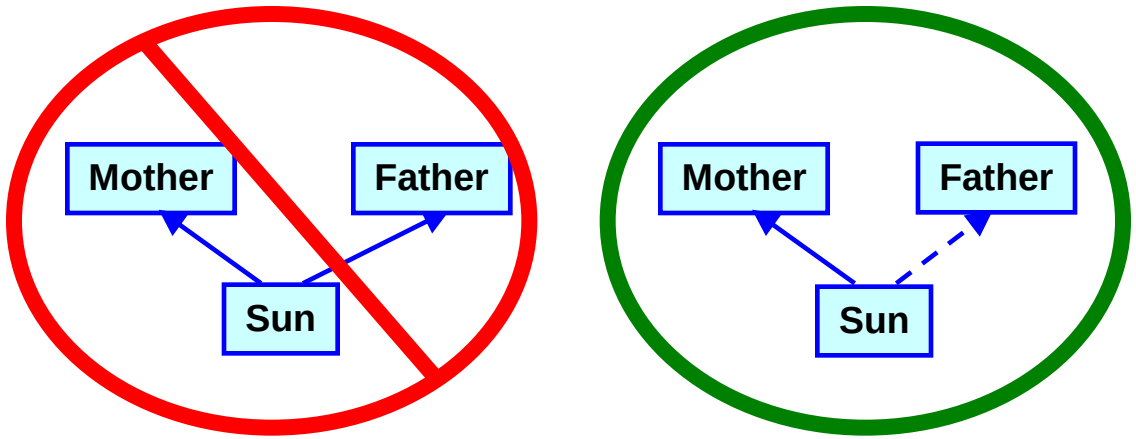


Множественное наследование

- Множественное наследование в языке Java не поддерживается

- Каждый класс, за исключением класса Object, имеет только один непосредственный суперкласс

- Для реализации множества общих методов различными классами можно использовать не только наследование, но и реализацию классами интерфейсов

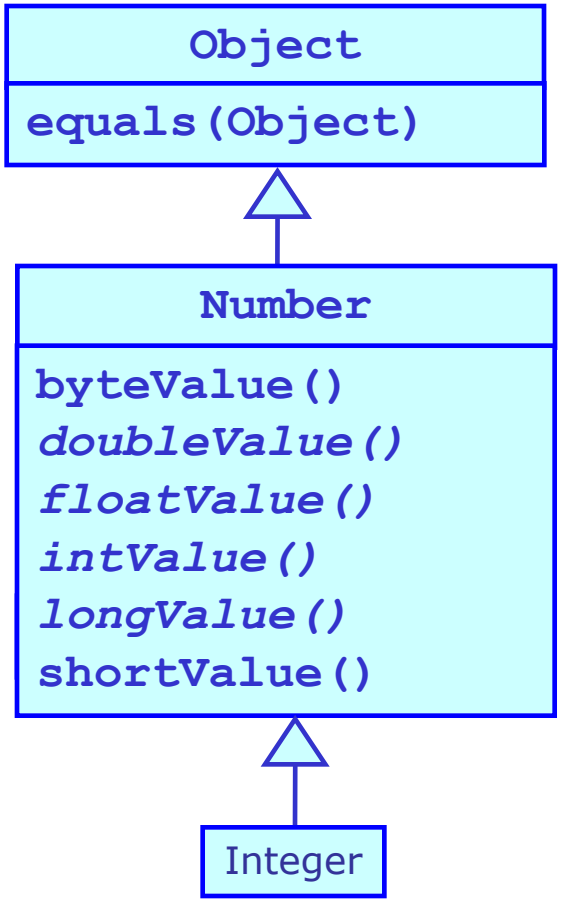


Наследование полей и методов

- Каждый подкласс наследует поля суперкласса и всех классов расположенных выше в иерархии наследования

- Каждый подкласс наследует методы суперкласса. Объект будет понимать все сообщения (вызов методов) его класса и суперклассов

```
Integer zero = new Integer(0);
if (zero.equals(x)) {
   byte b = zero.byteValue();
   …
}
```



Модификаторы доступа

- Поля и методы в Java имеют ограничения по доступу, описываемые следующими модификаторами:
  - *private* – доступ ограничен классом в котором объявлен данный член класса

```
private int x;
```

  - *умалчиваемый* (без модификатора) – доступ ограничен пакетом, в котором данный класс объявлен

```
int y;
```

  - *protected* – доступ ограничен пакетом, в котором данный класс объявлен, и подклассами данного класса

```
protected void setName(String name) {…}
```

  - *public* – доступ для всех классов всех пакетов

```
public String getName() {…}
```

Перекрытие наследуемых методов

- Имеется возможность дополнить или изменить поведение суперкласса перекрытием в подклассе унаследованного метода

- Перекрывающие метод должен иметь то же имя и список параметров (сигнатуру)

- Метод подкласса может заменять или уточнять метод суперкласса

```
public class MyClass extends Object {
   public boolean equals(Object o) {
      if (o==null)
      …
   }
}
```

Ограничения на перекрытие методов

- Список параметров должен совпадать со списком параметров перекрываемого наследуемого метода суперкласса

- Тип возвращаемого результата должен совпадать с типом возвращаемого результата метода суперкласса

- Модификатор доступа в суперклассе не может быть более ограничительным, чем модификатор доступа в подклассе

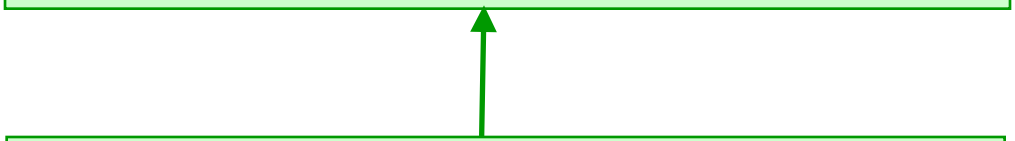
Пример. При перекрытии метода с модификатором *protected*, новый метод может быть *protected* или *public*, но не *private*

Наследование и статические методы

- Класс может использовать статические методы унаследованные от суперклассов и собственные статические методы

- Статические методы не могут быть перекрыты

```
static String t = "test";

public static String superTest(String s) {
    s += " was the arg.";
    return s;
}
```



```
public static void main(String[] args){
   System.out.println(superTest(t));
}
```

Наследование и конструкторы

- Вызываться могут только конструкторы класса, экземпляр которого создается, либо конструктор непосредственного суперкласса

- Для вызова конструктора суперкласса используется ключевое слово *super* и список параметров конструктора

- Для вызова конструктора того же класса используется ключевое слово *this* и список параметров конструктора

- Первой строкой конструктора может быть одна из:

```
this()
super()
```

- Вызов конструктора через this() или super() допустим только в конструкторе

Блоки инициализации. Инициализация экземпляров класса

- Блок инициализации экземпляра вызывается до вызова любого из конструкторов класса

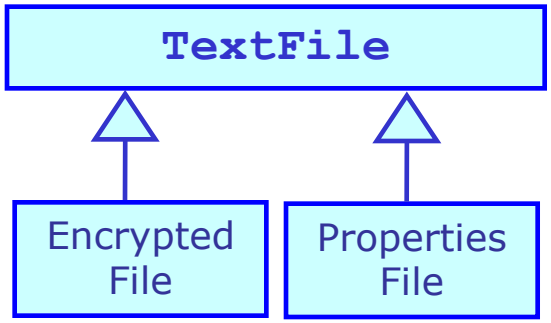
- В классе может быть неограниченное количество блоков инициализации

- Порядок расположения блоков инициализации в классе не имеет значения

Полиморфизм

- Переменной может быть присвоен объект типа переменной, либо объект типа – подкласс типа переменной:

```
TextFile file = new TextFile();
TextFile file = new EncryptedFile();
TextFile file = new PropertiesFile();
```



- Любой объект может быть присвоен переменной типа Object, поскольку он самый верхний в иерархии объектов

```
Object anything = new AnyType();
```

Пример инициализации экземпляра класса


```
package geometry;

public class Point {
    int x, y;

    {
        x = 1;
    }

    public Point() { }

    {
        y = 1;
    }
}
```

Блок инициализации экземпляра

Порядок расположения блоков инициализации не важен

Блоки инициализации. Инициализация статических полей класса

- Блок инициализации статических полей класса вызывается до вызова любого из конструкторов класса

- В классе может быть неограниченное количество блоков инициализации статических полей классов

- Порядок расположения в классе блоков инициализации статических полей классов не имеет значения

Пример инициализации статических полей класса


```
package graph;

public class Node {
    static Color defaultLineColor, defaultColorFill;

    static {
        defaultLineColor = Color.Blue;
    }

    public Node() { }

    static {
        defaultFillColer = Color.Cyan;
    }
}
```

Блок инициализации статических полей

Порядок расположения блоков инициализации не важен

Вложенность классов

- Способ группирования классов используемых только в одном месте. Вспомогательные классы внутри

- Увеличение инкапсуляции – вложенный класс работает со *скрытыми* (*private*) полями и методами *внешнего (outer)* класса.

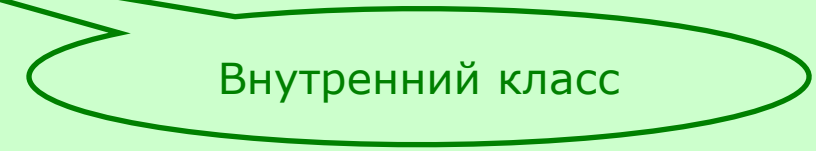
- Получаемый код проще понимать – вложенные классы обычно небольшие

- Получаемый код проще сопровождать – место использования класса расположено близко к месту объявления класса

Статические вложенные и внутренние классы

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

```
class OuterClass {
    ...

    static class StaticNestedClass {
        ...
    }

    class InnerClass {
        ...
    }
}
```



Доступ к статическим вложенным классам


```
class OuterClass {

        static class StaticNestedClass {
        }

} // class OuterClass

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

        OuterClass.StaticNestedClass nestedInstance
                    = new OuterClass.StaticNestedClass();

    } // main
} // class MainClass
```

Внутренние классы

- Внутренние классы могут существовать только внутри экземпляров внешнего класса

- Внутренние классы имеют прямой доступ к полям и методам экземпляра класса

- Перед созданием экземпляра внутреннего класса необходимо создать экземпляр внешнего класса

- Для внутренних классов можно задавать видимости: public, private, protected

Доступ к внутренним классам


```
class OuterClass {

     class InnerClass {
     }

} // class OuterClass

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

       OuterClass outerInstance = new OuterClass();
       OuterClass.InnerClass innerInstance
                    = outerInstance.new InnerClass();

    } // main
} // class MainClass
```

Внешний класс

Внутренний класс

Экземпляр внешнего класса

Внутренние локальные классы

- *Внутренние локальные классы* могут объявляются внутри методов класса

- Внутренние локальные классы имеют доступ к параметрам и локальным переменным класса

```
class OuterClass {
      public void outerMethod(int x) {
            int y;

            class InnerClass {
            } // class InnerClass

      } // outerMethod
} // class OuterClass
```

Внутренние анонимные классы

- *Внутренние анонимные классы* могут объявляются внутри методов класса

- Имя анонимного класса не указывается

```
class OuterClass {
      public void outerMethod(int x) {

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

      } // outerMethod
} // class OuterClass
```

Анонимный внутренний класс реализующий метод *run* интерфейса *Runable*

Романов Владимир Юрьевич ©2025

Перечисления в языке Java

Типы-перечисления

- *Типы-перечисления* – это типы поля которых содержат фиксированние множество констант

```
public enum BaseColor {
       RED, GREEN, BLUE
}

public class Test {
    static public void main(String[] args) {
        BaseColor color = BaseColor.RED;
        switch (color) {
           case RED:  System.out.println("Red");  break;
           case BLUE: System.out.println("Blue"); break;
        } // switch
    } // main
} // class Test
```

Поля и методы типов-перечислений


```
public enum Color {
        RED(255, 0, 0),
        GREEN(0, 255, 0),
        BLUE(0, 0, 255),
        WHITE(255, 255, 255),
        BLACK(0,0,0);

        public int red, green, blue;

        public int getIntensity()
            { return red + green + blue; }

        Color(int red, int green, int blue) {
                this.red = red;
                this.green = green;
                this.red = blue;
        }
}// enum Color
```

Возможно использование вне типа-перечисления

Возможно использование только внутри перечисления

Поля и методы типов-перечислений


```
public enum Dirs {
      LEFT_UP  (-1, -1),     UP  (+0, -1),     RIGHT_UP  (+1, -1),
      LEFT (-1, +0),                           RIGHT (+1, +0),
      LEFT_DOWN(-1, +1), DOWN(+0, +1), RIGHT_DOWN (+1, +1);

      public static final
      Dirs[] ALL = {
            LEFT_UP,       UP,       RIGHT_UP,
            LEFT,                    RIGHT,
            LEFT_DOWN, DOWN, RIGHT_DOWN
      };

      public int dv, dh;

      Dirs(int dv, int dh) {
            this.dv = dv;
            this.dh = dh;
      }
}
```

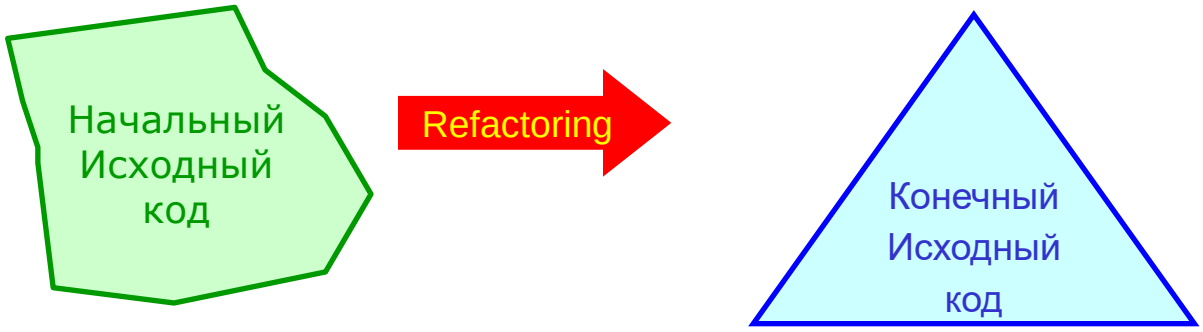
Романов Владимир Юрьевич

Рефакторинг программ на языке Java

Что такое рефакторинг

- Рефакторинг – процесс изменения программной системы таким образом, что внешнее поведение кода не изменяется, но внутренняя структура и архитектура улучшаются

- Рефакторинг – преобразование исходного кода сохраняющего поведение



Для чего необходим рефакторинг

- Улучшение читаемости и понятности
- Улучшение расширяемости кода
- Добавление гибкости
- Улучшение производительности

Как и когда выполняется рефакторинг

- Основа рефакторинга – обобщение. Абстракции находятся снизу вверх проверкой множества конкретных примеров

- Поиск методов с различными именами, но имеющими схожее поведение

- Параметризация различий у методов

- Разделение больших методов на методы меньшего размера, но допускающие большее переиспользование

- Выполняется во время сопровождения, тестирования, кодирования

Цикл рефакторинга

- Исходный код программы должен проходить через фазы расширения и упорядочения
  - Фаза расширения – код добавляется для реализации новых функциональных требований
  - Фаза упорядочивания – код удаляется и преобразуется для улучшения структуры кода и архитектуры системы
- За время жизни программы этот цикл повторяется многократно

Распространенные причины отказа от рефакторинга

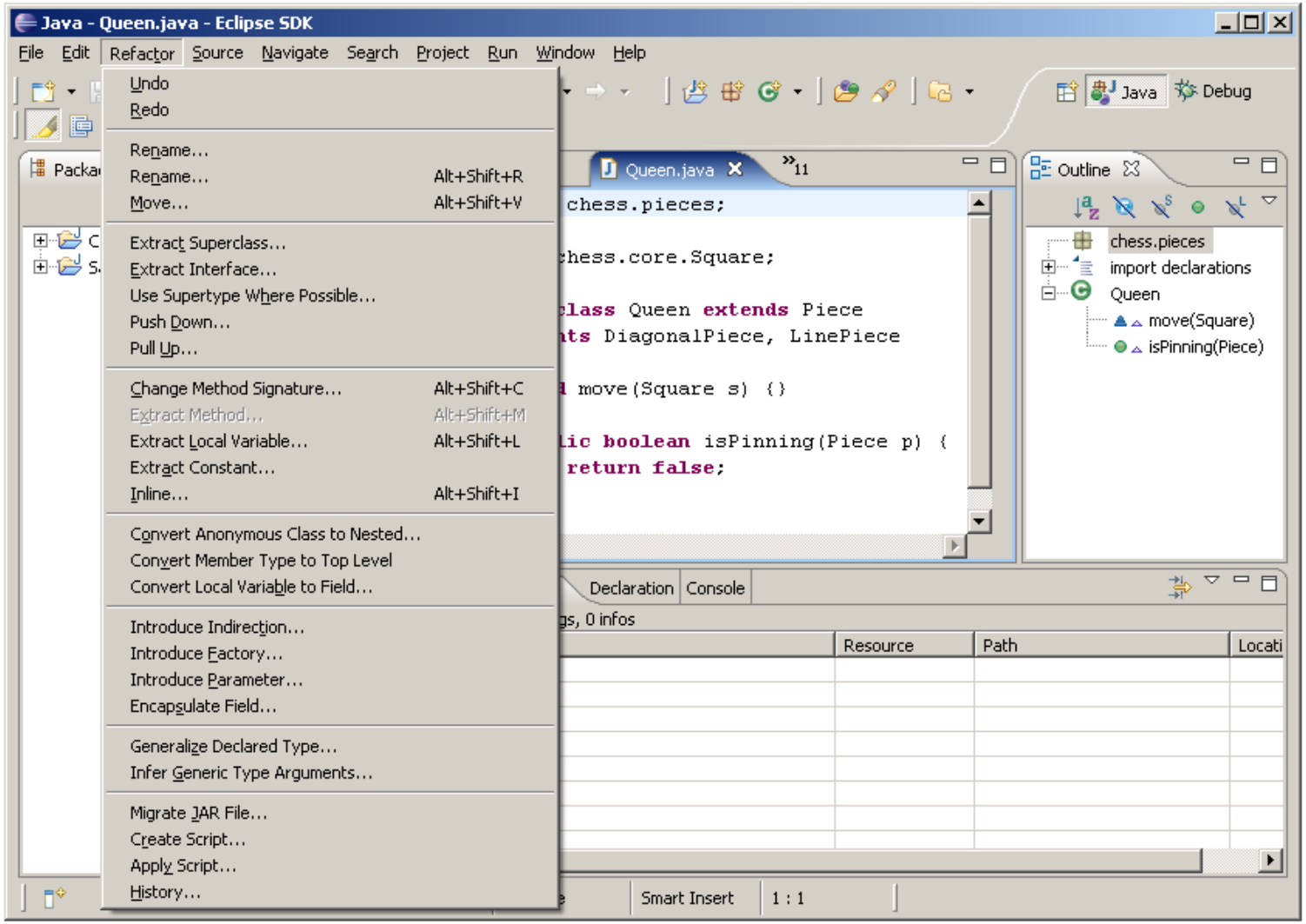
- Изменение кода/проекта системы может привести его нерабочее состояние
  - Автоматизация тестирования позволит устранить эту причину отказа системы
- Непонятно как система работает в данный момент
  - В процессе рефакторинга система может быть изучена
  - Процесс рефакторинга можно документировать
- Недостаточно времени для выполнения рефакторинга
  - Проведение рефакторинга позволит существенно сократить время разработки на более поздних фазах работы с программой

