

Языки и технологии программирования.

Объектно-ориентированное программирование

*О развитии языков программирования.
Основы Java.*

Цели курса

- 1. Изучить язык Java
- 2. Познакомиться с базовыми понятиями и принципами ООП
- 3. Познакомиться со стандартными подходами к созданию программ (шаблоны проектирования)
- 4. Получить практические навыки использования приемов ООП в программах небольшого и среднего размеров
- 5. Получить представления об особенностях применения приемов ООП в разных языках (на примере Java и C#)

Литература

- Герберт Шилдт "Java. Руководство для начинающих"
- Герберт Шилдт "Java. Полное руководство"
- Брюс Эккель "Философия Java"

- Герберт Шилдт "С# 4.0 Полное руководство"
- Эндрю Троелсен "Язык программирования С# 6.0 и платформа .Net 4.6"

- Марк Гранд "Шаблоны проектирования в Java"
- Мартин Фаулер "Рефакторинг. Улучшение существующего кода "
- Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес
"Приемы объектно-ориентированного проектирования. Паттерны проектирования"

- Barbara Liskov, John Guttag "Program Development in Java"

Полезные ссылки

- Материалы курса
<http://courses.imkn.urfu.ru/oop>
(или <http://courses.busin.usu.ru/oop>)
- The Principles of OOD (на английском)
<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Курс – 2 семестра:

1 семестр – основные понятия и принципы ООП, реализация ООП в языках Java и C#;

2 семестр – технологии промышленного программирования.

Курс – 2 семестра:

1 семестр – основные понятия и принципы ООП, реализация ООП в языках Java и C#;

2 семестр – технологии промышленного программирования.

Отчетность:

1 семестр – зачет, 2 семестр – экзамен.

Зачет/экзамен:

1. теоретический вопрос (экзамен)

2. анализ листинга (зачет/экзамен)

Допуск к зачету/экзамену:

Для допуска требуется сдать не менее 80% практических работ.

Курс – 2 семестра:

1 семестр – основные понятия и принципы ООП, реализация ООП в языках Java и C#;

2 семестр – технологии промышленного программирования.

Отчетность:

1 семестр – зачет, 2 семестр – экзамен.

Зачет/экзамен:

1. теоретический вопрос (экзамен)

2. анализ листинга (зачет/экзамен)

Допуск к зачету/экзамену:

Для допуска требуется сдать не менее 80% практических работ.

Аттестация:

Для получения аттестации требуется успешная работа на практических занятиях и посещаемость лекций.

Программная поддержка:

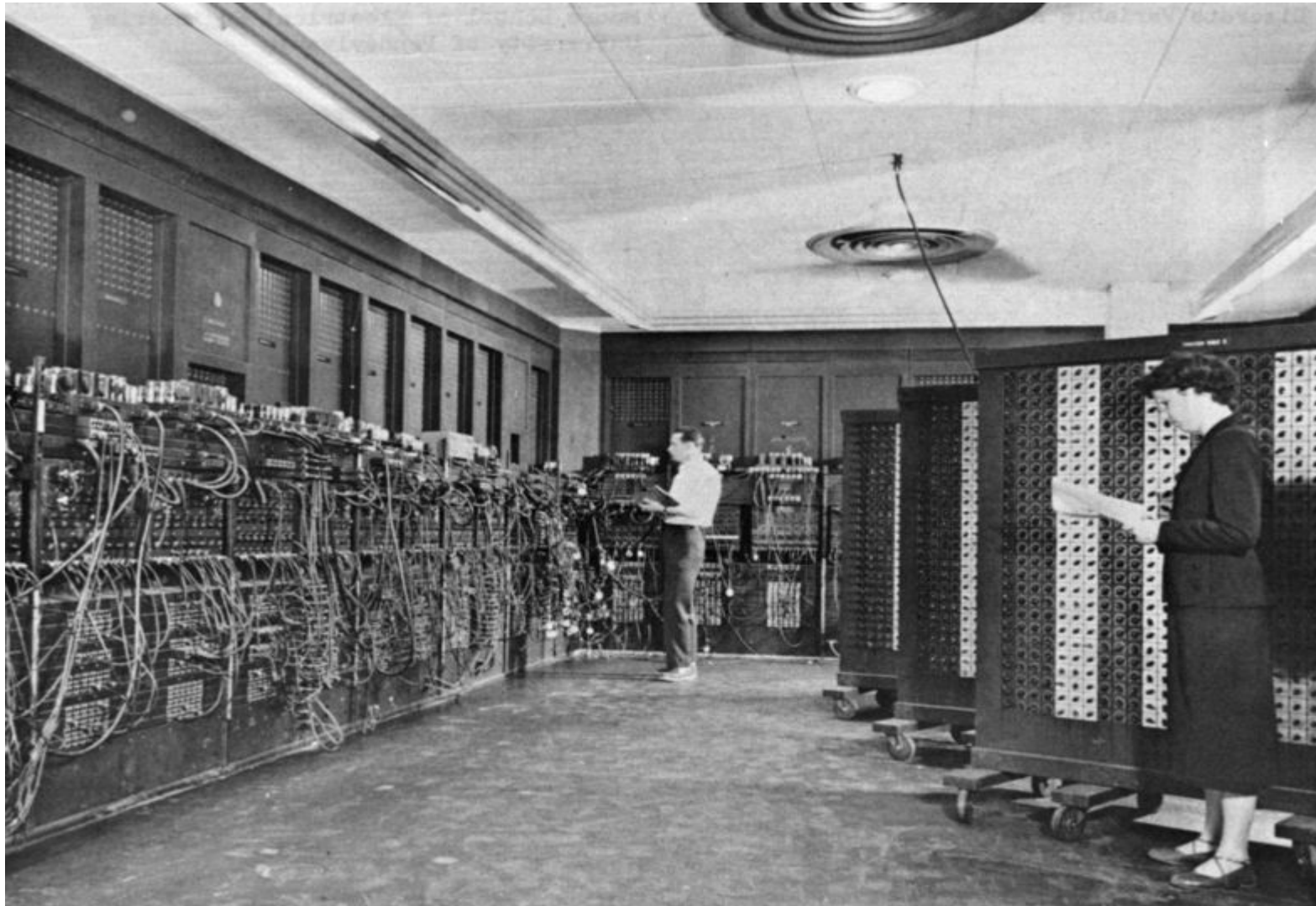
- Java 5 (JDK 1.5) или выше (<http://java.oracle.com>)
- Eclipse 3.3 или выше (<http://www.eclipse.org>)
- Компилятор C#, поставляемый в составе платформы .Net (для платформы Windows доступен для скачивания на сайте Microsoft)
- Для программирования на C# допустимо использование среды разработки Microsoft Visual Studio (наличие JetBrains Resharper крайне желательно)

Для программирования на Java допустимо использование альтернативных IDE (Oracle NetBeans, Oracle JDeveloper, IntelliJ Idea).

Развитие языков программирования

История языков программирования

1946 – первая ЭВМ, машина Эниак (Джон Экерт, Джон Мокли):



История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

ENIAC, сокр. от англ. *Electronic Numerical Integrator And Computer* —
Электронный численный интегратор и вычислитель

Вес - 27 тонн (17468 радиоламп)

Объём памяти: 20 число-слов

Потребляемая мощность – 174 кВт

Вычислительная мощность – 3 операции извлечения квадратного корня,
или 40 делений, или 357 операций умножения, или 5000 операций
сложения в секунду.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

ENIAC, сокр. от англ. *Electronic Numerical Integrator And Computer* —
Электронный численный интегратор и вычислитель

Для программирования этой машины никакого языка не требовалось, поскольку устройство программировалось посредством перекоммутации кабелей и переключением тумблеров.

Лишь позже, в 50-е годы появился язык Ассемблер, позволявший писать программы не в машинных кодах, а в виде, удобном для восприятия человеком.

Наблюдения:

1. Программа на Ассемблере предназначена для конкретного процессора.
То есть она не является переносимой.
2. Программа на языке Ассемблер всегда сложна для понимания.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

**1969-1973 – Кен Томпсон, Денис Ритчи создали
«универсальный ассемблер», ныне известный как язык С.**

Важными достоинствами языка С стали:

1. Переносимость на уровне исходных кодов
2. Возможность использования структурного программирования и подпрограмм
3. Близость многих инструкций языка к инструкциям ассемблера (что дает высокую эффективность программам на С)

История языков программирования

1946 – Джон Эkert, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт создает язык Паскаль, для обучения программированию

Важным достоинством языка Pascal являются его синтаксическая простота и ясность конструкций. Первые версии языка были бедны функционально ввиду отсутствия стандартных библиотек. Язык получил бурное развитие после выпуска компилятора Borland Turbo Pascal, который давал те же возможности, что и аналогичный компилятор языка C, но работал ощутимо быстрее.

История языков программирования

Языки C и Pascal относятся к семейству процедурных языков программирования. Такие языки предполагают наличие возможности использования:

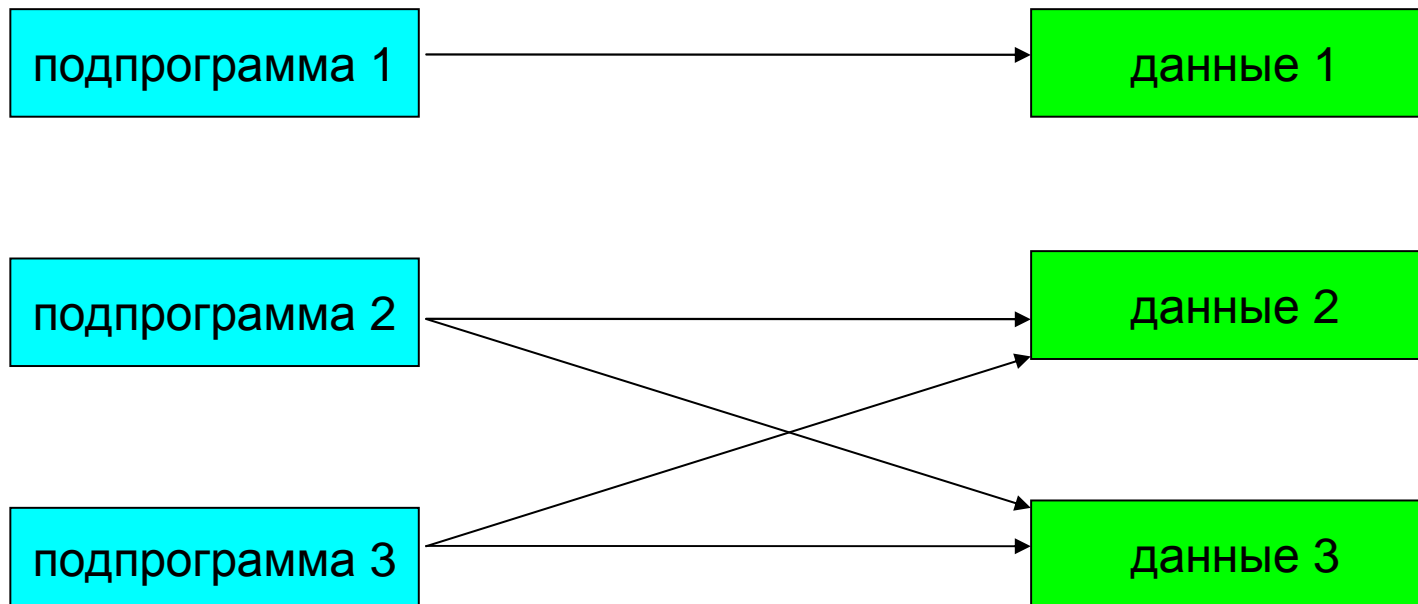
- стандартных инструкций управления ходом выполнения программы (ветвления и циклы)
- подпрограмм (группировка инструкций)
- структур (группировка данных)

Подобные возможности группировки позволили сделать программы более понятными человеку. Поэтому такие языки стали называть языками высокого уровня.

История языков программирования

Фактически, любая программа состоит из блоков, в которых какие-то данные обрабатываются какими-то подпрограммами. Таким образом, сложность понимания программы напрямую зависит от числа подпрограмм и количества используемых подпрограммами данных. Скорость роста числа связей при увеличении размера программы – квадратичная.

Поэтому возникает цель: улучшить языки программирования, чтобы уменьшить скорость роста количества логических связей в программе.

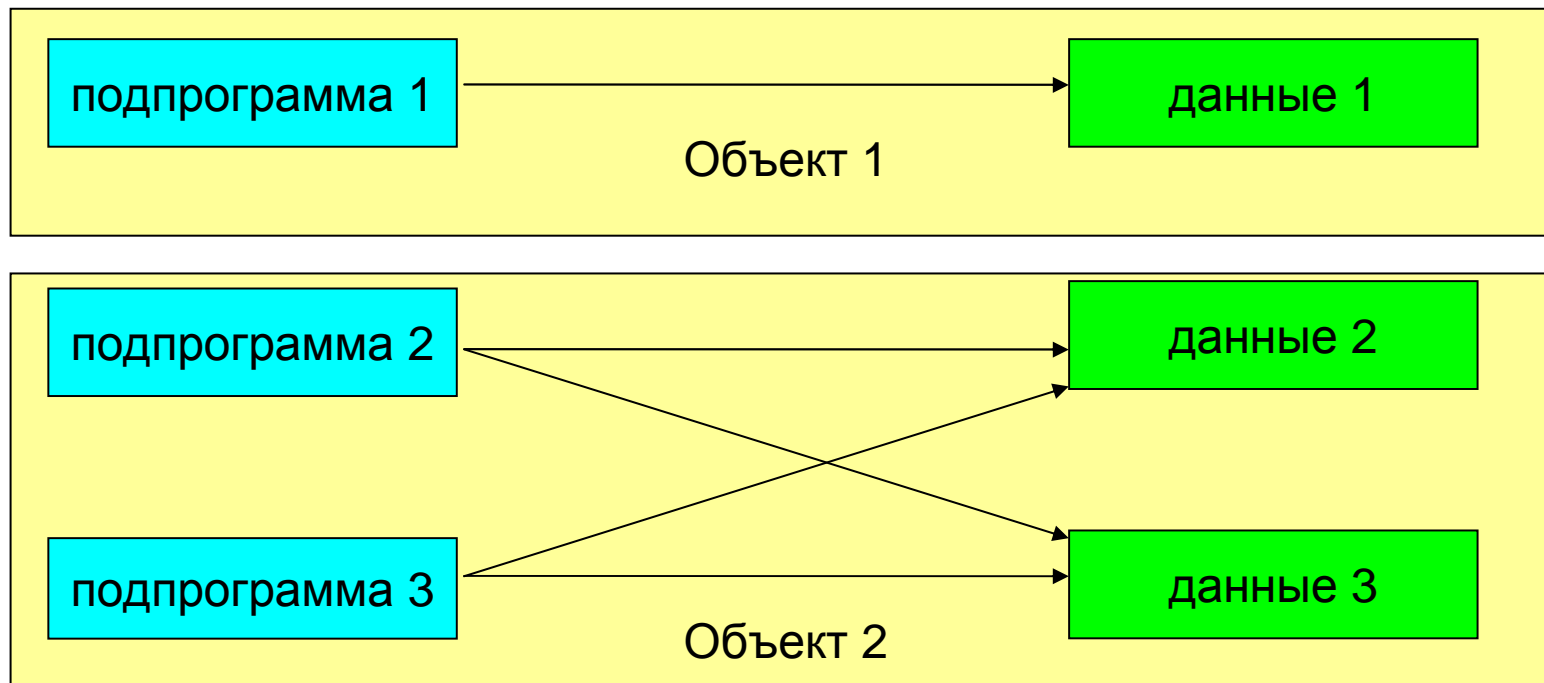


История языков программирования

В рамках объектно ориентированного программирования было предложено объединять данные и использующие их подпрограммы в единую сущность: объект.

Введение понятия «объект» позволило упростить программы за счет .

Также оказалось, что объекты прекрасно подходят для моделирования сущностей реального мира.



История языков программирования

В рамках объектно ориентированного программирования было предложено объединять данные и использующие их подпрограммы в единую сущность: **объект**.

В свою очередь множество объектов идентичной структуры оказалось удобно декларировать с помощью **классов**.

Например: что такое «стол»?

История языков программирования

В рамках объектно ориентированного программирования было предложено объединять данные и использующие их подпрограммы в единую сущность: **объект**.

В свою очередь множество объектов идентичной структуры оказалось удобно декларировать с помощью **классов**.

Например: что такое «стол»?

Если говорить о столах вообще (наличие столешницы на заданном уровне), то получится класс (**набор объектов** идентичной структуры).

Конкретный стол является **объектом** некоторого **класса** (имеет конструкцию определенного типа), но обладает некоторыми собственными характерными чертами (рисунок на столешнице).

Процесс поиска определения некоторого понятия, выявления ее существенных свойств называется **выделением абстракции**.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт: язык Паскаль

1967 – Оле-Йохан Даль и Кристен Нюгорд создают язык моделирования Simula-67

Simula-67 – это первый язык, поддерживающий концепции ООП. Он создавался для решения задач моделирования сложных систем.

Язык опередил свое время, лишь позже идеи этого языка получили вторую жизнь в других языках программирования. Например, если сравнивать язык Simula-67 и первые версии языка Java, то в Simula-67 нет лишь понятия интерфейса и наличия перегрузки конструкторов.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт: язык Паскаль

1967 – Оле-Йохан Даль, Кристен Ньюгорд: Simula-67

1971-1980 – в Xerox PARC создают язык Smalltalk-80

Smalltalk – первый полностью объектный язык. Интересен тем, что в нем появились многие современные идеи: компиляция в код виртуальной машины, автоматическое управление памятью, динамическая типизация, взаимодействие объектов через сообщения.

Удивительный факт: даже обычные управляющие инструкции (if, while) реализованы как объекты.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт: язык Паскаль

1967 – Оле-Йохан Даль, Кристен Ньюгорд: Simula-67

1971-1980 – Xerox PARC: Smalltalk-80

1983 – Бьерн Страуструп создает надстройку над C, ныне известную как C++

Язык быстро стал популярным.

В 1998 году язык C++ был стандартизован в комитете ANSI-ISO.

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт: язык Паскаль

1967 – Оле-Йохан Даль, Кристен Ньюгорд: Simula-67

1971-1980 – Xerox PARC: Smalltalk-80

1983 – Бьерн Страуструп: C++

1990-1996 – в Sun Microsystems создается язык Java

На данный момент Oracle предоставляет 8-ю версию комплекта разработки для Java (Java 8, JDK 1.8).

Важными являются следующие редакции языка:

Java, Java 2, Java 5, Java 8

История языков программирования

При создании языка Java преследовались следующие цели:

1. переносимость программ на уровне откомпилированных кодов;
2. возможность создания безопасных программ, пригодных для передачи через Internet с целью запуска на машине пользователя;
3. простота работы с памятью;
4. простота диагностики и локализации ошибок;
5. наличие богатой библиотеки стандартных модулей.

В основу языка положен синтаксис C/C++, что позволило получить язык, привычный большому числу программистов. Переносимость программ достигается трансляцией в специальный фиксированный по функциональности код (так называемый байт-код), который затем исполняется в виртуальной машине Java (JVM, Java Virtual Machine).

История языков программирования

1946 – Джон Экерт, Джон Мокли: первая ЭВМ, машина Эниак

1969-1973 – Кен Томпсон, Денис Ритчи: язык C

1970 – Никлаус Вирт: язык Паскаль

1967 – Оле-Йохан Даль, Кристен Ньюгорд: Simula-67

1971-1980 – Xerox PARC: Smalltalk-80

1983 – Бьерн Страуструп: C++

1990-1996 – Sun Microsystems: Java

2002-2003 – Microsoft выпускает язык C# и платформу .Net

Платформа .Net – альтернатива языку Java от Microsoft после неудачной попытки адаптации языка Java под платформу Windows. Создана с учетом недочетов в первых версиях языка Java и с изначальной ориентацией на платформу Windows.

История языков программирования

1946 – первая ЭВМ, машина Эниак (Экерт, Мокли)

1967 – язык Simula-67 (Даль, Ньюгорд)

1970 – язык Pascal (Вирт)

1973 – язык C (Томпсон, Ритчи)

1980 – язык Smalltalk-80 (Xerox PARC)

1983 – язык C++ (Страуструп)

1996 – язык Java (Oracle)

2003 – язык C# (Microsoft)

Рейтинг языков (TIOBE index) на август 2016 года:

1. Java – 19.010% -0.26%

2. C – 11.303% -3.43%

3. C++ – 5.800% -1.94%

4. C# – 4.907% +0.07%

5. Python – 4.404% +0.34%

...

Первое знакомство с Java

Java – полностью объектный язык. Поэтому даже в простейшей программе будет описание класса объектов:

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

Описание класса
HelloWorld

Описание точки запуска
программы

Описание класса должно находиться в одноименном файле с расширением .java

Для компиляции и запуска этой программы следует дать такие команды:

```
javac HelloWorld.java
```

```
java HelloWorld
```

Язык Java предлагает программисту следующие predefined типы данных (так называемые примитивные типы):

- int (32-битное знаковое целое)
- byte (8-битное знаковое целое)
- short (16-битное знаковое целое)
- long (64-битное знаковое целое)

- float (32-битное вещественное число, стандарт IEEE 754)
- double (64-битное вещественное число, стандарт IEEE 754)

- char (16-битный код символа в кодировке Unicode)

- boolean (логическое значение true или false)

Целочисленные типы::

- int (32-битное знаковое целое)
- byte (8-битное знаковое целое)
- short (16-битное знаковое целое)
- long (64-битное знаковое целое)

Для целочисленных типов предусмотрены следующие арифметические операции: + (сложение), - (вычитание), * (умножение), / (целочисленное деление), % (целочисленное взятие остатка), & (побитовое логическое «и», AND), | (побитовое логическое «или», OR), ^ (побитовое логическое исключающее «или», XOR), ~ (побитовая инверсия, NOT), << (арифметический сдвиг влево), >> (арифметический сдвиг вправо), >>> (логический сдвиг вправо), ++ (инкремент), -- (декремент).

Операции сравнения (>, >=, <, <=, ==, !=) сравнивают два числа и возвращают результат сравнения как значение типа boolean (true – истина, или false – ложь).

Целочисленные типы:

- int (32-битное знаковое целое)
- byte (8-битное знаковое целое)
- short (16-битное знаковое целое)
- long (64-битное знаковое целое)

```
int a = 1030; // 0x00000406
int b = -1030; // 0xFFFFFBFA
int c;
```

```
c = b / 3;      // -343
c = a % 3;      // 1
c = b % 3;      // -1 (а не 2, как в математике)
```

```
c = a & 0xFF;  // 6
c = ~b;        // 1029, 0x00000405
```

```
c = a++;       // c = 1030, a = 1031
c = ++a;       // c = 1032, a = 1032
```

```
c = b >> 4;     // 0xFFFFFBBF, старшие разряды заполняются знаковым битом
c = b >>> 4;    // 0x0FFFFFFB, старшие разряды заполняются нулем
```

```
c = Integer.MAX_VALUE; // 0x7FFFFFFF, это максимальное знаковое 4-байтовое целое
c = c + 1;             // 0x80000000, это минимальное знаковое 4-байтовое целое
                        // Важно: ошибки при переполнении не возникает!
```

Вещественные типы с плавающей точкой (IEEE 754) :

- float (32-битное вещественное число, стандарт IEEE 754)
- double (64-битное вещественное число, стандарт IEEE 754)

Для вещественных типов определены следующие арифметические операции: + (сложение), - (вычитание), * (умножение), / (деление), % (взятие остатка).

Операции сравнения (>, >=, <, <=, ==, !=) сравнивают два числа и возвращают результат сравнения как значение типа boolean (true – истина, или false – ложь).

Вещественные типы с плавающей точкой (IEEE 754) :

- float (32-битное вещественное число, стандарт IEEE 754)
- double (64-битное вещественное число, стандарт IEEE 754)

Наблюдение: в силу особенностей представления вещественных чисел в памяти сравнение на совпадение (==, !=) для типов float и double обычно не имеет смысла. Вместо сравнения на точное совпадение обычно проверяется, что два числа отличаются не более чем на указанную величину.

Наблюдение: вещественные типы могут хранить 3 специальных значения: +/- бесконечность и NaN (не число). Эти значения соответствуют ситуациям переполнения (деление на число, близкое к нулю) и неопределенности (деление нуля на ноль).

Наблюдение: в вещественных типах можно наблюдать ситуацию потери точности (добавление к большому числу маленького не изменяет исходное число).

Вещественные типы с плавающей точкой (IEEE 754) :

- float (32-битное вещественное число, стандарт IEEE 754)
- double (64-битное вещественное число, стандарт IEEE 754)

```
double a = 1.0;
double b = (a / 3) * 7;
double c = (a * 7) / 3;
// оба выражения равны 2.3(3), но
// b == c дает false

double d = a / 0; // не ошибка, результат - +бесконечность

double e = 0 / 0; // не ошибка, результат - NaN

double f = Math.sqrt(b - c); // не ошибка, результат - NaN
double g = Math.sqrt(c - b); // не ошибка, результат - 2.1073424255447017E-8
```

Символьный тип

- char (16-битный код символа в кодировке Unicode)

По характеристикам и операциям идентичен целочисленным типам.

По семантике его не следует смешивать в операциях с целочисленными типами.

Логический тип:

- `boolean` (логическое значение `true` или `false`)

Логический тип предназначен для вычисления логических выражений. Для него предусмотрены следующие операции: `&` (логическое «и»), `|` (логическое «или»), `^` (логическое исключающее «или»), `&&` (сокращенное логическое «и»), `||` (сокращенное логическое «или»), `!` (логическое отрицание).

Сокращенные операции отличаются от обычных тем, что если результат операции определен сразу после вычисления первого аргумента, то второй аргумент не вычисляется.

Операция присваивания.

Операция присваивания (=) для всех типов заносит в ячейку памяти, соответствующую первому аргументу (левой части операции) результат вычисления второго аргумента (правой части). При этом сама операция возвращает результат, равный присвоенному значению.

```
int a = b = c = 10; // устанавливает число 10 в переменные a, b и c

int d = ((b = 20) + 1); // устанавливает число 20 в переменную b
                        // и число 21 в переменную d
                        // Так оформлять код не рекомендуется!
```

Для операции присваивания допустима комбинированная форма, позволяющая совместить в сокращенную запись арифметическую или логическую бинарную операцию с сохранением результата в переменную первого операнда:

```
int a = 10;

a += 15;    // эквивалентно: a = a + 15;
a %= 3;     // эквивалентно: a = a % 3;
```

Приоритет операций (в порядке уменьшения):

1. (), [] (круглые и квадратные скобки)
2. ++, --, +, -, ~, ! (инкремент, декремент, унарные плюс и минус, побитовое и логическое отрицания)
3. *, /, % (умножение, деление, остаток от целочисленного деления)
4. +, - (сложение, вычитание)
5. >>, >>>, << (арифметические и логический сдвиги)
6. >, >=, <, <= (сравнение на больше, больше или равно, меньше, меньше или равно)
7. ==, != (сравнение на равенство, сравнение на неравенство)
8. & (побитовое и логическое «и»)
9. ^ (побитовое и логическое исключающее «или»)
10. | (побитовое и логическое «или»)
11. && (сокращенное логическое «и»)
12. || (сокращенное логическое «или»)
13. ?: (тернарная операция выбора)
14. op= (комбинированное присваивание)

Наблюдение: обычно стоит явно указывать порядок выполнения операций в выражении при помощи скобок, не полагаясь на умолчания языка. Такой подход улучшает читаемость кода, устраняет возможные неточности в восприятии кода программистом.

Язык Java разрешает использовать все основные конструкции структурного и процедурного программирования:

ВЕТВЛЕНИЕ:

```
if (<условие>)  
{  
    <операторы>  
}  
else  
{  
    <операторы>  
}
```

СОСТАВНОЕ ВЕТВЛЕНИЕ:

```
switch (<выражение>)  
{  
    case <константа 1>:  
    {  
        <операторы 1>;  
        break;  
    }  
    case <константа 2>:  
    {  
        <операторы 1>;  
        break;  
    }  
    ...  
    default:  
    {  
        <операторы>;  
        break;  
    }  
}
```


Язык Java разрешает использовать все основные конструкции структурного и процедурного программирования:

тернарная операция выбора:

`<условие> ? <значение 1> : <значение 2>`

Следующие фрагменты кода эквиваленты:

`a = <условие> ? <значение 1> : <значение 2>;`

```
if (<условие>)
{
    a = <значение 1>;
}
else
{
    a = <значение 2>;
}
```

Язык Java разрешает использовать все основные конструкции структурного и процедурного программирования:

ЦИКЛЫ:

```
while (<условие>)
{
    <тело цикла>
}
```

```
for (<инициализация цикла>; <условие>; <шаг цикла>)
{
    <тело цикла>
}
```

```
do
{
    <тело цикла>
}
while (<условие>)^
```

В теле цикла разрешается использование операторов `continue` (пропускает остаток тела цикла) и `break` (прерывает выполнение цикла).

В случае вложенных циклов в операторах `continue` и `break` можно указать метку цикла, к которому оператор относится:

```
loop1: while (<условие 1>)
{
    while (<условие 2>)
    {
        if (<условие 3>)
        {
            break loop1;           // прерывание внешнего цикла
        }
    }
}
```

```
while (<условие 1>)
{
    while (<условие 2>)
    {
        if (<условие 3>)
        {
            break;                 // прерывание внутреннего цикла
        }
    }
}
```

Язык Java разрешает использовать все основные конструкции структурного и процедурного программирования:

подпрограмма:

```
static public <тип результата> <имя подпрограммы>(<список параметров>)  
{  
    <операторы>;  
    return <значение>;  
}
```

Параметры подпрограммы (в ее описании) называются формальными.

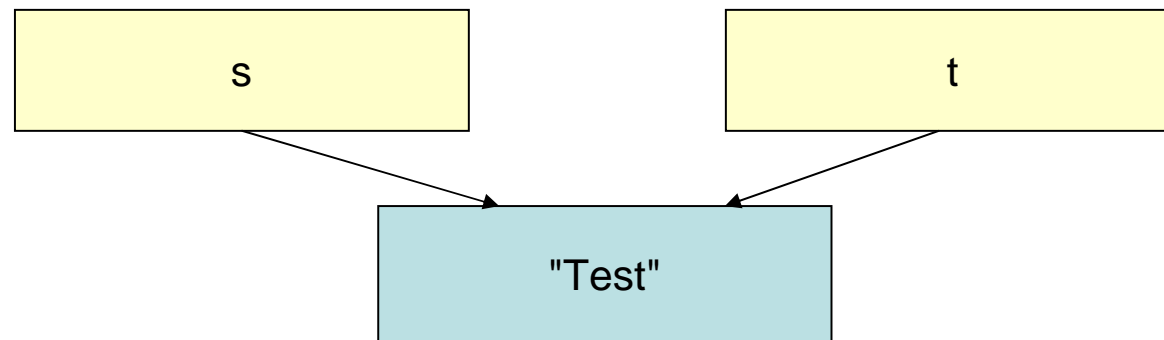
При вызове подпрограммы по сути происходит сопоставление формальных параметров фактическим:

```
// описание подпрограммы  
public static void sample(int value)    // формальный параметр value  
{  
    System.out.println(value);          // вывод на консоль значение параметра value  
}  
  
public static void main(String [] args)  
{  
    int a = 10;  
    sample(a);                          // формальному параметру value сопоставляется  
                                        // фактическое значение 10  
}
```

Объектные переменные (заглядываем вперед)

Для работы с объектами в Java используются объектные переменные, которые фактически хранят ссылки на объекты:

```
public static void main(String [] args)
{
    String s = "Test";           // строки в Java представляются объектами
    String t = s;                // переменные s и t ссылаются на один объект
}
```



Важно помнить о том, что работа ведется именно со ссылками (ибо возможно, что объект, с которым ведется работа через какую-либо ссылку, будет изменен через другие ссылки).

Для представления строк в язык Java встроен объектный тип String. Базовые операции с ним выглядят так:

```
String strMsg = "Hello, World";           // присваивание
strMsg = strMsg + "!";                     // конкатенация
System.out.println(strMsg);                // вывод на экран
System.out.println(strMsg.length());       // вывод на экран длины строки
System.out.println(strMsg.charAt(0));       // вывод на экран первого символа
System.out.println(strMsg.substring(7, 12)); // вывод подстроки "World"
```

Так можно работать с массивами:

```
int [] arrayOfInt = new int[10];           // массив из 10 элементов int
String [] values = new String[5];          // массив из 5 строк
values[0] = "Some text";                   // присваивание первому элементу
System.out.println(values[0]);              // выводит первый элемент
System.out.println(values.length);          // выводит число элементов в массиве
arrayOfInt = new int[]{17,31};              // массив из 2 элементов int,
                                           // проинициализированный значениями 17 и 31
```

Наблюдение: массивы это тоже объекты.

Для чтения строки с клавиатуры следует использовать класс Scanner:

```
import java.util.Scanner;      // сообщаем компилятору о намерении использовать класс

...

Scanner input = new Scanner(System.in); // начинаем процесс чтения с клавиатуры
...
String s = input.nextLine();          // читаем очередную строку
int i = input.nextInt();               // или число
double d = input.nextDouble();        // или произвольный примитивный тип
```

Важно:

согласно правилам оформления программ на Java, имена классов следует начинать с прописной буквы, а имена подпрограмм (методов) и переменных (полей) следует начинать со строчной буквы.

Пример программы, подсчитывающей количество вхождений символа 'b' в строку, введенную с клавиатуры:

```
import java.util.Scanner;  
  
public class Lesson1  
{  
    public static void main(String [] args)  
    {  
        Scanner input = new Scanner(System.in);  
        System.out.println("Введите произвольную строку:");  
        String strInput = input.nextLine();  
        int iCount = 0;  
        for (int iIdx = 0; iIdx < strInput.length(); iIdx++)  
        {  
            char c = strInput.charAt(iIdx);  
            if (c == 'b')  
            {  
                iCount++;  
            }  
        }  
        System.out.println("Символ \'b\' найден "+iCount+" раз");  
    }  
}
```

Сообщаем компилятору
о намерении использовать класс

Языки и технологии программирования.