Методы рефакторинга

- Создание и удаление
  - классов, методов, переменных
- Перемещение методов и переменных:
  - вниз/вверх по иерархии наследования
  - перемещение в другой класс
- Реорганизация
  - Иерархии наследования
  - Кода методов

Использование среды Eclipse как инструмента рефакторинга

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Рефакторинг в Eclipse. Переименование типа (1)

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

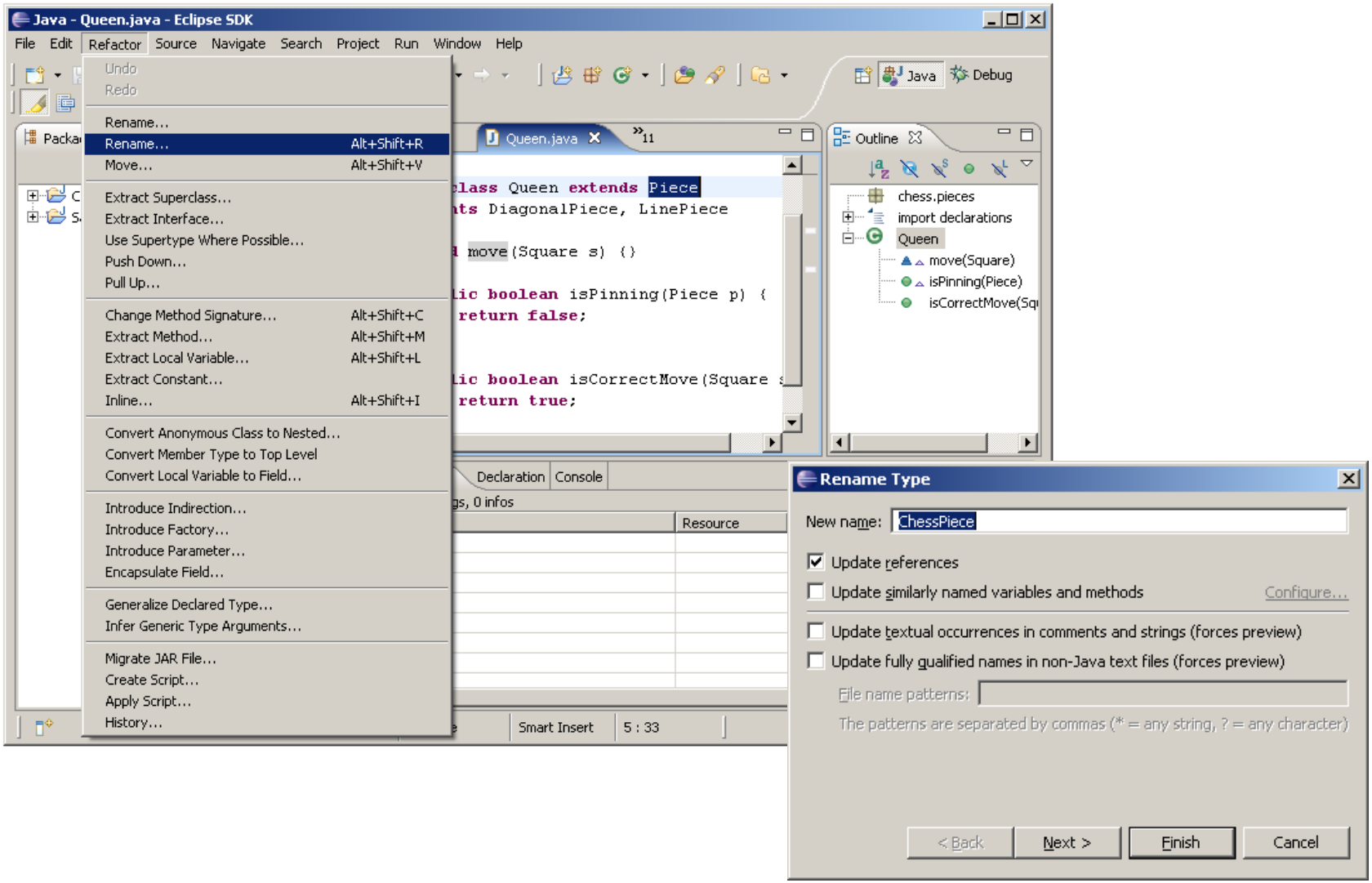
Среда Eclipse существенно упрощает проведение рефакторинга

- Поиск в программе всех мест, где используется переименовываемый тип
- Отображение предполагаемых изменений и их влияния на программу
- Выполнение изменений

Рефакторинг в Eclipse. Переименование типа (2)

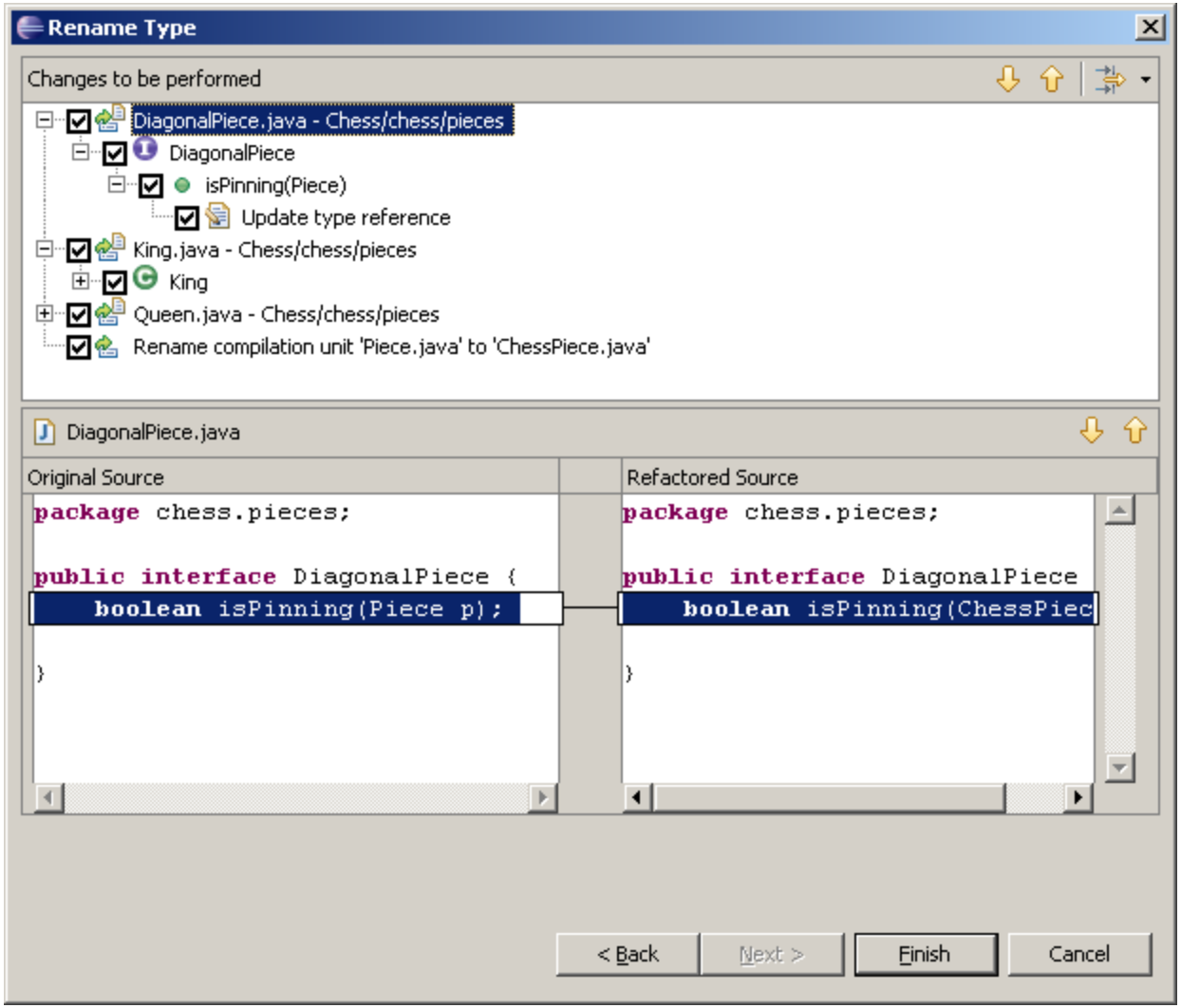
МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025



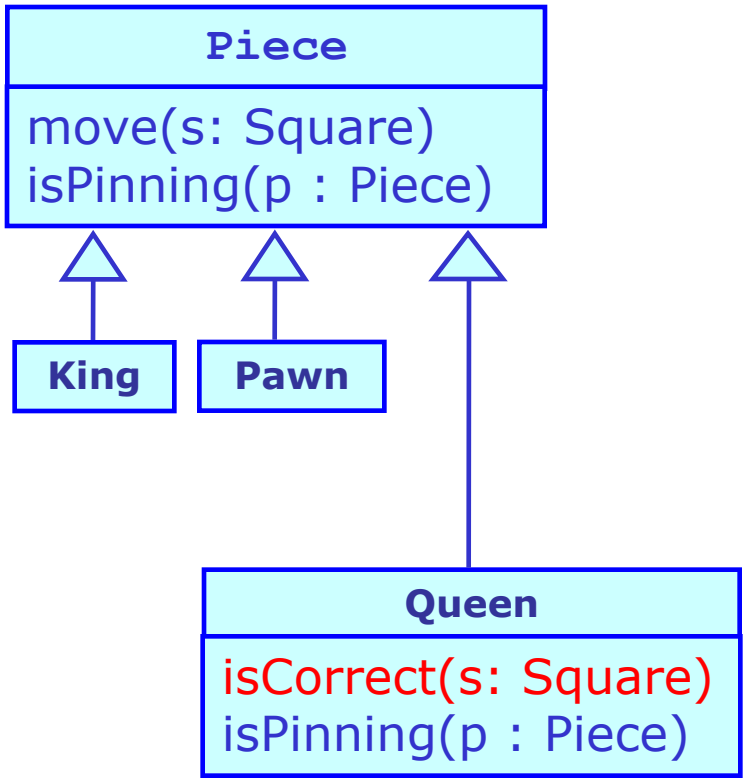
Рефакторинг в Eclipse. Переименование типа (3)

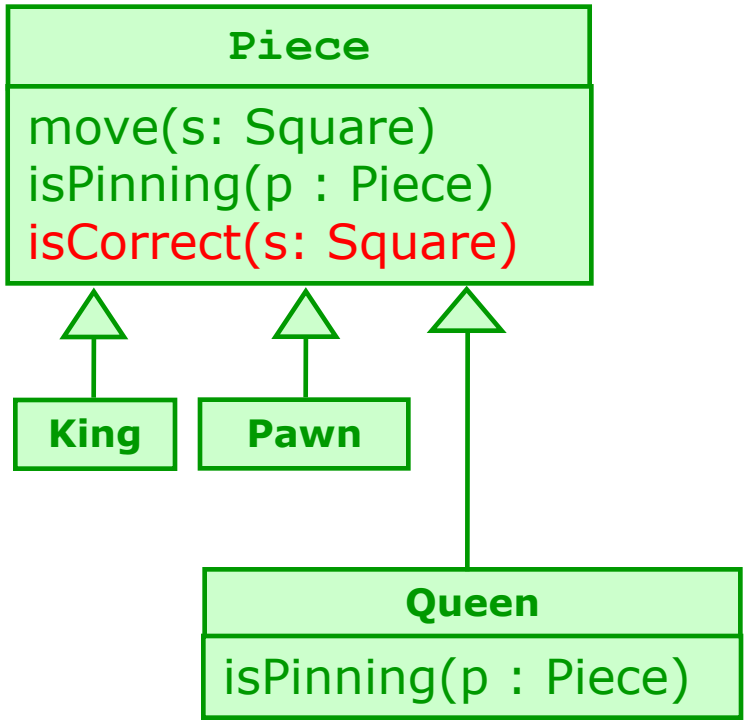
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Рефакторинг в Eclipse. Перемещение метода (1)

Романов Владимир Юрьевич ©2025

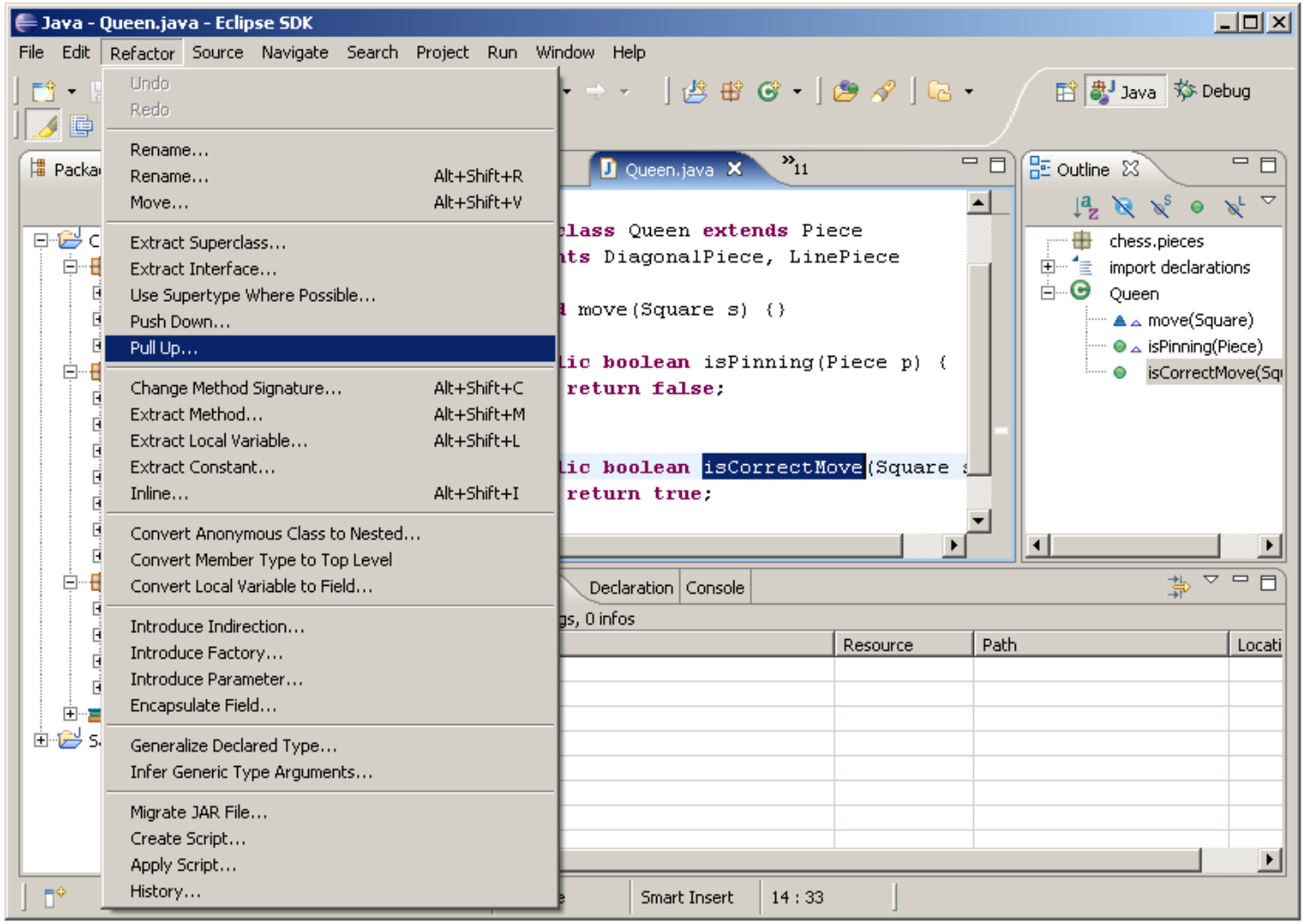




Рефакторинг в Eclipse. Перемещение метода (2)