Объектно-ориентированное программирование

*Основные понятия ООП:
Инкапсуляция*

Основные понятия ООП

- **Инкапсуляция**
- Абстракция
- Интерфейс
- Полиморфизм
- Наследование

Зачем нужна инкапсуляция

Программные продукты со временем могут становиться очень объемными:

Например:

IBM DataQuant (2012 год и продолжает развиваться): 13747 файлов, 101Мб исходных кодов (для сравнения: Толстой Л.Н. «Война и Мир» - примерно 3000000 знаков, 1.5 тысячи страниц – это всего 3Мб текста!).

Без применения специальных техник количество взаимосвязей между участками программы увеличивается пропорционально квадрату от объема исходных кодов.

Следовательно, нужно уметь:

1. объединять взаимосвязанные программные единицы в новые программные сущности;
2. изолировать внутренности таких составных программных сущностей от внешнего кода.

В итоге программа распадается на более простую конструкцию, состоящую из слабо зависящих друг от друга блоков.

Данный механизм называется **инкапсуляцией**.

Зачем нужна инкапсуляция

Инкапсуляция – это:

1. механизм, позволяющий изолировать отдельные программные сущности друг от друга;
2. механизм, позволяющий сгруппировать несколько взаимосвязанных элементов программы в одну новую программную сущность.

Вопрос:

На каких уровнях кода работает инкапсуляция?

Зачем нужна инкапсуляция

Инкапсуляция работает на следующих уровнях кода:

1. подпрограмма/метод

- инкапсулируется только логика

2. **класс**

- инкапсулируется модель данных
- инкапсулируется реализация модели поведения

3. модуль/библиотека

- инкапсулируется логика/совокупное поведение

4. приложение/сервис

- инкапсулируется логика/совокупное поведение
- накладываются дополнительные ограничения на модель взаимодействия

Это наша цель



Во всех названных случаях цель одна и та же: обеспечить изоляцию пользователя (сервиса) от поставщика (этого сервиса).

Зачем нужна инкапсуляция

Инкапсуляция на уровне класса:

1. позволяет **группировать** составные данные (поля) и подпрограммы работы с ними (методы) в логические единицы (объекты, определяемые классами);
2. дает возможность **защитить** данные объекта от спонтанного несанкционированного доступа извне за счет указания ограничений на доступ к ним;
3. обеспечивает **контролируемый доступ** к данным посредством методов;
4. дает возможность гарантировать **целостность и корректность** данных в объекте (состояние объекта) за счет соответствующей организации методов доступа;
5. позволяет обеспечить **прогнозируемость и локальность изменений** и последствий этих изменений при развитии кода программы.

Пример: хотим реализовать библиотеку для работы с рациональными числами.

Наблюдение: рациональное число удобно представить с помощью структуры.

РацЧисло
int: числитель
int: знаменатель

Вопрос: достаточно ли такой конструкции?

Пример: хотим реализовать библиотеку для работы с рациональными числами.

Наблюдение: рациональное число удобно представить с помощью структуры.

РацЧисло
int: числитель
int: знаменатель

Вопрос: достаточно ли такой конструкции?

Нет, недостаточно!

Требуется, чтобы всегда знаменатель был отличен от нуля и числитель со знаменателем были взаимопросты.

Наблюдение: обычно информация в полях структуры или объекта должна быть согласована (содержимое всех полей объекта определяет **состояние объекта**).

Наблюдение: как правило, набор допустимых состояний объекта тем или иным способом ограничен (то есть состояние объекта должно быть корректным).

Наблюдение: в рамках простой структуры невозможно наложить ограничения на содержимое полей, невозможно гарантировать целостность и корректность информации в структуре.

Вопрос: как обеспечить возможность подобного контроля?

Наблюдение: в рамках простой структуры невозможно наложить ограничения на содержимое полей, невозможно гарантировать целостность и корректность информации в структуре.

Вопрос: как обеспечить возможность подобного контроля?

1. Запретить прямой доступ к полям структуры с помощью модификаторов доступа
2. Предоставить контролируемый доступ к полям структуры посредством методов

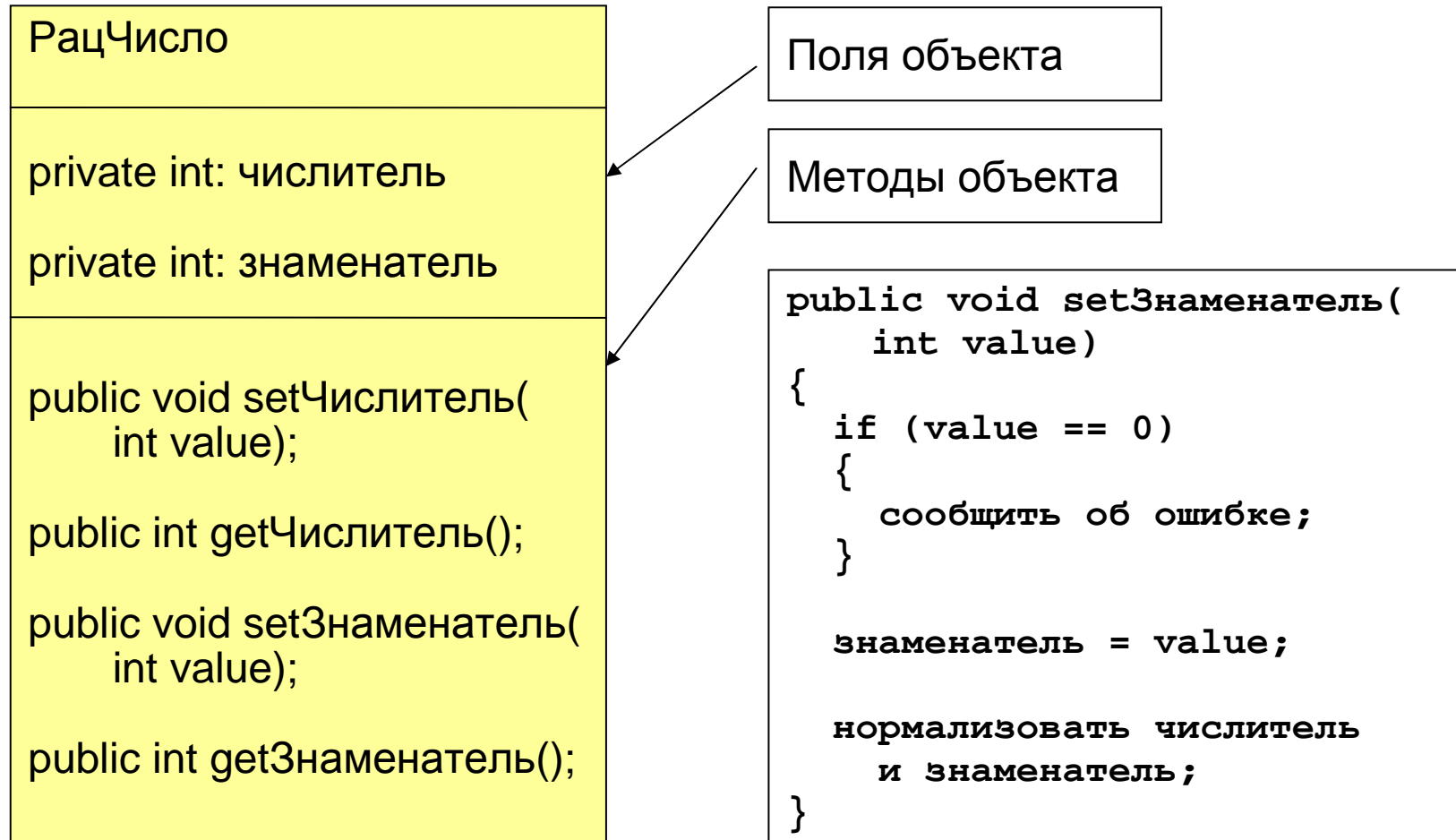
Модификаторы доступа:

public – разрешает доступ к члену/методу из любой точки программы

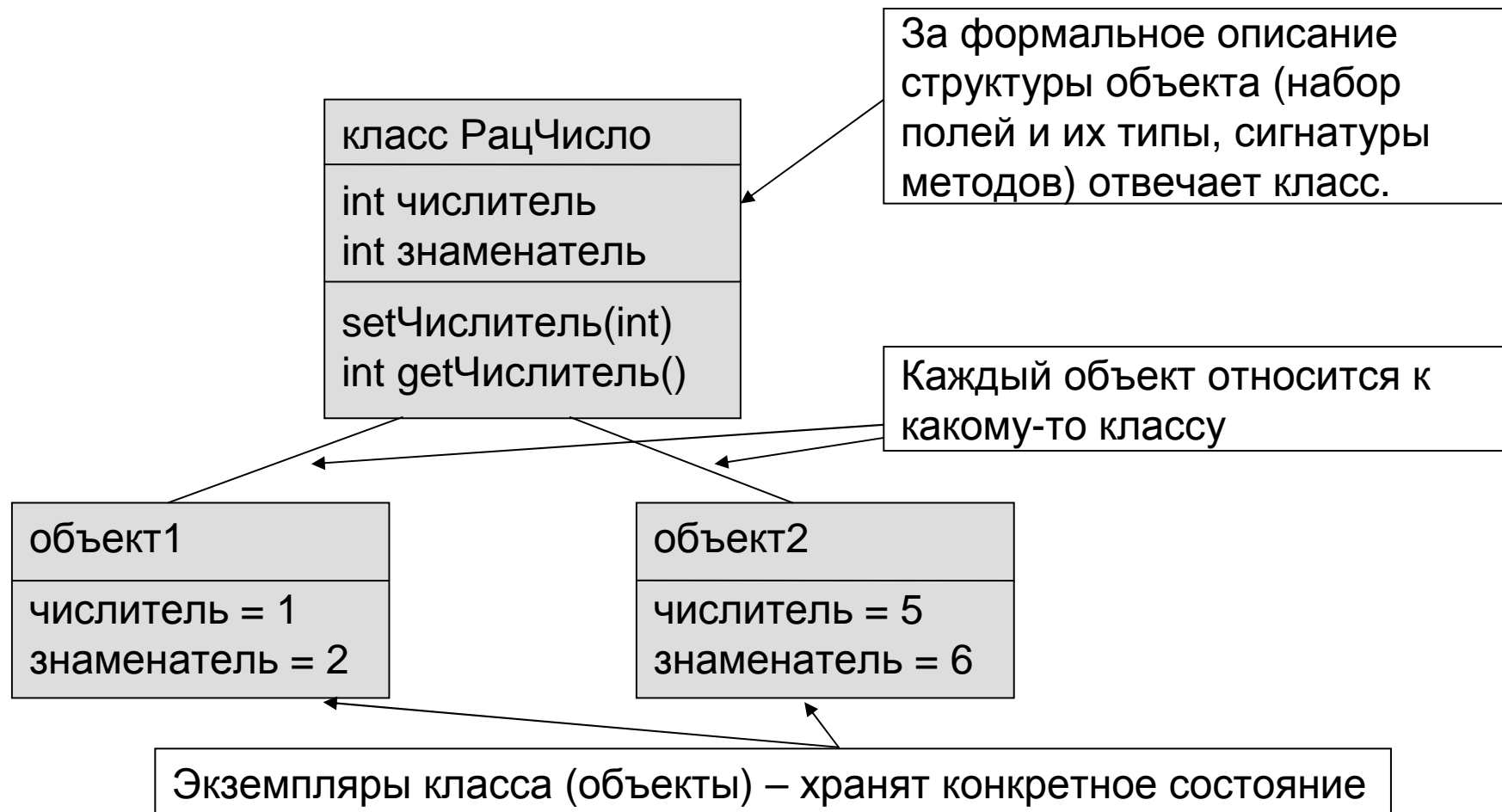
private – разрешает доступ к члену/методу только из того класса, в котором член/метод объявлен

Это простейший случай реализации инкапсуляции: модификаторы доступа формируют ограничения на доступ к элементу класса.

Так мог бы быть устроен объект для хранения рациональных чисел:



Так соотносятся между собой класс и объекты:



Как создать объект

Формальное описание структуры объекта (набор полей и их типы, общие свойства объекта, представленные методами) называется **классом**. Соответственно, каждый объект относится к какому-то классу (то есть конкретный объект является **экземпляром** некоторого **класса**).

Для создания объекта требуется:

- Описать класс (дать формальное описание полей и методов класса).
- Создать (в памяти машины) экземпляр объекта.

Часто формальное описание класса делят на две части: **прототип** (фиксирует набор полей и сигнатуры методов) и **реализацию** (описание кода методов).

Описание класса (Java):

```
public class RationalNumber
{
    private int m_iNumerator;
    private int m_iDenominator;

    public RationalNumber(int iNumerator, int iDenominator)
    {
        m_iNumerator = iNumerator;
        m_iDenominator = iDenominator;
    }

    public int getDenominator()
    {
        return m_iDenominator;
    }

    public void setDenominator(int iDenominator)
    {
        m_iDenominator = iDenominator;
    }
}
```

Декларация класса

Описание полей

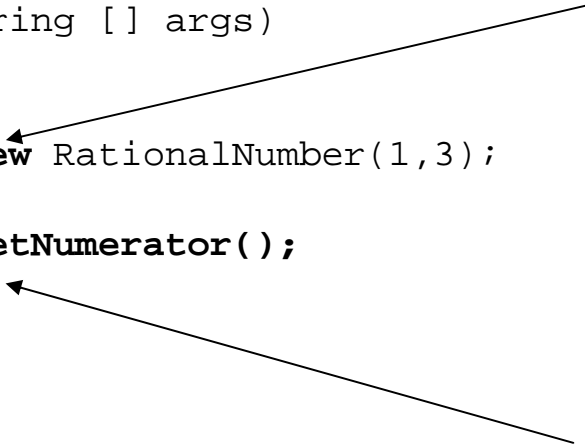
Описание методов

Создание объекта (Java):

```
public static void main(String [] args)
{
    RationalNumber num = new RationalNumber(1,3);

    int iNumerator = num.getNumerator();
}
```

Создание объекта
(конструирование)



Вызов метода
объекта

Демонстрация класса RationalNumber

Описание класса (C#):

```
public class RationalNumber
{
    private int m_iNumerator;
    private int m_iDenominator;

    public RationalNumber(int iNumerator, int iDenominator)
    {
        m_iNumerator = iNumerator;
        m_iDenominator = iDenominator;
    }

    public int GetDenominator()
    {
        return m_iDenominator;
    }

    public void SetDenominator(int iDenominator)
    {
        m_iDenominator = iDenominator;
    }
}
```

Декларация класса

Описание полей

Описание методов

Описание класса (C#):

```
public class RationalNumber
{
    public int Numerator {get; set;}
    public int Denominator {get; set;}

    public RationalNumber(int iNumerator, int iDenominator)
    {
        Numerator = iNumerator;
        Denominator = iDenominator;
    }
}
```

Описание свойств

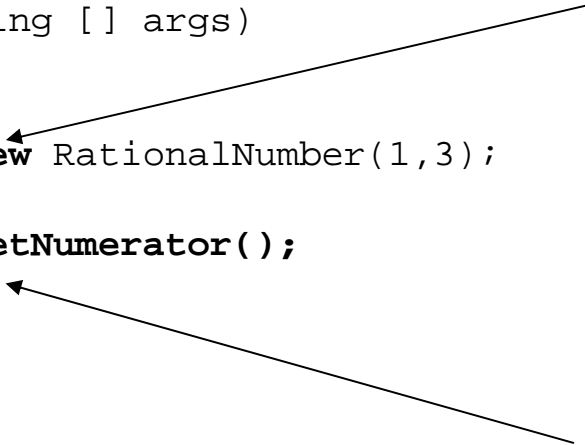


Язык C# позволяет описывать поля и методы доступа к ним более компактно. Такие поля называются **свойствами** (properties). По своей сути за одним свойством скрывается 3 сущности: поле для хранения информации и два метода для чтения и записи значения.

Создание объекта (C#):

```
public static int Main(string [] args)
{
    RationalNumber num = new RationalNumber(1,3);

    int iNumerator = num.GetNumerator();
}
```




Создание объекта
(конструирование)

Вызов метода
объекта

```
public static int Main(string [] args)
{
    RationalNumber num = new RationalNumber(1,3);

    int iNumerator = num.Numerator;
}
```



Тот же самый код с
использованием
свойств

Промежуточные итоги:

1. Инкапсуляция = сокрытие данных и деталей реализации
2. Объект = контейнер данных (структура) с набором операций (методов) для манипулирования данными

Наблюдение:

Поскольку инкапсуляция изолирует внутреннее пространство объекта от внешнего мира, то вопрос «Как объект устроен?» становится неактуальным. А важным становятся вопросы «Что можно с объектом делать?», «Какие внешние интерфейсы он предоставляет?», то есть программиста интересуют лишь набор методов и **поведение** объекта.

Языки и технологии программирования.

Объектно-ориентированное программирование

*Применение инкапсуляции:
неизменные классы (immutable classes)*

Преимущества от инкапсуляции

Основное преимущество от инкапсуляции – более аккуратная структура кода и меньшее количество внутренних взаимосвязей в программе.

Наблюдение: часто применение инкапсуляции может привести к ускорению программы.

Вопрос: за счет чего?

Преимущества от инкапсуляции

Основное преимущество от инкапсуляции – более аккуратная структура кода и меньшее количество внутренних взаимосвязей в программе.

Наблюдение: часто применение инкапсуляции может привести к ускорению программы.

Вопрос: за счет чего?

Ответ: за счет возможности контролировать и прогнозировать состояние объектов.

1. можно убрать ненужные проверки на корректность данных (классы за счет инкапсуляции данных позволяют гарантировать корректность состояния объектов);
2. можно исключить ненужные копирования данных (класс за счет инкапсуляции может контролировать доступ к данным, что позволяет использовать данные большого объема совместно в нескольких объектах).

Важный пример: классы для хранения строк в C++ и Java.

Пример: код на Java работает в 15-20 раз быстрее!

C++:

```
#include "string"
#include "iostream"
#include "windows.h"
using namespace std;

int main(int argc, char* argv[])
{
    long lTime = GetTickCount();

    string test("Hello world!");
    for (int iIdx = 0;
        iIdx < 1000000; iIdx++)
    {
        test.substr(1, 10);
    }
    lTime =
        GetTickCount() - lTime;

    cout << lTime << endl;
    return 0;
}
```

Причина в разном устройстве классов!

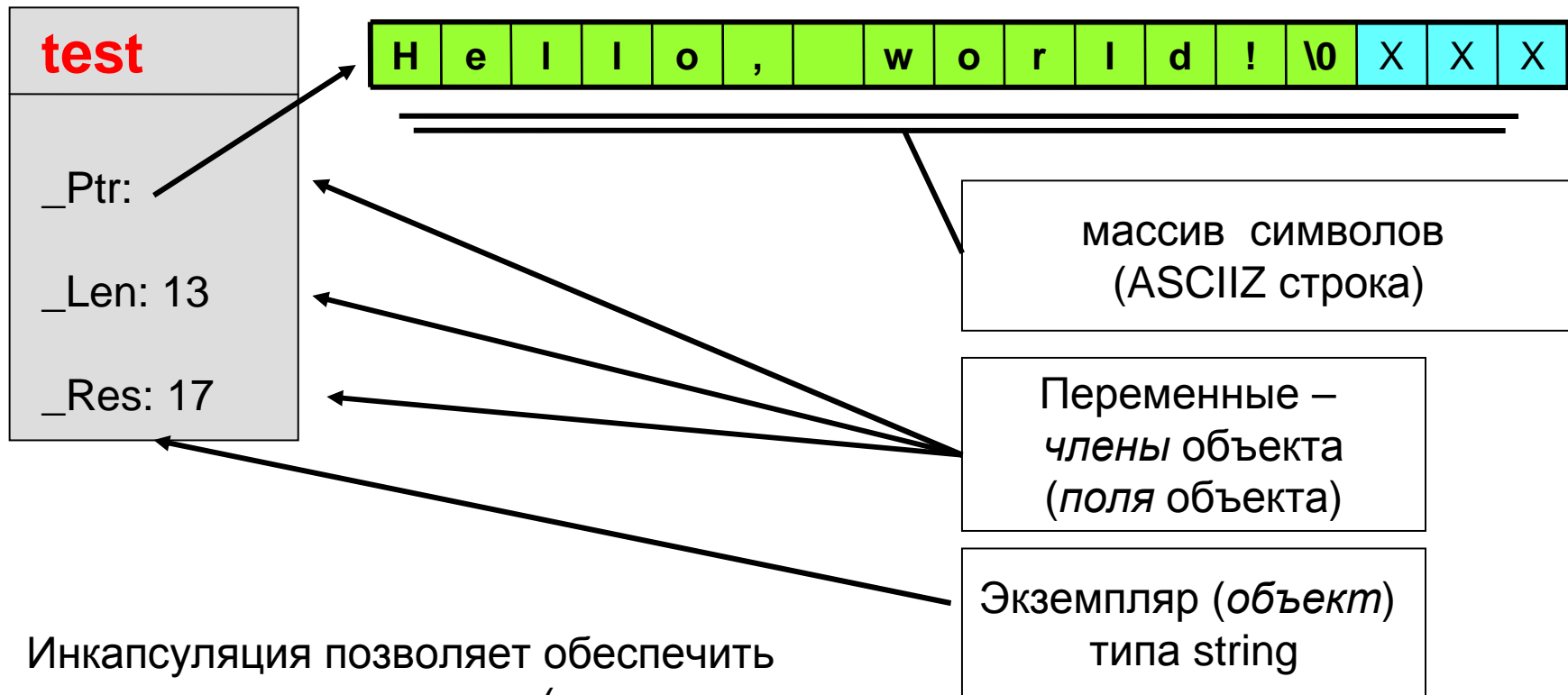
Java:

```
public class Test
{
    public static void main(String [] args)
    {
        long lTime =
            System.currentTimeMillis();
        String strTest = "Hello world!";
        for (int iIdx = 0;
            iIdx < 1000000; iIdx++)
        {
            strTest.substring(1, 11);
        }
        lTime =
            System.currentTimeMillis() - lTime;

        System.out.println(""+lTime);
    }
}
```

Для эксперимента нужна
Java версии 6 или ниже

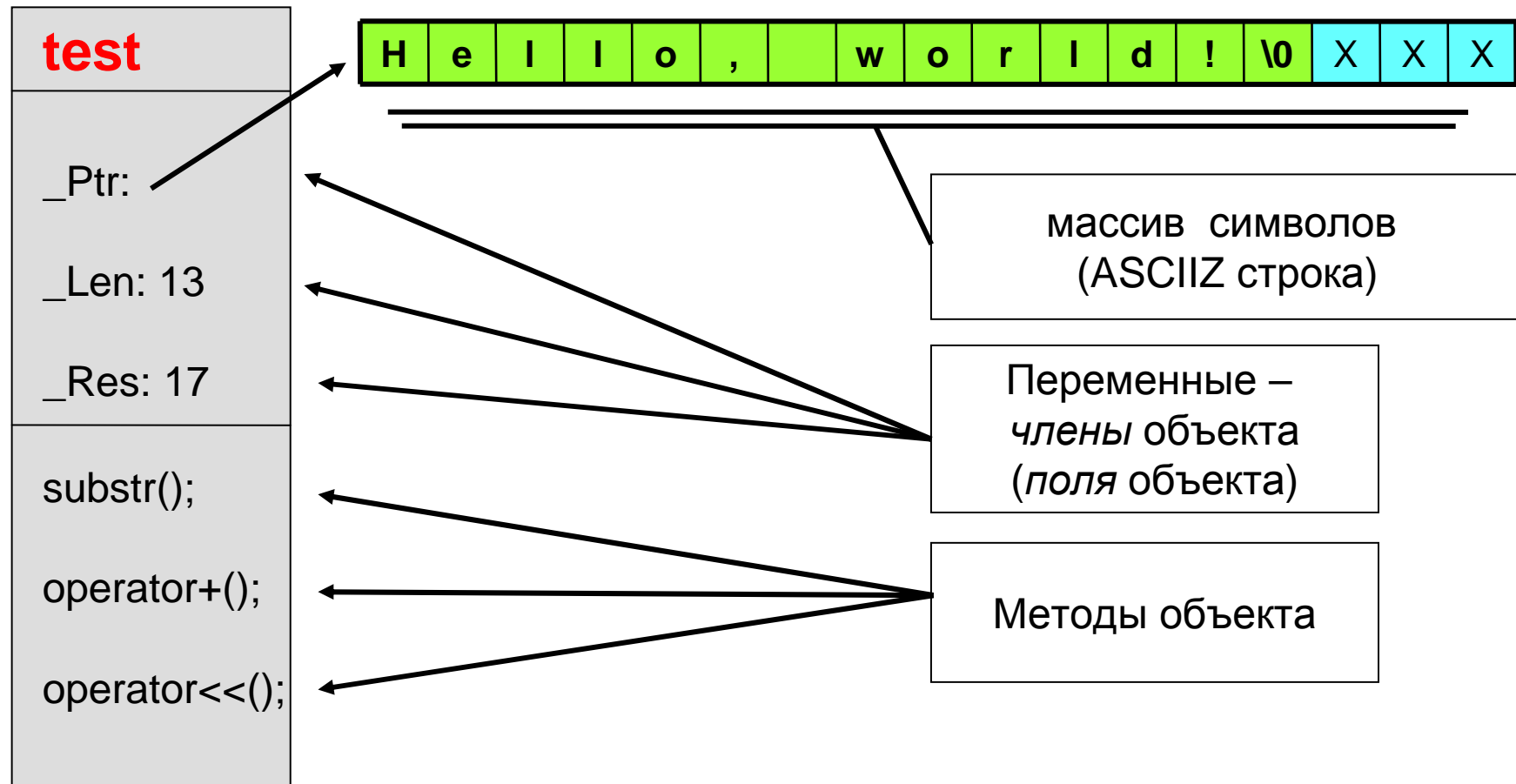
Внутри типа string (C++):



Инкапсуляция позволяет обеспечить согласованность данных (длина строки не превышает размер выделенного сегмента памяти).

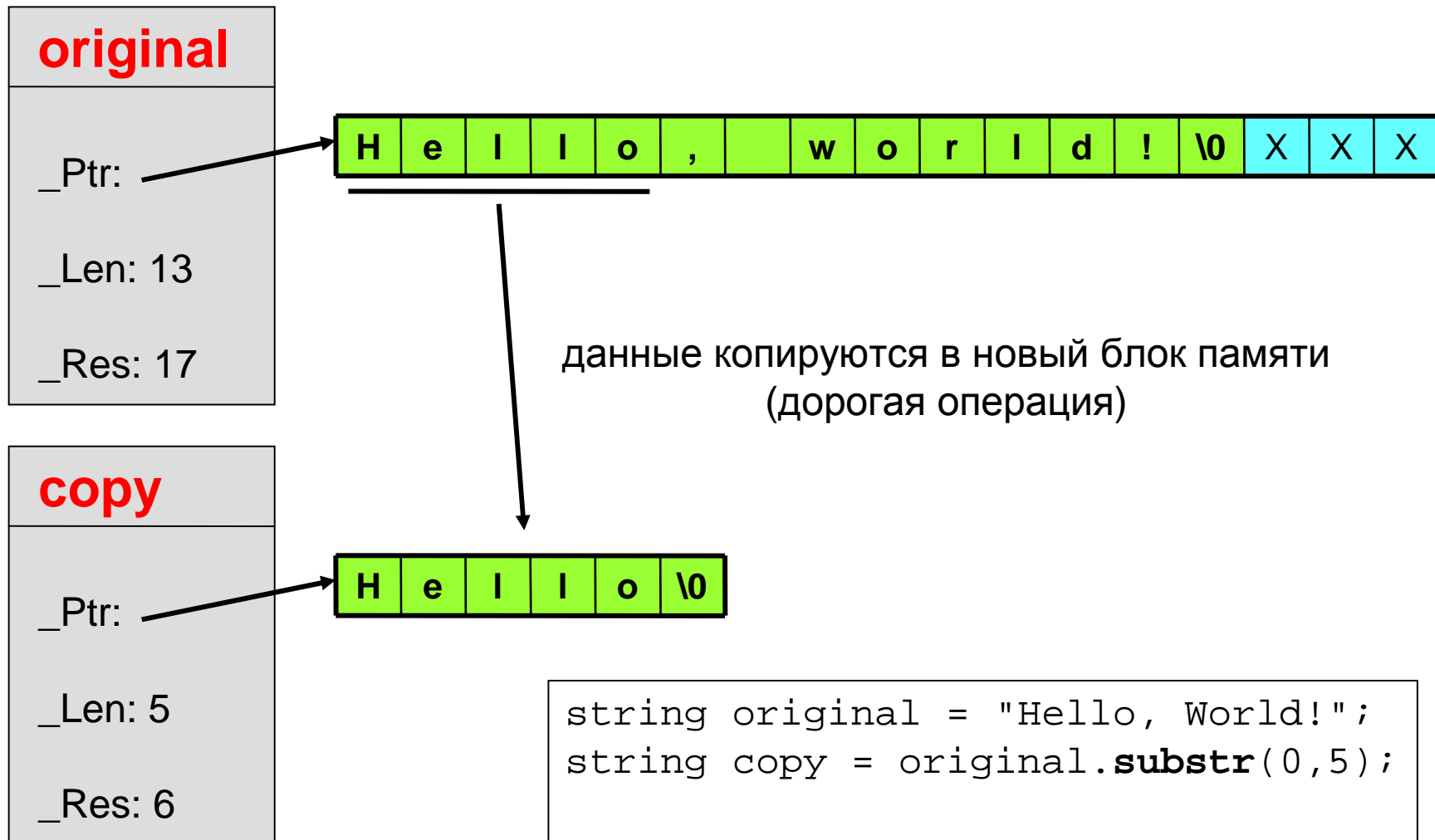
Прямой доступ к полям привел бы к разрушению состояния объекта.

Внутри типа string (C++):



Методы служат для изменения **состояния** объекта, которое определяется содержимым полей объекта

Получение подстроки (C++):



Наблюдение:

Организация класса `string` в C++ требует выделения нового блока памяти и копирования информации при каждой операции взятия подстроки. То есть эта операция оказывается дорогой.

Вопрос:

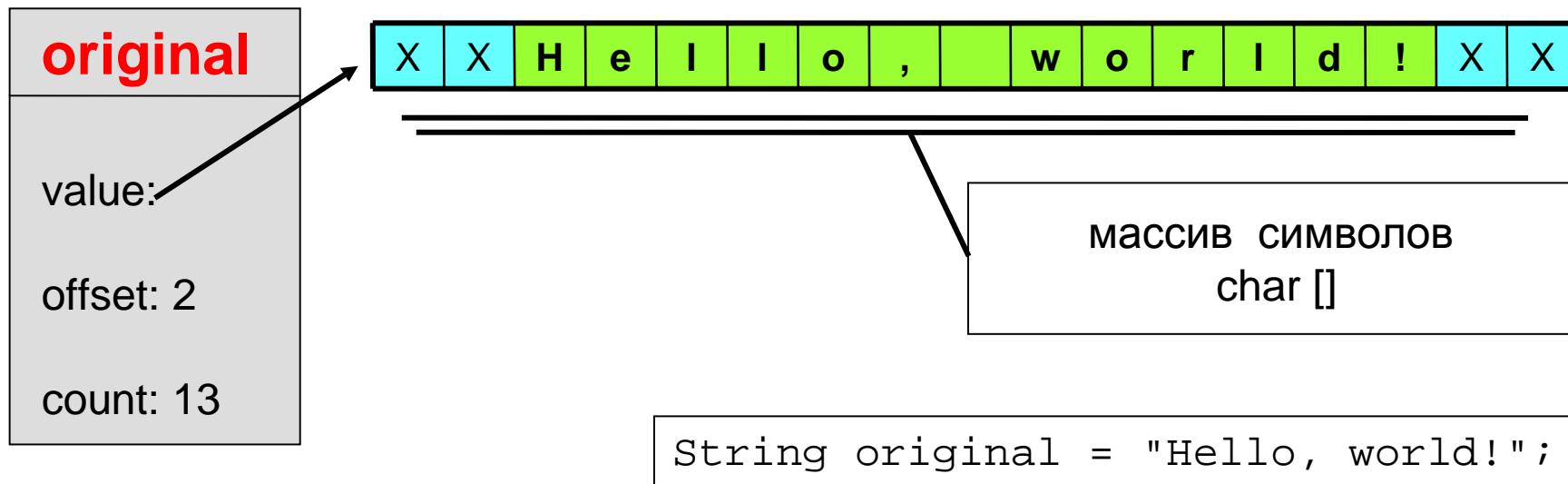
Каким образом можно сделать эту операцию дешевой?

Ответ:

Использовать общую область памяти для хранения данных исходной и производной строк!

Но тогда нужно обеспечить неизменность этой общей области памяти, то есть применить подход (шаблон проектирования), который называется **неизменный класс** (immutable class). Именно этот подход применяется в Java.

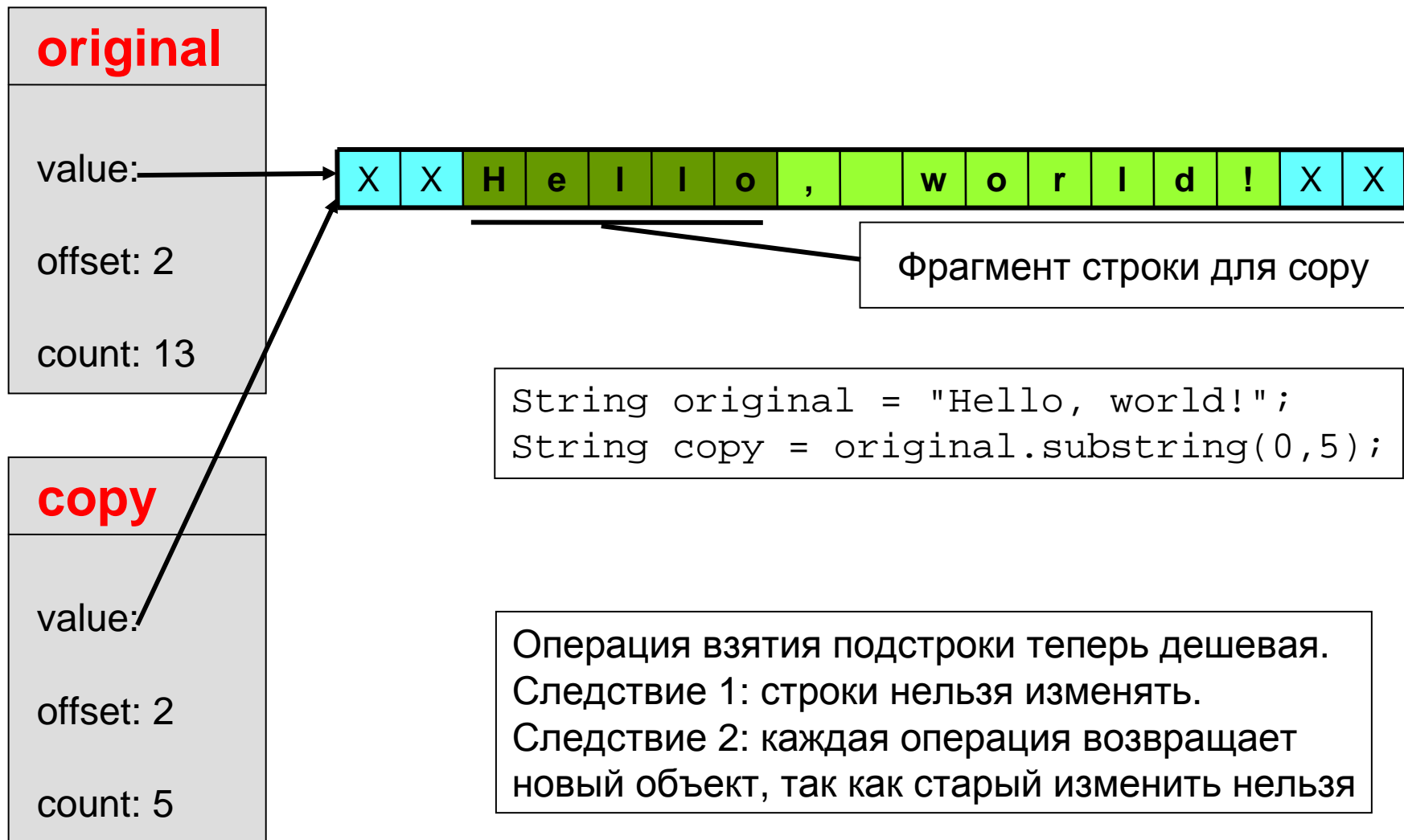
Внутри типа String (до Java 7):



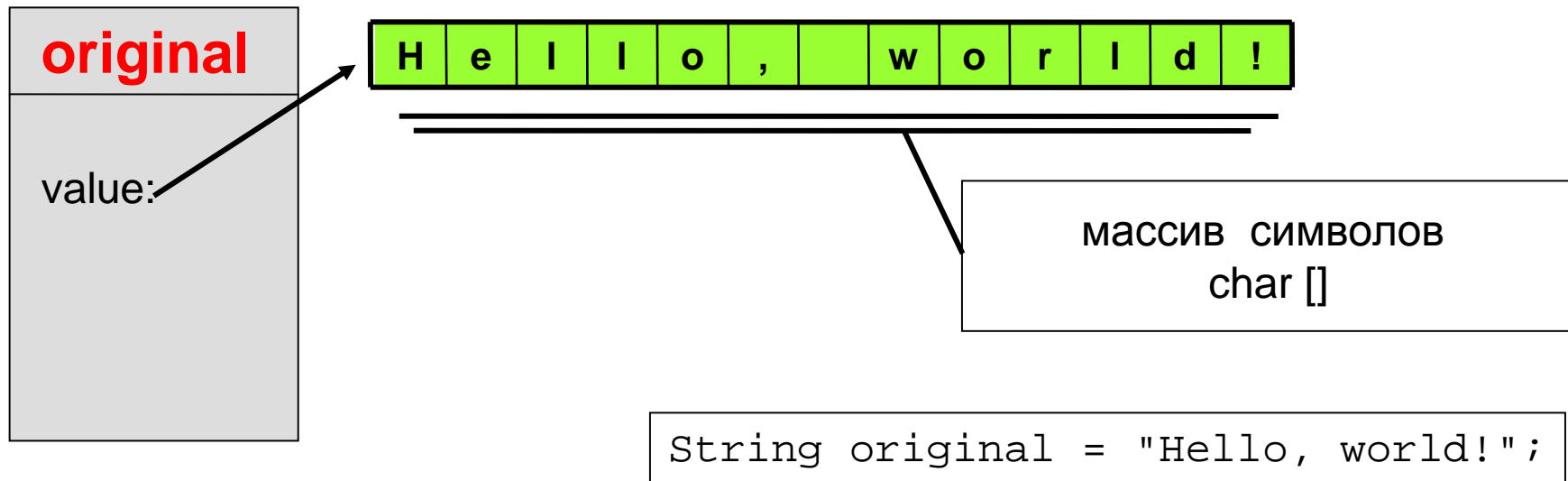
Наблюдения:

- отсутствует символ-терминатор
- отсутствует информация о длине массива (в Java это атрибут массива)

Внутри типа String (до Java 7):



Внутри типа String (Java 7 и выше):



Наблюдения:

- + экономим память в объекте (нет полей со смещением и длиной)
- + экономим память в массиве (нет неиспользуемых ячеек)
- + можем создавать строки в стеке (нет совместного использования)
- снижение быстродействия (требуется копировать данные)

Строки по-прежнему нельзя изменять!

Шаблон «Неизменный объект»

Строки в Java построены с использованием шаблона **«неизменный объект» (Immutable object)**.

Суть этого шаблона в том, что объект, построенный по этому шаблону, не содержит методов, позволяющих изменить его состояние (а прямое изменение членов объекта невозможно в силу применения инкапсуляции).

Получается, что объект, будучи созданным, в дальнейшем не может быть изменен.

Преимущества подхода:

1. При передаче неизменного объекта как параметра в подпрограмму нет необходимости создавать его копию, чтобы предотвратить объект от возможных его модификаций в подпрограмме.
2. Если неизменный объект хранит данные в каких-то внутренних структурах (например буфер в классе String), то эти внутренние структуры можно разделять между несколькими экземплярами объектов класса (например, при взятии подстроки объекты класса String могут совместно использовать один буфер).

Как манипулировать со строками в Java

Класс String в Java не позволяет изменять строки.

Для изменения строк следует использовать класс StringBuilder (или StringBuffer). Этот класс позволяет произвести изменения в строке, а затем перейти к неизменному, потенциально более эффективному, представлению (преобразовать свое состояние в класс String).

Пример:

```
StringBuilder strBuffer = new StringBuilder(20);  
strBuffer.append("Hello, ");  
strBuffer.append("World!");  
String strMessage = strBuffer.toString();  
System.out.println(strMessage);
```

Класс StringBuilder является более быстрым, но при этом не является потокобезопасным (не гарантирует корректной работы в случае одновременного использования из нескольких программных потоков).

Как манипулировать со строками в Java

Наблюдение: операция + для строк сводится к использованию класса `StringBuilder`. Поэтому следующие фрагменты кода эквивалентны:

```
String strMsg1 = "Hello, ";  
String strMsg2 = "World!";  
String strMessage = strMsg1 + strMsg2;
```

```
String strMsg1 = "Hello, ";  
String strMsg2 = "World!";  
StringBuilder strBuffer = new StringBuilder(strMsg1);  
strBuffer.append(strMsg2);  
String strMessage = strBuffer.toString();
```

Вывод:

Если требуется провести серию конкатенаций, может оказаться выгодно явно создать объект `StringBuilder` с достаточным размером буфера, чтобы минимизировать затраты на перевыделение памяти в процессе выполнения операций `append()`.

Классы Java, реализованные по шаблону «неизменный класс»

- Контейнеры для примитивных типов данных (Integer, Long, Short, Byte, Character, Boolean, Float, Double)
- Классы, реализующие арифметику неограниченной точности (BigInteger и BigDecimal)

Языки и технологии программирования.

Объектно-ориентированное программирование

Объекты как контейнеры данных

Использование данных объекта в методах

Как в методе добраться до полей объекта?

C#:

```
public class Test
{
    private int iValue;

    public void SetValue(int iValue)
    {
        this.iValue = iValue;
    }

    public int GetValue()
    {
        return iValue;
    }
}
```

Java:

```
public class Test
{
    private int iValue;

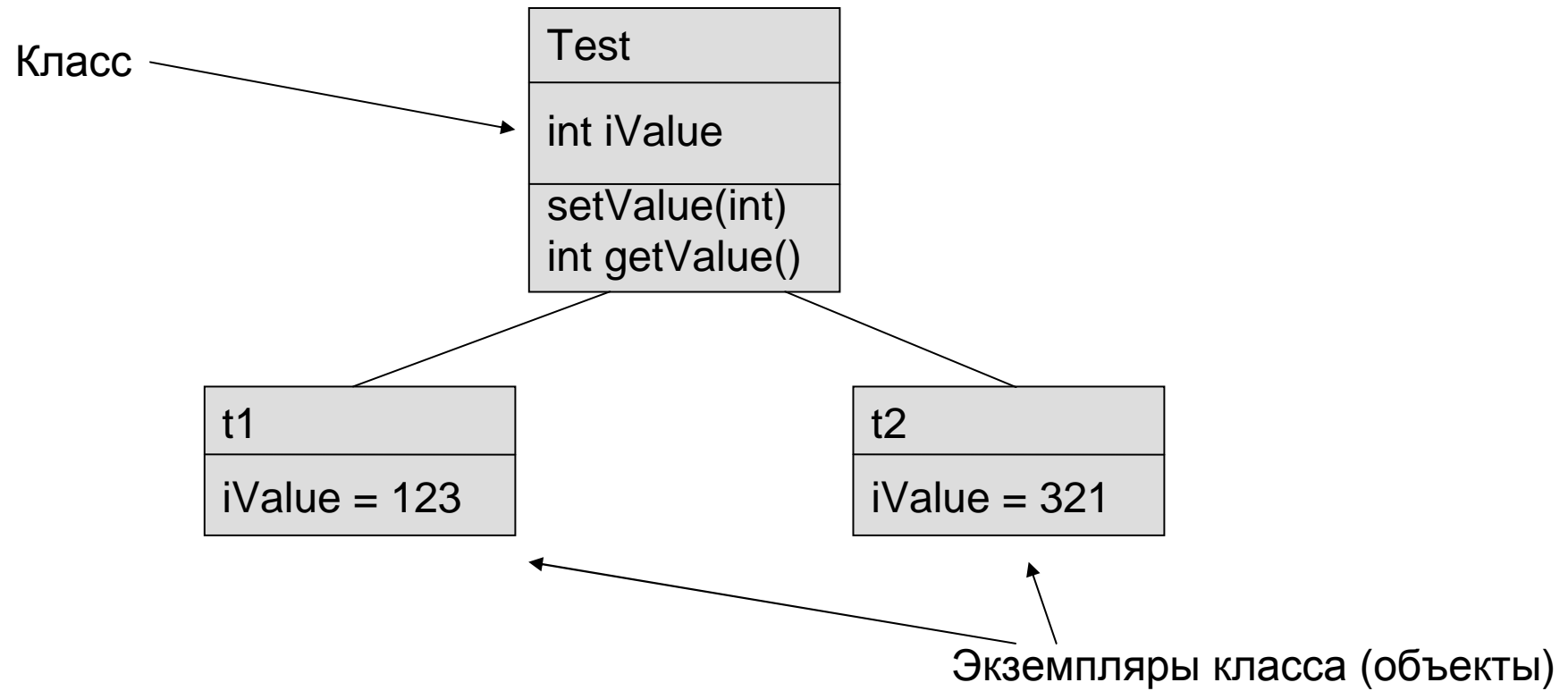
    public void setValue(int iValue)
    {
        this.iValue = iValue;
    }

    public int getValue()
    {
        return iValue;
    }
}
```

Наблюдение: в данном коде искусственно создана ситуация конфликта имен. Лучше так не делать.

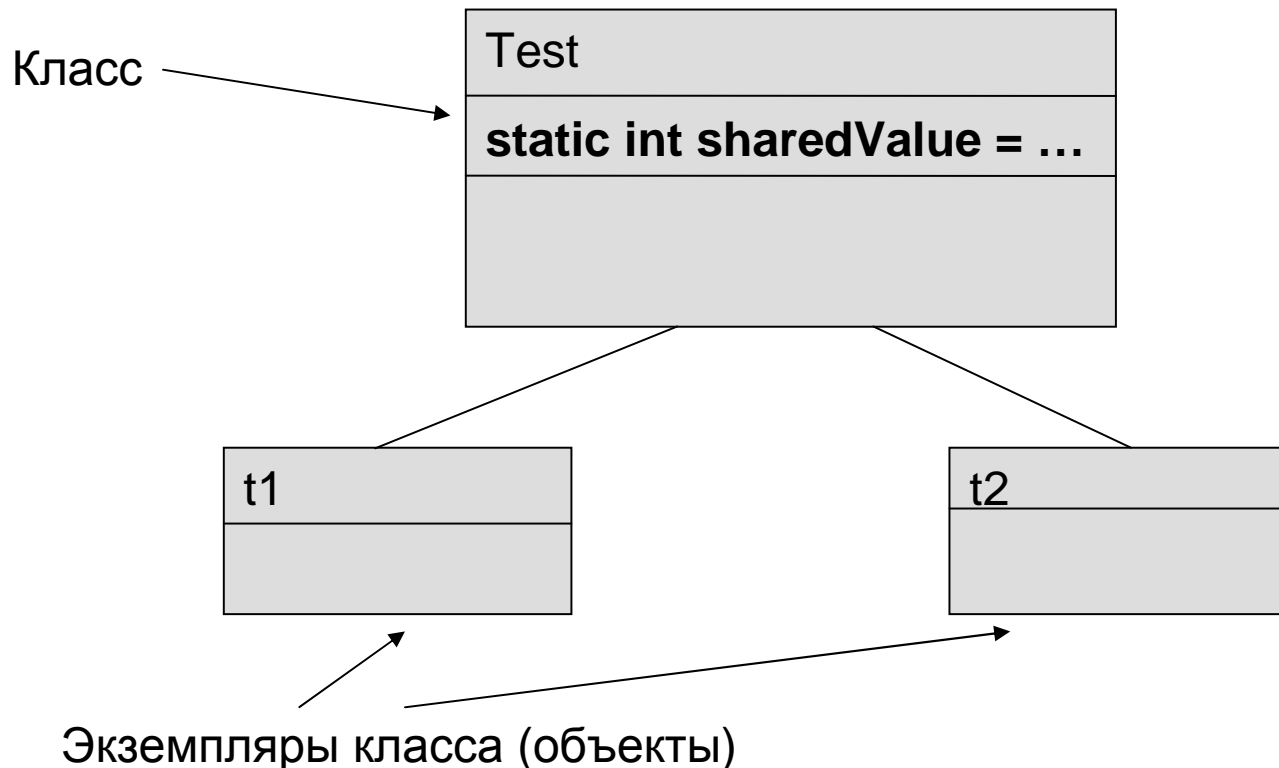
Java (C# - аналогично):

```
public static void main(String [] args)
{
    Test t1 = new Test();
    Test t2 = new Test();
    t1.setValue(123);
    t2.setValue(321);
}
```



Использование ключевого слова static

Статические члены класса связаны с классом, а не с конкретным объектом. Соответственно, **статические поля** размещаются в области памяти класса, и могут быть использованы совместно всеми объектами класса. **Статические методы** также не связаны с конкретным экземпляром класса, то есть они не требуют объекта при вызове и могут работать только со статическими полями:



Использование ключевого слова static

Java:

```
public class Test
{
    private static int sharedValue;
    private int localValue;

    public void setValues(int iValue)
    {
        Test.sharedValue = iValue;
        localValue = iValue;
    }

    public int getSharedValue()
    {
        return Test.sharedValue;
    }

    public int getLocalValue()
    {
        return localValue;
    }
}
```

C#:

```
public class Test
{
    private static int sharedValue;
    private int localValue;

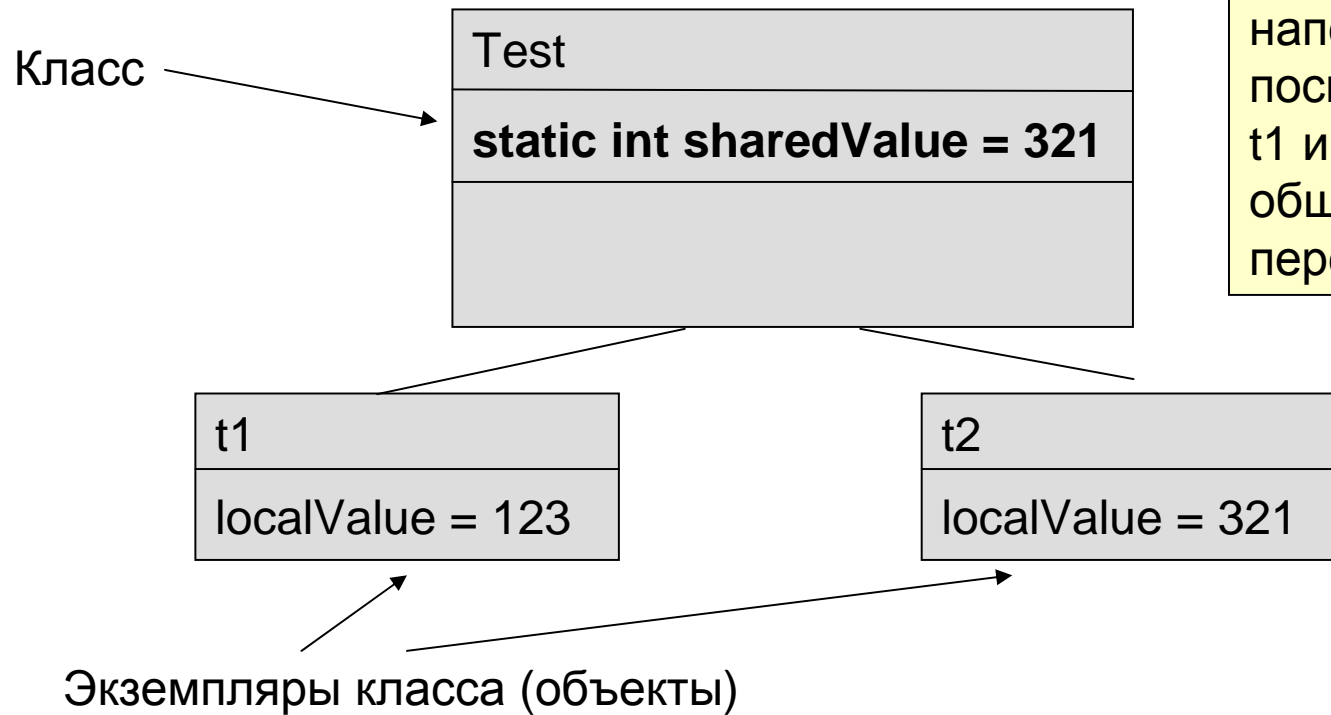
    public void SetValues(int iValue)
    {
        Test.sharedValue = iValue;
        localValue = iValue;
    }

    public int GetSharedValue()
    {
        return Test.sharedValue;
    }

    public int GetLocalValue()
    {
        return localValue;
    }
}
```


Java:

```
public static void main(String [] args)
{
    Test t1 = new Test();
    Test t2 = new Test();
    t1.setValues(123);
    t2.setValues(321);
    System.out.println(t1.getSharedValue());
}
```



На экране будет напечатано 321, поскольку объекты t1 и t2 используют общую статическую переменную.

Константы в Java. Ключевое слово final

Константы в Java реализуются как члены класса с модификаторами `static` и `final`.

Модификатор `static` позволяет использовать константу без создания экземпляра класса.

Модификатор `final` запрещает изменение значения члена класса (то есть, собственно, и формирует константу).

```
public class ConstTest
{
    public static final double PI = 3.14159265358979323846;

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

Константы в C#. Ключевые слова readonly и const

Константы в C# реализуются как члены класса с модификатором const (для примитивных типов) или комбинацией модификаторов static и readonly (для ссылочных объектных типов).

Модификатор static позволяет использовать константу без создания экземпляра класса.

Модификатор readonly запрещает изменение значения члена класса (то есть, собственно, и формирует константу).

Модификатор const по сути формирует макроподстановку (константа подставляется во все точки использования).

```
public class ConstTest
{
    public const double PI = 3.14159265358979323846;
    public static readonly MESSAGE = "Value of Pi is: ";

    public static int Main(String [] args)
    {
        Console.WriteLine(ConstTest.MESSAGE + ConstTest.PI);
    }
}
```

Языки и технологии программирования.

Объектно-ориентированное программирование

Пакеты в Java

Полное имя каждого класса в Java состоит из двух частей:



Язык Java **требует** уникальности полного имени класса.

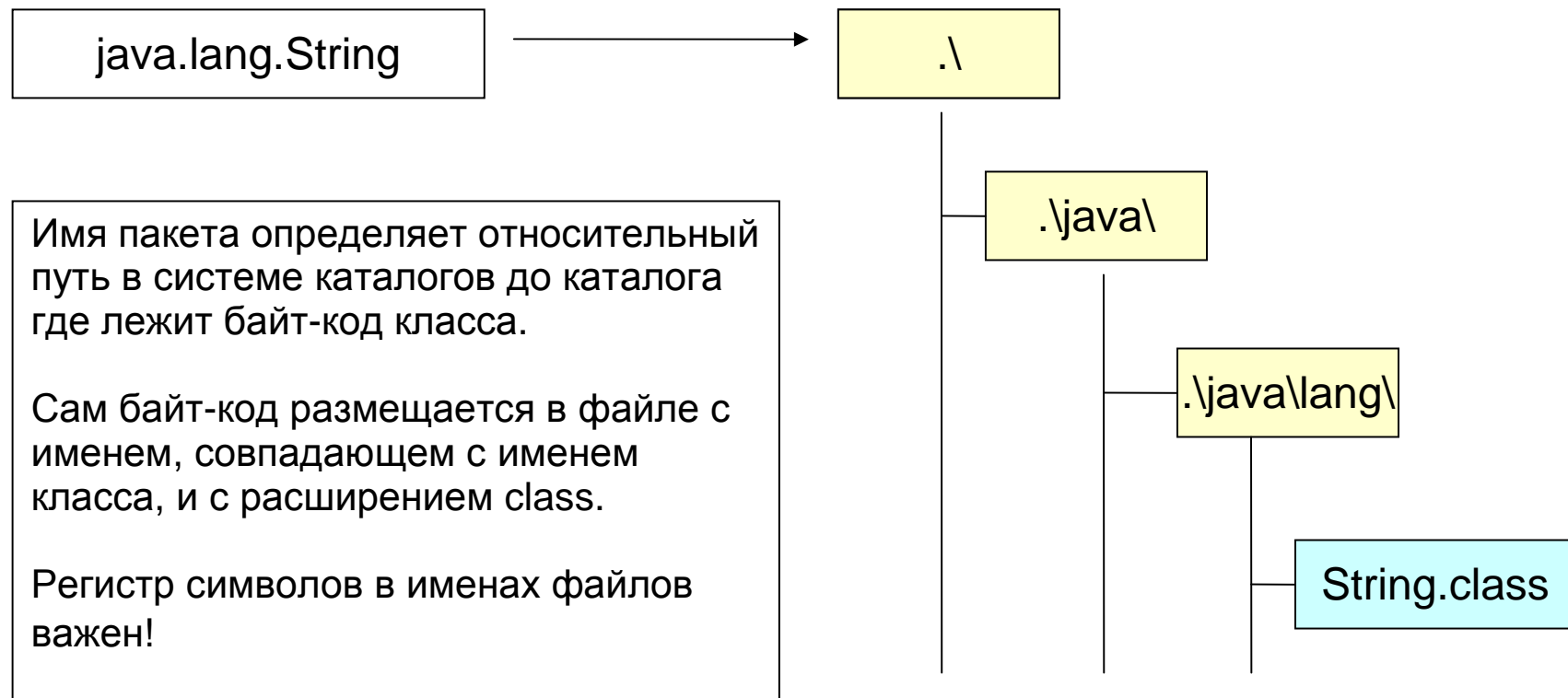
Это требование обусловлено тем, как JVM ищет байт-код класса, который нужно загрузить в память.

При загрузке класса производятся следующие шаги:

- специальному модулю (ClassLoader) дается команда на поиск и загрузку байт-кода класса с заданным полным именем (наблюдение: если имя класса не является уникальным, то операция некорректна)
- загруженный байт-код проверяется на целостность и корректность
- загруженный байт-код по возможности компилируется в машинный код для ускорения работы
- названный алгоритм повторяется для всех классов, на которые есть ссылки из загруженного класса

Загрузчики классов могут быть разными (например, байт-код можно загружать из базы данных или из сети).

Стандартный загрузчик классов работает так:



Принадлежность класса пакету определяется ключевым словом **package**

```
package java.lang;
public final class String
{
    ...
}
```

The diagram consists of two rectangular boxes with arrows pointing to specific parts of the code. The first box, labeled "имя пакета" (package name), has an arrow pointing to the text "java.lang" in the "package" statement. The second box, labeled "имя класса в пакете" (class name in package), has an arrow pointing to the text "String" in the "class" declaration.

Пусть хотим создать класс `some.package.MyClass`. Тогда:

- Файл с текстом класса должен размещаться здесь:
`.\some\package\MyClass.java`
- Для того, чтобы откомпилировать класс, требуется дать команду
`javac .\some\package\MyClass.java`
- Для того, чтобы запустить класс, требуется дать команду
`java some.package.MyClass`

Импортирование классов из пакетов

```
import java.io.Reader;  
import java.util.*;
```

```
public final class String  
{  
  
    java.io.InputStream is = ...  
  
    Reader r = ...  
  
    ArrayList list = ...  
  
    ...  
}
```

класс идентифицируется полным именем,
импорт необязателен

класс идентифицируется кратким именем,
(импорт через имя класса:
import java.io.Reader)

класс идентифицируется кратким именем,
(импорт через имя пакета: import java.util.*)

Инструкция **import** сообщает компилятору правило поиска полного имени класса по его короткому имени (так, например, в примере выше короткому имени `Reader` в коде соответствует полное имя класса `java.io.Reader`).

Языки и технологии программирования.

Объектно-ориентированное программирование

*Основные понятия ООП:
Абстракции и интерфейсы*

Основные понятия ООП

- Инкапсуляция
- **Абстракция**
- **Интерфейс**
- Полиморфизм
- Наследование

Задача:

написать библиотеку шифрования/дешифрования данных

Форматы кодирования:

DES, Triple DES, Blowfish, IDEA

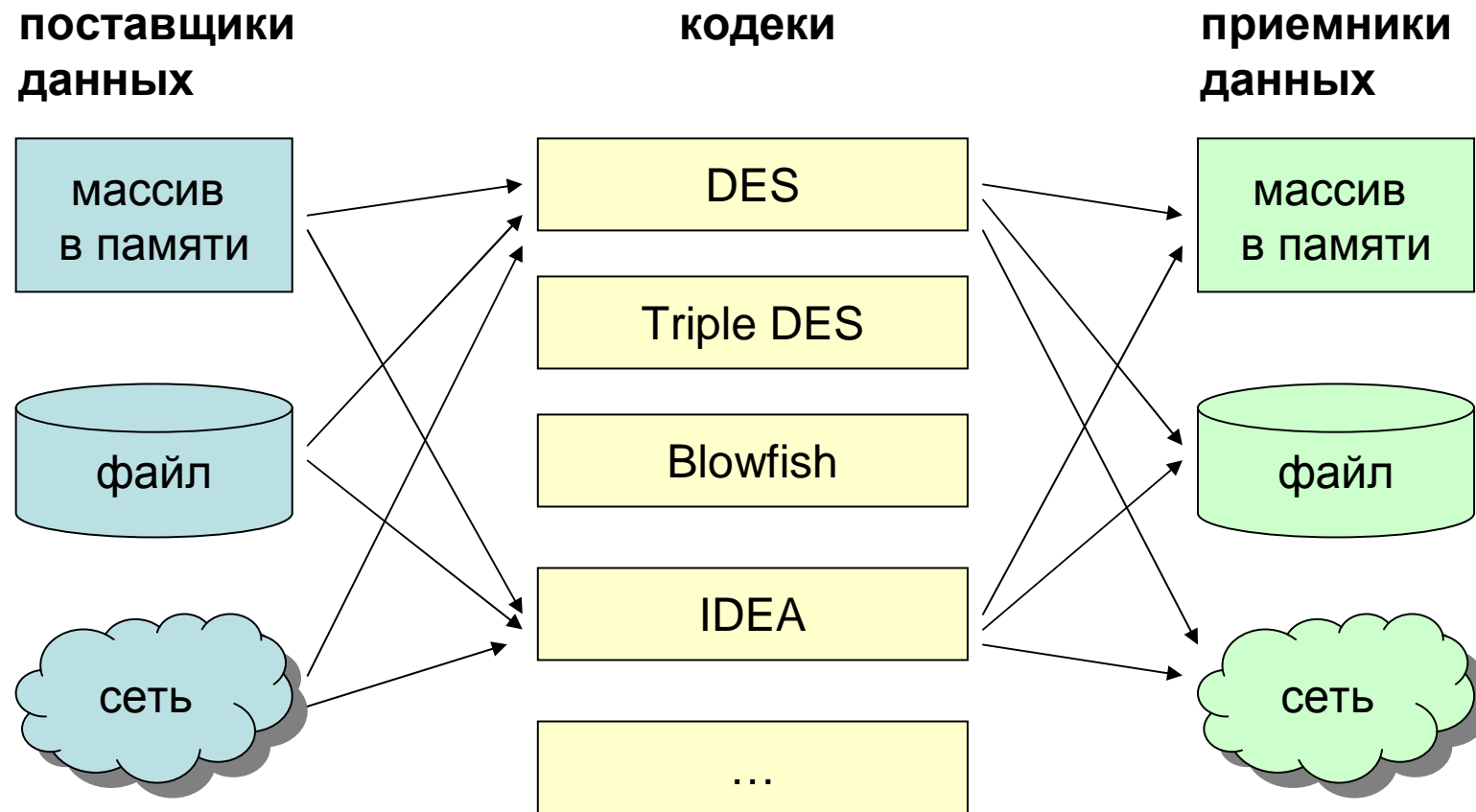
Входные форматы:

массив в памяти, файл на диске, данные из сети

Выходные форматы:

массив в памяти, файл на диске, посылка данных в сеть

На диаграмме задача выглядит так:



Применение структурного подхода приводит к серии подпрограмм:

```
encodeDESMemoryToMemory(...);  
encodeDESMemoryToFile(...);  
encodeDESMemoryToNet(...);  
encodeDESFileToMemory(...);  
...
```

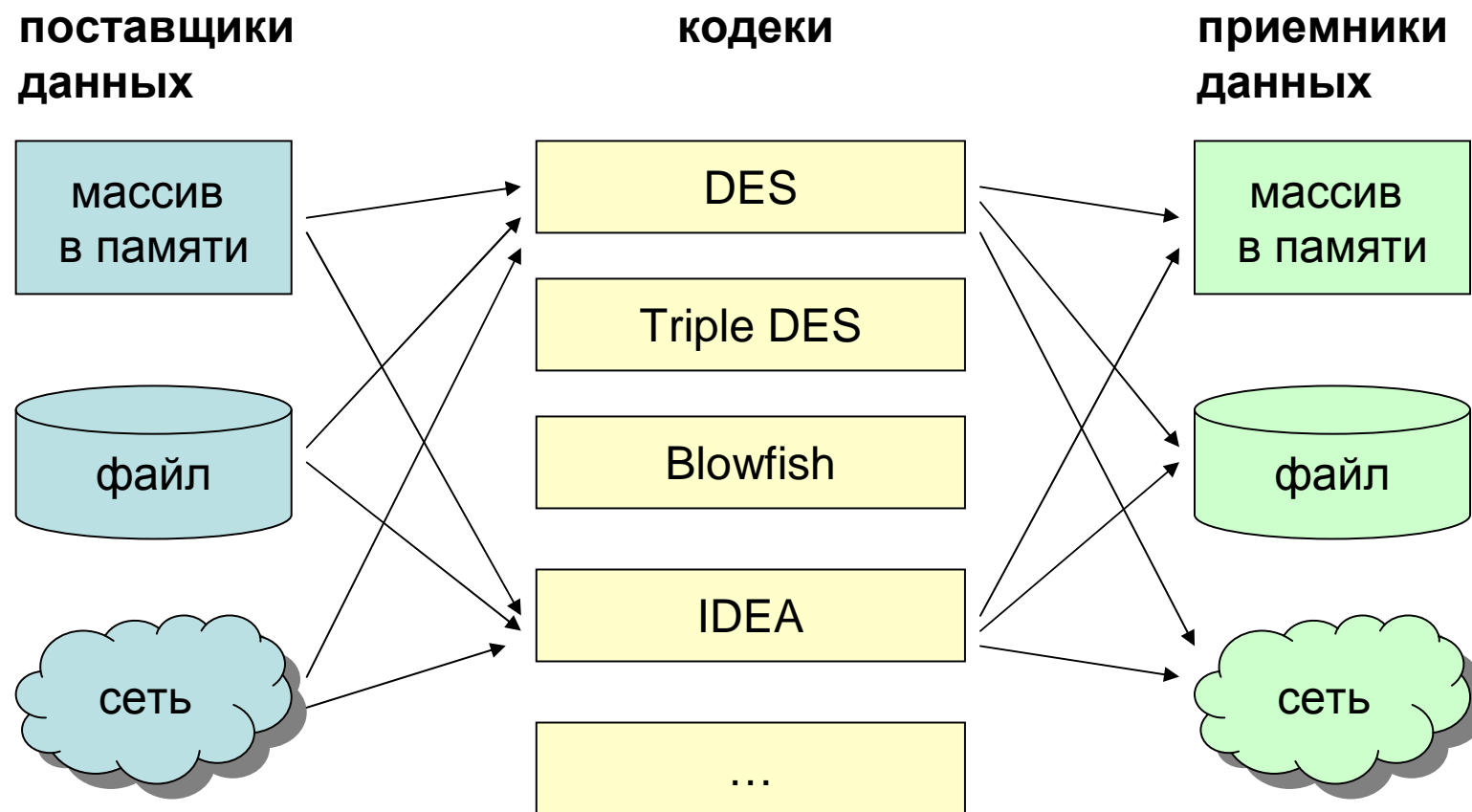
Всего $3 * 4 * 3 = 36$ подпрограмм.

Недостатки подхода:

- большой объем похожего кода
- сложность отладки (надо проверить все 36 вариантов)
- сложность поддержки (для добавления нового формата кодирования требуется дописать 9 подпрограмм)

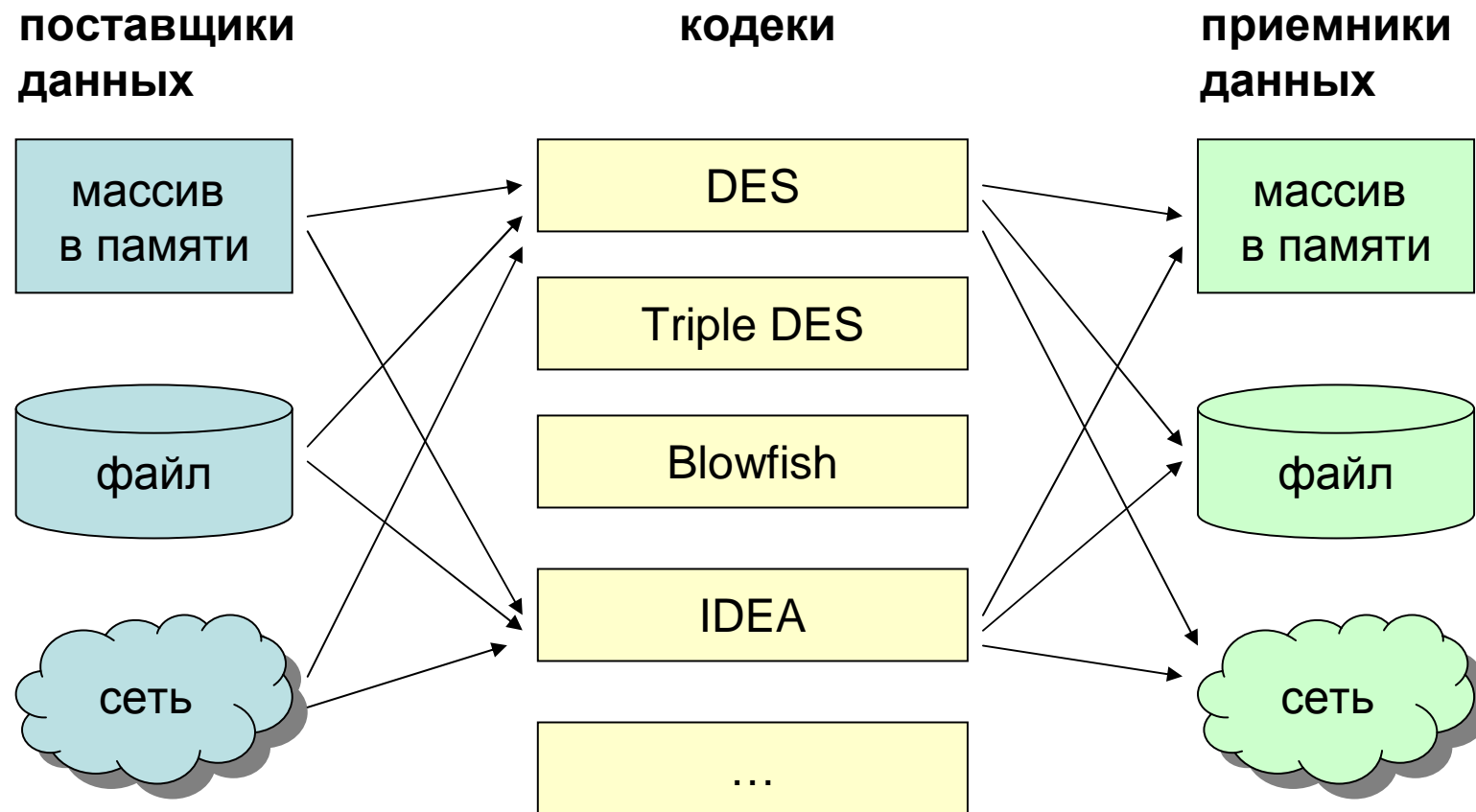
Вывод: структурное программирование плохо справляется с задачами, которые могут расширяться сразу в нескольких направлениях.

Попробуем выделить абстракции (ролевые сущности):



Что общего есть у всех поставщиков данных?

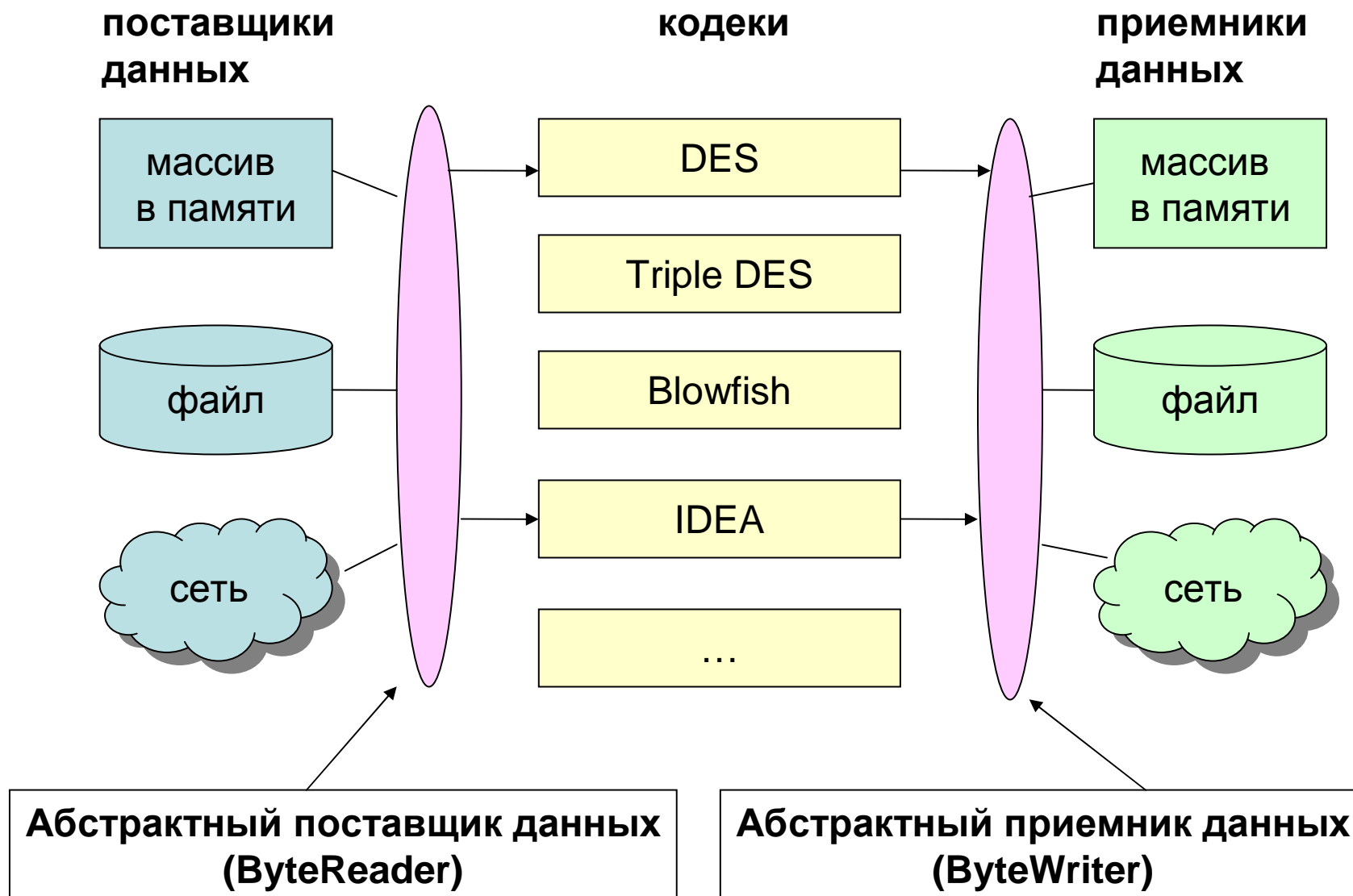
Попробуем выделить абстракции (ролевые сущности):



Что общего есть у всех поставщиков данных?

Способность читать данные байт за байтом!

Абстракции фиксируют общие свойства семейства сущностей:



Таким образом выделяются следующие абстракции (роли) и их реализации:

Роль:

ByteReader – выдает байт за байтом из некоторого источника

Реализации:

FileByteReader – выдает байт за байтом из некоторого файла

MemoryByteReader – выдает байт за байтом из массива в памяти

NetworkByteReader – выдает байт за байтом из сетевого канала связи

Роль:

ByteWriter – принимает байт за байтом для записи куда-то

Реализации:

FileByteWriter – пишет байт за байтом в некоторый файл

MemoryByteWriter – пишет байт за байтом в массив в памяти

NetworkByteWriter – передает байт за байтом через сетевой канал связи

Наблюдение 1:

Было бы удобно иметь возможность описать подпрограмму в терминах абстракций. Например, так:

```
void decodeDES(ByteReader input, ByteWriter output)
{
    while (input.hasMoreData())
    {
        byte val = input.readByte();
        ... // закодировать val
        output.writeByte(val);
    }
    output.flush();
}
```

И затем использовать его примерно следующим способом:

```
decodeDES(someFileByteReader, someMemoryByteWriter);
```

То есть получили бы возможность **передавать поведение** (абстракцию) в качестве параметра.

Наблюдение 2:

Сам алгоритм кодирования тоже является абстракцией. То есть, если выделить абстракцию `Encoder`, то код мог бы выглядеть так:

```
Encoder encoder = new DESEncoder();  
ByteReader input = new FileByteReader("some_file.txt");  
ByteWriter output = new MemoryByteWriter ();  
encoder.encode(input, output);
```

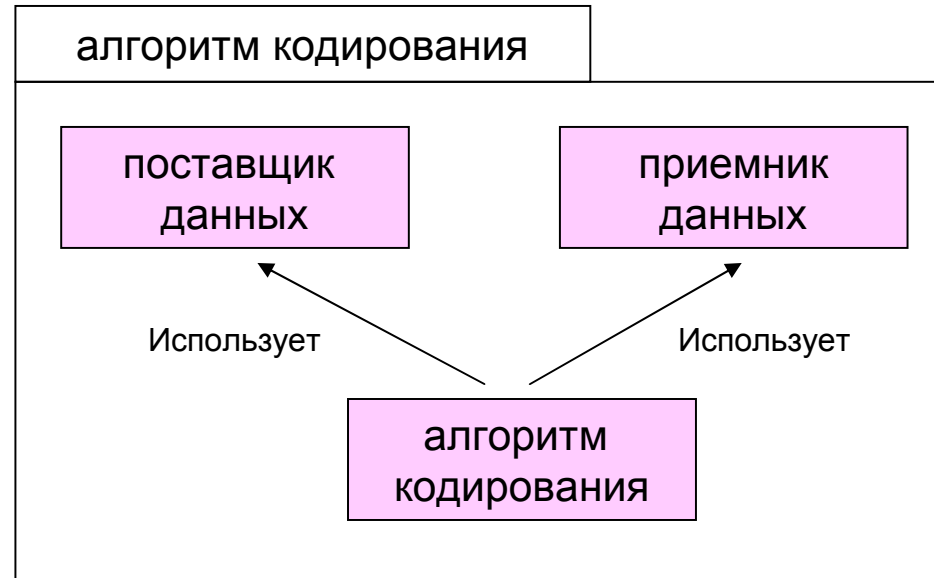
Вывод:

Чтобы защитить код от изменчивости условий его использования, его нужно писать так, чтобы он по возможности работал с абстракциями, а не с конкретикой.

Наблюдение 3:

Суть абстракции не в том, как она устроена внутри, а в том, что она умеет делать. То есть важны не детали реализации, а доступный набор операций (методов) – вспомним принцип инкапсуляции.

Введение абстракций позволяет строить модель программы с точки зрения поведения отдельных функциональных блоков:



Наблюдение: модель программы позволяет моделировать поведение программы (проводить функциональную отладку еще до создания реального кода!), тем самым, давая возможность выявить логические проблемы на ранней стадии.

Наблюдение: для описания модели особенности реализации каждой из абстракций не важны (работает инкапсуляция)!

Абстракции в объектно-ориентированных языках программирования

Чтобы язык программирования позволял работать с абстракциями, в нем должны быть следующие возможности:

- определять свои (новые) абстракции;
- писать код в терминах абстракций;
- создавать конкретные реализации абстракций.

Рассмотрим, как эти возможности реализованы в языке Java.

Определение абстракций

Абстракция характеризуется своим набором операций, то есть внешним **интерфейсом** (интерфейс = набор операций). Соответственно, описание интерфейса похоже на описание методов класса. Вот так, например, выглядят описания абстракций `ByteReader` и `ByteWriter` в языке Java.

```
public interface ByteReader
{
    byte readByte();
    boolean hasMoreData();
}
```

Наблюдения:

← ключевое слово `interface`
а не `class`

```
public interface ByteWriter
{
    void writeByte(byte value);
    void flush();
}
```

← у методов нет тела
(поскольку это абстракция)

Интерфейс определяет контракт (функциональность), который должен выполняться любой реализацией этого интерфейса.

Реализация абстракций

Для реализации абстракции (интерфейса) требуется:

- формально зафиксировать, что класс реализует абстракцию;
- реализовать функциональность, декларируемую абстракцией.

```
public class FileReader implements Reader
{
    byte readByte()
    { код метода }

    boolean hasMoreData()
    { код метода }
}
```

формальная декларация,
что класс реализует интерфейс

реализация всех (!) методов
интерфейса

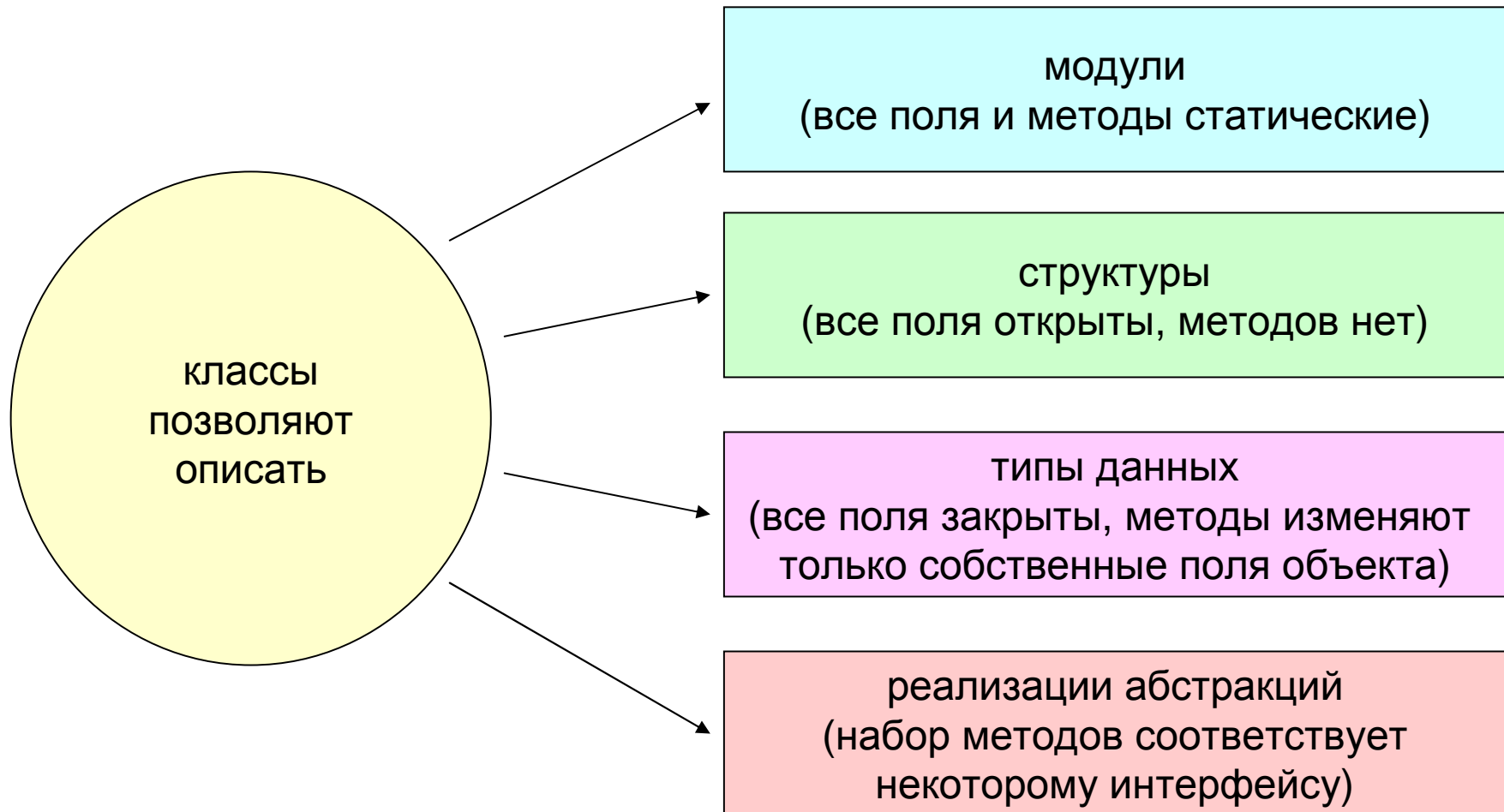
То есть такой класс `FileReader` реализует интерфейс `Reader`.

Вывод:

Классы могут реализовывать интерфейсы.

Таким образом, классы – это способ создания конкретных реализаций абстракций.

Использование классов:



Языки и технологии программирования.

Объектно-ориентированное программирование

*Основные понятия ООП:
интерфейсы, наследование и
полиморфизм*

Основные понятия ООП

- Инкапсуляция
- Абстракция
- **Интерфейс**
- **Полиморфизм**
- **Наследование**

Задача:

написать библиотеку шифрования/дешифрования данных

Форматы кодирования:

DES, Triple DES, Blowfish, IDEA

Входные форматы:

массив в памяти, файл на диске, данные из сети

Выходные форматы:

массив в памяти, файл на диске, посылка данных в сеть

абстракции являются точками сочленения функциональности

**поставщики
данных**

массив
в памяти

файл

сеть

Интерфейс ByteReader

кодеки

DES

Triple DES

Blowfish

IDEA

...

**приемники
данных**

массив
в памяти

файл

сеть

Интерфейс ByteWriter

Интерфейс декларирует функциональность. То, что класс реализует заданный интерфейс, означает, что среди, возможно, большого числа методов класса есть набор методов, соответствующий интерфейсу.

В рамках рассматриваемой задачи для случая хранения данных в памяти реализации интерфейсов `ByteReader` и `ByteWriter` можно было бы объединить. То есть создать класс, реализующий сразу два интерфейса:

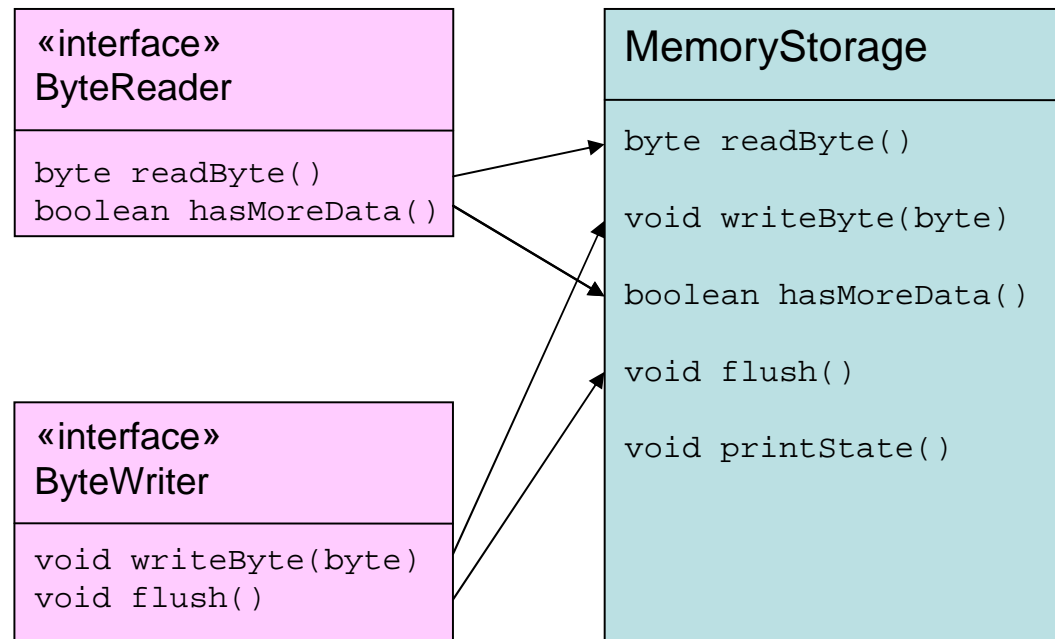
```
public class MemoryStorage
    implements ByteReader,
               ByteWriter
{
    byte readByte()
    { ... }

    void writeByte(byte value)
    { ... }

    boolean hasMoreData()
    { ... }

    void flush()
    { ... }

    void printState()
    { ... }
}
```



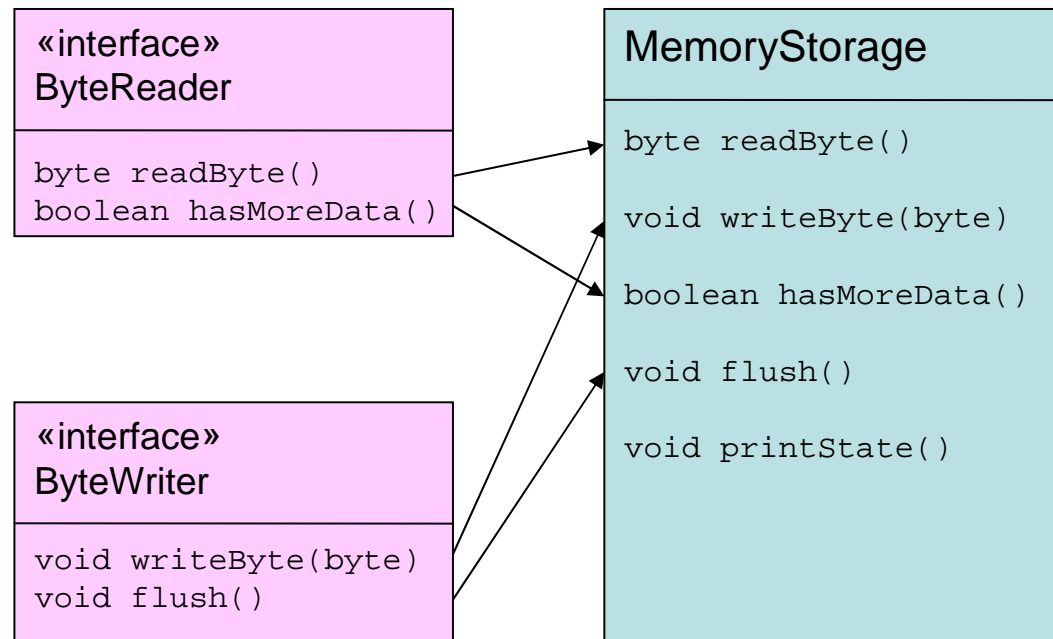
Теперь на класс `MemoryStorage` можно смотреть как на самостоятельный класс, как на реализацию интерфейса `ByteReader` и как на реализацию интерфейса `ByteWriter`.

```
// создали экземпляр класса
MemoryStorage storage = new MemoryStorage();
// корректные вызовы методов
storage.printState();
storage.hasMoreData();
```

```
// использование storage
// как реализации интерфейса
ByteReader reader = storage;
byte value =
    reader.readByte();
```

```
ByteWriter writer = storage;
writer.writeByte(value);
```

```
// некорректные вызовы
reader.flush();
writer.hasMoreData();
reader.printState();
```



Первое правило совместимости по присваиванию в ООП (правило согласованности интерфейсов)

Пусть TypeA – какой-то класс или интерфейс, а InterfaceB – какой-то интерфейс. Тогда присваивание значения переменной varB в коде

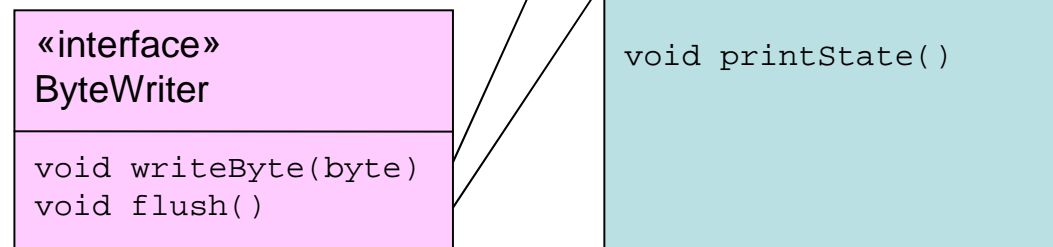
TypeA varA = ...;

InterfaceB varB = varA;

является корректным в том и только в том случае, когда TypeA реализует интерфейс InterfaceB.

```
// создали экземпляр класса
MemoryStorage storage = new MemoryStorage();
ByteWriter reader = storage;

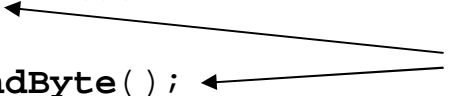
ByteWriter another = new MemoryStorage();
```



Первое правило согласованности интерфейсов позволяет писать код в терминах абстракций.

Вот так мог бы выглядеть код алгоритма кодирования данных в терминах введенных интерфейсов:

```
public void encodeXXX(ByteReader reader, ByteWriter writer)
{
    while (reader.hasMoreData())
    {
        byte b = reader.readByte();
        byte encodedByte = encodeByteXXX(b);
        writer.writeByte(encodedByte);
    }
    writer.flush();
}
```



В коде можно использовать методы,
описанные в интерфейсе

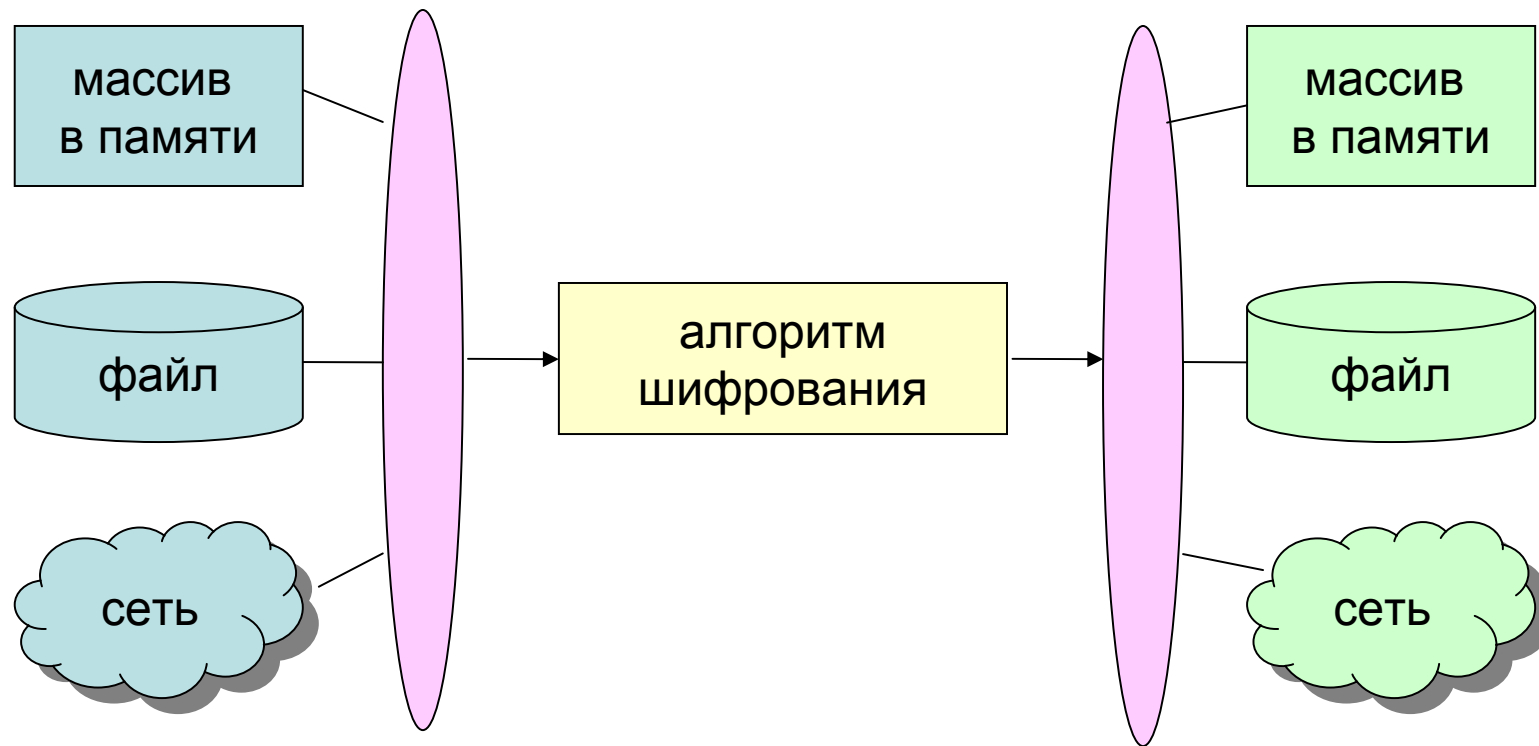
И код, его использующий:

```
MemoryStorage data = new MemoryStorage();
MemoryStorage encodedData = new MemoryStorage();
encodeXXX(data, encodedData);
```

Согласованность интерфейсов обеспечивает корректность кода.

Возможность взаимодействовать с объектом через реализуемый им интерфейс без привязки к специфике объекта называется **полиморфизмом**.

То есть полиморфизм позволяет работать с объектами различных классов одинаковым образом.



Интерфейсы позволяют отделить функциональные модули программы, скрыть детали реализации. В то же время в чистом виде использование интерфейсов может приводить к дублированию кода.

Например, для удобства создания новых алгоритмов кодирования было бы удобно иметь в составе интерфейса `ByteReader` следующие методы:

```
public interface ByteReader
{
    byte readByte();
    boolean hasMoreData();

    // читает не один байт, а заданное число байтов
    byte[] readBytes(int count);
    // читает заданное число байтов в переданный массив
    void readBytes(byte[] buffer, int startIndex, int count);
}
```

Отметим, что новые методы являются лишь более удобными аналогами первых двух методов.

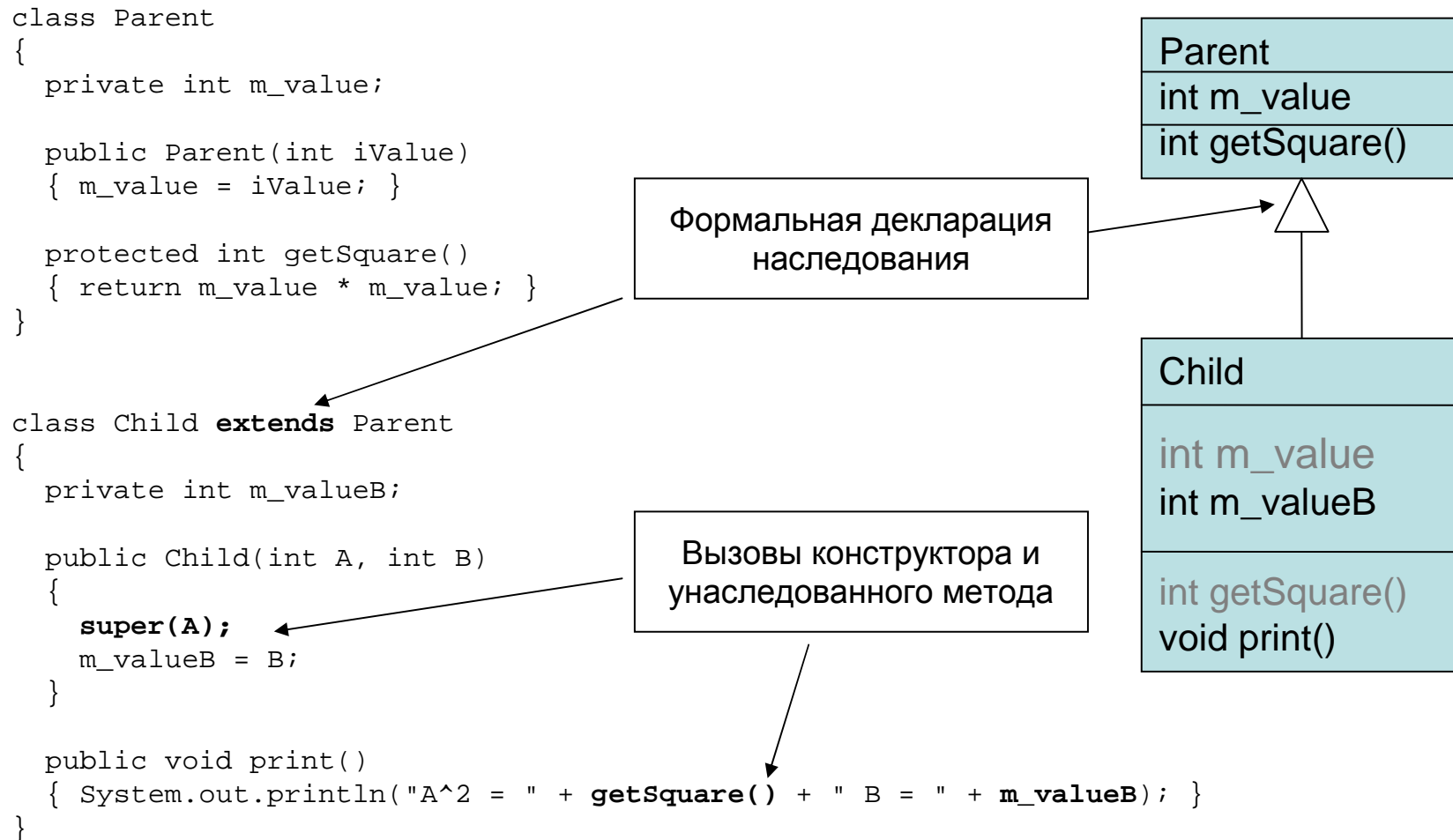
Заметим, что для новых методов для всех реализаций интерфейса `ByteReader` можно было бы написать примерно следующую реализацию:

```
byte[] readBytes(int count)
{
    byte[] buffer = new byte[count];
    for (int i = 0; i < count; i++)
    {
        if (hasMoreData())
            buffer[i] = readByte();
        else
            buffer[i] = 0;
    }
    return buffer;
}
```

Этот код годится для любой реализации интерфейса `ByteReader`, но при описании классов мы вынуждены его тиражировать.

Проблему тиражирования кода при реализации некоторого интерфейса позволяют решить применение механизма наследования классов и использование абстрактных классов.

Новые классы можно строить либо явным описанием класса, либо путем наследования от уже существующего класса. В последнем случае класс **наследует** все характеристики уже существующего класса. То есть он получает в свое распоряжение все члены и все методы уже существующего класса.



Класс, от которого наследуется функциональность, называется **базовым классом** или **суперклассом**. Класс, который наследует функциональность, называется **наследником**, **подклассом**, или **потомком** соответствующего суперкласса.