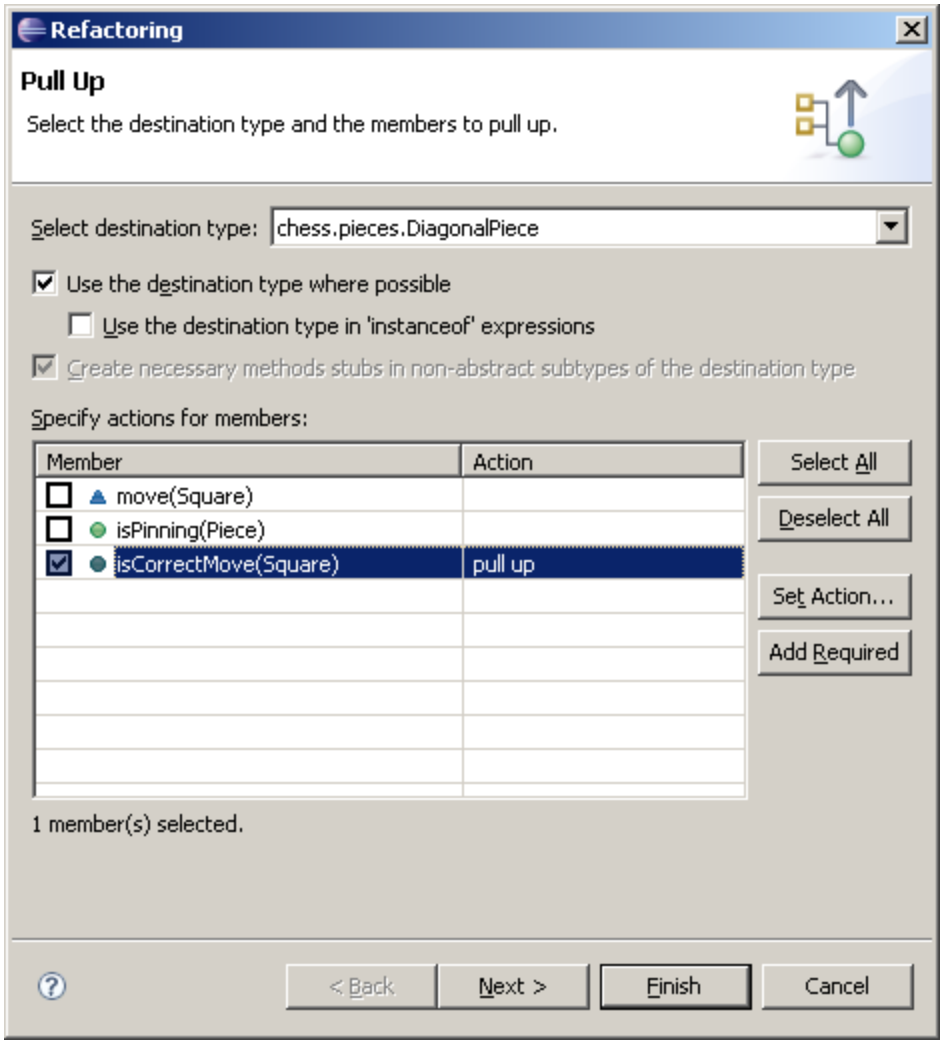
МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025



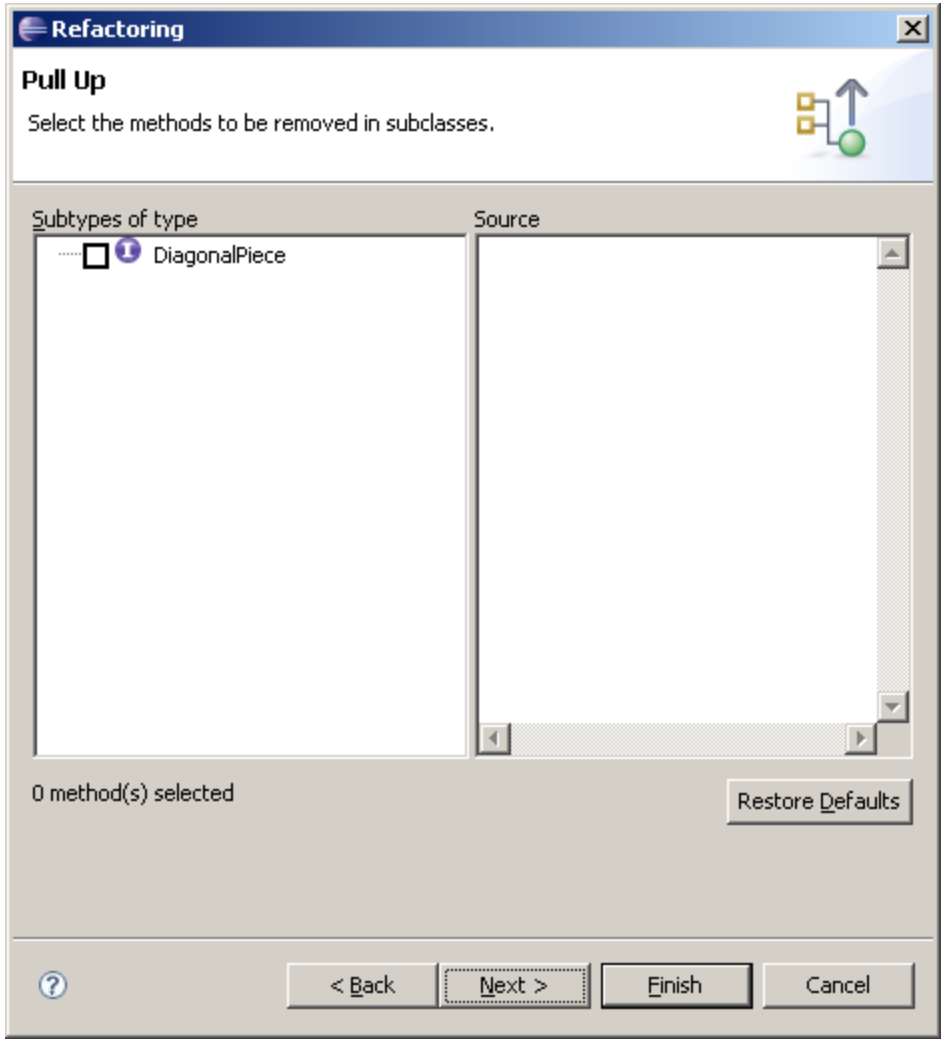
Рефакторинг в Eclipse. Перемещение метода (3)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



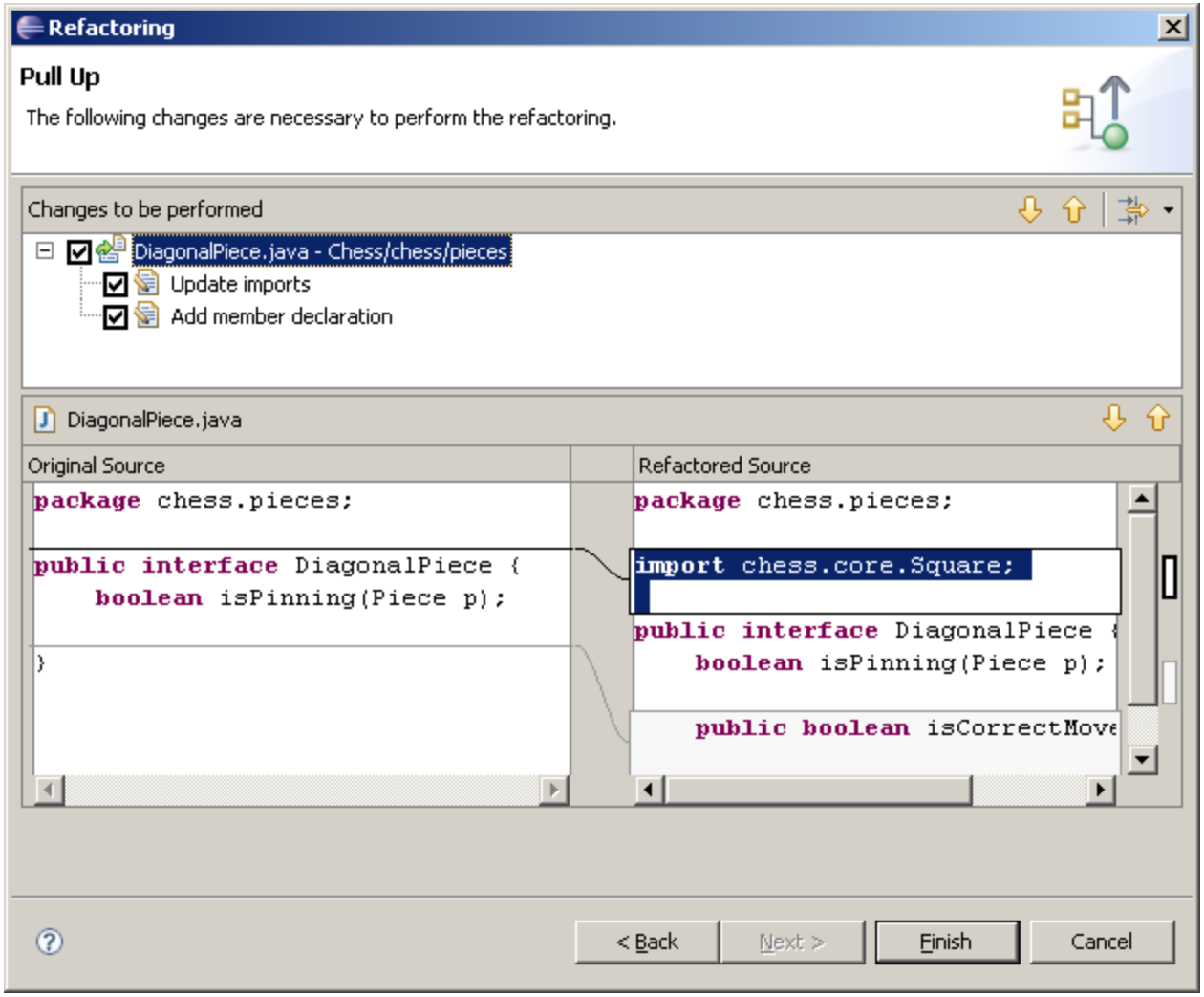
Рефакторинг в Eclipse. Перемещение метода (4)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Рефакторинг в Eclipse. Перемещение метода (5)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Рефакторинг в Eclipse. Использование полиморфизма (1)

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

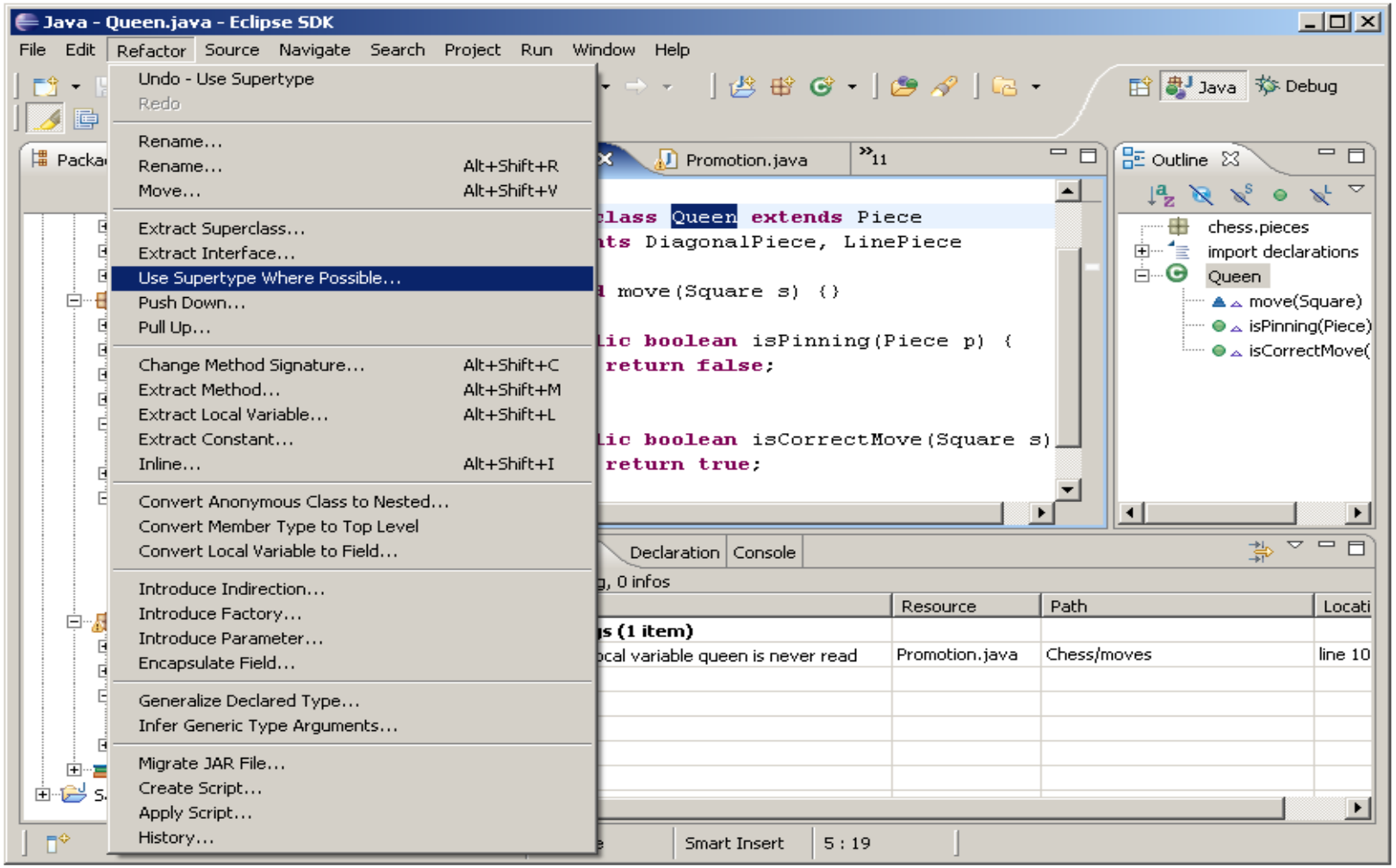
- Замена имени класса-потомка на имя класса-предка

- После этого программа работает со всеми потомками класса-предка

Рефакторинг в Eclipse. Использование полиморфизма (2)

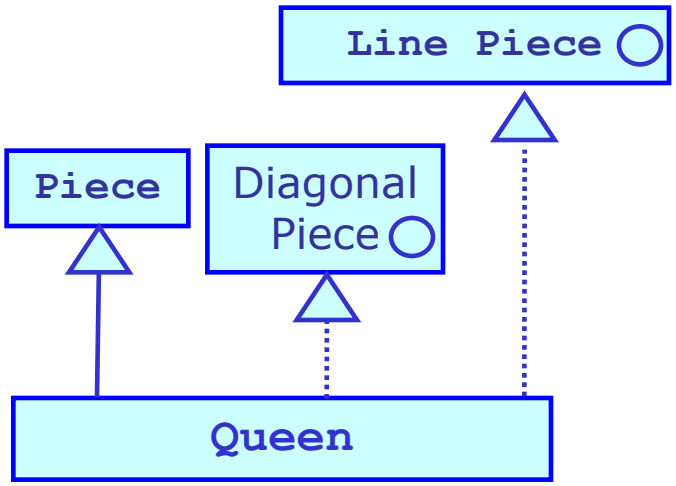
МГУ им. М.В.Ломоносова. Факультет ВМК.

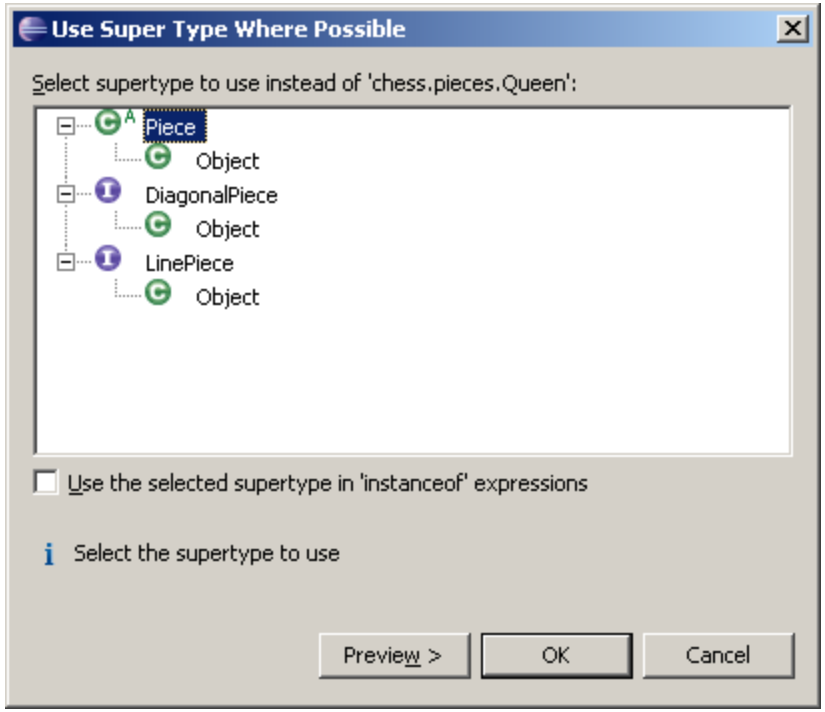
Романов Владимир Юрьевич ©2025



Рефакторинг в Eclipse. Использование полиморфизма (3)

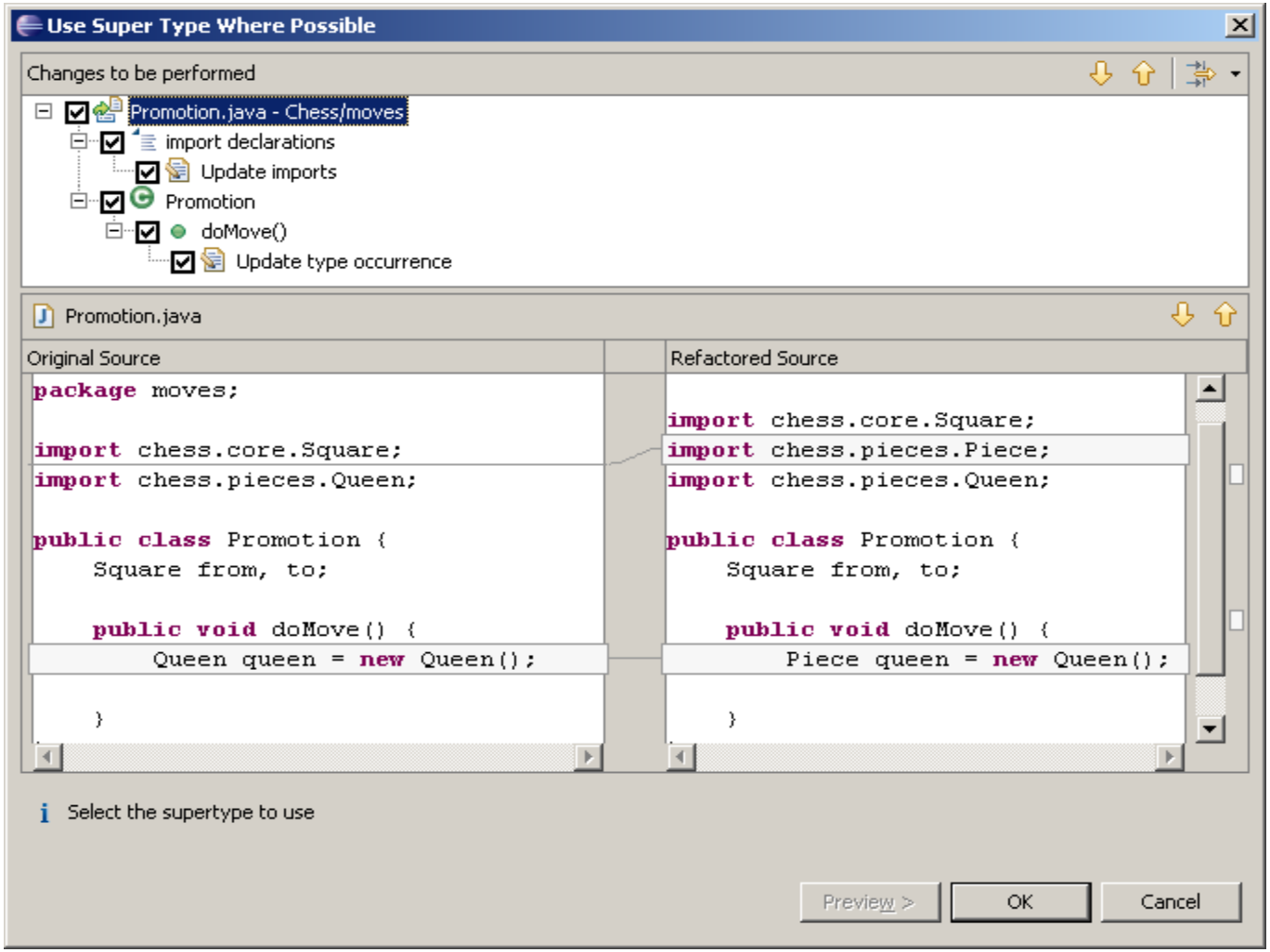
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025