```
class Parent
{
    private int m_value;

    public Parent(int iValue)
    { m_value = iValue; }

    protected int getSquare()
    { return m_value * m_value; }
}
```

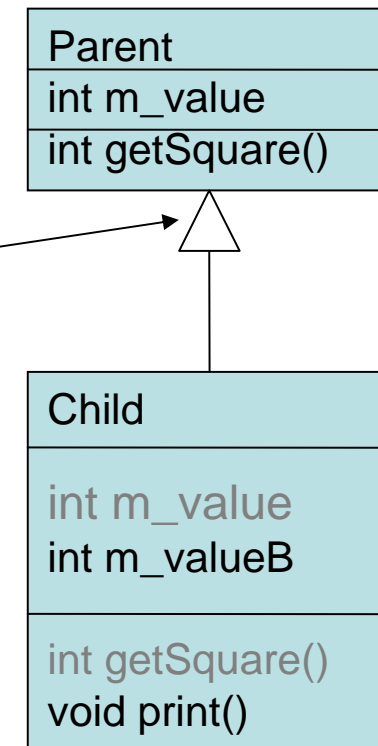
```
class Child extends Parent
{
    private int m_valueB;

    public Child(int A, int B)
    {
        super(A);
        m_valueB = B;
    }
}
```

```
public void print()
{ System.out.println("A^2 = " + getSquare() + " B = " + m_valueB); }
```

Формальная декларация наследования

Вызовы конструктора и унаследованного метода



Класс потомок наследует все члены и все методы суперкласса.
Значит ли это, что подкласс может напрямую работать с любыми членами и с любыми методами суперкласса?

Класс потомок наследует все члены и все методы суперкласса.
Значит ли это, что подкласс может напрямую работать с любыми членами и с любыми методами суперкласса?

Ответ: нет, не значит. Все зависит от модификаторов доступа.

public – разрешает доступ к члену/методу из любой точки программы

private – разрешает доступ к члену/методу только из того класса, в котором член/метод объявлен

protected – разрешает доступ к члену/методу из того класса, в котором член/метод объявлен, из любого его подкласса и из любого класса того же пакета, в котором класс объявлен

Отсутствие модификатора доступа означает так называемый доступ в пределах пакета (package – доступ). В этом случае доступ к члену/методу разрешается из того класса, в котором член/метод объявлен и из любого класса того же пакета, в котором класс объявлен.

Наблюдение: принцип инкапсуляции остается справедливым и в случае механизма наследования. Внутреннюю структуру класса следует защищать и от подклассов.

Поэтому обычно следует поступать так:

- все члены класса изначально снабжать модификатором `private`
- в случае необходимости доступа к члену класса из подкласса уровень доступа можно повысить до `protected`, но более предпочтительным является использование методов для доступа к членам класса.

Наблюдение: использование методов для доступа к полям позволяет более полно изолировать классы, но увеличивает время доступа к полям. Программист должен исходя из условий задачи достигать компромисса между скоростью работы программы и изолированностью классов.

Важной особенностью механизма наследования является возможность заменить часть методов суперкласса в потомке.

```
class Parent
{
    public void printMessage()
    { System.out.println(getMessage()); }

    protected String getMessage()
    { return "It is a parent class!"; }
}
```

```
class Child extends Parent
{
    protected String getMessage()
    { return "It is a child class!"; }
}
```

```
Parent parent = new Parent();
parent.printMessage();           // печатает "It is a parent class!"
```

```
Child child = new Child();
child.printMessage();           // печатает "It is a child class!"
```

Наблюдение: при наследовании подкласс, в частности, наследует внешний (public) и защищенный (protected) интерфейсы суперкласса. Отсюда следует:

Второе правило совместимости по присваиванию в ООП (правило согласованности при наследовании)

Пусть TypeA и TypeB – какие-то классы. Тогда в коде

```
TypeA varA = ...;  
TypeB varB = varA;
```

присваивание является корректным в том и только в том случае, когда TypeB совпадает с TypeA или является его суперклассом.

Это правило корректно в силу того, что подкласс всегда обладает всеми свойствами суперкласса.

Демонстрация полиморфизма при наследовании

```
class Parent
{
    public void printMessage()
    { System.out.println(getMessage()); }

    protected String getMessage()
    { return "It is a parent class!"; }
}
```

```
class Child extends Parent
{
    protected String getMessage()
    { return "It is a child class!"; }
}
```

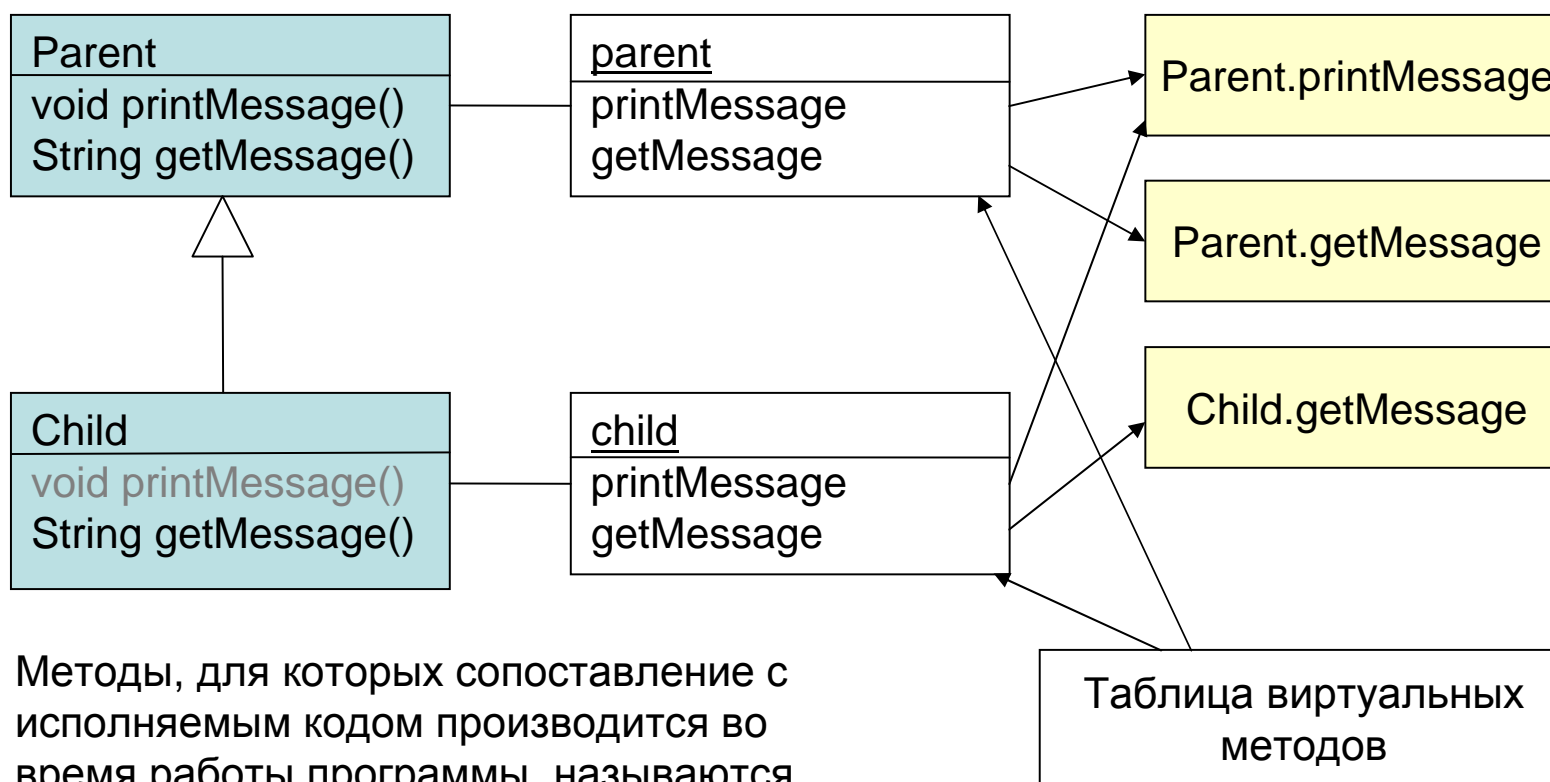
```
Parent parent = new Parent();
parent.printMessage();           // печатает "It is a parent class!"
```

```
Child child = new Child();
child.printMessage();           // печатает "It is a child class!"
```

```
Parent something = new Child(); // корректное присваивание
something.printMessage();       // печатает "It is a child class!"
```

Наблюдение:

полиморфизм невозможен без специальной организации вызовов методов, поскольку информацию о том, какой метод вызывать, следует получать непосредственно в момент вызова метода во время работы программы (так называемое **позднее связывание**).



Методы, для которых сопоставление с исполняемым кодом производится во время работы программы, называются **виртуальными**.

Еще раз про наследование

У любого класса явно выделяется три группы методов:

1. собственные (private) методы. Эти методы фактически составляют набор подпрограмм, доступных только из данного класса;
2. защищенные (protected) методы. Эти методы составляют **защищенный интерфейс** класса (то есть интерфейс, доступный лишь потомкам класса);
3. открытые (public) методы. Эти методы составляют **внешний интерфейс** класса (доступный всем).

Parent
private: someMethod1(); someMethod2();
protected: someMethod3(); someMethod4();
public: someMethod5(); someMethod6();

Еще раз про наследование

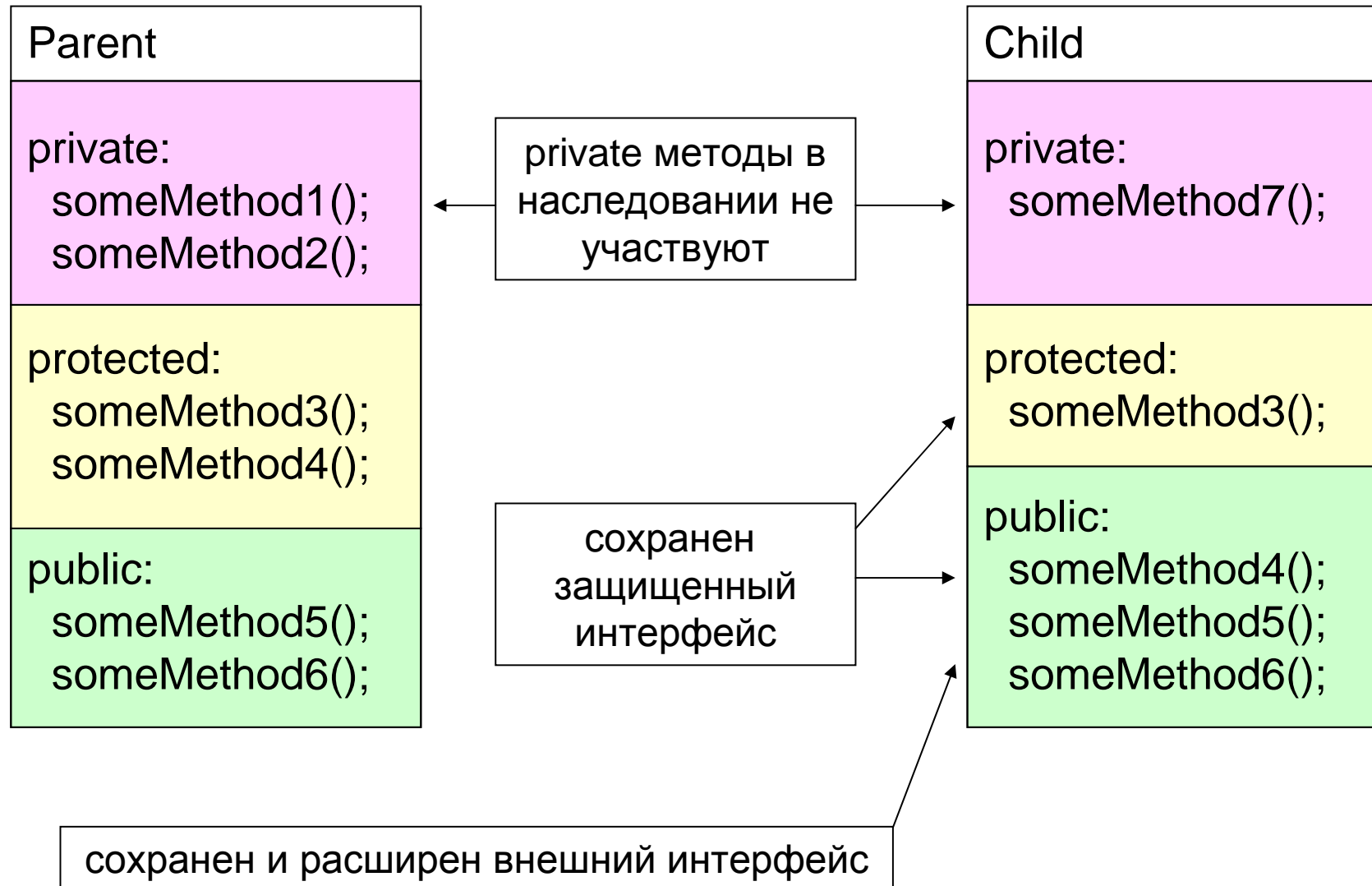
Наследование означает получение подклассом всей функциональности суперкласса. Это означает, что фактически наследуются **и функциональность класса (защищенный интерфейс), и внешний интерфейс**. Если класс сделал какую-то функциональность видимой извне, то и все потомки обязаны предоставлять эту функциональность.

Отсюда следствие: если в предке метод имеет модификатор `public`, то в потомке этот метод также должен иметь модификатор `public` (потомок не имеет права скрывать методы, открытые в предке).

А открывать методы в потомках можно.
В Java это правило заложено в язык.

Child
private: someMethod7();
protected: someMethod3();
public: someMethod4(); someMethod5(); someMethod6();

Еще раз про наследование



Наличие виртуальных методов позволяет вводить конструкцию, занимающую промежуточное положение между интерфейсами и настоящими классами. Эта конструкция называется «**абстрактный класс**».

```
abstract class Parent
{
    public void printMessage()
    { System.out.println(getMessage()); }

    abstract protected String getMessage();
}
```

```
class Child extends Parent
{
    protected String getMessage()
    { return "It is a child class!"; }
}
```

```
Parent parent = new Parent();           // недопустимо
```

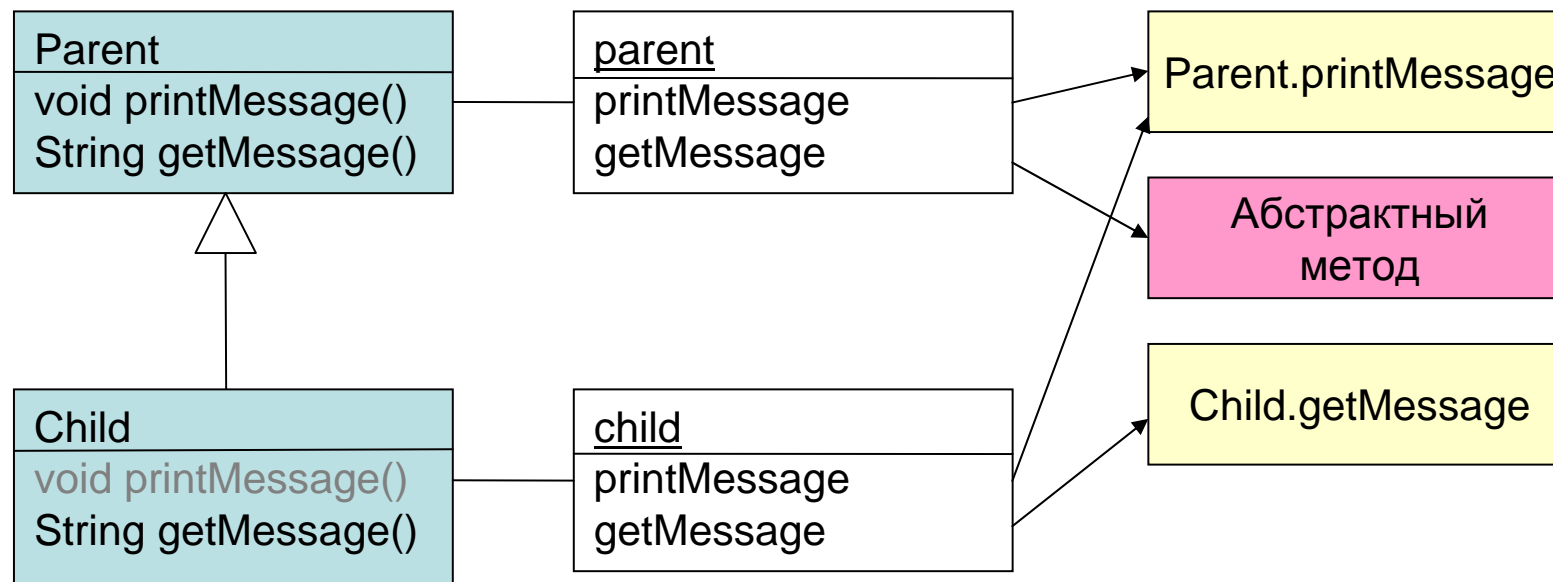
```
Child child = new Child();
child.printMessage();                     // печатает "It is a child class!"
```

```
Parent something = new Child();          // корректное присваивание
something.printMessage();               // печатает "It is a child class!"
```

Описание абстрактного класса полностью повторяет синтаксис описания обычного класса. Но часть методов (как и в случае с интерфейсом) лишь задекларирована.

Наблюдение:

Так изменится таблица виртуальных методов для абстрактного класса Parent:



Абстрактный метод резервирует запись в таблице виртуальных методов, но не заполняет ее. При переопределении абстрактного метода в подклассе эта запись окажется корректно заполнена.

Решение задачи организации алгоритмов кодирования:

```
abstract class BaseByteReader
    implements ByteReader
{
    // эти методы оставляем
    // на реализацию в подклассах
    abstract public byte readByte();
    abstract public boolean hasMoreData();

    // а эти методы будут унаследованы
    public byte[] readBytes(int count)
    {
        byte[] buffer = new byte[count];
        for (int i = 0; i < count; i++)
        {
            if (hasMoreData())
                buffer[i] = readByte();
            else
                buffer[i] = 0;
        }
        return buffer;
    }

    public void readBytes(byte[] buffer,
        int startIndex, int count)
    { ... }
}
```

```
class FileByteReader
    extends BaseByteReader
{
    // достаточно реализовать
    // эти два метода
    public byte readByte()
    { return <очередной байт из файла>; }

    public boolean hasMoreData()
    { return <есть ли еще данные>; }

    public byte[] readBytes(int count)
    { <эффективная реализация блочного
        чтения>;
    }
}

class NetworkByteReader
    extends BaseByteReader
{
    byte readByte()
    { return <очередной байт из сети>; }

    boolean hasMoreData()
    { return <есть ли еще данные>; }
}
```

Решение задачи организации алгоритмов кодирования (Java 8+):

```
interface ByteReader
{
    // эти методы нужно реализовать
    public byte readByte();
    public boolean hasMoreData();

    // а эти методы будут "унаследованы"
    default public
        byte[] readBytes(int count)
        {
            byte[] buffer = new byte[count];
            for (int i = 0; i < count; i++)
            {
                if (hasMoreData())
                    buffer[i] = readByte();
                else
                    buffer[i] = 0;
            }
            return buffer;
        }

    default public
        void readBytes(byte[] buffer,
            int startIndex, int count)
        { ... }
}
```

```
class FileByteReader
    implements ByteReader
{
    // достаточно реализовать
    // эти два метода
    public byte readByte()
    { return <очередной байт из файла>; }

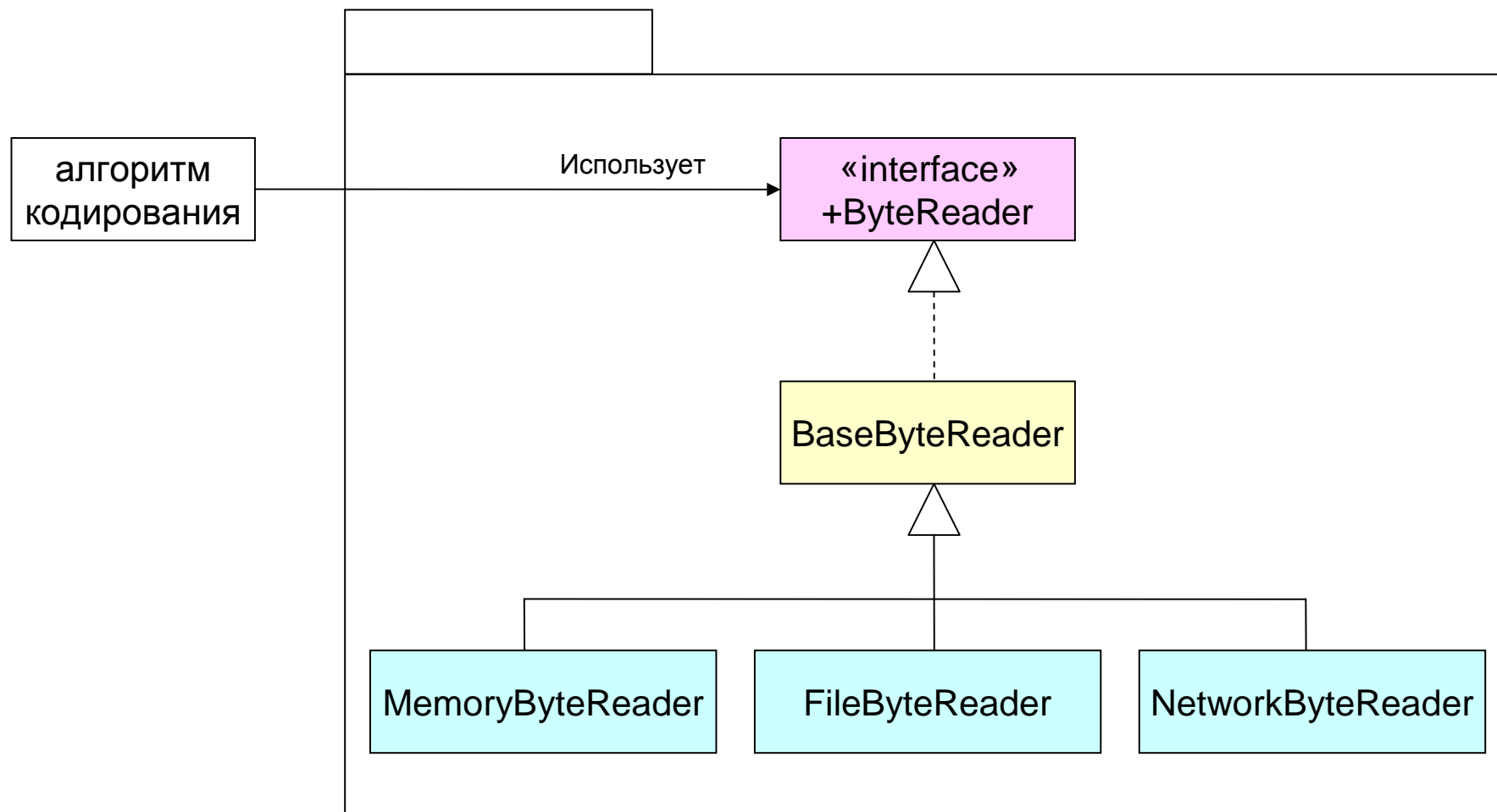
    public boolean hasMoreData()
    { return <есть ли еще данные>; }

    public byte[] readBytes(int count)
    { <эффективная реализация блочного
        чтения>;
    }
}

class NetworkByteReader
    implements ByteReader
{
    byte readByte()
    { return <очередной байт из сети>; }

    boolean hasMoreData()
    { return <есть ли еще данные>; }
}
```

Структурная диаграмма шаблона «Интерфейс и абстрактный класс»



Структурная диаграмма шаблона «Интерфейс и абстрактный класс»

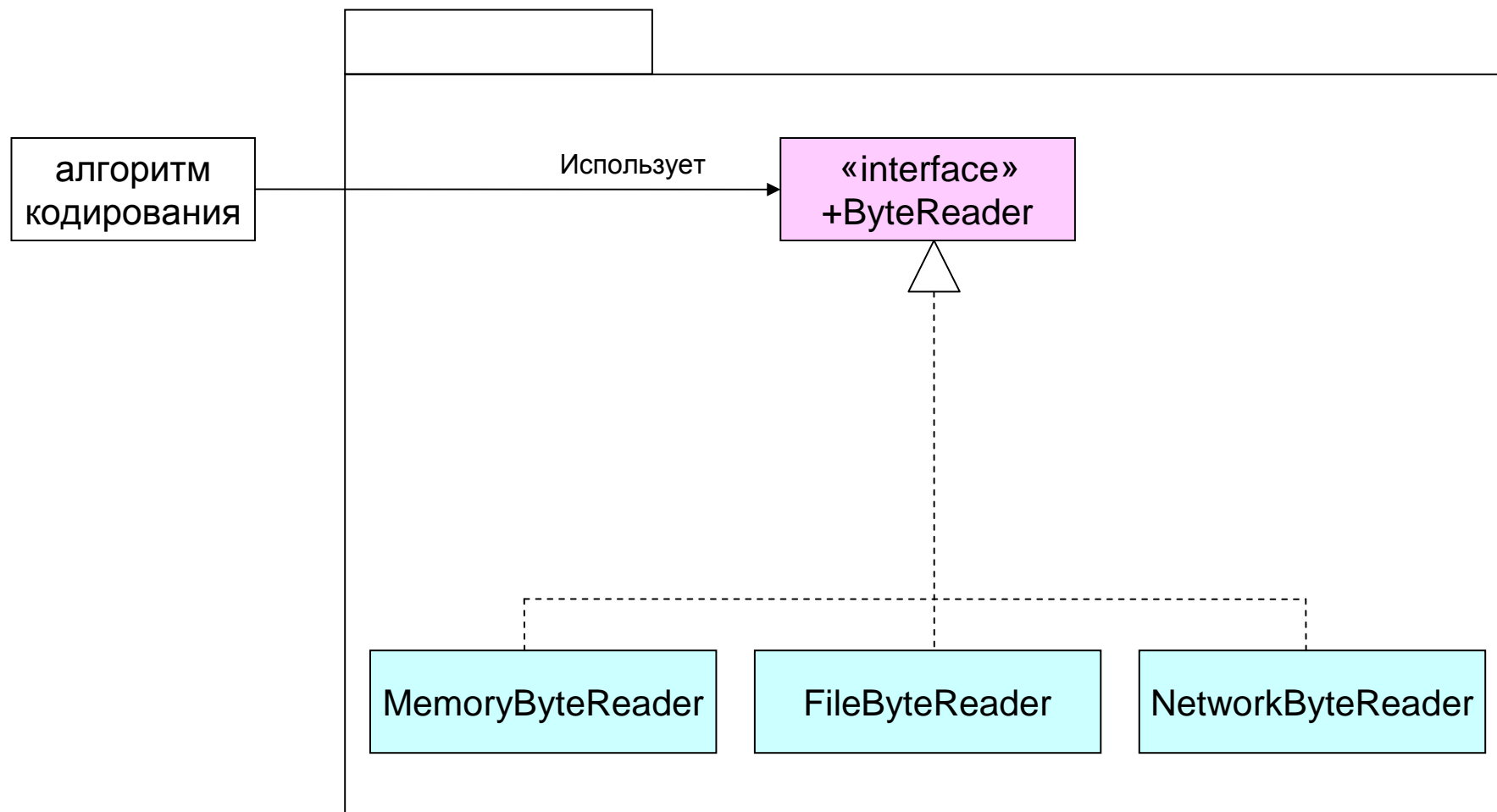
Интерфейс отвечает за фиксацию поведения семейства объектов, за их общий контракт.

Абстрактный класс отвечает за хранение кода, общего для нескольких реализаций интерфейса.

Листовой класс (в нижнем ряду диаграммы) отвечает за конкретную реализацию интерфейса.

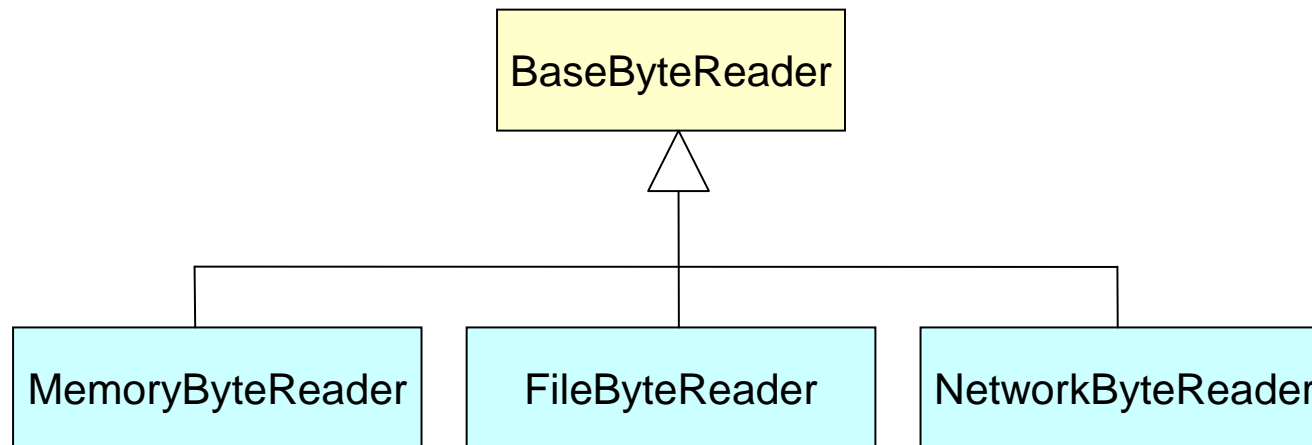
Подобная система классов всегда используется через интерфейс. Прямое использование листовых классов является нежелательным (и часто делается невозможным при помощи инкапсуляции).

Структурная диаграмма шаблона «Интерфейс»



Если общего кода нет, абстрактный класс не нужен.

Структурная диаграмма шаблона «абстрактный суперкласс»



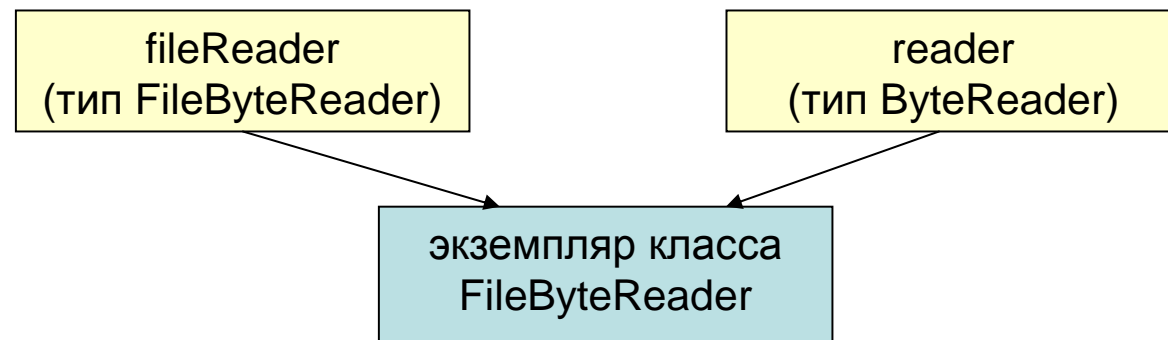
В некоторых случаях роль интерфейса может выполнять абстрактный класс. Это возможно за счет того, что любой класс определяет некоторый интерфейс (внешний интерфейс класса), который и заменяет явный интерфейс.

На практике такой подход обычно является нежелательным. В подобных случаях лучше явно определять интерфейс.

Важно!

Присваивание для объектных типов копирует ссылку на объект, но не сам объект:

```
FileByteReader fileReader = new FileByteReader();  
ByteReader reader = fileReader;
```



Наблюдение: раз переменные разных типов могут ссылаться на один и тот же объект, следовательно, должна быть возможность определения, какой тип реально имеет объект, на который ссылается переменная.

Для такой проверки служит оператор **instanceof**

```
if (reader instanceof FileByteReader)  
{  
    FileByteReader fileReader = (FileByteReader)reader;  
    fileReader.printFileName();  
}
```

Зачем нужна возможность определения типа объекта.

В большинстве случаев необходимость определения типа объекта является признаком плохой объектной структуры, поскольку явное определение типа объекта мешает полиморфизму.

Рассмотрим пример, когда анализ типа объекта является оправданным.

Рассмотрим две версии интерфейса `ByteReader`: простую и усложненную.

```
public interface ByteReader
{
    byte readByte();
    boolean hasMoreData();
}
```

```
public interface BlockByteReader
    extends ByteReader
{
    public byte[] readBytes(int count);

    public void readBytes(byte[] buffer,
        int startIndex, int count);
}
```

Обратим внимание, что один интерфейс является расширением другого.

Наблюдение: мы уже знаем, что методы из интерфейса `BlockByteReader` являются более удобными, но не являются существенными для чтения данных.

Код алгоритма кодирования мог бы быть устроен следующим образом:

```
public void encodeXXX(
    BufferedReader reader, ByteWriter writer)
{
    if (reader instanceof BlockByteReader) {
        BlockByteReader fastReader =
            (BlockByteReader) reader;
        byte [] data = fastReader.readBytes(...);
        encodeBlockXXX(data);
        writeData(data);
    }
    else {
        while (reader.hasMoreData())
        {
            byte b = reader.readByte();
            byte encodedByte = encodeByteXXX(b);
            writer.writeByte(encodedByte);
        }
        writer.flush();
    }
}
```

Поскольку в подобном коде интерфейс как бы помечает наличие дополнительной функциональности, такие интерфейсы часто называют **маркирующими**.

Наблюдение: в некоторых случаях маркирующий интерфейс может вообще не иметь методов!

То есть за счет проверки типа полиморфный код может получать дополнительную информацию о функциональности полученного класса. Важно: полиморфный код всегда должен быть способен работать с любой реализацией используемого интерфейса!

При использовании наследования часто требуется зафиксировать некоторую функциональность или поведенческую модель класса и запретить ее модификацию в подклассах.

Это можно сделать при помощи модификатора **final**: в описании метода он запрещает переопределение метода в подклассе, а в описании класса он запрещает дальнейшее наследование.

```
abstract class BaseByteReader
    implements ByteReader
{
    abstract public byte readByte();
    abstract public boolean hasMoreData();

    public final byte[] readBytes(
        int count)
    {
        byte[] buffer = new byte[count];
        for (int i = 0; i < count; i++)
        {
            if (hasMoreData())
                buffer[i] = readByte();
            else
                buffer[i] = 0;
        }
        return buffer;
    }
}

public void readBytes(byte[] buffer,
    int startIndex, int count)
{ ... }
```

```
final class FileByteReader
    extends BaseByteReader
{
    byte readByte()
    { return <очередной байт из файла>; }

    boolean hasMoreData()
    { return <есть ли еще данные>; }
}
```

Языки и технологии программирования.

Объектно-ориентированное программирование

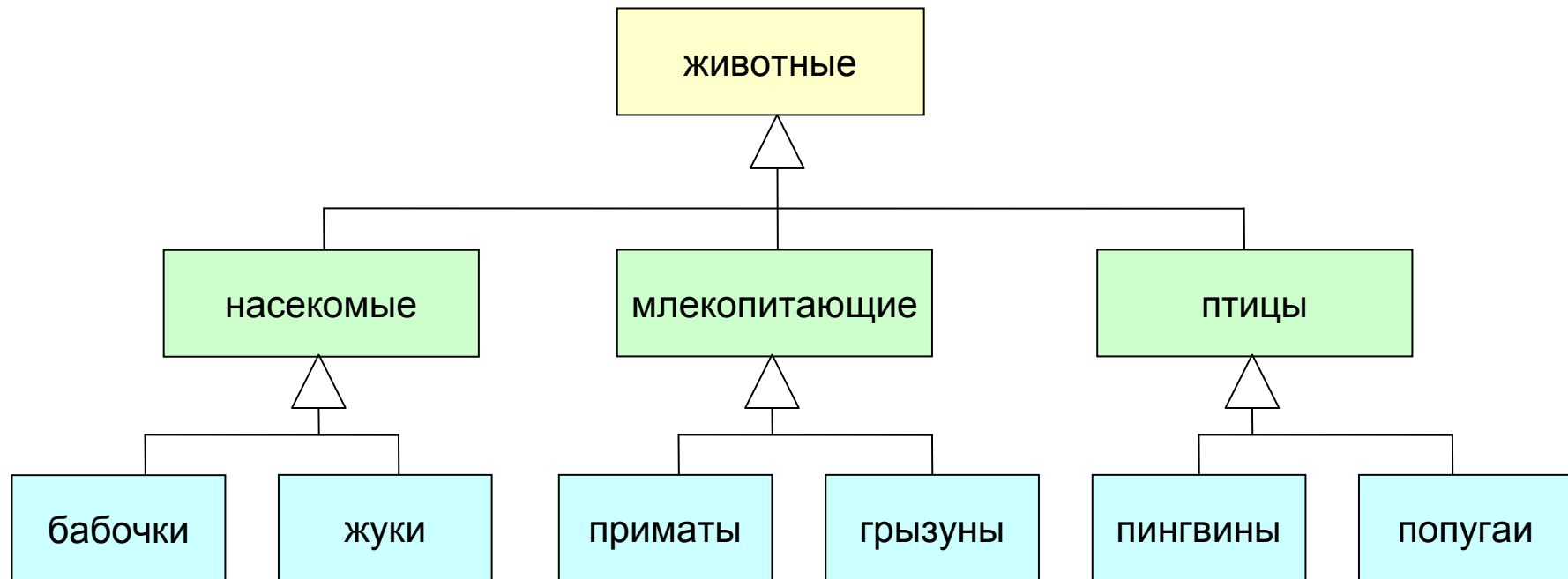
*Особенности применения ООП:
Примеры использования наследования*

Наблюдение:

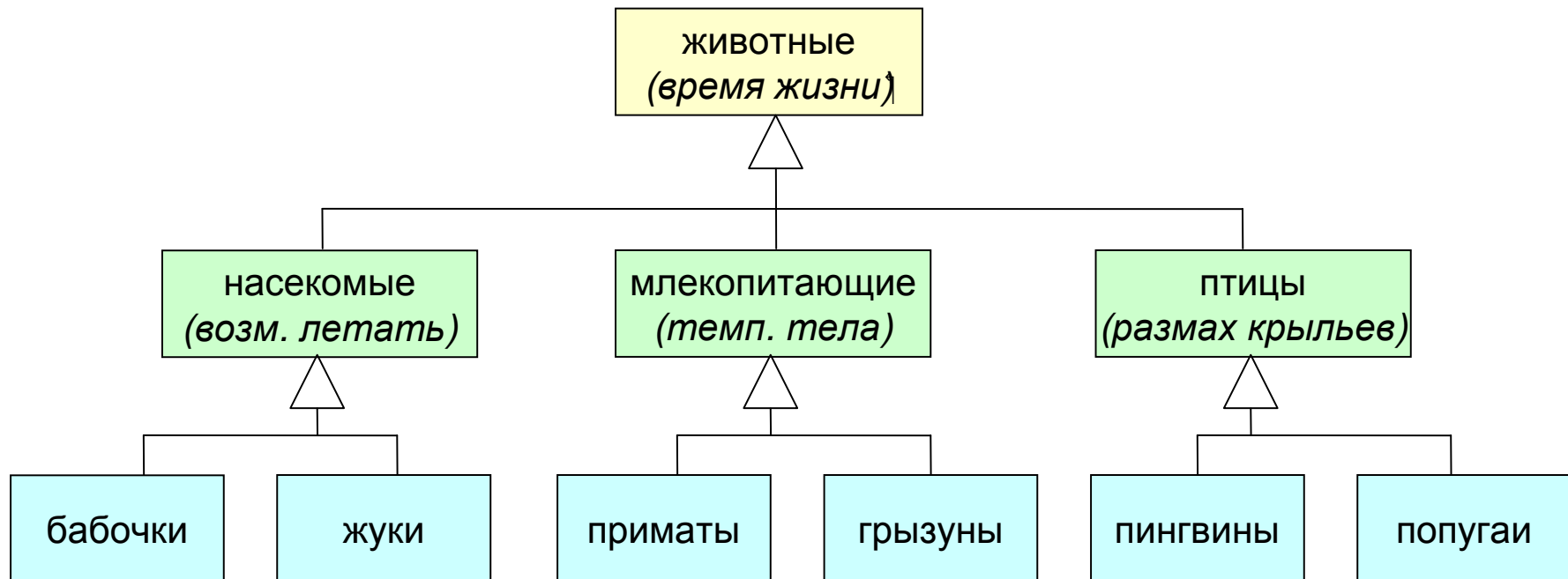
Отношение наследование не может изменяться в процессе работы программы. Поэтому часто отношение наследования (класс В – подкласс класса А) характеризуется фразой:

«класс В – это разновидность класса А»

Наследование часто возникает там, где можно ввести какую-либо классификацию:



В этом случае структура наследования повторяет структуру классификатора, и каждый уровень наследования уточняет структуру класса путем введения новых атрибутов (полей класса):



То есть при наследовании расширяется открытый интерфейс за счет введения новых атрибутов и методов в классах-потомках.

Аналогичным образом строится иерархия графических объектов при построении графического интерфейса пользователя и иерархия геометрических объектов при реализации графического редактора.

Но в этих случаях на первый план выходит возможность построения полиморфного кода:

```
abstract class Figure
{
    private Point m_position;
    private Color m_color;

    abstract public void show();
    abstract public void hide();

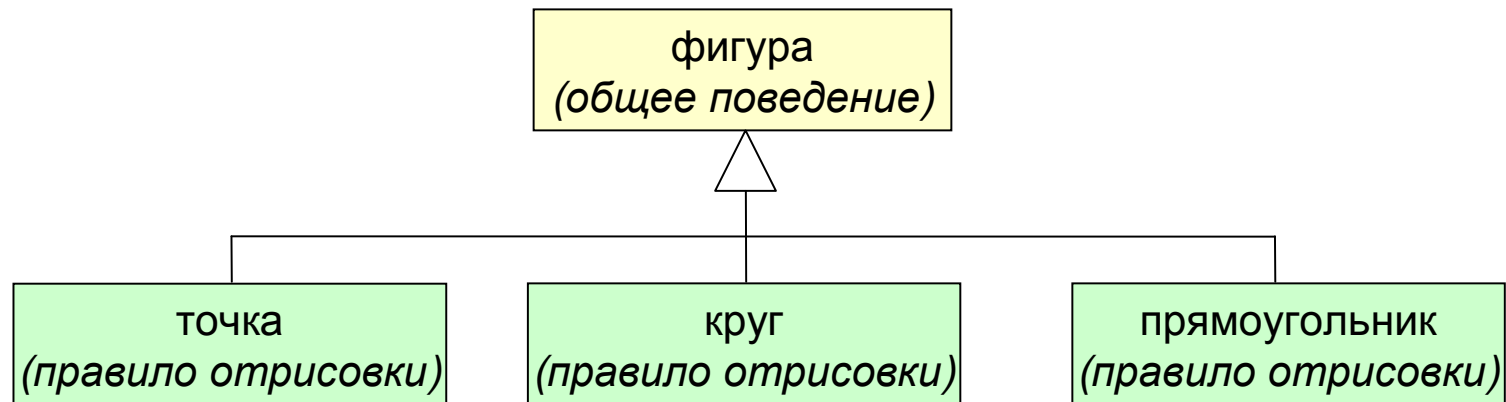
    public void moveTo(Point point)
    {
        hide();
        m_position = point;
        show();
    }

    public void setColor(Color color)
    {
        hide();
        m_color = color;
        show();
    }
}
```

```
public class Square
    extends Figure
{
    public void show()
    {
        нарисовать квадрат с левым верхним
        углом в точке m_position и цветом
        m_color
    }

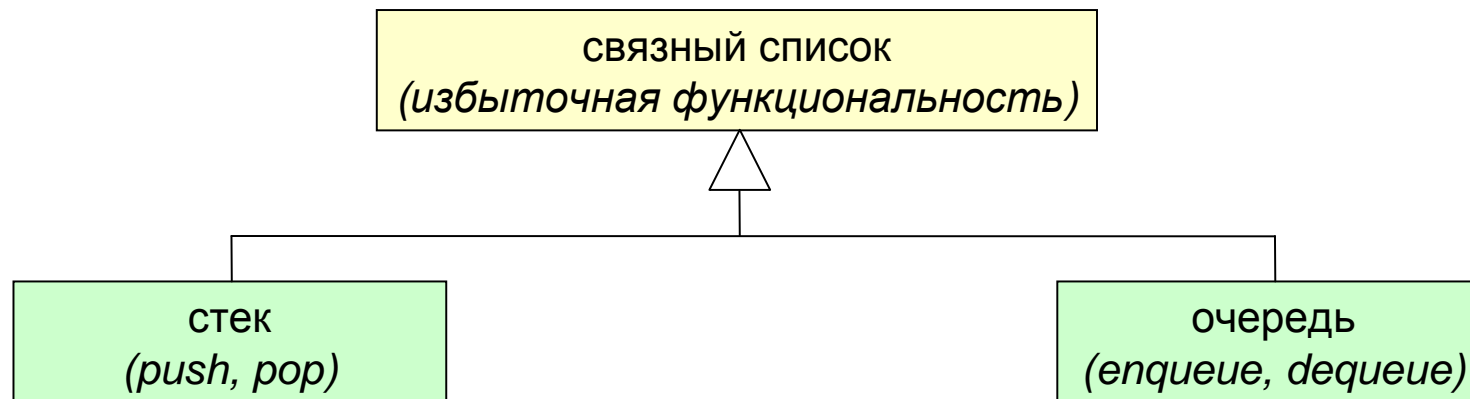
    public void hide()
    {
        нарисовать квадрат с левым верхним
        углом в точке m_position и цветом фона
    }
}
```


Соответственно возникает иерархия:



То есть при наследовании расширяется открытый интерфейс за счет введения новых особенностей поведения в классах-потомках.

Иерархии классов могут возникать и при фиксации в подклассах тех или иных возможностей более функционального суперкласса

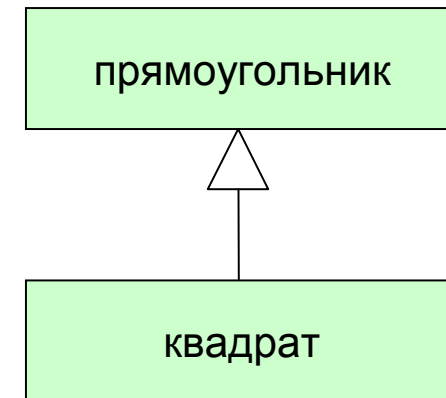


В этом примере при наследовании целью является не расширение открытого интерфейса класса, а обеспечение доступа к закрытому интерфейсу класса.

По сути, данный пример наследования говорит, что «класс предок используется классом потомком». В языке С++ для этого типа наследования есть специальная языковая конструкция: закрытое наследование.

Возможность наследования определяется фразой
«класс В – это разновидность класса А»
Как понимать слово «разновидность»?

Вопрос:
Можно ли реализовать «квадрат» как потомка
«прямоугольника»?



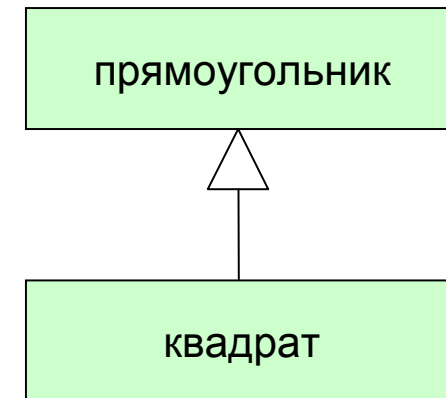
Можно ли реализовать «квадрат» как потомка «прямоугольника»?

Квадрат – это прямоугольник с **ограничениями**.

Наличие ограничений не позволяет выполнить главное требование наследования: полного наследования внешнего интерфейса, то есть поведенческой модели объекта.

Наблюдение. Внутри интерфейса методы условно делятся на две группы: читающие состояние объекта (get-методы) и изменяющие состояние объекта (set-методы).

При наследовании с ограничениями, как правило, легко обеспечить реализацию get-методов, но обычно невозможно обеспечить реализацию set-методов.



```

class Rectangle
{
    private int m_height;
    private int m_width;
    private Point m_location;

    public Rectangle(int height,
        int width, Point location)
    {
        m_height = height;
        m_width = width;
        m_location = location;
    }

    public int getHeight()
    { return m_height; }

    public int getWidth()
    { return m_width; }

    public Point getLocation()
    { return m_location; }

    public void draw() {...}
}

```

```

class Square extends Rectangle
{
    public Square(int size,
        Point location)
    {
        super(size, size, location);
    }
}

```

Во внешнем интерфейсе класса
Rectangle нет set-методов.
Наследование корректно.

```

class Rectangle
{
    private int m_height;
    private int m_width;
    private Point m_location;

    public Rectangle(int height,
        int width, Point location)
    {
        m_height = height;
        m_width = width;
        m_location = location;
    }

    public int getHeight()
    { return m_height; }

    public int getWidth()
    { return m_width; }

    public Point getLocation()
    { return m_location; }

    public void draw() {...}

```

```

    public void setDimensions(
        int height, int width)
    {
        m_height = height;
        m_width = width;
    }
}

class Square extends Rectangle
{
    public Square(int size,
        Point location)
    {
        super(size, size, location);
    }
}

```

Во внешнем интерфейсе класса Rectangle появился set-метод. Наследование некорректно, поскольку невозможно повторить **поведенческую** модель.

```

class Rectangle
{
    private int m_height;
    private int m_width;

    public int getHeight()
    { return m_height; }

    public int getWidth()
    { return m_width; }

    ...

    public void setDimensions(
        int height, int width)
    {
        m_height = height;
        m_width = width;
    }
}

class Square extends Rectangle
{
    public Square(int size,
        Point location)
    {
        super(size, size, location);
    }
}

```

```

public static void RectangleTest(
    Rectangle r)
{
    r.SetDimensions(5, 10);
    if ((r.getHeight() != 5)
        || (r.getWidth() != 10))
    {
        // сообщить об ошибке
    }
}

```

В соответствии с идеей наследования мы можем ожидать, что если метод **RectangleTest** корректно работает с объектами класса **Rectangle**, то и с объектами класса **Square** он также будет работать корректно. Но по причине ограничений на допустимые состояния в классе **Square** это оказывается не так!

Важно понимать, что с точки зрения семантики, класс предок является обобщением класса потомка (то есть имеет более простое описание модели поведения, меньше свойств, меньше методов).

Класс потомок является **уточнением** этой более общей модели за счет **расширения** этой модели. Он не должен сужать унаследованное множество допустимых состояний за счет наложения ограничений на унаследованные атрибуты и операции. Множество состояний может лишь расширяться за счет новых атрибутов.

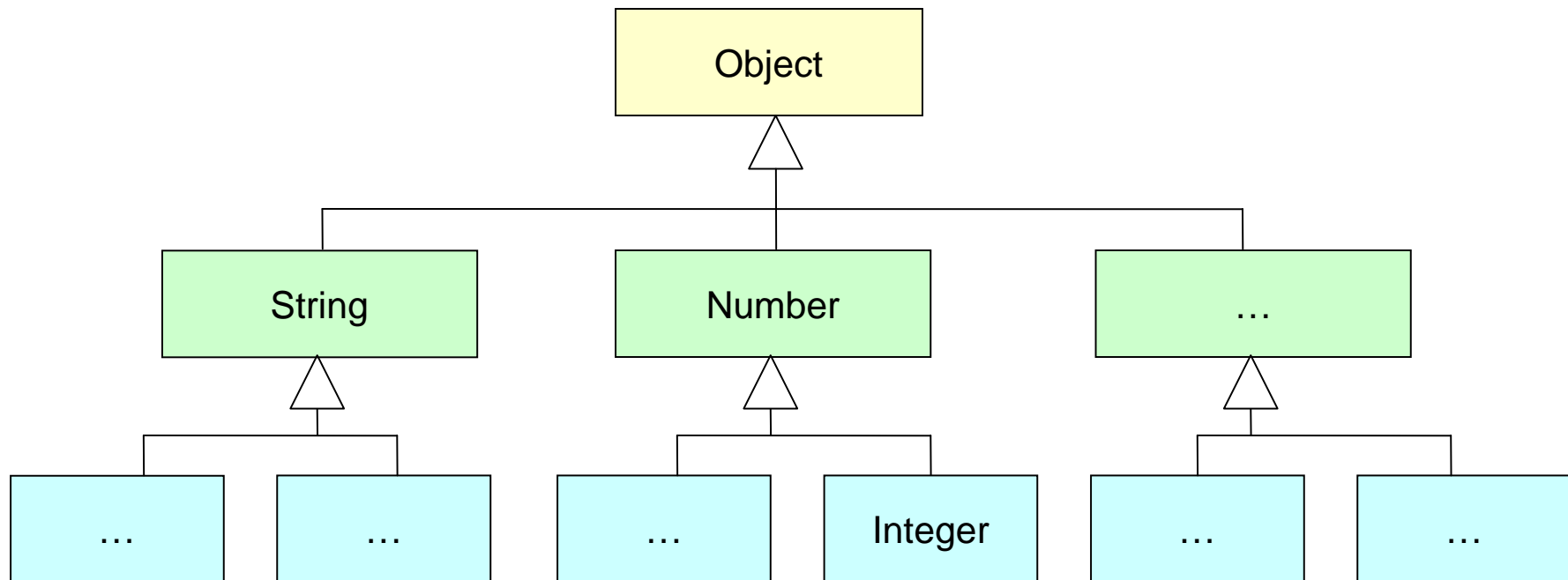
Один из принципов ООП (принцип замены Лисков, LSP) как раз и требует, чтобы класс потомок можно было бы использовать везде, где можно использовать класс предок. Что и означает, в частности, что множество состояний класса потомка не может быть меньше множества состояний класса предка.

Языки и технологии программирования.

Объектно-ориентированное программирование

*Основы Java:
Иерархия объектов в Java*

Все классы в Java выстроены в единую иерархию, в корне которой находится класс Object.



Это позволяет:

- получить тип, который совместим по присваиванию с любым объектным типом
- зафиксировать функциональность, присущую ВСЕМ объектам в Java

Методы класса Object:

- **public** String toString()
- **public int** hashCode()
- **public boolean** equals(Object obj)
- **protected** Object clone()
- **protected void** finalize()
- **public** Class getClass()
- **public final native void** notify()
- **public final native void** notifyAll()
- **public final void** wait()

Методы класса Object:

- **public** String toString()

Любой объект можно превратить в строку, вызвав метод toString().

Например:

```
Integer myInt = new Integer(5);  
System.out.println(myInt.toString());
```

Замечание:

Метод toString() удобно переопределять в целях отладки (отладчик в Eclipse «отображает» объект, вызывая его метод toString).

Методы класса Object:

- **public** String toString()

Вопрос: как определить метод toString() для класса ComplexNumber?

Например, так, чтобы на печати получался такой формат:

5 + 10i

Методы класса Object:

- **public** String toString()

Вопрос: как определить метод toString() для класса ComplexNumber?

Или такой формат:

(5 , 10)

Или такой:

(Re : 5 , Im : 10)

Наблюдение: у одного класса может быть несколько текстовых представлений! Поэтому обычно неочевидно, какое из представлений следует сопоставить методу toString()

Методы класса Object:

- **public** String toString()

Вывод:

Метод toString() нежелательно использовать для реализации операции преобразования объекта в строку в неотладочных целях, поскольку:

1. не всегда очевиден формат, в котором будет выводиться информация
2. высок риск, что другой программист изменит формат на более удобный для отладки, тем самым разрушив логику других компонентов системы, опирающихся на знание формата результата метода toString().
3. сложно отслеживать полиморфные цепочки вызовов, поскольку этот метод присутствует во всех классах Java.

Также нежелательно включать метод toString() в состав интерфейса, поскольку компилятор не сможет проверить корректность реализации интерфейса (метод toString() может оказаться унаследованным от Object, а не реализованным в рамках полной реализации интерфейса).

Методы класса Object:

- **public** String toString()

Более удобным является следующий шаблон реализации метода toString() через делегирование:

```
//для бизнес-логики
public String toStringValue()
{
    return <строковое представление>;
}
```

```
// для отладочной печати
public String toString()
{
    return toStringValue();
}
```


Методы класса Object:

- **public int** hashCode()
- **public boolean** equals(Object obj)

Метод equals() служит для проверки, является ли содержимое двух разных объектов одним и тем же (совпадает ли состояние двух разных объектов).

У класса Object он реализован так:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

То есть изначально объект равен только самому себе (поскольку операция == в данном случае сравнивает равенство ссылок на объект).

Методы класса Object:

- **public int** hashCode()
- **public boolean** equals(Object obj)

Следующий пример показывает, что могут быть равны разные объекты. То есть с точки зрения размещения в памяти объекты разные, но содержимое (состояние объектов) совпадает.

```
String string1 = new String("Test");  
String string2 = new String("Test");  
boolean bTest1 = (string1 == string2); // = false  
boolean bTest2 = (string1.equals(string2)); // = true  
boolean bTest3 = (string2.equals(string1)); // = true
```

Операция equals() должна быть симметричной!

Результат вызова x.equals(null) всегда должен быть false!

Методы класса Object:

- **public int** hashCode()
- **public boolean** equals(Object obj)

Метод hashCode () должен просто вернуть некоторое целое число, согласованное с состоянием объекта.

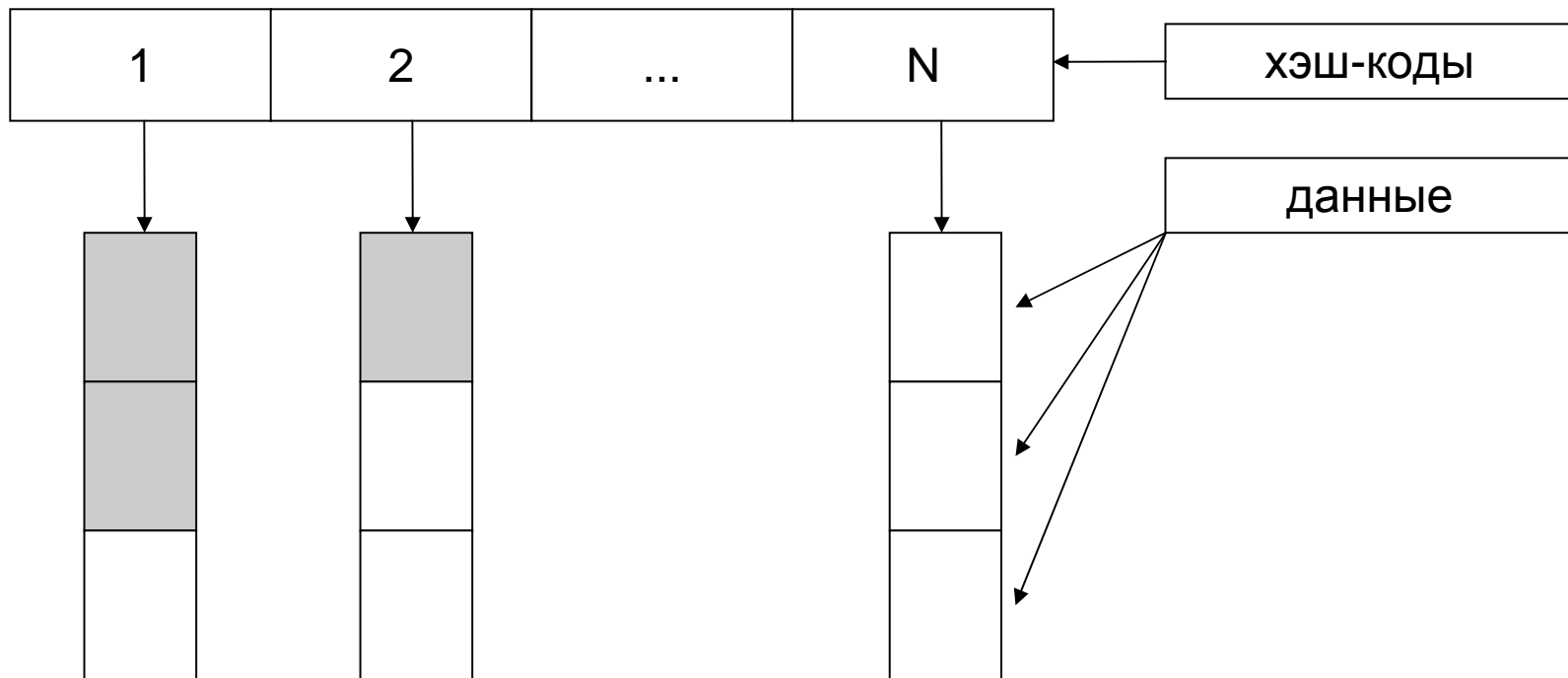
Единственное формальное требование: если a.equals(b) возвращает true, то a.hashCode() и b.hashCode() обязаны совпадать.

Наблюдение: Из неравенства хэш-кодов следует неравенство объектов. Но из равенства хэш-кодов не следует равенство объектов (возможных состояний объекта обычно гораздо больше чем возможных хэш-кодов).

Наличие метода hashCode () позволяет:

1. повысить эффективность поиска объекта среди множества объектов;
2. эффективно хранить любые объекты Java в хэш-таблицах.

Хэш-таблица – структура данных, позволяющая хранить информацию и эффективно находить хранимую информацию за счет использования хэширования.



Для получения наилучшей производительности хэш-таблиц, следует строить метод `hashCode()` так, чтобы он обеспечивал равномерное распределение возвращаемых значений, но не в ущерб скорости.

Методы класса Object:

- **protected** Object clone()

При помощи операции = невозможно создать копию объекта, поскольку реально создается лишь новая ссылка на уже существующий объект.

Метод clone() позволяет создавать точную копию объекта.