Рефакторинг в Eclipse. Использование полиморфизма (4)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

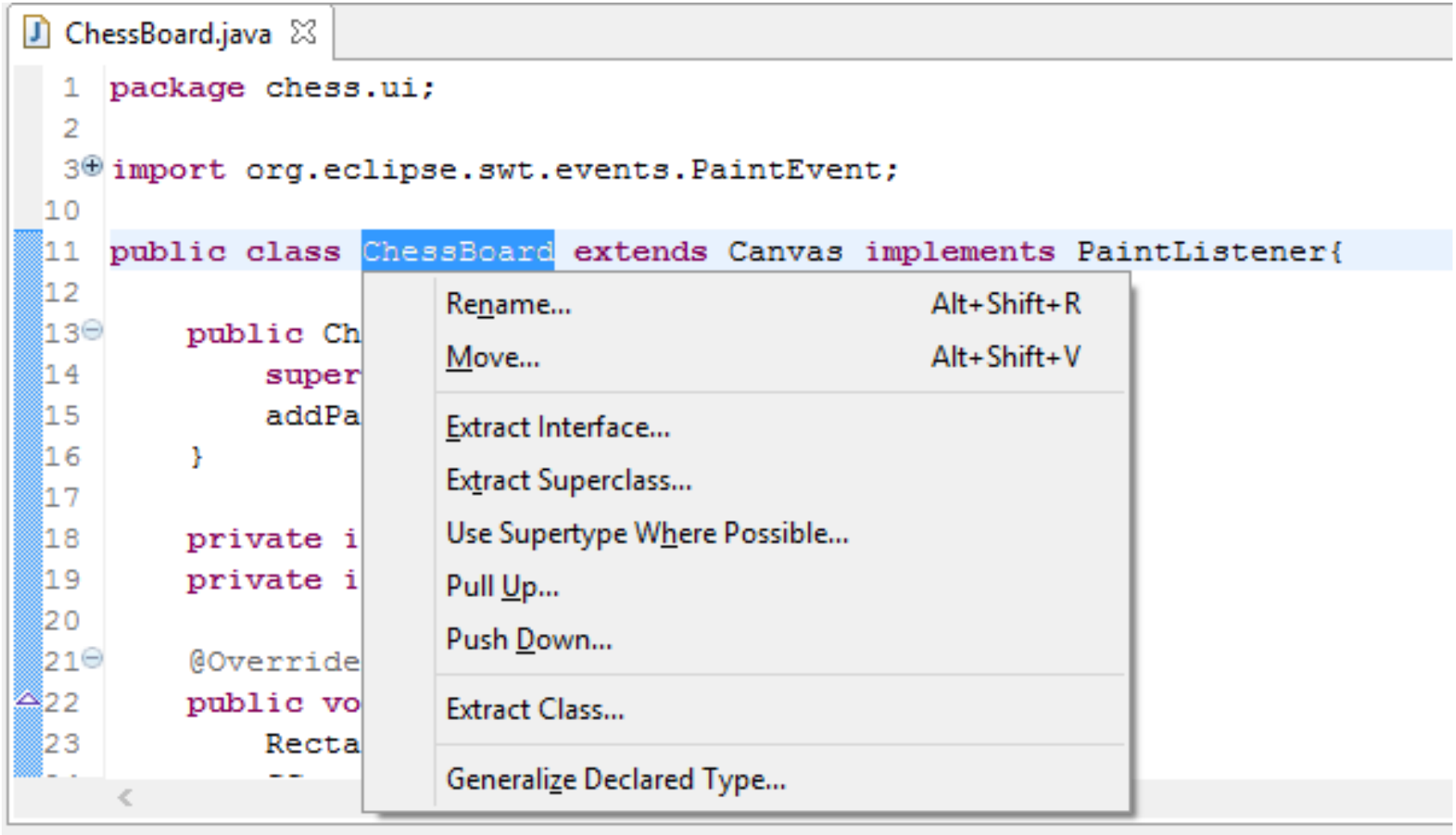


Контексто-зависимый рефакторинг. Клавиши **Alt + Shift + T**

МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025

Преобразования применимые к классу ChessBoard

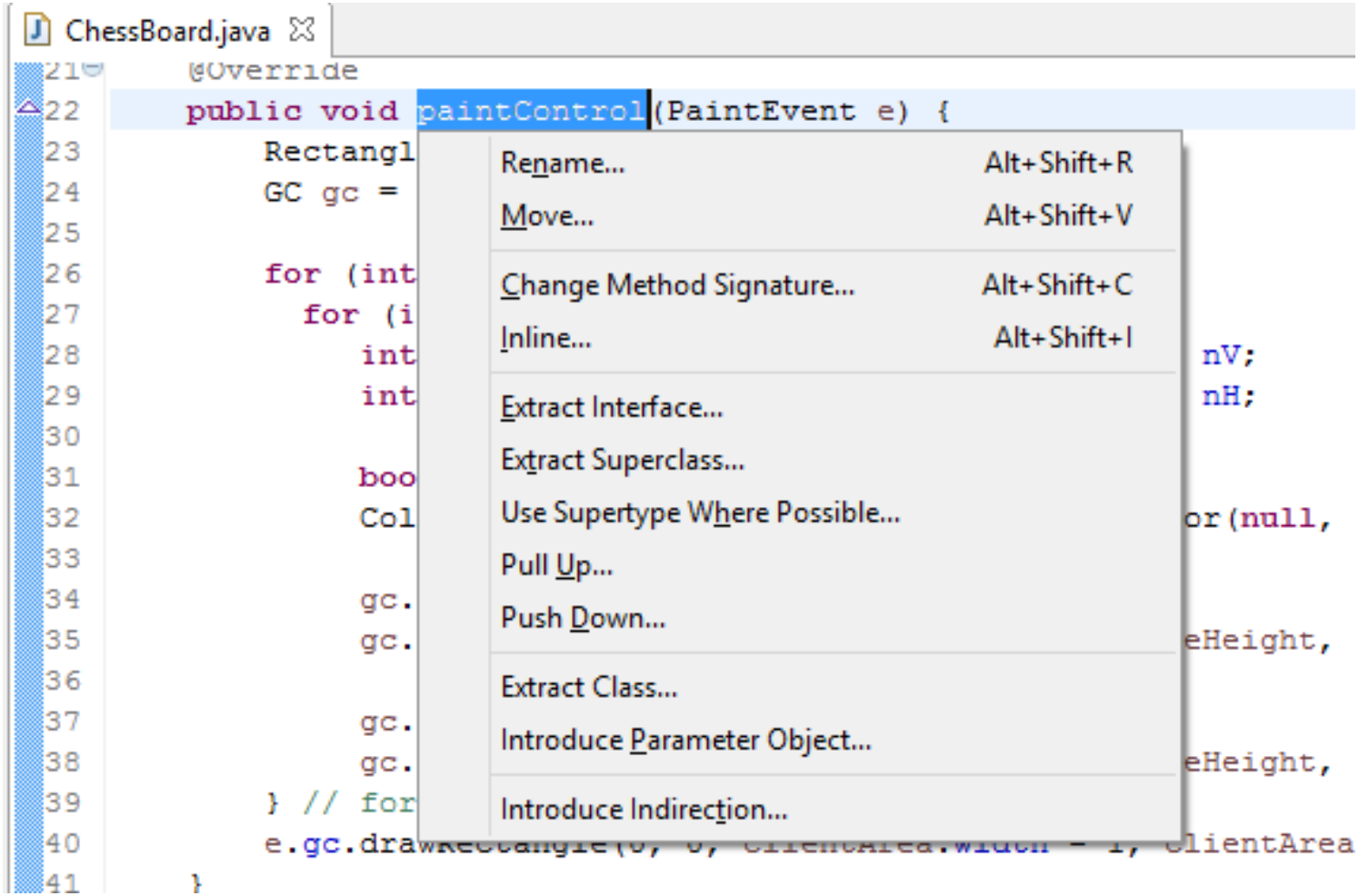


Контексто-зависимый рефакторинг. Клавиши **Alt + Shift + T**

МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025

Преобразования применимые к методу paintControl

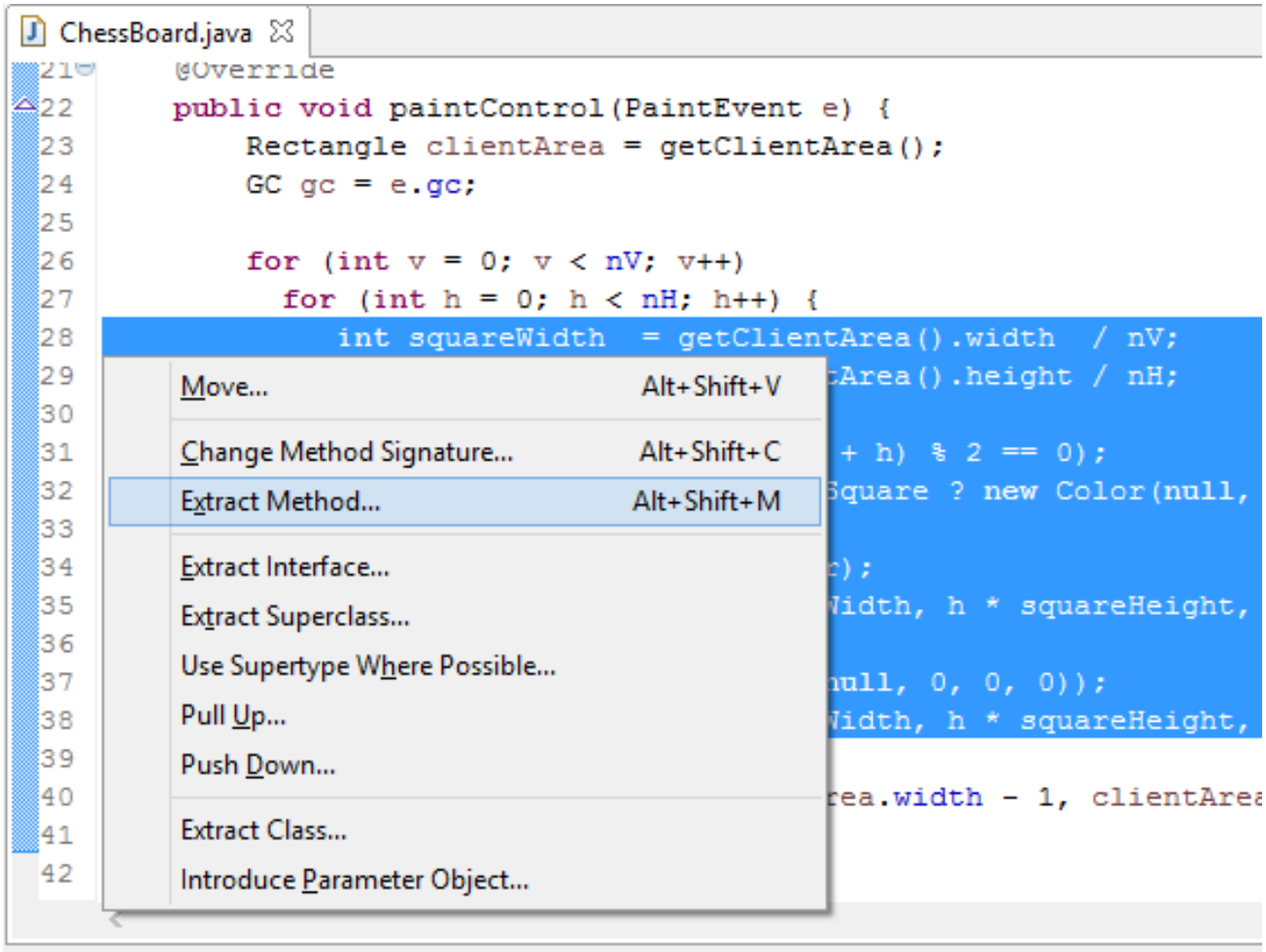


Контексто-зависимый рефакторинг. Клавиши **Alt + Shift + T**

МГУ им. М.В.Ломоносова. Факультет ВМК.

Романов Владимир Юрьевич ©2025

Преобразования применимые к выделенному тексту



Романов Владимир Юрьевич ©2025

Исключения и утверждения в языке Java

Исключения в языке Java. Что изучается

- Как исключения используются для сигнализации об ошибках
- Как использовать конструкции ***try*** и ***catch*** для обработки исключений
- Как порождать исключения
- Как использовать предложение ***assert***

Исключения в языке Java. Определение

- Исключение
  - Событие или условие которое нарушает нормальных ход выполнения программы
  - Условие, которое приводит к *порождению* (***throw***) исключения системой
  - Поток управления прерывается и обработчик исключения будет *перехватывать* (***catch***) исключение
- Обработка исключения объектно-ориентированная
  - Локализует в объекте стандартные условия выполнения программы
  - Предоставляет простой способ сделать программу более надежной
  - Позволяет разделить нормальный и ненормальный ход выполнения программы

Исключения в языке Java. Источники исключений

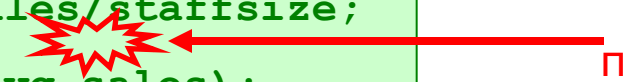
МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Источник исключений виртуальная машина Java
  - Не может быть загружен класс
  - Используется нулевая (***null***) ссылка на объект

- Ситуацию может породить и код, который пишет программист, и класс, который программист использует
  - **IOError**
  - Деление на ноль
  - Проверка корректности данных
  - Исключение обусловленное логикой работы программы

- Если исключение не перехвачено, оно завершает работу программы

```
float sales = getSales();
int staffsize = getStaff().size;
float avg_sales = sales/staffsize;
System.out.println(avg_sales);
```



Деление на 0 порождает ситуацию

Исключения в языке Java. Иерархия исключений

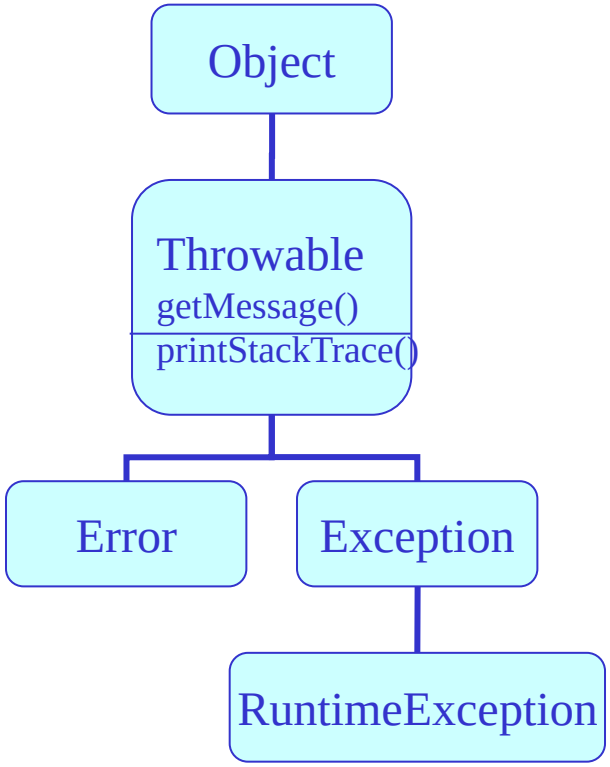
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- ***Throwable*** – базовый класс, предоставляющий общий интерфейс и реализацию большинства исключений

- ***Error*** – отмечает серьезные проблемы, которые не могут быть перехвачены

- ***Exception*** – описывает класс условий, которые должны быть перехвачены или описаны как порождаемые

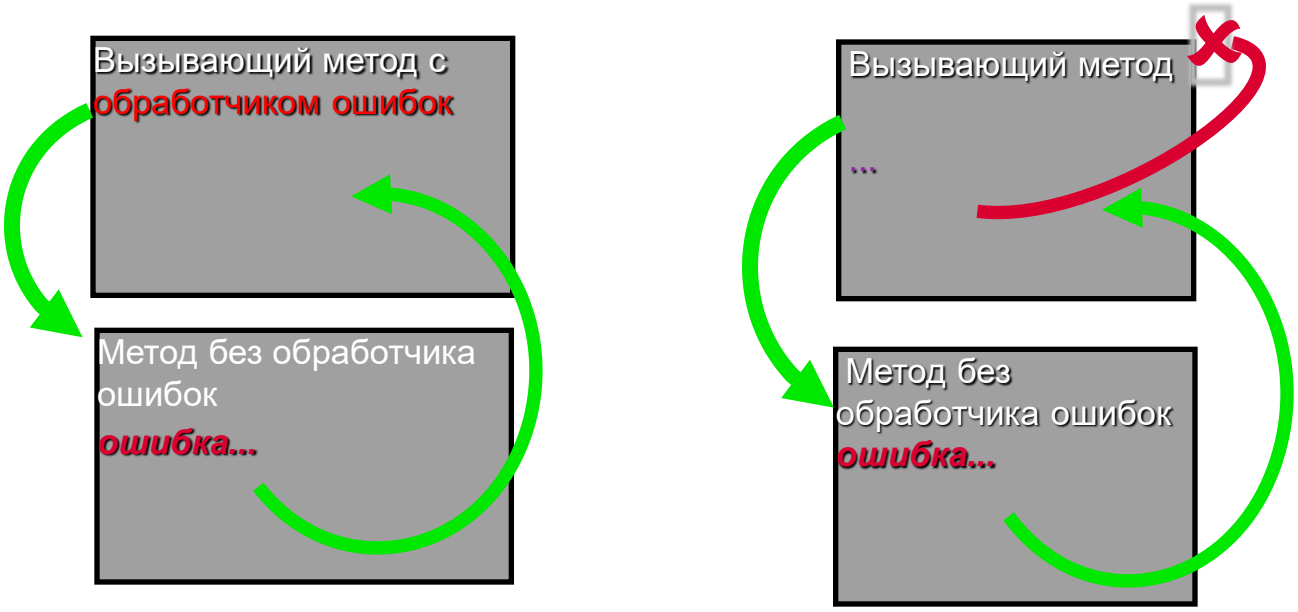
- ***RuntimeException*** – это исключение может быть порождено при нормальном выполнении виртуальной машины
  - **ArithmeticException**
  - **BufferOverflowException**



Исключения в языке Java. Обработка исключений

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Порожденные исключения обрабатываются либо в методе, в котором они были порождены, либо делегируются в вызывающий метод



Исключения в языке Java. Конструкции обработки исключений

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- ***throws*** – фрагмент объявления метода, содержащий список исключений который могут быть делегированы вверх по стеку вызовов
  - public int doIt() **throws** SomeException, …

- ***try*** – представляет блок кода с присоединенными обработчиками ошибок. Ошибки в try-блоке будут обработаны обработчиками ошибок

- ***catch*** – блок кода для обработки конкретного исключения

- ***finally*** – необязательный блок который следует после блоков catch. Выполняется всегда независимо от того, какое исключение порождено и было ли оно порождено

- ***throw*** – явно порождает исключение
  - **throw** new SomeException();

Исключения в языке Java. Блоки *try* и *catch*

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Используются для обработки исключений
- Код, который может породить ошибку, заключается в блок ***try***
- Сразу за блоком ***try*** должен идти блок ***catch***

```
try {
  // код выполняющий чтение из файла
}
catch (IOException ioe){
  // код обрабатывающий ошибки ввода/вывода
}
```

Исключения в языке Java. Блок *catch*

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Блок ***catch*** всегда содержит один аргумент определяющий тип перехватываемого исключения

- Аргументом может быть ссылка на объект класса ***Throwable*** или подкласс этого класса

- За одним блоком ***try*** может идти несколько блоков ***catch***

```
try {
  // код выполняющий чтение из файла
}
catch (FileNotFound fnf) {
  // код обрабатывающий ошибку файл не найден
}
catch (IOException ioe){
  System.out.println(”I/O error " + e.getMessage() );
}
```

Исключения в языке Java. Блок *finally*

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Блок ***finally*** необязателен. Выполняет действия независимо от того, было порождено исключение или нет
- Могут быть блоки ***try*** и ***finally*** без блоков ***catch***
- Блок ***catch*** выполняется после любого блока ***catch***, даже после того, который содержит предложение ***return***

```
try {
  // код выполняющий чтение из файла
}
catch (FileNotFound fnf) {
  // код обрабатывающий ошибку файл не найден
}
finally{
  // закрытие файла
}
```

Исключения в языке Java. Предложение *throw*

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Предложение может быть использовано в блоке ***try*** при необходимости порождения исключения

- Предложение ***throw*** требует только одного аргумента – объекта класса, являющегося потомком класса ***Throwable***

- Для того, что бы инкапсулировать условие, создается новый экземпляр класса ***Throwable***

- Поток управления завершается немедленно после предложения ***throw***

```
throw new java.io.IOException("msg");
```

Исключения в языке Java. Предложение *throw*

- Перехват исключения в вызывающем методе

```
public class ThrowDemo {

    public void demoThrowMethod(int n) throws Exception {
        if (n < 0)
         throw new Exception("n < 0");
    }

    public static void main(String[] args) {
      ThrowDemo app = new ThrowDemo();
      try {
            app.demoThrowMethod(-1);
      } catch (Exception e) {
        System.err.println(e.getMessage());
      }
    }
}
```

Исключения в языке Java. Предложение *throw*

- Перехват исключения в порождающем ситуацию методе

```
public class ThrowDemo {

    public void demoThrowMethod(int n) {
      try {
         if (n < 0)
          throw new Exception("n < 0");
      } catch (Exception e) {
        System.err.println(e.getMessage());
      }
    }

    public static void main(String[] args) {
      ThrowDemo app = new ThrowDemo();
       app.demoThrowMethod(-1);
    }
}
```

Исключения в языке Java. Предложение *throw*


```
public void doMove() throws GameOver {
    target.setPiece(piece);
    changeCapturedColor();

    Board board = target.getBoard();

    List<Square> empties = board.getEmptySquares();
    if (!empties.isEmpty()) return;

    int enemies = piece.getEnemies().size();
    int friends = piece.getFriends().size();

    if (enemies == friends)
        throw new GameOver(GameResult.DRAWN);
}
```

Утверждения в языке Java. Когда использовать утверждения

- Не рекомендуется использовать утверждения
  - Для проверки аргументов в публичных методах. Для этого требуются ***RuntimeException***, например ***IllegalArgumentException***
- Рекомендуется использовать утверждения для проверки
  - Внутренних инвариантов (значения, которые никогда не должны возникать). Например, вставить `default: assert false` в конец выбирающего предложения
  - Инварианты потоков управления. Например, вставить **assert false** в те части программы, которые никогда не должны быть достигнуты
  - Предусловия, постусловия и инварианты классов. Например, проверка аргументов скрытых методов

Утверждения в языке Java. Использование утверждений

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Две формы утверждений

```
assert <boolean expression> ;
assert <boolean expression> : <value expression> ;
```

- Если утверждение ложно
  - первая форма утверждения порождает исключение ***AssertionError*** *без сообщений*
  - вторая форма утверждения порождает исключение ***AssertionError*** с сообщением, определенным при вычислении второго выражения

- По умолчанию утверждения не работают (игнорируются). Для их включения необходимо в командной строке для компилятора **java** использовать ключ enableassertions

- Утверждения **включаются/выключаются** для **классов/пакетов**

Утверждения в языке Java. Использование утверждений

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Две формы утверждений

```
assert <boolean expression> ;
assert <boolean expression> : <value expression> ;
```

- Если утверждение ложно
  - первая форма утверждения порождает исключение ***AssertionError*** *без сообщений*
  - вторая форма утверждения порождает исключение ***AssertionError*** с сообщением, определенным при вычислении второго выражения

- По умолчанию утверждения не работают (игнорируются). Для их включения необходимо в командной строке java использовать ключ **enableassertions**

- Утверждения включаются/выключаются для классов/пакетов

Романов Владимир Юрьевич ©2025

Интерфейсы в языке Java

Интерфейсы в языке Java. Объявление интерфейса

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Интерфейс – именованный список объявлений методов
  - Методы только объявляются, а не реализуются
  - Интерфейс похож на абстрактный класс, но тем не менее отличается от него

- Объявляемые в Java типы либо классы, либо интерфейсы, либо перечисления

- Интерфейс можно трактовать как контракт – обязательство объектов реализовать некоторый набор услуг

```
package game.moves;

interface Move {
    void doMove();
    void undoMove();
}
```

Интерфейсы в языке Java. Иерархия интерфейсов (1)

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Между интерфейсами возможно отношение ***наследования***

- Интерфейс, расширяющий другой интерфейс, ***наследует*** все объявления методов интерфейса - предка

- Иерархия наследования интерфейсов независима от иерархии наследования классов

```
public interface IPutMove extends Move {
    /**
     * Вернуть клетку куда пошла фигура.
     */
    Square getTarget();
}
```

```
public interface ITransferMove extends Move {
        Square getSource();
        Square getTarget();
}
```

```
public interface ICaptureMove extends Move {
        /**
         * Вернуть клетки на которых стоят захваченные фигуры.
         */
        List<Square> getCaptured();
}
```

Интерфейсы в языке Java. Реализация интерфейсов классом

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
package chess.moves;

public class Promotion implements ITransferMove , ICaptureMove {
      void doMove() {
            //…
      }
      void undoMove() {
            // …
      }
      Square getSource() {
            // …
      }
      Square getTarget() {
            // …
      }
      List<Square> getCaptured() {
            // …
      }
}
```

Интерфейсы в языке Java. Реализация интерфейса

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Методы объявленные в интерфейсе реализуются в классе, поддерживающим данный интерфейс

Определение

```
public interface ITransferMove extends Move {
        Square getSource();
        Square getTarget();
}
```

```
public interface ICaptureMove extends Move {
        List<Square> getCaptured();
}
```

Определение

Реализация

```
package reversi.moves;

public class Capture implements ITransferMove ,
ICaptureMove {
        void doMove() { … }
        void undoMove() { … }
       Square getSource() { … }
       Square getTarget() { … }
       List<Square> getCaptured() { … }
}
```

Интерфейсы в языке Java. Синтаксис реализации интерфейса

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Объявление суперкласса всегда предшествует объявлению интерфейсов реализуемых классом:

```
public class Directory extends Secure
             implements File {
     ...
}
```

- Если класс реализует несколько интерфейсов, то имена этих интерфейсов перечисляются через запятую:

```
public class Directory
             implements File, ISecure {
     ...
}
```

Интерфейсы в языке Java. Типизация и интерфейсы

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Типом переменной и параметра может быть интерфейс
- Переменной и параметру может быть присвоен только объект, реализующий этот интерфейс
- Переменная и параметр могут быть использованы только для вызова методов, определенных в интерфейсе
- Имя интерфейса не может быть в выражении ***new***

```
Move m1 = new Move();       // Ошибка

Move m2 = new Castling();   // Допустимо
```

Интерфейсы в языке Java. Использование интерфейсов (2)

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Использование интерфейсов позволяет
  - Использовать полиморфизм независимо от иерархии классов
  - Осуществлять доступ к методам в отдельных независимых деревьях классов
  - Использовать в переменных и параметрах объекты не связанные иерархией наследования классов
- Классы, реализующие один и тот же интерфейс, понимают те же самые сообщения независим от положения в иерархии классов

Интерфейсы в языке Java.
Соглашения об именах интерфейсов

- Суффикс “able” в именах интерфейсов
  - Cloneable, Serializable
- Существительное + Impl
  - Bank, BankImpl
  - BankAccount, BankAccountImpl
- Префикс I перед существительным
  - Bank, IBank
  - BankAccount, IBankAccountImpl

Интерфейсы в языке Java. Использование интерфейсов (2)

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Программист может определить параметры метода как интерфейсы
  - Это ограничит использование этих параметров только типами, которые реализуют этот интерфейс
  - Более четкое указание программисту какие методы он может использовать
- Увеличивает повторное использование кода, использующего преимущественно типы - интерфейсы

Вложенные классы и интерфейсы

- Допускается вложенность интерфейсов в классы и другие интерфейсы

- Вложенность классов и интерфейсов – дополнительный способ структурирования программы. Вложенность возможна не только в пакеты.

- Вложенные интерфейсы могут иметь видимости как у полей и методов классов и интерфейсов
  - **public**
  - **protected**
  - **private**
  - **package**

- Вложенные классы имеют доступ к полям и методам охватывающих классов и интерфейсов.

Романов Владимир Юрьевич ©2025

Коллекции в языке JAVA

Коллекции языка Java. Что изучается

- Понимание основных понятий коллекций
- Оценка основных интерфейсов коллекций
  - интерфейсы
  - абстрактные типы
  - конкретные реализации
- Понять как "устаревшие" классы и интерфейсы связаны с новыми интерфейсами и классами

Коллекции языка Java. Что такое коллекция

- Коллекция – это объект который группирует несколько элементов в единую сущность

- Массивы языка Java также рассматриваются как коллекции

- Виды коллекций
  - *Set (множество)* – не может иметь повторяющиеся элементы. Например, книги в библиотеке
  - *List (список)* – упорядоченная коллекция, может содержать повторения. Например, список чисел
  - *Map (карта)* – объекты, которые отображают ключи на значения. Дублирование ключей недопустимо. Например, словарь, список свойств

Коллекции языка Java.
Java Collection Framework

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- JCF - унифицированная архитектура для представления и манипулирования коллекциями

- Состоит из трех частей
  - Интерфейсы
  - Реализации
  - Алгоритмы

Коллекции языка Java.
Интерфейсы, реализации, алгоритмы

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

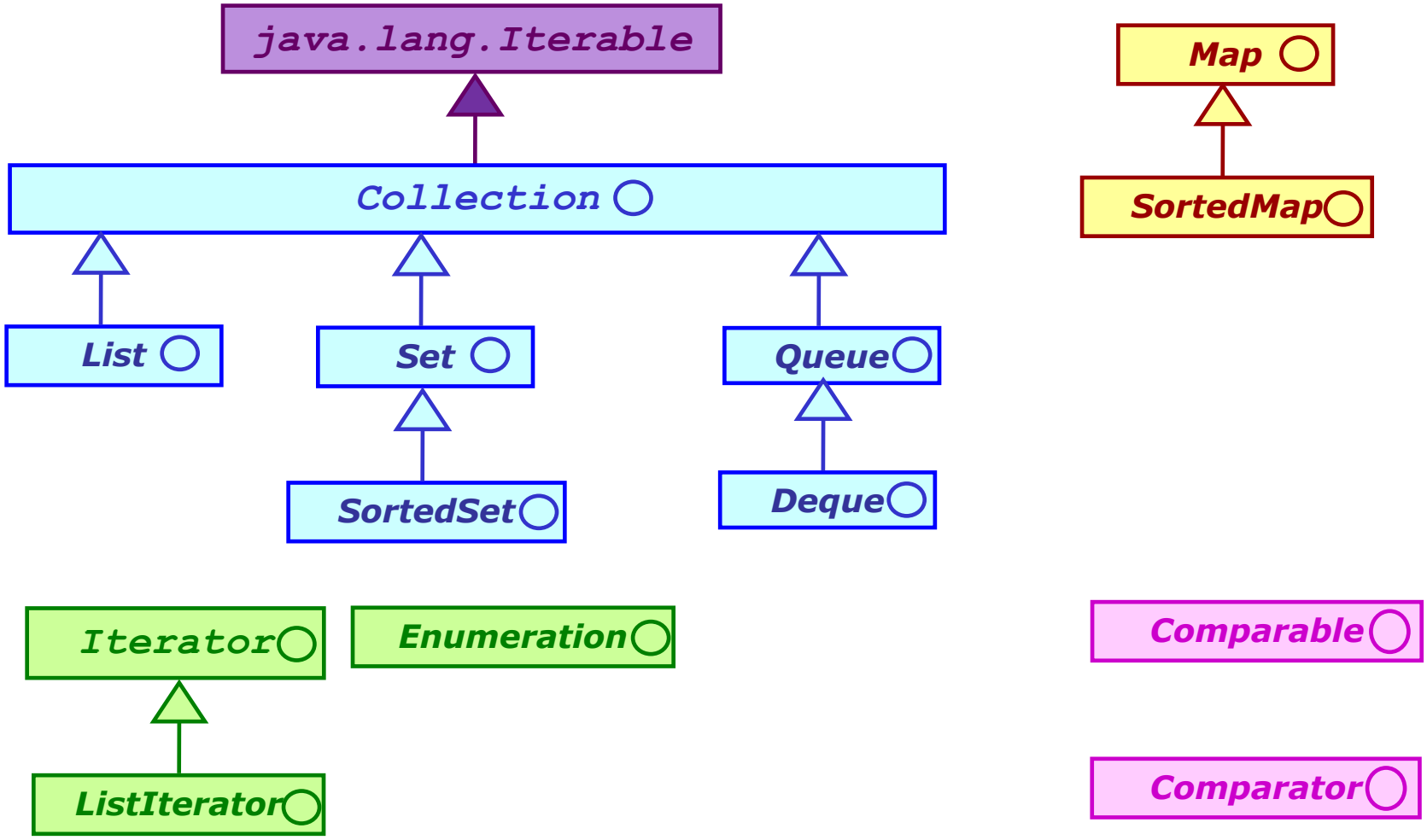
- **Интерфейсы** – абстрактные типы данных представляющие коллекции. Назначение коллекций:
  - позволить манипулирование коллекциями независимо от их деталей их представления
  - предоставить точки расширения для добавления новых типов коллекций и из реализаций
- **Реализации** – конкретная реализация интерфейса коллекции
- **Алгоритмы** – методы, выполняющие полезные вычисления над объектами, которые реализуют интерфейс коллекций. Например, поиск и сортировку.
  - Предоставляют повторно используемую функциональность посредством полиморфизма – один алгоритм для разных реализаций

Коллекции языка Java.
Достоинства использования JCF

- Уменьшает усилия по программированию
- Уменьшает усилия по изучению и использованию нового API
- Увеличивает скорость и надежность программы
- Допускает переносимость среди связанных API

Коллекции языка Java.
Интерфейсы в коллекциях

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

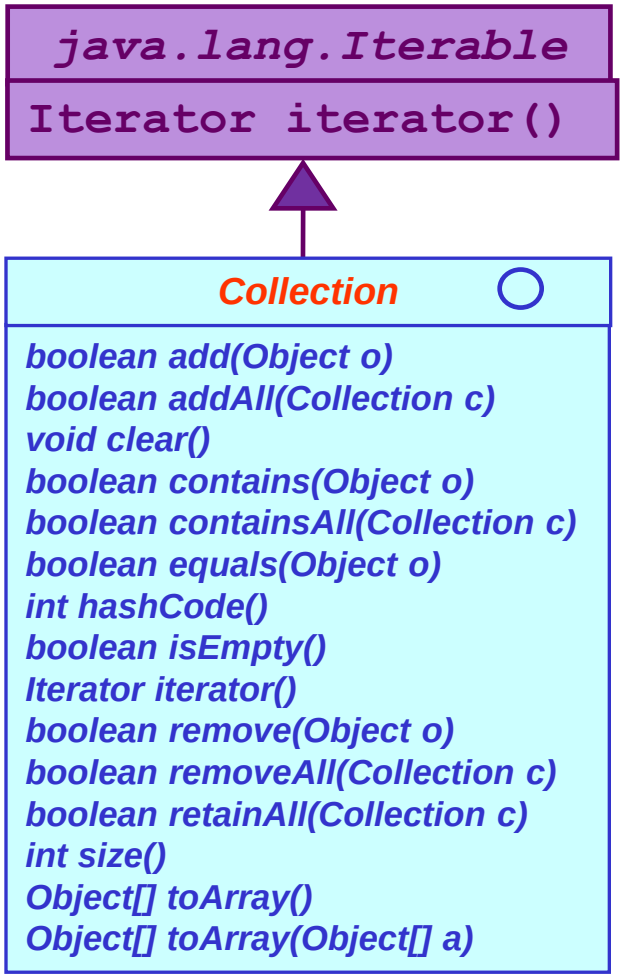


Коллекции языка Java.
Интерфейс *Collection*

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

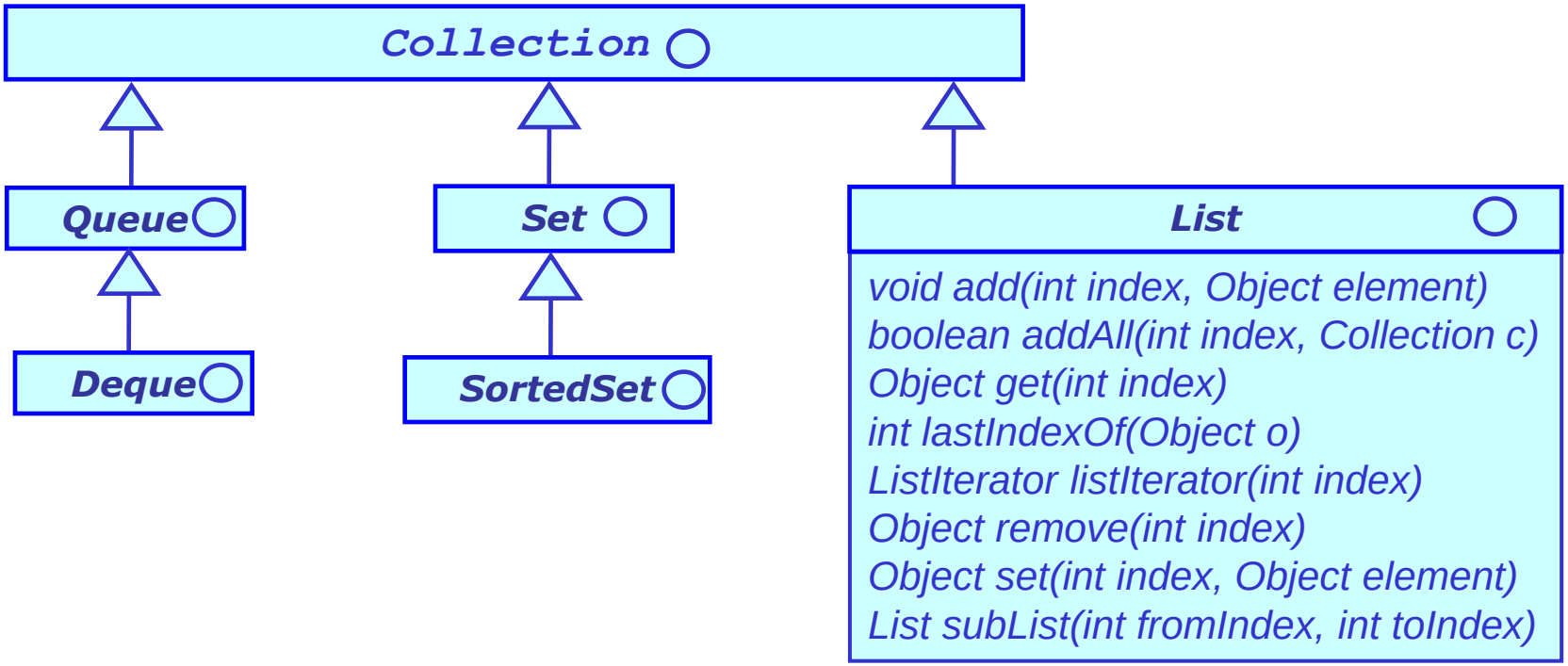
- Этот универсальный интерфейс для изменения коллекций и прохода по элементам коллекций
- Проверки принадлежности элемента к коллекции
- Добавления элемента к коллекции
- Удаления элемента из коллекции