```

public class ValueHolder
    implements Cloneable
{
    private int m_iValue;

    public ValueHolder(int iValue)
    {
        m_iValue = iValue;
    }

    public void setValue(int iValue)
    {
        m_iValue = iValue;
    }

    public int getValue()
    {
        return m_iValue;
    }

    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}

```

```

ValueHolder val1 = new ValueHolder(1);
ValueHolder val2 = val1;
val2.setValue(5);

// сейчас val1.getValue() вернет 5
System.out.println(val1.getValue());

ValueHolder val3 =
    (ValueHolder)val1.clone();

// сейчас val3.getValue() также вернет 5
System.out.println(val3.getValue());

val3.setValue(10);

// поскольку val3 - это отдельная копия
// объекта класса ValueHolder,
// то val3.getValue() вернет 10,
// но val1.getValue() вернет 5
System.out.println(val3.getValue());
System.out.println(val1.getValue());

```

```

public class ValueHolder
    implements Cloneable
{
    private int [] m_values =
        new int[1];

    public ValueHolder(int iValue)
    {
        m_values[0] = iValue;
    }

    public void setValue(int iValue)
    {
        m_values[0] = iValue;
    }

    public int getValue()
    {
        return m_values[0];
    }

    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}

```

```

ValueHolder val1 = new ValueHolder(1);
ValueHolder val2 = val1;
val2.setValue(5);

// сейчас val1.getValue() вернет 5
System.out.println(val1.getValue());

ValueHolder val3 =
    (ValueHolder)val1.clone();

// сейчас val3.getValue() также вернет 5
System.out.println(val3.getValue());

val3.setValue(10);

// теперь поверхностное копирование
// создало val3 как отдельную копию
// объекта класса ValueHolder,
// но сохранило ссылку на тот же самый
// массив m_iValue
// Поэтому val3.getValue() вернет 10,
// и val1.getValue() вернет 10
System.out.println(val3.getValue());
System.out.println(val1.getValue());

```

```

public class ValueHolder
    implements Cloneable
{
    private int [] m_values =
        new int[1];

    public ValueHolder(int iValue)
    { m_values[0] = iValue; }

    public void setValue(int iValue)
    { m_values[0] = iValue; }

    public int getValue()
    { return m_values[0]; }

    public Object clone()
    {
        try
        {
            ValueHolder result =
                (ValueHolder)super.clone();
            result.m_values =
                (int [])m_values.clone();
            return result;
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}

```

```

ValueHolder val1 = new ValueHolder(1);
ValueHolder val2 = val1;
val2.setValue(5);

// сейчас val1.getValue() вернет 5
System.out.println(val1.getValue());

ValueHolder val3 =
    (ValueHolder)val1.clone();

// сейчас val3.getValue() также вернет 5
System.out.println(val3.getValue());

val3.setValue(10);

// теперь за счет глубокого копирования
// вновь val3.getValue() вернет 10,
// а val1.getValue() вернет 5
System.out.println(val3.getValue());
System.out.println(val1.getValue());

```


Методы класса Object:

- **protected void** finalize()

Финализатор. Вызывается Java-машиной в тот момент, когда объект уничтожается.

Не путать с деструктором!

- **public Class** getClass()

Позволяет программно узнать структуру класса, к которому относится данный объект.

- **public void** notify()
- **public void** notifyAll()
- **public void** wait()

Служат для реализации межпоточной синхронизации при программировании многопоточных приложений.

Языки и технологии программирования.

Объектно-ориентированное программирование

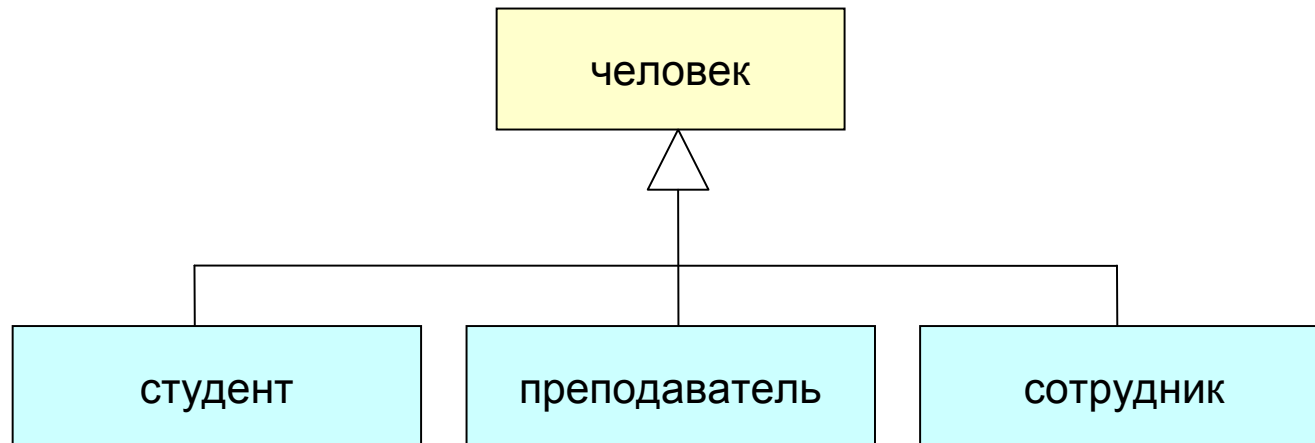
*Основные понятия ООП:
Делегирование*

Когда наследование не работает...

Задача: построить систему классов для описания людей, находящихся в университете.

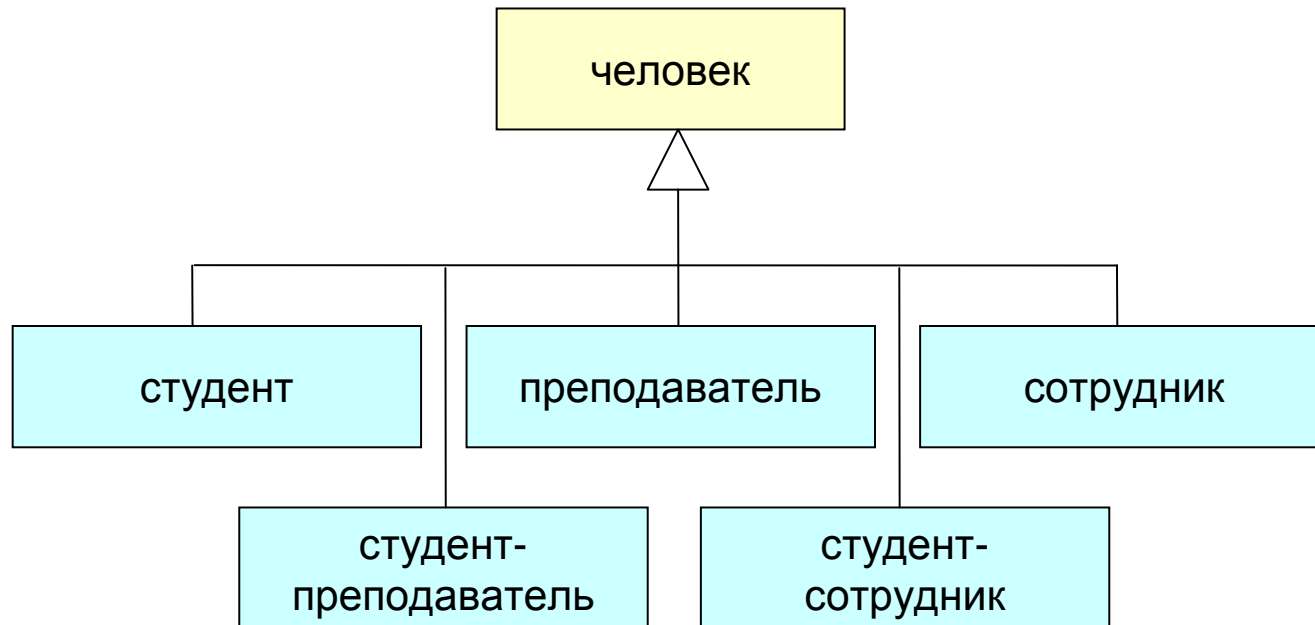
Наблюдение: есть по крайней мере три роли, которые может выполнять человек, находящийся в университете: студент, преподаватель, сотрудник.

Применение наследования выглядит на первый взгляд вполне обоснованным:

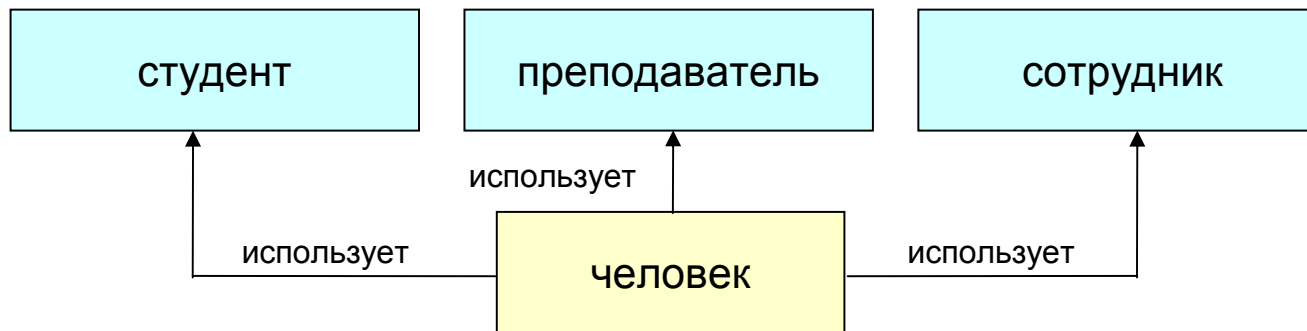


Однако, несложно понять, что роль человека может меняться во времени (например, студент может временно стать преподавателем на младших курсах).

Выходом из положения могло бы стать создание комбинированных ролей: сотрудник-студент, студент-преподаватель и т.п. Но это ведет к необоснованному росту числа классов в модели.



Правильным в такой ситуации является применение шаблона «делегирование» для организации динамического отношения между классами. То есть получаем следующую схему:



То есть классы, представляющие роли, не связаны отношением наследования, вместо этого класс «человек» делегирует свою функциональность тому классу, роль которого конкретный экземпляр класса «человек» сейчас исполняет.

```

public interface LectureVisitor
{
    public void visitLecture();
}

public class Person
    implements LectureVisitor
{
    private LectureVisitor
        m_visitorBehaviour;

    public void visitLecture()
    {
        m_visitorBehaviour.visitLecture();
    }

    public void switchToStudent()
    {
        m_visitorBehaviour =
            new StudentBehaviour();
    }

    public void switchToTeacher()
    {
        m_visitorBehaviour =
            new TeacherBehaviour();
    }

    ...
}

```

```

public class StudentBehaviour
    implements LectureVisitor
{
    public void visitLecture()
    {
        <послушать и поспать>
    }
}

public class TeacherBehaviour
    implements LectureVisitor
{
    public void visitLecture()
    {
        <рассказать и разбудить>
    }
}

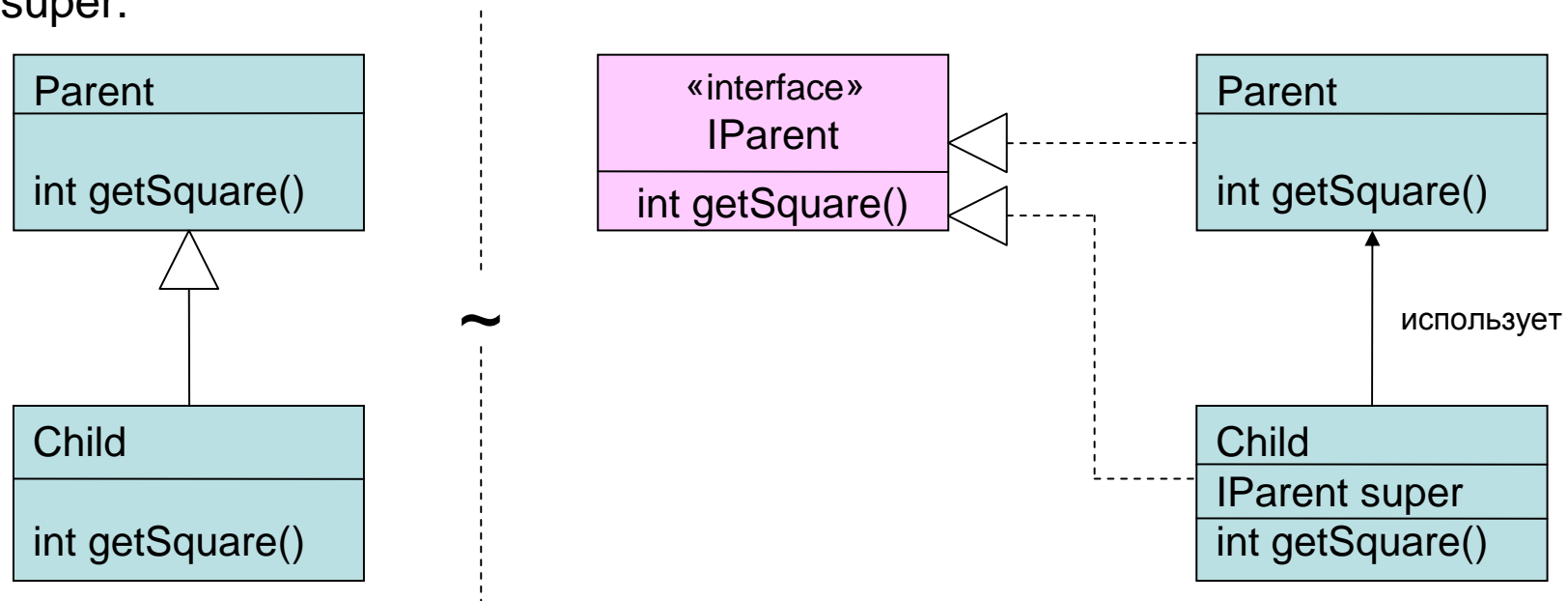
public class StafferBehaviour
    implements LectureVisitor
{
    public void visitLecture()
    {
        <принести ноутбук>
    }
}

```

Наблюдение: применение делегирования позволяет динамически изменять поведение объекта, при использовании наследования поведение объекта наследуется жестко.

Наблюдение: наследование – частный (неявный) случай делегирования, когда в роли делегата выступает класс предок:

1. родительский класс Parent индуцирует интерфейс IParent, который реализуется и суперклассом, и подклассом;
2. подкласс получает доступ к суперклассу через «специальное» поле `super`.



Итоги

- Абстракции – основа для построения полиморфного кода
- Интерфейсы – способ описания абстракций
- Наследование – механизм повторного использования кода и способ создания устойчивых поведенческих и структурных отношений между классами
- Полиморфизм – механизм работы с объектами через абстракции, позволяющий создавать модульный расширяемый код
- Абстрактные классы – способ частичной фиксации модели поведения
- Делегирование – способ реализации изменчивой модели поведения

Языки и технологии программирования.

Объектно-ориентированное программирование

Структурная обработка ошибок

Наблюдение:

любой программист допускает ошибки в программах, которые он создает. Даже если это очень аккуратный и высококвалифицированный специалист.

Важно:

на отладку уходит до 60% от общего времени разработки программного продукта.

Этапы разработки программного обеспечения:

1. Постановка задачи
2. Разработка архитектуры программы
3. Программирование
4. Отладка
5. Внедрение и сопровождение

1. Постановка задачи

Цель – перевести требования заказчика на язык, понятный программистам. Результатом являются техническое задание (ТЗ) и сценарии использования (тесты). Выполняется аналитиками.

2. Разработка архитектуры программы

Результатом является функциональная объектная модель программы. Такая модель позволит спланировать и распределить дальнейшие работы. Выполняется программистом-архитектором (опытным программистом).

3. Программирование

Результатом является код программы. Выполняется программистом.

4. Отладка

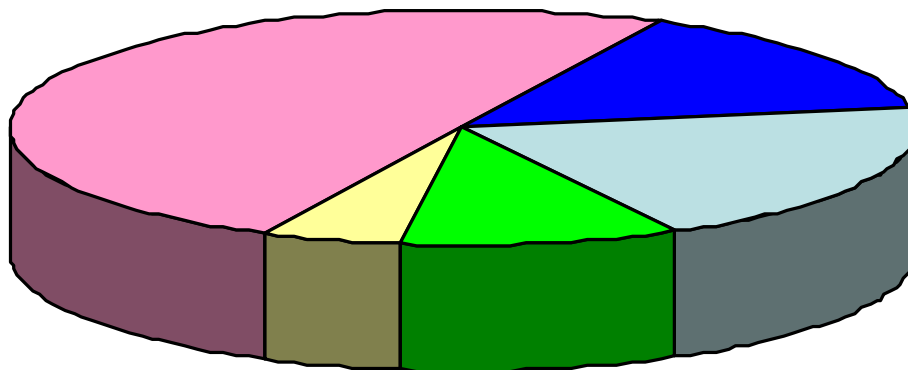
Результатом является работающий код программы. Выполняется программистом с помощью тестировщиков.

5. Внедрение и сопровождение

Результатом является программа, работающая на машинах заказчика. Выполняется аналитиком, программистом, тестировщиком, менеджером по внедрению.

Этапы разработки программного обеспечения:

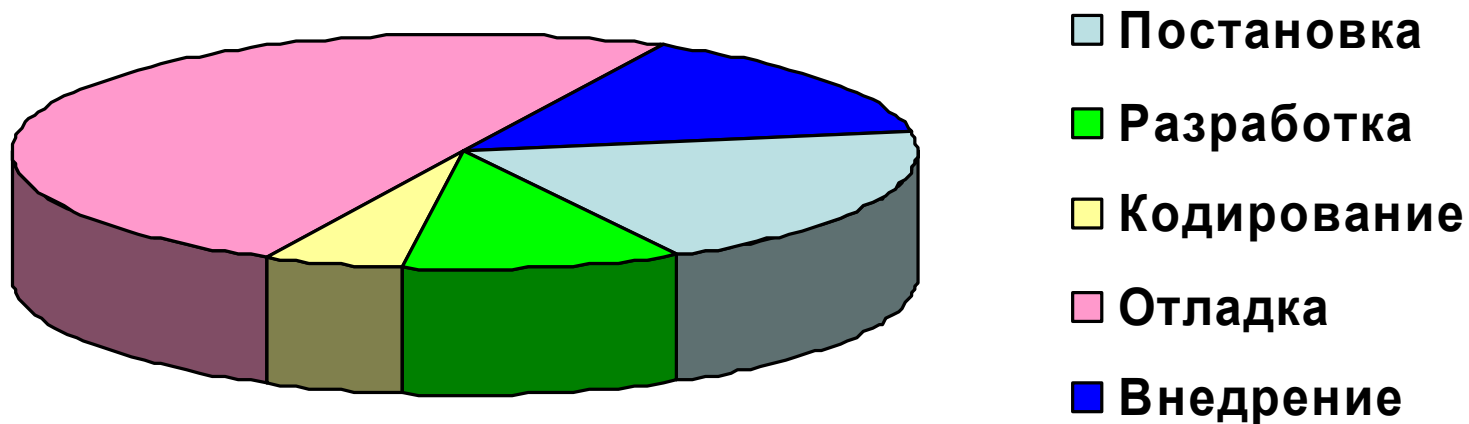
1. Постановка задачи
2. Разработка архитектуры программы
3. Программирование
4. Отладка
5. Внедрение и сопровождение



- Постановка
- Разработка
- Кодирование
- Отладка
- Внедрение

Этапы разработки программного обеспечения:

1. Постановка задачи
2. Разработка архитектуры программы
3. Программирование
4. Отладка
5. Внедрение и сопровождение



Факт: отладка является самым дорогим этапом разработки продукта

Цель: сократить расходы на отладку

Вопросы:

Что эта программа делает? Есть ли в ней ошибки? Если есть, то сколько?

```
#include "iostream"

double av(double * values, int n)
{
    double res = 0;
    for (int i = 0; i <= n; i++)
    {
        res = res + values[i];
    }
    return res / n;
}

int main()
{
    double arr[] = {1.0, 2.0, 4.0, 9.0};
    double val = 0;
    val = av(arr, 4);
    std::cout << val << "\n";    // печатает ?
    val = av(arr, 4);
    std::cout << val << "\n";    // печатает ?
    return 0;
}
```

Вопросы:

Ошибки есть! И они не обнаруживаются при компиляции!

```
#include "iostream"
```

```
double av(double * values, int n)
{
    double res = 0;
    for (int i = 0; i <= n; i++)
    {
        res = res + values[i];
    }
    return res / n;
}
```

выход за границы массива

потенциальное деление на ноль

```
int main()
{
    double arr[] = {1.0, 2.0, 4.0, 9.0};
    double val = 0;
    val = av(arr, 4);
    std::cout << val << "\n";    // печатает 4
    val = av(arr, 4);
    std::cout << val << "\n";    // печатает 5
    return 0;
}
```

нестабильное поведение
(нестабильность зависит
от компилятора!)

Попробуем переписать код на Java:

```
public class Sample
{
    static double av(double [] values, int n)
    {
        double res = 0;
        for (int i = 0; i <= n; i++)
        {
            res = res + values[i];
        }
        return res / n;
    }

    public static void main(String [] args)
    {
        double arr[] =
            new double[]{1.0, 2.0, 4.0, 9.0};
        double val = 0;
        val = av(arr, 4);
        System.out.println(val);
        val = av(arr, 4);
        System.out.println(val);
    }
}
```

```
#include "iostream"

double av(double * values, int n)
{
    double res = 0;
    for (int i = 0; i <= n; i++)
    {
        res = res + values[i];
    }
    return res / n;
}

int main()
{
    double arr[] =
        {1.0, 2.0, 4.0, 9.0};
    double val = 0;
    val = av(arr, 4);
    std::cout << val << "\n";
    val = av(arr, 4);
    std::cout << val << "\n";
    return 0;
}
```

Попробуем переписать код на Java:

```
public class Sample
{
    static double av(double [] values, int n)
    {
        double res = 0;
        for (int i = 0; i <= n; i++)
        {
            res = res + values[i];
        }
        return res / n;
    }

    public static void main(String [] args)
    {
        double arr[] =
            new double[]{1.0, 2.0, 4.0, 9.0};
        double val = 0;
        val = av(arr, 4);
        System.out.println(val);
        val = av(arr, 4);
        System.out.println(val);
    }
}
```

```
#include "iostream"

double av(double * values, int n)
{
    double res = 0;
    for (int i = 0; i <= n; i++)
    {
        res = res + values[i];
    }
    return res / n;
}

int main()
{
    double arr[] =
        {1.0, 2.0, 4.0, 9.0};
    double val = 0;
    val = av(arr, 4);
    std::cout << val << "\n";
    val = av(arr, 4);
    std::cout << val << "\n";
    return 0;
}
```

Результат запуска:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at Sample.av(Sample.java:8)
    at Sample.main(Sample.java:17)
```

Вывод: язык Java содержит встроенные средства для диагностики и выявления ошибок.

Цель: разобраться, как эти средства работают.

Для диагностики ошибок во время работы программы требуется обеспечить следующее:

1. внедрить в программу код обнаружения ошибок
2. сохранить информацию о месте возникновения ошибки
3. программа должна перейти в специальное состояние «обработки ошибки», чтобы либо каким-то образом преодолеть ошибочное состояние, либо прервать работу

Для многих потенциальных ошибок Java автоматически внедряет код контроля. Автоматически проверяются:

1. соответствие типов данных при присваивании,
2. обращение к объекту через незаданную (null) ссылку
3. выход за границы массива,
4. ошибки арифметики и т.п.

Также программист может размещать в программе собственный код обнаружения и преодоления ошибок.

Пусть ошибка обнаружена. Надо запомнить точку ее появления.

Наблюдение:

запомнить только имя класса и номер строки обычно недостаточно, поскольку один и тот же метод может вызываться из разных частей программы, вызовы методов также могут быть рекурсивными.

Вывод: надо запоминать **стек вызовов**.

Стеком вызовов называется последовательность вложенных вызовов подпрограмм, начинающаяся с точки запуска программы.

Наблюдение:

если при каждом входе в метод запоминать его имя в стеке, а при каждом выходе из стека удалять элемент в вершине стека, то информация в стеке как раз и будет составлять стек вызовов.

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:5)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)


```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:5)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:8)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:5)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:6)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)
Sample.fact(Sample.java:8)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)
Sample.fact(Sample.java:8)

```
public class Sample
{
    static double fact(int n)
    {
        if (n == 1)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main(String [] args)
    {
        System.out.println(
            String.valueOf(fact(3)));
    }
}
```

Стек вызовов:

Sample.main(Sample.java:13)

Отслеживанием стека вызовов занимается Java-машина.

Поэтому в точке возникновения ошибки достаточно создать специальный класс-контейнер, в котором эта информация будет размещаться.

Такой класс контейнер может создаваться явно или неявно:

```
int a = 5;  
int b = 0;  
int c = a / b;
```

неявное создание

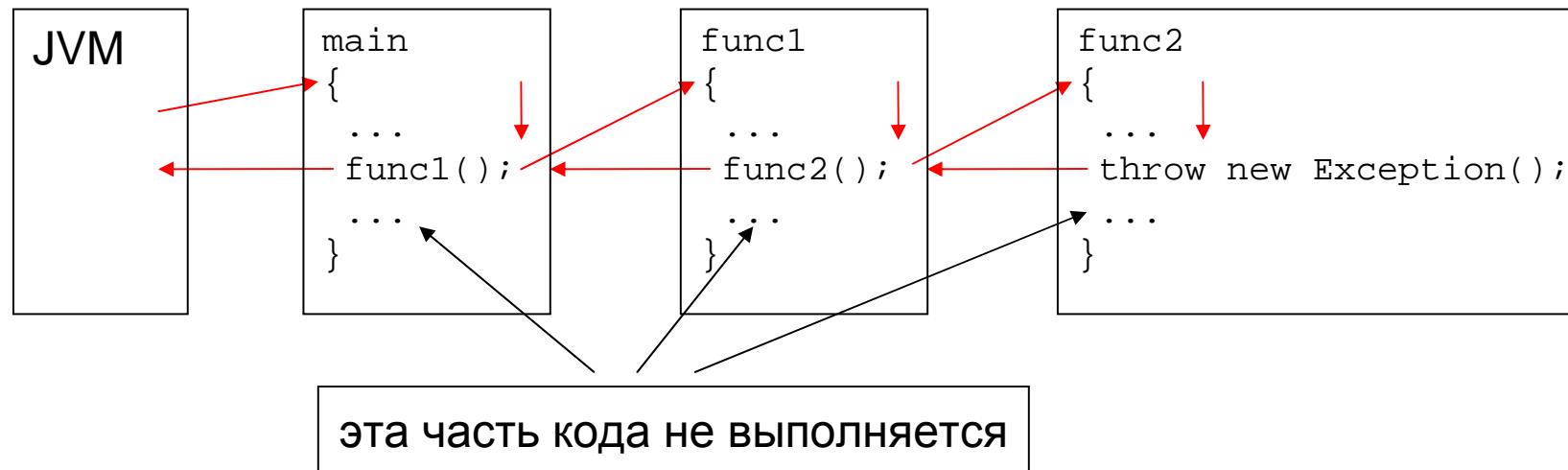
```
int a = 5;  
int b = 0;  
if (b == 0)  
{  
    throw new ArithmeticException("деление на 0");  
}  
int c = a / b;
```

явное создание

Ключевое слово `throw` говорит о том, что нужно инициировать состояние обработки ошибки с данным контейнером информации об ошибке.

Состояние ошибки называется **исключительной ситуацией** или **исключением** (англ. exception). Соответственно, классы-контейнеры с информацией об ошибке называются **исключениями**.

В момент возбуждения состояния ошибки операцией `throw` нормальный ход выполнения программы прерывается, и Java-машина начинает последовательно выходить из методов, в которых обнаружена ошибка:



То есть исключение «двигается» или «пробрасывается» через подпрограммы (сквозь стек вызовов) до тех пор, пока не достигнет Java-машины, где информация из контейнера извлекается и выводится на консоль.

Наблюдение: по умолчанию в случае обнаружения ошибки программа принудительно останавливается, тем самым предотвращая возможное нестабильное поведение или нарушения в логике работы.

Ошибки, критические для конкретного участка кода, не всегда бывают критическими для логики программы.

Например, попытка прочитать данные из несуществующего файла порождает исключение (для кода чтения данных отсутствие файла явно является критической ошибкой). Но в каких-то случаях программа может продолжить работу (например, уточнить имя файла у пользователя).

Чтобы в случае обнаружения подобной ошибки программа не прерывалась, следует обеспечить выход программы из состояния ошибки путем внедрения инструкций **обработки ошибки**.

В Java для этого служит конструкция **try/catch/finally**.

```

try
{
    ... // (1)
    if (ошибка)
        throw new Exception(); // (2)
    ... // (3)
}
catch (Exception ex)
{
    System.out.println(
        ex.getMessage()); // (4)
    ex.printStackTrace(
        System.out);
}
finally
{
    ... // (5)
}

```

В случае отсутствия ошибки конструкция try/catch/finally работает по схеме (1) – (3) – (5).

И по схеме (1) – (2) – (4) – (5), если ошибка произошла.

Наблюдения:

Блок finally выполняется всегда!

Вход в блок catch выводит программу из состояния ошибки.

Блок catch может отсутствовать. В этом случае блок finally выполняется, но программа остается в состоянии ошибки.

Блок finally может отсутствовать.

Блоков catch может быть несколько:

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```

В этом случае управление получает первый (в порядке появления) блок catch такой, что класс исключения в его сигнатуре является предком обрабатываемого исключения (то есть первый блок catch, аргументу которого можно присвоить пробрасываемое исключение).

Блоков catch может быть несколько:

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```

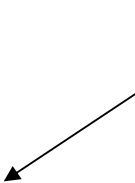
ВВОД: 15 3

ОТВЕТ ПРОГРАММЫ: 5

ЭТО СООТВЕТСТВУЕТ РАБОТЕ БЕЗ ОШИБОК

Блоков catch может быть несколько:

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```



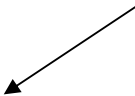
throw new ArithmeticException
внутри операции деления,
не подкласс InputMismatchException,
подкласс Exception.

ВВОД: 15 0

ответ программы: Арифметическая ошибка: / by zero

Блоков catch может быть несколько:

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```



throw new InputMismatchException
внутри метода nextInt,
не подкласс ArithmeticException,
подкласс Exception.

ВВОД: 15 abc

ОТВЕТ ПРОГРАММЫ: Ошибка ввода

Блоков catch может быть несколько:

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```

throw new NoSuchElementException
внутри метода nextInt,
не подкласс ArithmeticException,
не подкласс InputMismatchException,
подкласс Exception.

ввод (Windows): 15 ^Z

или ввод (Linux): 15 ^D

ответ программы: Произошла ошибка: null

Наблюдение:

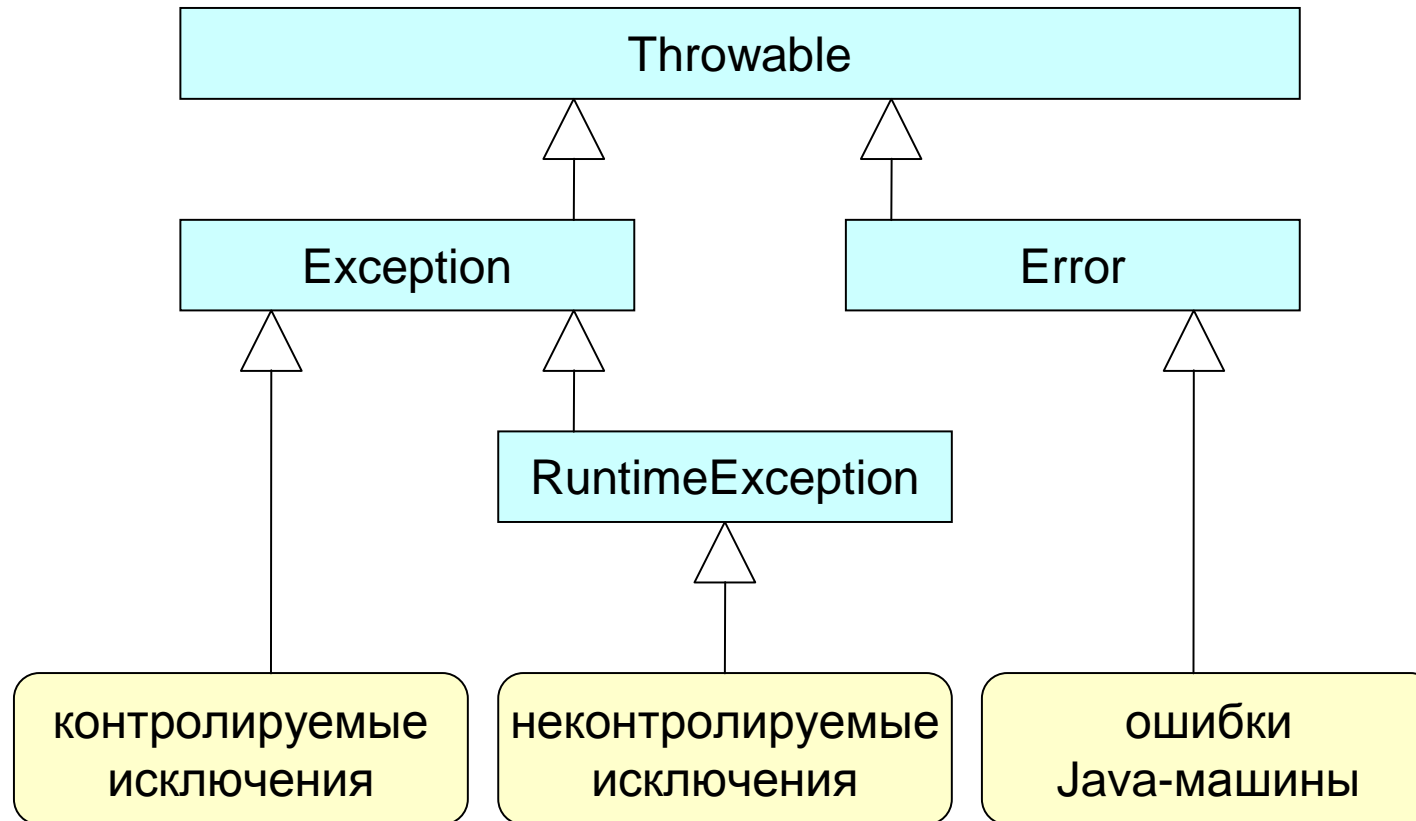
Просто обеспечить наличие средств диагностики ошибок на уровне языка недостаточно по причине существования человеческого фактора (программист никогда не делает ошибки сознательно, но они постоянно появляются).

Поэтому в Java введен механизм **принудительного** контроля за ошибками.

Цель введения механизма принудительного контроля за ошибками в том, чтобы обязать программиста:

1. декларировать в сигнатуре метода, какие ошибки могут возникать в конкретном методе;
2. требовать явной реакции (специальной обработки) в тех точках кода, где могут возникать ошибки.

Исключения в Java являются классами и образуют такую иерархию:



Механизм принудительного контроля за ошибками работает только для контролируемых исключений.

Контролируемые исключения

```
class MyException
    extends Exception
{
    public MyException(String strMessage)
    { super(strMessage); }
}

public class Sample4
{
    public static void main(String [] args)
    {
        process(args);
    }
    public static void process(String [] args)
    {
        if (args.length == 0)
        {
            throw new MyException("Отсутствуют аргументы");
        }
    }
}
```

В сигнатуре метода отсутствует информация о том, что метод может инициировать какие-либо ошибки. Поэтому ошибка обязана быть обработана внутри метода.

Исключение `MyException` относится к контролируемым (поскольку наследуется от `Exception`, но не от `RuntimeException`), поэтому компилятор выдает диагностику:

```
unreported exception MyException; must be caught or declared to be thrown
```

Контролируемые исключения

```
class MyException
    extends Exception
{
    public MyException(String strMessage)
    {    super(strMessage); }
}

public class Sample4
{
    public static void main(String [] args)
    {
        process(args);
    }
    public static void process(String [] args)
        throws MyException ←
    {
        if (args.length == 0)
        {
            throw new MyException("Отсутствуют аргументы");
        }
    }
}
```

явное описание того, что метод
может инициировать состояние
ошибки типа MyException

Теперь методу разрешается инициировать состояние ошибки, поэтому можно не обрабатывать исключение MyException внутри метода.

Контролируемые исключения

```
class MyException
    extends Exception
{
    public MyException(String strMessage)
    { super(strMessage); }
}

public class Sample4
{
    public static void main(String [] args)
    {
        process(args);
    }
    public static void process(String [] args)
        throws MyException
    {
        if (args.length == 0)
        {
            throw new MyException("Отсутствуют аргументы");
        }
    }
}
```

теперь наличие необработанной
ошибки диагностируется
в этой точке

метод может инициировать
состояние ошибки MyException

Вновь имеем диагностику при компиляции:

unreported exception MyException; must be caught or declared to be thrown

поскольку теперь метод process может инициировать контролируемое исключение, которое следует обработать в методе main().

Контролируемые исключения

```
class MyException
    extends Exception
{
    public MyException(String strMessage)
    {    super(strMessage); }
}

public class Sample4
{
    public static void main(String [] args)
    {
        try
        {
            process(args);
        }
        catch (Exception ex)
        {
            ex.printStackTrace(System.out);
        }
    }
    public static void process(String [] args)
        throws MyException
    {
        if (args.length == 0)
        {
            throw new MyException("Отсутствуют аргументы");
        }
    }
}
```

Контролируемые исключения

```
class MyException
    extends Exception
{
    public MyException(String strMessage)
    {    super(strMessage); }
}

public class Sample4
{
    public static void main(String [] args)
    {
        try
        {
            process(args);
        }
        catch (Exception ex)
        {
            ex.printStackTrace(System.out);
        }
    }

    public static void process(String [] args)
        throws MyException
    {
        if (args.length == 0)
        {
            throw new MyException("Отсутствуют аргументы");
        }
    }
}
```

Наблюдения:

Наличие подобных «глобальных» ловушек исключений с выдачей информации на экран или в файл протокола обеспечивает получение информации о любых ошибках в программе.

Локальные ловушки ошибок также важны, поскольку только в локальном контексте можно обеспечить корректное восстановление работы после ошибки.

Выводы:

Исключения, для которых Java проверяет обязательное наличие обработчика называются **контролируемыми**. К ним относятся все подклассы класса Exception, кроме класса RuntimeException и его подклассов.

Исключения, которые могут возникать практически в каждой строке (ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException и подобные) выделены в особую группу (подклассы класса RuntimeException), для которой не проводится принудительная проверка наличия обработчика. Такие исключения называются **неконтролируемыми**.

Исключения, которые возникают в результате внутренних проблем во время работы JVM, выделены в отдельную группу **ошибок Java машины** (подклассы класса Error). Программист может обработать такие ошибки, но в большинстве случаев этого делать не приходится.

Преимущества структурной обработки исключений

1. Более простая семантика и структура кода

Наличие исключений позволяет описать основную логику работы кода и указать, как действовать в случае возникновения исключительных (ошибочных) ситуаций

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (InputMismatchException ex)
{
    System.out.println("Ошибка ввода");
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```

Преимущества структурной обработки исключений

2. Обеспечение гарантии освобождения ресурсов даже в случае ошибки

Поскольку блок `finally` срабатывает всегда (в том числе в случае ошибки), его следует использовать для освобождения ресурсов.