```
Collection c;

for(Object o : c) {
    System.out.println(o);
}
```



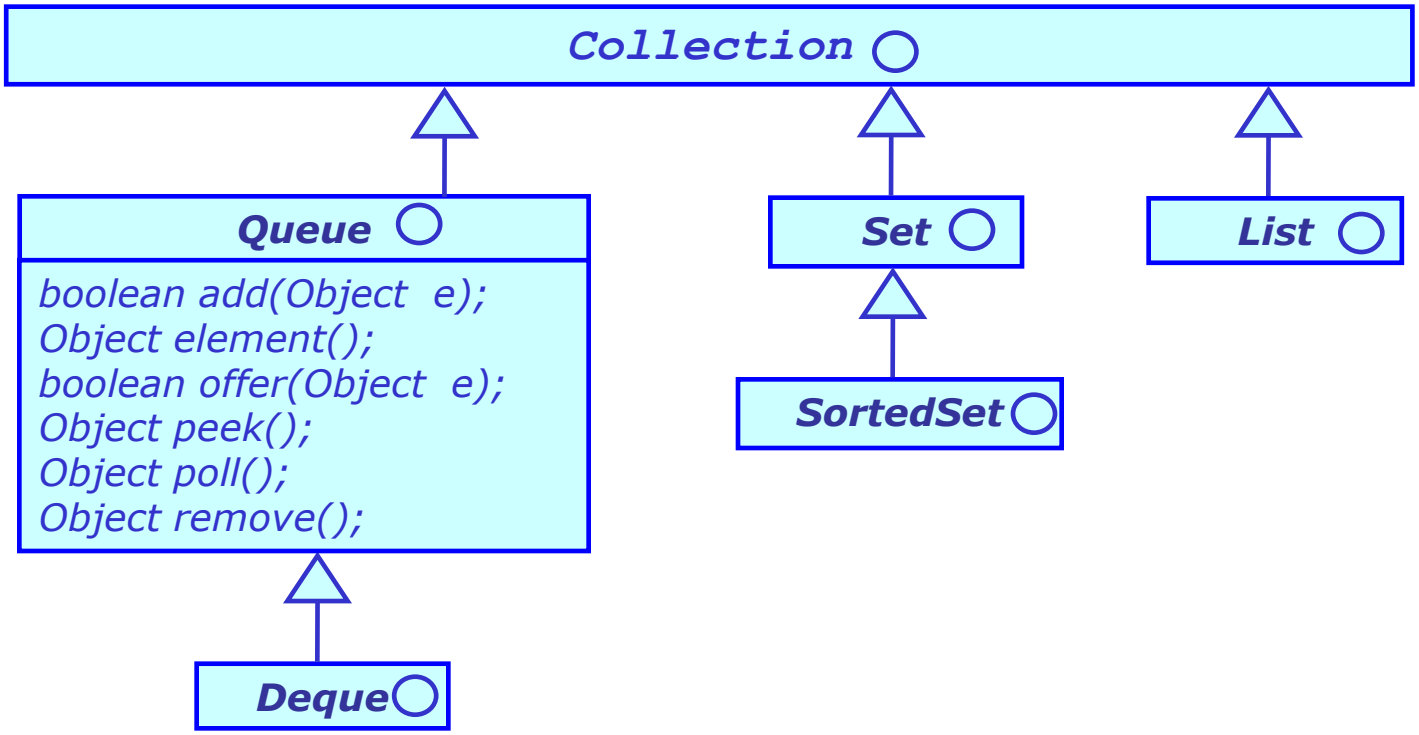
Коллекции языка Java. Интерфейс списка

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



Коллекции языка Java.
Интерфейс очереди Queue

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

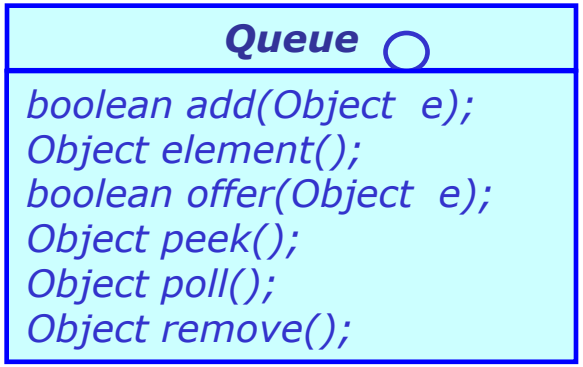


Коллекции языка Java.
Интерфейс очереди Queue

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

Действия по добавлению, удалению и проверки элемента в очереди.

Различие – что делается если действие выполнить невозможно



	Порождение ситуации	**Возвращается значение или null**
Вставка	*boolean add(Object e)*	*boolean offer(Object e)*
Удаление	*Object remove()*	*Object poll()*
Проверка	*Object element()*	*Object peek()*

Коллекции языка Java.
Интерфейс очереди Queue

boolean add(E e)
Вставляет указанный элемент в эту очередь, если это возможно сделать немедленно, не нарушая ограничения емкости, возвращая значение ***true*** в случае успеха и вызывая исключение ***IllegalStateException***, если в настоящее время нет свободного места.

E element()
Извлекает, но не удаляет заголовок этой очереди. Этот метод отличается от ***peek*** только тем, что выдает исключение, если эта очередь пуста.
Возвращает: заголовок этой очереди
Порождает ситуацию ***IllegalStateException*** если очередь пуста.

E remove()
Возвращает, удаляя, элемент в заголовке очереди.
Порождает ситуацию ***IllegalStateException*** если очередь пуста.

Коллекции языка Java.
Интерфейс очереди Queue

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

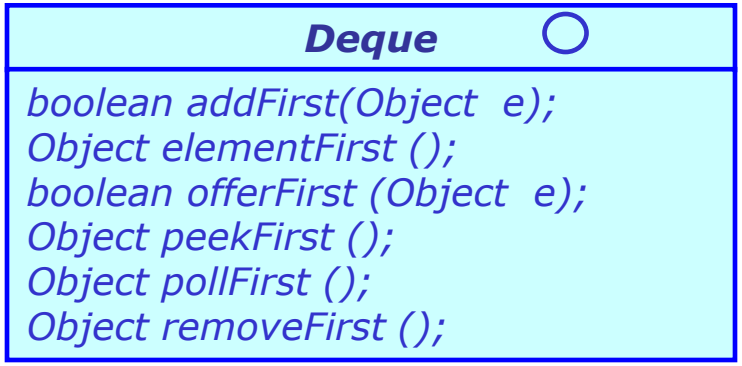
boolean offer(E e)
Вставляет описанный элемент в очередь, если возможно сделать это немедленно без нарушения ограничений по плотности.
Возвращает значение **true** если вставка прошла успешно и ***false*** иначе.

E peek()
Возвращает, не удаляя, элемент в заголовке очереди.
Возвращает ***null*** если очередь пуста.

E poll()
Возвращает, удаляя, элемент в заголовке очереди.
Возвращает ***null*** если очередь пуста.

Коллекции языка Java.
Интерфейс очереди Deque (1)

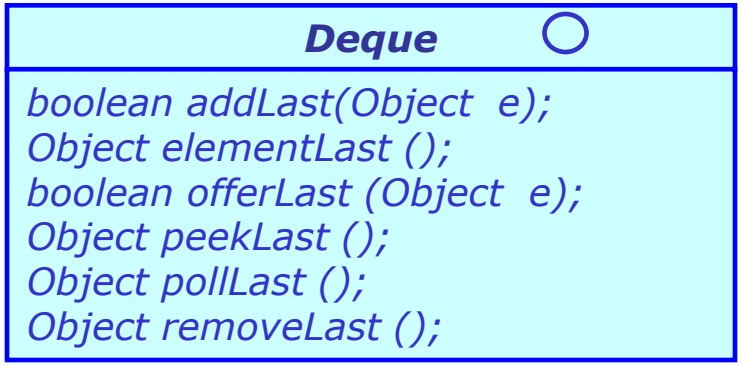
МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025



	Ситуация	Значение или null
Вставка	*boolean addFirst(Object e)*	*boolean offerFirst(Object e)*
Удаление	*Object removeFirst()*	*Object pollFirst()*
Проверка	*Object elementFirst()*	*Object peekFirst()*

Коллекции языка Java.
Интерфейс очереди Deque (2)

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

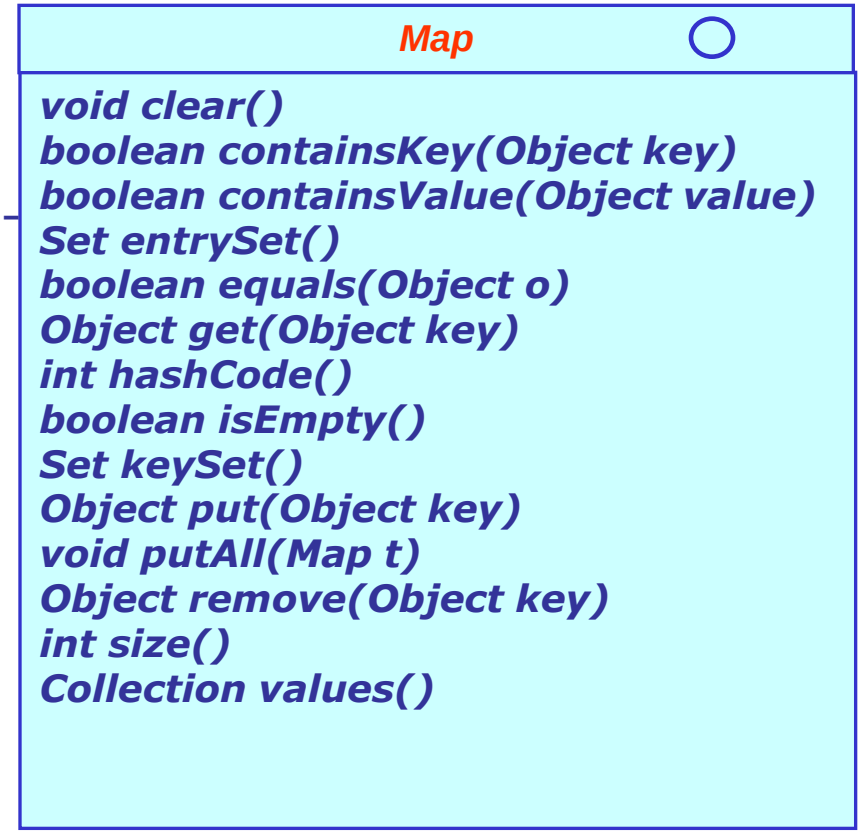


	Ситуация	Значение или null
Вставка	*boolean addLast(Object e)*	*boolean offerLast(Object e)*
Удаление	*Object removeLast()*	*Object pollLast()*
Проверка	*Object elementLast()*	*Object peekLast()*

Коллекции языка Java.
Интерфейс *Map*

Романов Владимир Юрьевич ©2025

- Карта отображает ключи на значения
- Добавление/удаление пары ключ-значение
- Взять значение для данного ключа
- Проверить наличие элемента в карте
- Можно рассматривать карту как:
  - Множество ключей
  - Множество пар ключ-значение
  - Коллекцию значений



Сравнение объектов. Упорядочивание объектов

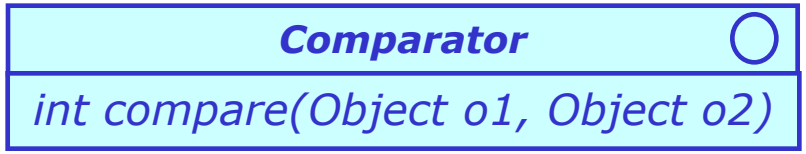
- Для сортировки объектов в коллекциях, необходим способ упорядочивания объектов
  - Объект A идет перед объектом B
  - Объект B идет перед объектом A
  - Объект A и объект B равны
- Существует два способа упорядочения объектов при сортировке:
  - Интерфейс Comparable
  - Интерфейс Comparator

Сравнение объектов.
Интерфейсы *Comparable* и *Comparator*

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Интерфейс *Comparable*
  - Реализуют классы, объекты которых способны сравнить себя с другими объектами сами
  - Такие классы называются классами с естественным упорядочиванием

- Интерфейс *Comparator*
  - Реализует класс, назначение которого сравнивать один объект с другим
  - Два объекта могут сравниваться по разному

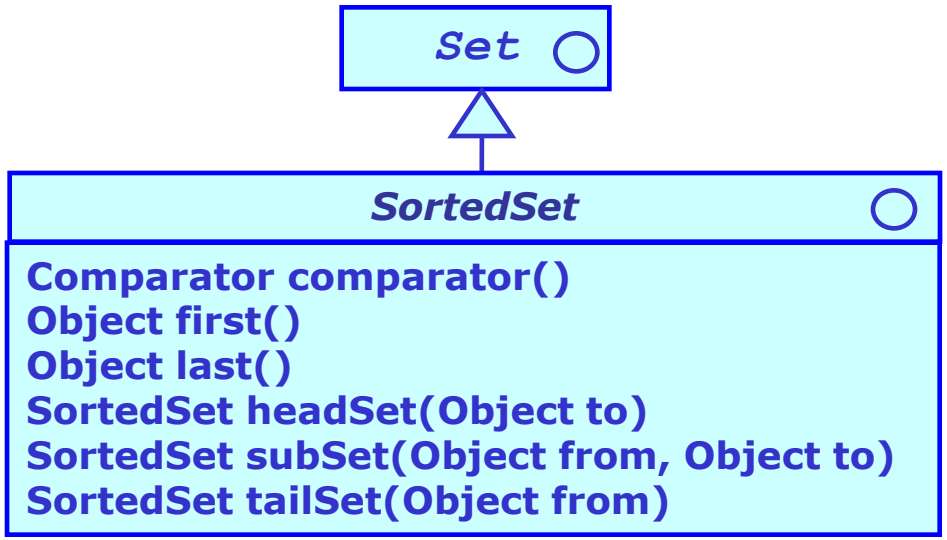


Сортированные коллекции.

- *SortedSet (Сортированное Множество)* – это множество (*Set*) с встроенным и автоматически поддерживаемым упорядочением
  - Существуют методы для использования этого порядка

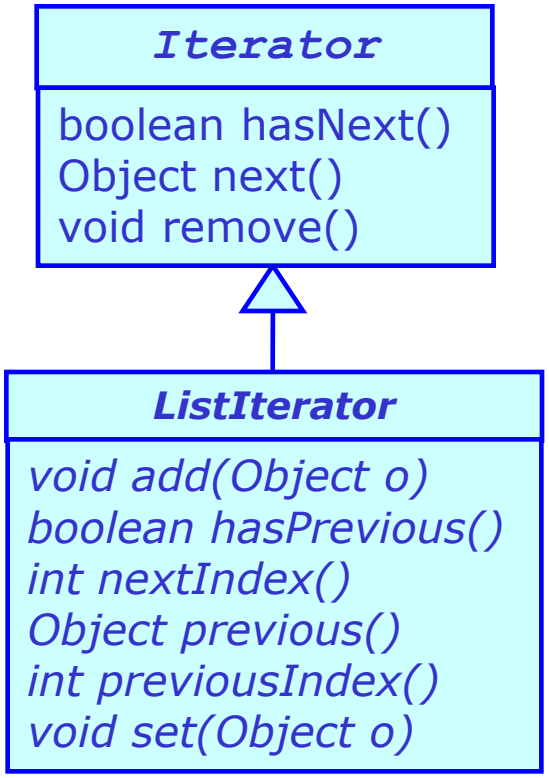
- *SortedMap (Сортированная Карта)* – это карта (*Map*) имеет аналогичные свойства, основанные на упорядочении ключей карты

- Порядок может определяться как с помощью естественного порядка, так и с помощью класса реализующего интерфейс Comparator



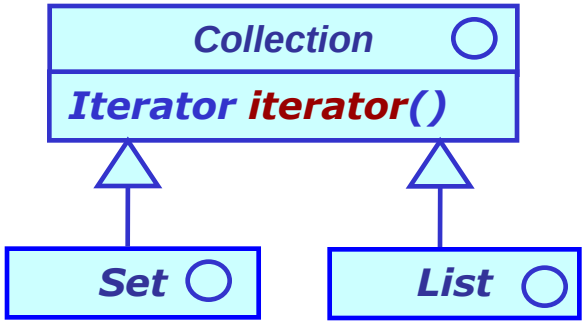
Итераторы.

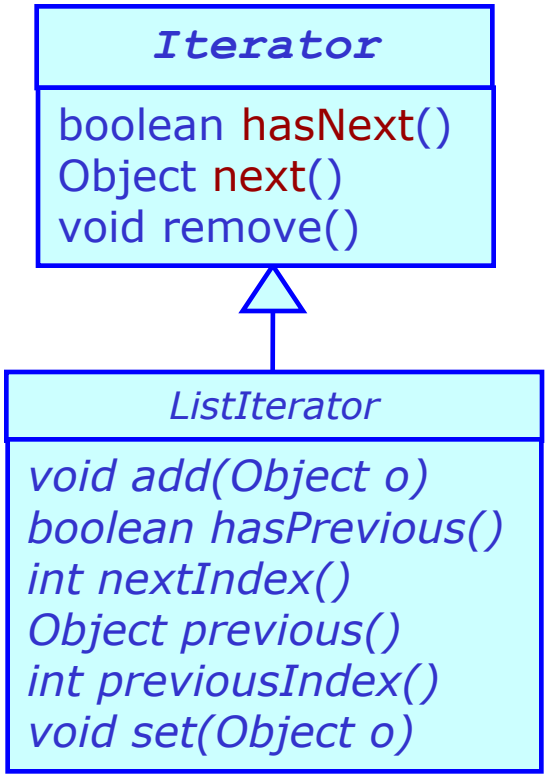
- Итератор предоставляет удобный способ перебора элементов коллекции
- *ListIterator* – добавляет методы представляющие последовательность элементов этой коллекции
- Методы *add* и *remove* позволяют изменять коллекцию во время ее прохода
- Итераторы для сортированных коллекций учитывают порядок заданный при сортировке



Шаблон кода итератора.

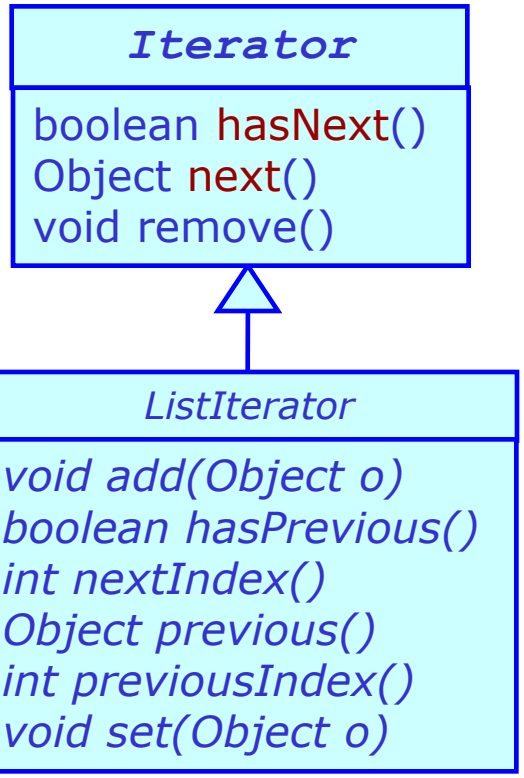

```
Collection c;

Iterator i = c.iterator();
while (i.hasNext()) {
   Object o = i.next();
   System.out.println(o);
}
```

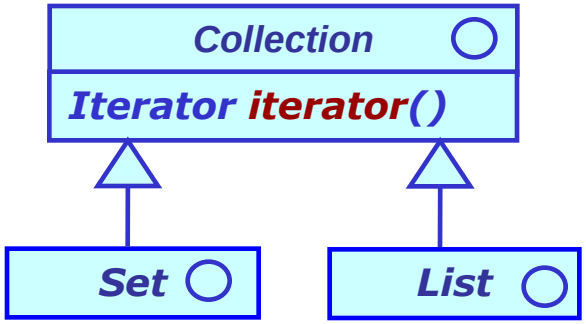




Итератор встроенный в язык Java.


```
Collection c;
//…
for(Object o : c) {
   // process this object
}
```





Интерфейсы и реализации.

		Реализации				
		Хэш таблица	Массив	Дерево	Связанный список	Устаревшие
Интерфейсы	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties
	Queue		ArrayDeque		LinkedList	
	Deque		ArrayDeque		LinkedList	

Рекомендация: при кодировании использовать интерфейсы, а не реализацию

Пример. Использование карты. Частота слов в командной строке


```
public class MapExample {
  public static void main(String args[]) {
      Map map = new HashMap();
      Integer ONE = new Integer(1);

      for (String key : args) {
          Integer frequency = (Integer) map.get(key);
          if (frequency == null) {
                frequency = ONE;
          } else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
          }
          map.put(key, frequency);
      }
      System.out.println(map);
      Map sortedMap = new TreeMap(map);
      System.out.println(sortedMap);
  }
}
```

Пример. Использование карты в Java8. Частота слов в командной строке


```
public class MapExample {
public static void main(String args[]) {
      Map<String, Integer> map = new HashMap<>();

      for (String key : args) {
            int frequency = !map.containsKey(key) ? 1 : map.get(key) + 1;
            map.put(key, frequency);
      }

      System.out.format("HashMap: %s %n%n", map);

      Map<String, Integer> sortedMap = new TreeMap<>(map);
      System.out.format("TreeMap: %s %n%n", sortedMap);
 // …
```

Пример. Использование карты в Java8. Частота слов в командной строке


```
        System.out.format("Keys:     %s %n%n",  sortedMap.keySet());
        System.out.format("Values:   %s %n%n",  sortedMap.values());
        System.out.format("EntrySet: %s %n%n", sortedMap.entrySet());

        for (Entry<String, Integer> entry : sortedMap.entrySet()) {
            System.out.format("Город: %10s Частота=%d %n",
                              entry.getKey(),
                              entry.getValue());
        }
}
}
```

Пример. Использование карты. Частота слов в командной строке

Ввод

```
Москва Тула Москва Петербург Москва Тула
```

Вывод

```
HashMap:   {Петербург=1, Тула=2, Москва=3}

TreeMap:   {Москва=3, Петербург=1, Тула=2}

Keys:      [Москва, Петербург, Тула]

Values:    [3, 1, 2]

EntrySet:  [Москва=3, Петербург=1, Тула=2]

Город:      Москва Частота=3
Город:  Петербург Частота=1
Город:       Тула Частота=2
```

Сравнение реализаций. Множества и карты

- **Set / Map**
  - HashSet / HashMap
    - Очень быстрая, без упорядочения
    - Выбирается начальная плотность (*initial capacity)* и коэффициент загрузки (*load factor)* для улучшения представления
  - TreeSet / TreeMap
    - Хранит сбалансированное дерево, хорошо для сортированных вставок
    - Нет параметров настройки
  - HashTable
    - Синхронизирована
    - Рекомендуется использовать через интерфейс **Map**

Сравнение реализаций. Списки

- **List**
  - ArrayList
    - Очень быстрый
    - Можно использовать «native» метод **System.arraycopy**
  - LinkedList
    - Хорошо использовать для меняющихся коллекций или для вставки в начало списка (для очередей Queue и Deque)
  - Vector
    - Синхронизированный
    - Рекомендуется использовать через интерфейс **List**

Сравнение реализаций. Списки

- **Queue и Deque**
  - LinkedList
    - Хорошо использовать для меняющихся коллекций или для вставки в начало списка (для очередей Queue и Deque)
  - DeckArray
    - Очень быстрый
    - Если не надо уделять элементы из середины
    - Можно использовать «native» метод **System.arraycopy**

Сравнение реализаций. Устаревшие коллекции

- Классы устаревший коллекций по прежнему доступны, но их реализации были изменены.
  - java.util.Vector
    - Расширяемый индексируемый список
  - java.util.Stack
    - Расширяет вектор операциями *push* и *pop*
  - java.util.BitSet
    - Расширяемое множество «флагов» True/False
  - java.util.Dictionary
    - Этот класс заменен на java.util.Map
  - java.util.Hashtable
    - Эффективное хранение данных без сортировки
  - java.util.Properties
    - Хранит пары ключ-значение. Ключ есть имя свойства.

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparable**

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

Comparable

```
class Point implements Comparable<Point> {
    public int x, y;

    static public Point p0 = new Point(0, 0);

    public Point(int x, int y)
        { this.x = x; this.y = y; }

    public int compareTo(Point p) {
        double d = p0.distance(p) - p0.distance(this);
        return d == 0 ? 0 : (d > 0 ? -1 : 1) ;
    }

    private double distance(Point p) {
        int dx = x - p.x; int dy = y - p.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparable**

Comparable

```
public class ComparableExample {
    public static void main(String args[]) {
        SortedSet<Point> points = new TreeSet<>();

        points.add( new Point(10, 9) );
        points.add( new Point(  2, 2) );
        points.add( new Point(  1, 3) );

        for (Point p : points)
            System.out.format("(%2s, %2s) %n", p.x, p.y);
    }
}
```

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparable**

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
( 2,  2)
( 1,  3)
(10,  9)
```

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparator**

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

Comparator

```
class Person {
        public String name;
        public int age;
        public Point location;

        public Person(String name, int age, Point location) {
                this.name = name;
                this.age = age;
                this.location = location;
        }

        public String toString() {
                return String.format("name=%7s age=%7s, location=(%2s, %2s)%n",
                                                name, age, location.x, location.y);
        }
}
```

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparator**

Comparator

```
public class ComparatorExample {
    public static void main(String args[]) {
        List<Person> persons = new ArrayList<>();
        persons.add( new Person("Bill",   19, new Point(40, 50) ) );
        persons.add( new Person("Alex",    9, new Point(60, 40) ) );
        persons.add( new Person("John",   3, new Point(10, 90) ) );

        persons.sort ( new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        });

        System.out.println();
        for (Person p : persons)
            System.out.print("by name: " + p);
```

Коллекции языка Java.
Интерфейсы в коллекциях. **Comparator**

Comparator

```
persons.sort( new Comparator<Person>() {
        @Override
        public int compare(Person p1, Person p2) {
                int d = p1.age - p2.age;
                return d == 0 ? 0 : (d > 0 ? 1 : -1);
        }
});

System.out.println();
for (Person p : persons)
        System.out.print("by age: " + p);
```

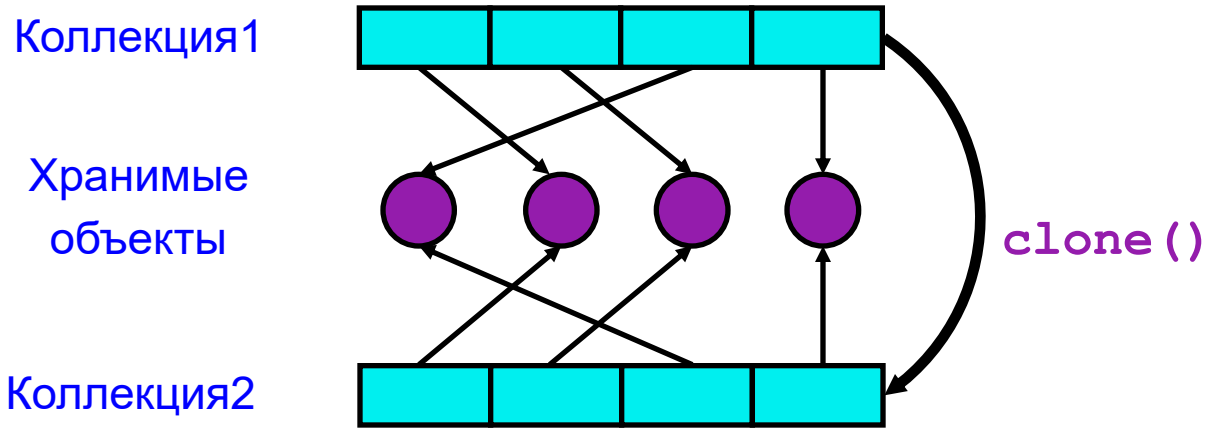
Коллекции языка Java.
Интерфейсы в коллекциях. **Comparator**

Comparator

```
persons.sort( new Comparator<Person>() {
        @Override
        public int compare(Person p1, Person p2) {
                return p1.location.compareTo(p2.location);
        }
});

System.out.println();
for (Person p : persons)
        System.out.print("by location: " + p);
```

Клонирование коллекций

- Для большинства коллекций можно сделать копию, не создавая копию для хранимых объектов («неглубокое» копирование)



Алгоритмы. Класс *Collections*

- *java.util.Collections* содержит только статические методы для работы с коллекциями или для создания коллекций. Он содержит:
  - Полиморфные алгоритмы для работы с коллекциями, например:
    - **binarySearch**
    - **copy**
    - **min и max**
    - **replace**
    - **reverse**
    - **rotate**
    - **shuffle**
    - **sort**
    - **swap**
  - «Обертки» – возвращают новые коллекции на основе имеющихся
    - **Синхронизированные коллекции**
    - **Не модифицируемые коллекции**

Алгоритмы. Класс *Collections*. Добавление строк и сортировка

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

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

Collections.addAll(strings, "Moscow", "London", "Berlin");

System.out.println("До:    " + strings);
    Collections.sort(strings);
System.out.println("После: " + strings);
```

```
До:    [Moscow, London, Berlin]
После: [Berlin, London, Moscow]
```

Алгоритмы. Класс *Collections*. Добавление массива строк и сортировка

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

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

String[] cities = { "Moscow", "London", "Berlin" };

Collections.addAll(strings, cities);

System.out.println("До:    " + strings);
    Collections.sort(strings);
System.out.println("После: " + strings);
```

```
До:    [Moscow, London, Berlin]
После: [Berlin, London, Moscow]
```

Алгоритмы. Класс *Collections*. Сортировка без учета регистра

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
Comparator<String> comparator = new Comparator<String>(){
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};

List<String> strings = new ArrayList<>();

String[] cities = { "Moscow", "berlin", "London", "moscow", "Berlin" };

Collections.addAll(strings, cities);

System.out.println("До:    " + strings);
    Collections.sort(strings, comparator);
System.out.println("После: " + strings);
```

```
До:    [Moscow, berlin, London, moscow, Berlin]
После: [berlin, Berlin, London, Moscow, moscow]
```

Алгоритмы. Класс *Collections.*
Java8 – переходим к лямбда-выражениям


```
Comparator<String> comparator = new Comparator<String>(){
    public int compare(String s1, String s2) {
        return  s1.compareToIgnoreCase(s2);
    }
};

List<String> strings = new ArrayList<>();

String[] cities = { "Moscow", "berlin", "London", "moscow", "Berlin" };

Collections.addAll(strings, cities);

System.out.println("До:    " + strings);
    Collections.sort(strings, comparator);
System.out.println("После: " + strings);
```

```
До:    [Moscow, berlin, London, moscow, Berlin]
После: [berlin, Berlin, London, Moscow, moscow]
```

Алгоритмы. Класс *Collections*
Внешний сортировщик – лямбда выражение

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
Comparator<String> comparator = (a, b) -> a.compareToIgnoreCase(b);

List<String> strings = new ArrayList<>();

String[] cities = { "Moscow", "berlin", "London", "moscow", "Berlin" };

Collections.addAll(strings, cities);

System.out.println("До:    " + strings);
    Collections.sort(strings, comparator);
System.out.println("После: " + strings);
```

```
До:    [Moscow, berlin, London, moscow, Berlin]
После: [berlin, Berlin, London, Moscow, moscow]
```

Алгоритмы. Класс *Collections*

Лямбда выражение в месте вызова сортировки

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

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

String[] cities = { "Rome", "berlin", "Bern", "moscow", "Minsk" };

Collections.addAll(strings, cities);

System.out.println("До:    " + strings);
Collections.sort(strings, (s1, s2) -> s1.length() - s2.length() );
System.out.println("После: " + strings);
```

```
До:    [Rome, berlin, Bern, moscow, Minsk]
После: [berlin, moscow, Minsk, Rome, Bern]
```

Алгоритмы. Класс *Arrays*

Лямбда выражение в месте вызова сортировки

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
public static void main(String[] args) {
    List<String> strings = Arrays.asList(args);

    strings.forEach(str -> System.out.format(" '%s'", str));
}
```

```
'Moscow' 'London' 'Rome'
```

Алгоритмы. Класс *Collections*

Случайное расположение элементов коллекции

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

```
// Запрашиваем все правильные ходы на доске
// фигурами заданного цвета.
List<Move> correctMoves = getCorrectMoves(board, color);

// Случайным образом переставим ходы в списке
// чтобы игра не повторялась.
Collections.shuffle(correctMoves);

// В переменой brain хранится ссылка на класс реализующий
// интерфейс Comparator.
// Каждая программа-игрок сортирует ходы по своему.
correctMoves.sort(brain);

// После сортировке первый ход – лучший ход.
Move bestMove = correctMoves.get(0);
```

Романов Владимир Юрьевич ©2025

Родовые типы в языке Java

Родовые типы языка Java.
1. Необходимость родовых типов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Безопасность типов. Контроль типов должен выполняться на этапе компиляции.

```
Map m = new HashMap();
m.put("key", "blarg");
String s = (String)  m.get("key");
```

- Удаление излишних преобразований типов. Текст программы должен быть более читаемый и содержать меньше ошибок.

```
Map m = new HashMap();
m.put("key", 1);
String s = (String)  m.get("key");
```

- Код должен быть более эффективным по времени выполнения.

Родовые типы языка Java.
2. Пример использования родовых типов

- Объявление родового типа-интерфейса Map

```
public interface Map<K, V> {
      public void put(K key, V value);
      public V get(K key);
}
```

- Использование родового типа-интерфейса Map

```
Map<String, String> m = new HashMap<String, String>();
m.put("key", "blarg");
m.put("key", 1); // Ошибка компиляции.
String s = m.get("key");
```

Родовые типы языка Java.
3. Не ковариантность родовых типов

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Ковариантность массивов языка Java

```
Integer[] intArray = new Integer[10];
Number[] numberArray = intArray;
```

- Не ковариантность родовых типов

```
List<Integer> intList = new ArrayList<Integer>();
List<Number> numberList = intList; // Ошибка компиляции
```

- Number есть супертип для Integer.

  Но List<Number> не есть супертип List<Integer>

  Object супертип для List<Integer>

Родовые типы языка Java.
4. Wildcard (Неизвестный тип)

- Метод *printList* не работает с типом *List<String>*

```
void printList(List l) {
   for (Object o : l)
      System.out.println(o);
}
```

- Метод printList работает только с типом List<Object>

```
void printList(List<Object> l) {
   for (Object o : l)
        System.out.println(o);
}
```

- Метод printList работает со списком элементов любого типа: List<Object>, List<Integer>, …

```
void printList(List<?> l) {
     for (Object o : l)
          System.out.println(o);
}
```

Родовые типы языка Java.
5. Родовые методы

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Родовые методы не обязательно должны быть у родовых типов.
- Родовые методы также имеют placeholders.

```
public <T> T ifThenElse(boolean b, T first, T second) {
        return b ? first : second;
}
```

- Пример: метод ifThenElse работает со всем типами, если у 2-го и 3-го параметров типы одинаковые.

```
String s  = ifThenElse(b, "a", "b");
Integer i = ifThenElse(b, new Integer(1), new Integer(2));
```

- Родовые методы приемлемы для статических методов, когда не используются типы заданные для родового класса.
- Когда тип родового класса применяется только к методу. В этом случае упрощается сигнатура родового класса.

Родовые типы языка Java.
6. Ограниченные типы. Ограничение сверху

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2025

- Параметр типа **V** не ограничен

```
public class Matrix<V> { ... }
```

- Параметр типа **V** ограничен типом **Number**

```
public class Matrix<V extends Number> { ... }
```

Ограничение сверху.
V класс Number
и его потомки

Родовые типы языка Java.
6. Ограниченные типы. Ограничение снизу

МГУ им. М.В.Ломоносова. Факультет ВМК.
Романов Владимир Юрьевич ©2025

- Параметр типа **V** ограничен типом **Number**

```
public class A<V super Number> { ... }
```

Ограничение снизу.
V класс Number
и его предки

Java 8.

Владимир Юрьевич Романов
Московский Государственный Университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
vromanov@cs.msu.su,
vladimir.romanov@gmail.com
http://master.cmc.msu.ru

Java 8. Lambda - выражения

Владимир Юрьевич Романов
Московский Государственный Университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
vromanov@cs.msu.su,
vladimir.romanov@gmail.com
http://master.cmc.msu.ru

Назначение Lambda-выражений

- Трактовать блоки кода как объекты (данные)
- Это не только полезно, но и необходимо
- Lambda (**λ**)-выражение – это просто блок кода с параметрами, которые необходимо передать в блок (анонимная функция).

Обычные функции

- Могут получать объекты
- Могут создавать объекты
- Могут возвращать объекты

Функции высшего порядка (появились в Java 8)

- Могут получать функции
- Могут создавать функции
- Могут возвращать функции

Пример: Lambda – выражение вместо анонимного класса

Романов Владимир Юрьевич ©2025

```
public class RunnableTest {

public static void main(String[] a) {

 // Anonymous Runnable
 Runnable r = new Runnable() {
    @Override
    public void run(){
      System.out.println("Hello");
    }
  };
  r.run();
}
```

```
public class RunnableTest {

public static void main(String[] a){

  // Lambda Runnable
  Runnable r = () ->
      System.out.println("Hello");

  r.run();
}
```

```
Runnable r = new Runnable
    @Override
    public void run(){
      System.out.println("Hello");
    }
  }
```

Пример: Lambda – выражение вместо анонимного класса


```
final String name = "John";
Button button = new Button();

button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
            System.out.println("hi " + name);
      }
});
```

```
final String name = "John";
Button button = new Button();

button.addActionListener( event -> System.out.println("hi " + name) );
```

Пример: Lambda – выражение в операторе сравнения при сортировке


```
List<Person> personList = Person.createShortList();

Collections.sort(personList,
  new Comparator<Person>(){
    public int compare(Person p1, Person p2){
         return p1.getName().compareTo(p2.getName());
    }
  }
);
```

```
List<Person> personList = Person.createShortList();

Collections.sort(personList,
    (Person p1, Person p2) ->
       p1.getSurName().compareTo(p2.getSurName()));
);
```

Пример: Lambda – выражение в операторе сравнения при сортировке


```
List<Person> personList = Person.createShortList();

Collections.sort(personList,
    new Comparator<Person>(){
        public int compare(Person p1, Person p2){
            return p1.getName().compareTo(p2.getName());
        }
    }
);
```

```
List<Person> personList = Person.createShortList();

Collections.sort(personList,
    (p1, p2) -> p1.getSurName().compareTo(p2.getSurName()));
);
```

Пример: Lambda – выражение при работе с коллекциями


```
List<String> features = Arrays.asList("A",  "B", "C", "D");

// Внешний итератор.
for (String feature : features)
     { System.out.println(feature); }
```

```
List<String> features = Arrays.asList("A",  "B", "C", "D");

// Внутренний итератор.
features.forEach(n -> System.out.println(n));

// Ссылка на метод.
features.forEach(System.out::println);
```

Назначение Lambda-выражений

- Лямбда – выражение **SAM**-тип:
  (**S**ingle **A**bstract **M**ethod) – интерфейс с единственным *абстрактным* методом

- Другое название **SAM**-типа – *функциональный интерфейс*

- В Java8 могут быть интерфейсы с **НЕ** абстрактными методами (*default* – методы)

Функциональный интерфейс. Пример


```
package java.util.function;

@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        // ...
    }
}
```

- **@FunctionalInterface** – аннотация для синтаксического контроля: только один абстрактный метод.

- **default – может быть неограниченное число умалчиваемых методов**

Синтаксис Lambda-выражений

Параметры **Стрелка** **Тело**

```
(int x, int y) ->                x + y

()             ->                100

(String s)     -> {
     System.out.println(s);
}
```

Синтаксис Lambda-выражений


```
Параметры      Стрелка          Тело

(x, y)           ->             x + y

()               ->               100

s          -> { System.out.println(s); }
```

Функциональные интерфейсы. java.util.function

Романов Владимир Юрьевич ©2025

Имя интерфейса	Аргумент	Возвращает
Predicate<T>	T	Boolean
Consumer<T>	T	Void
Function<T,R>	T	R
Supplier<T>	Нет	T
UnaryOperator<T>	T	T
BinaryOperator<T>	(T, T)	T

Функциональный интерфейс. Примеры из пакета java.util.function

- **Consumer<T> – выполняет действия на типом T. Результат не возвращается**

- **Supplier<T> – возвращает результат типа T. Ввод не требуется**

- **Predicate<T> – получает экземпляр типа T как параметр. Возвращает логическое значение**

- **Function<T,R> – получает экземпляр типа T как параметр. Возвращает результат типа R**

Пример: Lambda – выражение для функционального программирования


```
package java.util.function;

interface Predicate<T> {
        boolean test(T t);
        //…
}
```

```
// Функциональный интерфейс
interface Predicate<Person> {
        boolean test(Person p);
}
```

```
// Функциональный интерфейс
interface CheckPerson {
        boolean test(Person p);
}
```

Пример: Lambda – выражение для функционального программирования


```
// Объявление
public static
void printWithPredicate( List<Person> roster,  Predicate<Person> tester)  {
    for (Person p : roster) {
            if (tester.test(p)) {
               p.printPerson();
            }
    }
}
```

```
// Вызов
printWithPredicate( roster,
          p ->  p.getGender() == Person.Sex.MALE &&
                p.getAge() >= 18 &&
                p.getAge() <= 25
);
```

Умалчиваемые методы интерфейсов. Новые методы для коллекций


```
interface Collection {
//…

default void forEach (Consumer<? super T> action) {

        for (T  t  :  this)
            action.accept(t);

}
```

```
@FunctionalInterface
interface Consumer<T> {
        //…
        void accept(T t);
}
```

Умалчиваемые методы интерфейсов. Новые методы для коллекций


```
List<String> features = Arrays.asList("A", "B", "C", "D");

// Внутренний итератор.
features.forEach (n -> System.out.println(n));

// Внутренний итератор  со ссылкой на метод.
features.forEach (System.out::println);
```

Java 8. Stream - потоки

Владимир Юрьевич Романов
Московский Государственный Университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
vromanov@cs.msu.su,
vladimir.romanov@gmail.com
http://master.cmc.msu.ru

Определение потоков

Последовательность элементов из ***источника*** которые поддерживают ***составные операции***

Последовательность элементов:
поток предоставляет интерфейс для упорядоченного множества значений для заданного типа элементов.
Однако потоки не хранят значения, а вычисляют их по требованию.

Источник:
потоки потребляют информацию из источников предоставляющих данные: коллекций, массивов, ресурсов ввода и вывода.

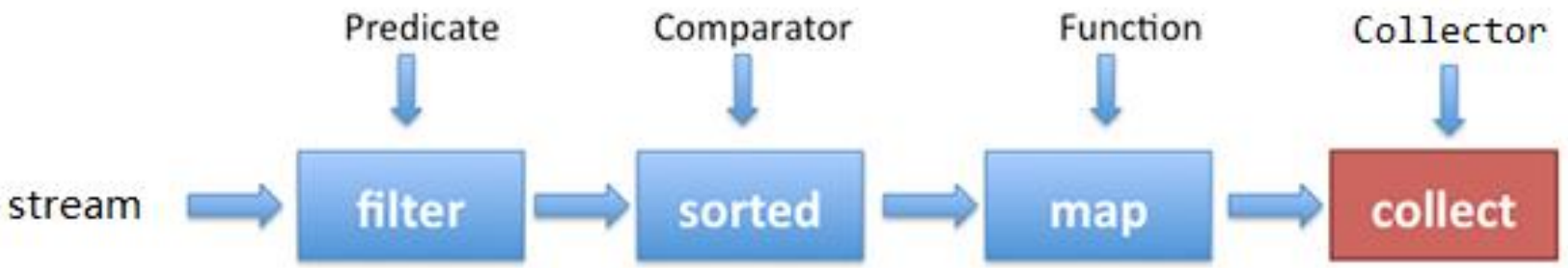
Составные операции:
Потоки поддерживают операции из функциональных языков программирования: filter, map, reduce, find, match, sorted, ...

Отличие операций потоков от операций коллекций

Конвейер (Pipelining):
Многие операции потока возвращают сами потоки.
Это позволяет объединять операции в форме большого конвейера.

Внутренние итерации:
В отличие от коллекций, которые проходятся явно (*внешняя итерация*), в потоковых операциях итераций скрыта.

Обработка данных в потоке



Интерфейс Stream. Преобразования конвейера


```
public interface Stream<T> extends BaseStream<T,Stream<T>>
{
    static <T> Stream<T> of(T... values);

    Stream<T> filter(Predicate<? super T> predicate);
    Stream<T> distinct();
    Stream<T> sorted();
    Stream<T> sorted(Comparator<? super T> comparator);
    Stream<T> peek(Consumer<? super T> action);
    <R> Stream<R> map(Function<? super T,? extends R> mapper);
    IntStream mapToInt(ToIntFunction<? super T> mapper);

    //…
```

Интерфейс Stream. Окончание конвейера


```
public interface Stream<T> extends BaseStream<T,Stream<T>>
{
    //…

    void forEach(Consumer<? super T> action);
    Object[] toArray();
    Optional<T> min(Comparator<? super T> comparator)
    long count();
    boolean allMatch(Predicate<? super T> predicate);
    boolean anyMatch(Predicate<? super T> predicate);
    <R,A> R collect(Collector<? super T,A,R> collector);
}
```

Интерфейс BaseStream


```
interface BaseStream<T,S extends BaseStream<T,S>> {
   boolean isParallel();
   Iterator<T> iterator();
   S parallel();
   S sequential();
   S unordered();
   S onClose(Runnable closeHandler);
}
```

Java8. Потоки
Порождение потоков из коллекций


```
interface Collection {
    //...

    default Stream<E> stream();

    //...
}
```

```
List<String> collection = Arrays.asList("A", "B", "C", "D");

// sequential version
Stream stream = collection.stream();

// parallel version
Stream parallelStream = collection.parallelStream();
```

Java8. Потоки
Порождение потоков


```
Stream<Integer> numbersFromValues = Stream.of (1, 2, 3, 4);

int[] numbers = {1, 2, 3, 4};
IntStream numbersFromArray = Arrays.stream(numbers);

List<String> list = Arrays.asList("Monkey", "Lion", "Giraffe”);
Stream<String> streamFromList = list.stream();

Set<String> set = new HashSet<>(list);
Stream<String> streamFromSet = set.stream();

Stream<String> lines =
     Files.lines(Paths.get("File.txt"), Charset.defaultCharset());

long numberOfLines =
     Files
          .lines(Paths.get("File.txt”), Charset.defaultCharset())
          .count();
```

Java8. Потоки
Порождение потоков


```
Stream<Integer> numbers = Stream.iterate (0, n -> n + 10);
numbers
        .limit(5)
        .forEach( System.out::println ); // 0, 10, 20, 30, 40

IntStream oneTwoThree = IntStream.of(1, 2, 3);

IntStream positiveSingleDigits = IntStream.rangeClosed(1, 9);


IntStream powersOfTwo = IntStream.iterate(1, i -> i * 2);

IntStream randomInts = new Random().ints();

IntStream chars = "ABC".chars();
```

Пример: Обработка списка из строк


```
List<String> collection = Arrays.asList(
"alfred", "Adam", "Alex", "David", "Catrin", "Anita", "ann" );


List<String> result = collection .stream()
        .filter( e -> e.startsWith("A") || e.startsWith("a") )
        .sorted( String.CASE_INSENSITIVE_ORDER )
        .peek( System.out::println )
        .sorted()
        .map(e -> "Name is: " + e)
        .collect( Collectors.toList()  );

System.out.println(result);
```

Пример: Обработка списка из строк


```
Adam
Alex
alfred
Anita
ann
[Name is: Adam, Name is: Alex, Name is: Anita, Name is: alfred, Name is: ann]
```

Пример: редукция


```
int sum = Stream.of(1, 2, 3)
        .reduce (0, (acc, element) -> acc + element);
```

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Пример: Сборка


```
List<String> collected = Stream.of("a", "b", "c")
   .collect( Collectors.toList() );
```

```
<R,A> R collect(Collector<? super T,A,R> collector);
```

interface Collector <T,A,R>

T – Тип входных параметров для операции редукции.

A – Тип для аккумуляции операции редукции

R – Результирующий тип операции редукции

Пример: Подсчет количества фигур противника атакующих заданную фигуру


```
public long enemyAttacks(Piece piece) {
    Board board = piece.square.getBoard();

    PieceColor myColor = piece.getColor();
    PieceColor enemyColor = Board.getOponentColor(myColor);

    List<Square> squares= board.getSquares();
    long n = squares
        .stream()
        .filter(s -> !s.isEmpty())
        .map(s -> s.getPiece())
        .filter(p -> p.getColor() == enemyColor)
        .filter(p -> attacksSquare(p, square))
        .count();

    return n;
}
```

Пример: Лямбда выражение при сравнении ходов. Выбор ходов с наибольшим шагом


```
/**
 * Приоритет у хода делающего больший шаг к цели -
 * противоположному углу доски.
 */
Comparator<ITransferMove> maxStep =
    (move1, move2) -> {
        // Направление шага.
        int dir = move1.getPiece().isWhite() ? 1 : -1;

        int step1 = move1.getSource().shift( move1.getTarget() );
        int step2 = move2.getSource().shift( move2.getTarget() );

        return dir * (step2 - step1);
    };
```

Пример: Лямбда выражение при сравнении ходов. Выбор ходов с удаленной от цели клетки


```
/**
 * Приоритет у хода с более дальней от цели клетки
 * (ход отстающими фигурами).
 */
Comparator<ITransferMove> fromBack =
    (move1, move2) -> {
        Square goal = Halma.getPieceGoal(move1.getPiece());

        // Расстояние до клетки - цели.
        int distance1 = goal.distance(move1.getSource());
        int distance2 = goal.distance(move2.getSource());

        return Math.abs(distance2) - Math.abs(distance1);
    };
```

Пример: Использование умалчиваемых методов интерфейса **Comparator**


```
protected Comparator <ITransferMove> getComparator() {
    return maxStep.thenComparing(fromBack);
}
```

Использование умалчиваемого метода **thenComparing** для задания порядка сравнения