```
Scanner sc = new Scanner(System.in);
try
{
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
finally
{
    sc.close();
}
```

Начиная с Java 7, с классами, реализующими `java.lang.AutoCloseable`, можно работать так:

```
try (Scanner sc = new Scanner(System.in))
{
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
```

Преимущества структурной обработки исключений

3. Возможность внедрить в контейнер ошибки дополнительную информацию.

Поскольку исключение – это класс, он может содержать полезную информацию, позволяющую точнее идентифицировать причину ошибки (например, путь к файлу в случае ошибки доступа к файлу).

Преимущества структурной обработки исключений

4. Возможность разделять или группировать ошибки при обработке

Поскольку исключения – это классы, их можно выстраивать в иерархию, тем самым объединяя классы ошибок при их обработке.

```
try
{
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(String.valueOf(a / b));
}
catch (ArithmeticException ex)
{
    System.out.println("Арифметическая ошибка: "+ex.getMessage());
}
catch (Exception ex)
{
    System.out.println("Произошла ошибка: "+ex.getMessage());
}
```

Исключения в C#

1. Модель работы идентична используемой в Java.
2. Проброс исключений в C# работает в разы медленней!
3. Есть predetermined иерархия исключений (более бедная, чем в Java)
4. Нет контролируемых исключений
5. Ловушка для исключения выбирается так же, как и в Java, то есть путем сопоставления классов по иерархии наследования
6. Для перехвата всех исключений можно использовать блок catch без указания типа перехватываемой ошибки (будет подразумеваться тип Exception)
7. По причине низкой эффективности исключений и отсутствия контролируемых исключений в C# ошибочное состояние часто возвращают через результат подпрограммы, а не через проброс исключения.

```

using System;

public class BaseException
    : Exception
{
    protected int m_iCode;
    public BaseException(int iCode)
    { m_iCode = iCode; }

    virtual public void Print()
    {
        Console.Out.WriteLine(
            "Ошибка: {0}", m_iCode);
    }
};

public class ChildException : BaseException
{
    protected int m_iExtendedCode;
    public ChildException(int iCode,
        int iExtendedCode) : base(iCode)
    {
        m_iExtendedCode = iExtendedCode;
    }

    override public void Print()
    {
        base.Print();
        Console.Out.WriteLine(
            "Доп. код: {0}", m_iExtendedCode);
    }
};

```

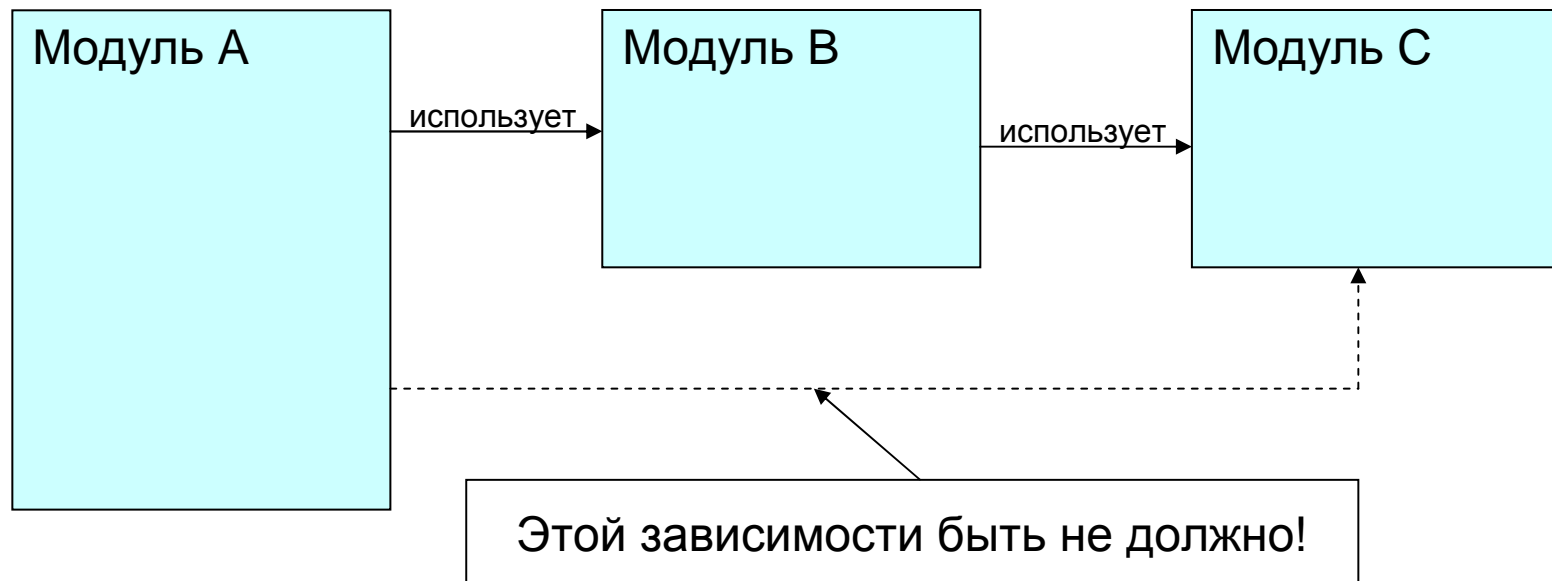
```

public class Tester
{
    public static void Main(string [] args)
    {
        try
        {
            throw new ChildException(10,135);
            // throw new BaseException(15);
            // throw new ArgumentException();
            Console.Out.WriteLine(
                "Эта строка не выполняется");
        }
        catch (ChildException ex)
        {
            Console.Out.WriteLine(
                "catch для ChildException");
            ex.Print();
        }
        catch (BaseException ex)
        {
            Console.Out.WriteLine(
                "catch для BaseException");
            ex.Print();
        }
        catch
        {
            Console.Out.WriteLine(
                "catch для любых ошибок");
        }
    }
}

```

Инкапсуляция в модульных системах

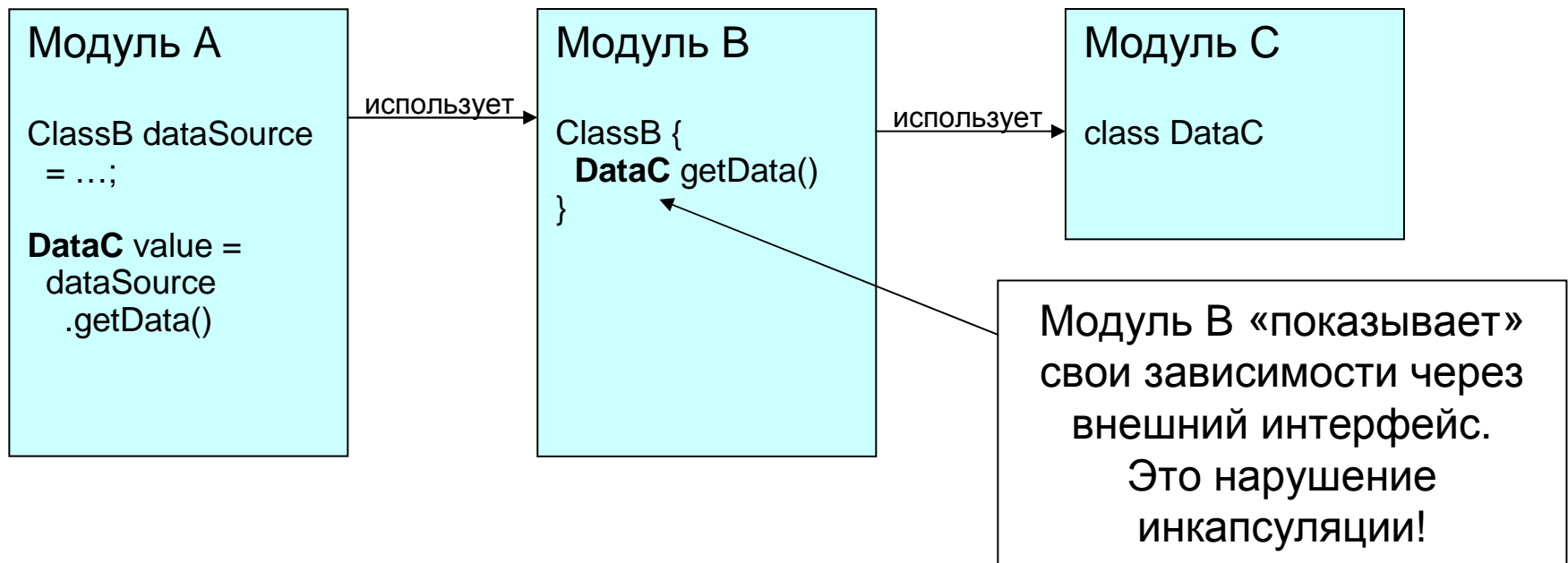
Модуль является самостоятельной программной единицей с собственным программным интерфейсом, скрывающим собственные зависимости модуля (так, зависимость модуля В от какого-то используемого им модуля С не должна быть видна через программный интерфейс модуля В).



Инкапсуляция в модульных системах

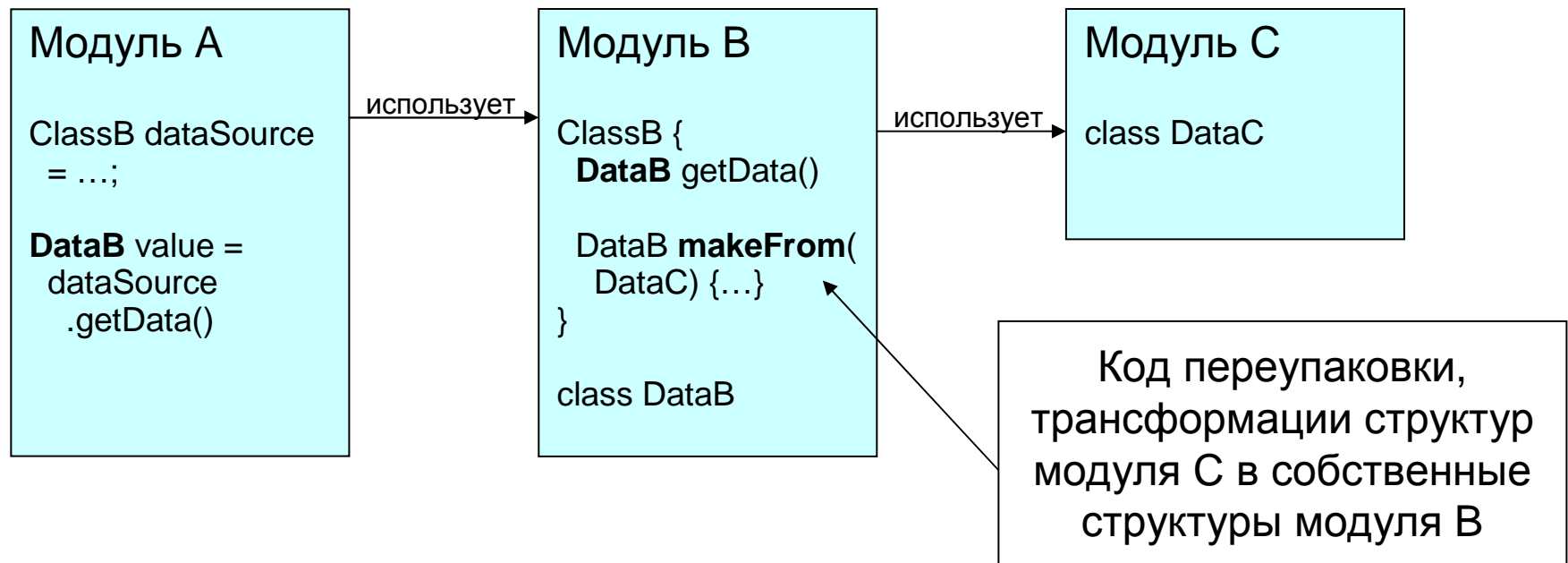
Часто в подобных ситуациях программист может спроектировать интерфейсы модуля В с использованием сущностей из модуля С.

Но это плохая идея, поскольку в этом случае изменение в модуле С приводит к изменениям в модуле А, хотя в рамках инкапсуляции эти изменения должны были затронуть только модуль В, но не зависимые от модуля В части программы.



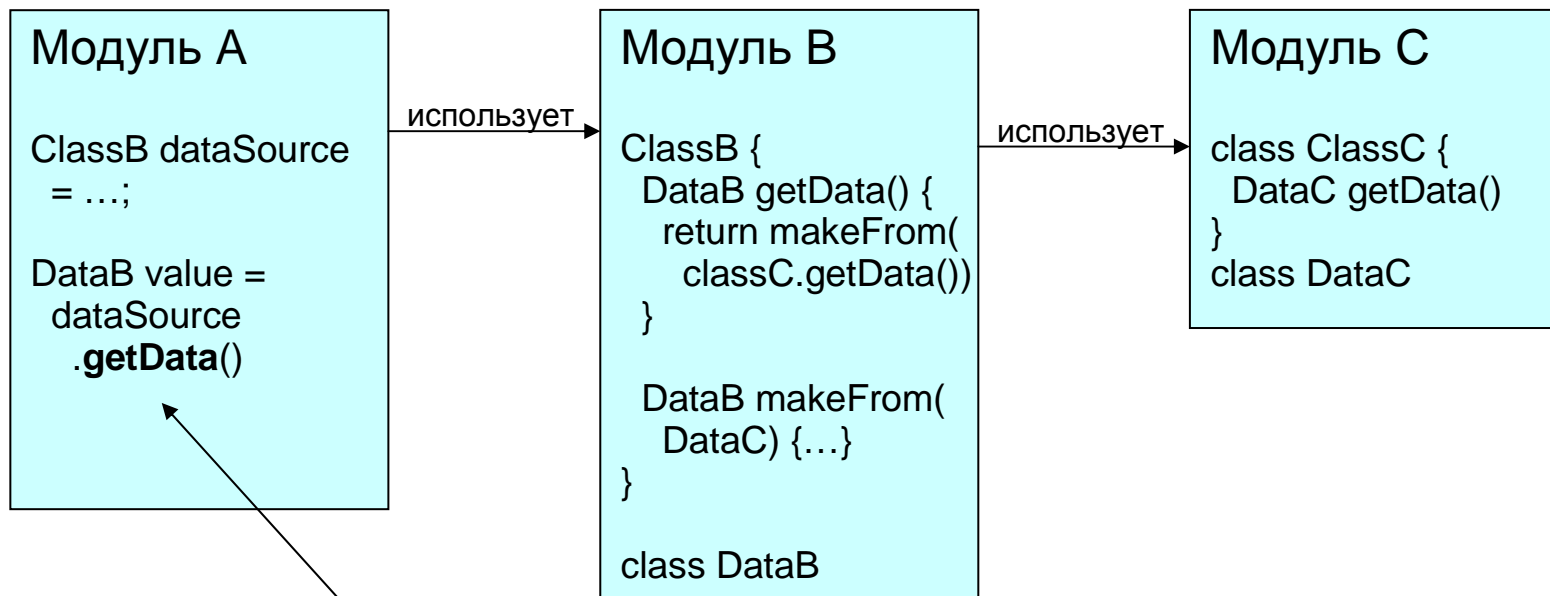
Инкапсуляция в модульных системах

Правильный подход состоит в том, чтобы переупаковать данные, получаемые из модуля С в собственные структуры, объявленные в модуле В.



Инкапсуляция в модульных системах

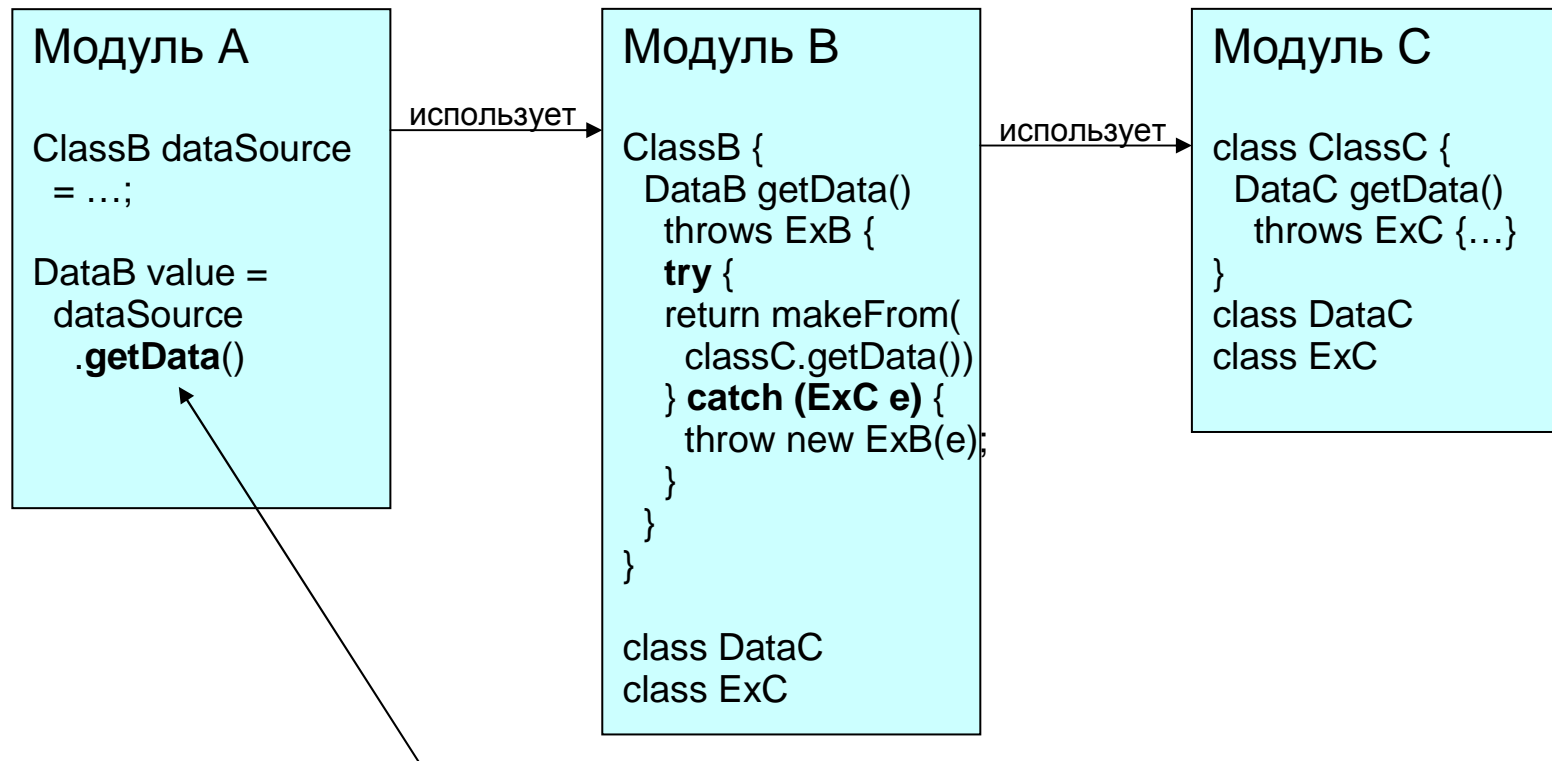
Наблюдение: исключения являются частью интерфейса модуля. Поэтому неконтролируемые исключения порождают неявную зависимость между модулями. И модуль А никак не может проверить, что он обрабатывает все ошибочные ситуации, порождаемые зависимыми модулями.



Здесь можем получить исключение из модуля C. Это опять нарушает инкапсуляцию.

Инкапсуляция в модульных системах

Правильнее делать исключения контролируемыми и явно переупаковывать их при необходимости:



Здесь явно видим, какие ошибки могут возникать при вызове метода. Изменения в модуле C не могут изменить этот набор ошибок (ExB в данном случае).

О контрактах...

Исключения предоставляют удобный механизм обработки ошибок.

Java и C# предоставляют еще один механизм для внедрения блоков контроля в код программы.

Рассмотрим метод, решающий квадратное уравнение:

```
public static double [] solveSqEq(double a, double b, double c)
{
    double [] result;
    double discr = b * b - 4 * a * c;
    if (discr < 0)
    {
        result = new double[0];
    }
    else if (discr == 0)
    {
        result = new double[1];
        result[0] = -b / (2 * a);
    }
    else
    {
        result = new double[2];
        result[0] = (-b + Math.sqrt(discr)) / (2 * a);
        result[1] = (-b - Math.sqrt(discr)) / (2 * a);
    }
    return result;
}
```

Формально код правильный. Но он завершится аварийно, если на вход подать некорректные значения (например, 0 в качестве значения для a). Либо логика кода может оказаться нарушенной в результате чужих правок (например, добавлен код для решения кубических уравнений).

Поэтому полезно в код встраивать блоки самодиагностики.

```

public static double [] solveSqEq(double a, double b, double c)
{
    if (a == 0)
    {
        throw new IllegalArgumentException("a = 0");
    }

    double [] result;
    double discr = b * b - 4 * a * c;
    if (discr < 0)
    {
        result = new double[0];
    }
    else if (discr == 0)
    {
        result = new double[1];
        result[0] = -b / (2 * a);
    }
    else
    {
        result = new double[2];
        result[0] = (-b + Math.sqrt(discr)) / (2 * a);
        result[1] = (-b - Math.sqrt(discr)) / (2 * a);
    }
    if ((result == null) || (result.length > 2))
    {
        throw new IllegalStateException("Something wrong");
    }
    return result;
}

```

блоки самодиагностики

Блоки самодиагностики проверяют выполнение **контракта** метода, то есть правильность входных параметров и корректность результата.

Наблюдение:

Блоки самодиагностики, как правило, нужны лишь на этапе отладки, поскольку в рабочей версии программы подобные нарушения контракта просто не должны возникать. Но рассмотренный пример внедрения самодиагностики не позволяет отключать дополнительные проверки в рабочей версии программы, тем самым замедляя работу программы.

Для решения задачи внедрения проверок контракта в код программы служит инструкция **assert**

Синтаксис в Java:

```
assert <проверяемое условие> [ : <текст сообщения> ] ;
```

Выполнение инструкции `assert` приводит к пробросу исключения `java.lang.AssertionError` и остановке программы, если проверяемое условие ложно. Но сама инструкция `assert` выполняется только при запуске программы в режиме отладки (см. ключи `-ea` и `-da` в параметрах запуска виртуальной машины).

```
public static double [] solveSqEq(double a, double b, double c)
{
    assert (a != 0) : "a = 0";
    double [] result;
    double discr = b * b - 4 * a * c;
    if (discr < 0)
    {
        result = new double[0];
    }
    else if (discr == 0)
    {
        result = new double[1];
        result[0] = -b / (2 * a);
    }
    else
    {
        result = new double[2];
        result[0] = (-b + Math.sqrt(discr)) / (2 * a);
        result[1] = (-b - Math.sqrt(discr)) / (2 * a);
    }
    assert ((result != null) && (result.length <= 2)) : "Something wrong";
    return result;
}
```

Diagram illustrating the self-diagnostic blocks in the code. A box labeled "блоки самодиагностики" (self-diagnostic blocks) has two arrows pointing to the `assert` statements in the `solveSqEq` method: one pointing to `assert (a != 0) : "a = 0";` and another pointing to `assert ((result != null) && (result.length <= 2)) : "Something wrong";`.

Теперь код блоков самодиагностики стал более компактным, и блоки самодиагностики отключаются при обычном (не отладочном) запуске программы.

Assert в C#:

Использование Assert в C# имеет свои особенности:

Синтаксис в C#:

```
using System.Diagnostics;
```

```
...
```

```
Debug.Assert(<проверяемое выражение>[ , <текст  
сообщения> ] );
```

Методы класса Debug будут исполняться только в отладочной версии программы, которую следует собирать, определяя макропеременную DEBUG, например, так:

```
csc /define:DEBUG <файл с исходным кодом>
```

Автоматизированное тестирование

В процессе разработки крупных программ предполагается использование автоматизированного тестирования. В этом случае программист пишет не только код конкретного функционального модуля, но и код серии тестов, позволяющих автоматически проверять правильность работы модуля.

Современные системы разработки содержат готовые средства для прогона систем тестов (например, JUnit в Java и NUnit в C#).

Применение модульных тестов позволяет вовремя обнаруживать наведенные ошибки, когда правки в одной части программы приводят к нарушению функциональности другой части.

Языки и технологии программирования.

Объектно-ориентированное программирование

*Особенности реализации концепций
ООП в языке C#*

Основные возможности Java и C# в области ООП идентичны. Но есть следующие отличительные особенности:

1. виртуальные методы описываются явно;
2. при реализации нескольких интерфейсов можно различать реализации одноименных методов с одной и той же сигнатурой из разных интерфейсов;
3. помимо полей и методов есть свойства (property);
4. имеются делегаты;
5. доступна перегрузка операторов;
6. класс Object устроен немного иначе;
7. описания классов могут быть разделены на несколько файлов.
8. методы-расширения (extension methods)

Наследование в C#

Язык C# требует явного указания, будет ли метод вызываться как виртуальный (ключевое слово `virtual`). Ключевые слова `override` и `new` отвечают за переопределение и замену метода соответственно.

```
class Parent {
    public void NonVirtual() {
        Console.WriteLine(
            "Non virtual - parent");
    }
    public virtual void Virtual() {
        Console.WriteLine(
            "Virtual - parent");
    }
}

class Child : Parent {
    public new void NonVirtual() {
        Console.WriteLine(
            "Non virtual - child");
    }
    public override void Virtual() {
        Console.WriteLine(
            "Virtual - child");
    }
}

Parent p = new Parent();
p.NonVirtual();           // -> Non virtual - parent
p.Virtual();              // -> Virtual - parent
Child c = new Child();
c.NonVirtual();           // -> Non virtual - child
c.Virtual();              // -> Virtual - child
Parent pc = new Child();
pc.NonVirtual();          // -> Non virtual - parent
pc.Virtual();             // -> Virtual - child
```

Наследование в C#

Для запрета наследования от класса или запрета переопределения виртуального метода в потомке следует пометить соответствующие класс или метод ключевым словом **sealed**.

Если требуется выяснить тип ссылки во время исполнения программы, можно использовать любую из следующих инструкций:

```
<переменная> is <тип>
```

Возвращает истину, если переменная содержит ссылку на объект, которую можно присвоить переменной типа <тип>. Это аналог оператора instanceof в языке Java.

```
<переменная> as <тип>
```

Если переменная содержит ссылку на объект, которую можно присвоить переменной типа <тип>, то возвращает эту ссылку, как ссылку на тип <тип> (то есть последующее понижающее преобразование типа не требуется). Иначе возвращает null.

Множественная реализация в C#

Язык C# допускает независимую реализацию разных интерфейсов, содержащих метод с одной и той же сигнатурой. В этом случае при определении метода надо явно указать реализуемый интерфейс:

```
public interface ISample1
{
    void DoIt();
}

public interface ISample2
{
    void DoIt();
}

public class Sample
    : ISample1, ISample2
{
    void ISample1.DoIt() {
        Console.WriteLine(
            "DoIt for ISample1");
    }
    void ISample2.DoIt() {
        Console.WriteLine(
            "DoIt for ISample2");
    }
}

Sample s = new Sample();

ISample1 s1 = s;
s1.DoIt();           // -> DoIt for ISample1

ISample2 s2 = s;
s2.DoIt();           // -> DoIt for ISample2
```

Свойства в С#

Часто в программе требуется описать поле и пару методов (get/set) для доступа к нему. В С# такой код можно упростить:

```
class Sample
{
    public string Name { get; set; }
}
```

```
class Sample
{
    private string name;

    public string GetName()
    { return name; }

    public void SetName(string name)
    { this.name = name; }
}
```


Свойства в С#

Помимо более компактного описания свойства, язык предлагает более компактную инициализацию объекта и более компактный код доступа:

```
class Sample
{
    public string Name { get; set; }
}
```

```
Sample test = new Sample()
{
    Name = "Petrov"
}
```

```
test.Name = "Sidorov";
Console.Out.WriteLine(
    test.Name);
```

```
class Sample
{
    private string name;

    public string GetName()
    { return name; }

    public void SetName(string name)
    { this.name = name; }
}
```

```
Sample test = new Sample();
test.SetName("Petrov");
```

```
test.SetName("Sidorov");
Console.Out.WriteLine(
    test.GetName());
```

Свойства в С#

Свойства могут быть устроены более сложно (так называемые свойства поверх поля / properties with backing field):

```
class Sample
{
    private string name;

    public string Name {
        get
        { return name; }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException();
            name = value;
        }
    }
}
```

Свойства в С#

Свойства могут быть неизменными:

```
class Sample
{
    public string Name { get; private set; }
}
```

Точнее, такие свойства можно изменять только внутри класса.

Свойства в С#

Поскольку свойство – это пара методов, они могут наследоваться (и участвовать в полиморфизме):

```
class Sample
{
    public virtual string Name
        { get; set; }
}

class SampleChild : Sample
{
    public override string Name {
        get
        {
            return base.Name + base.Name;
        }
        set
        {
            base.Name = value;
        }
    }
}
```

```
class EntryPoint
{
    public static void Main(string [] args)
    {
        Sample test = new Sample()
            {Name = "Petrov"};
        Console.Out.WriteLine(test.Name);
        // печатает Petrov

        test = new SampleChild()
            {Name = "Petrov"};
        Console.Out.WriteLine(test.Name);
        // печатает PetrovPetrov
    }
}
```

Свойства в С#

Свойства могут быть частью интерфейса (и участвовать в полиморфизме):

```
public interface ISampleGetter
{
    string Name { get;}
}

public interface ISampleSetter
{
    string Name { set;}
}

class Sample : ISampleGetter, ISampleSetter
{
    public string Name { get; set; }
}
```

Делегаты в С#

Интерфейс – это описание набора методов. То есть переменную типа интерфейс можно трактовать как ссылку на набор методов. В частности, интерфейс может быть из одного метода. То есть получим «ссылку на метод».

Для этой ситуации С# предлагает более компактный код получения такой ссылки и ее использования.

```
class Sample
{
    private static void PrintMsg(string msg) {
        Console.Out.WriteLine(msg);
    }

    delegate void PrintAction(string arg);    // описание типа делегата

    public static void Main(string [] args) {
        PrintAction p = Sample.PrintMsg;    // описание переменной делегата
        p("Hello!");                        // вызов делегата
    }
}
```

Делегаты в С#

Вот так можно имитировать делегаты в языках, где их нет:

```
class Sample {  
    private static void  
        PrintMsg(string msg) {  
        Console.Out.WriteLine(msg);  
    }  
}
```

```
delegate void PrintAction(  
    string arg);
```

```
public static void Main(  
    string [] args) {  
  
    PrintAction p = Sample.PrintMsg;  
    p("Hello!");  
}  
}
```

```
class Sample {  
    private static void  
        PrintMsg(string msg) {  
        Console.Out.WriteLine(msg);  
    }  
}
```

```
interface IPrintAction {  
    void PrintAction(string arg);  
}
```

```
class PrintActionImpl : IPrintAction {  
    public void PrintAction(string arg) {  
        Sample.PrintMsg(arg);  
    }  
}
```

```
public static void Main(  
    string [] args) {  
  
    IPrintAction p =  
        new PrintActionImpl();  
    p.PrintAction("Hello!");  
}  
}
```

Делегаты в С#

Использование делегата всегда подразумевает описание некоторого интерфейса и создание анонимного класса с реализацией этого интерфейса. Поэтому можно создавать делегаты более компактно и гибко:

```
class Sample
{
    private static void PrintMsg(string msg) {
        Console.Out.WriteLine(msg);
    }

    delegate void PrintAction(string arg);    // описание типа делегата

    public static void Main(string [] args) {
        PrintAction p = (s) => {
            Sample.PrintMsg("Message: " + s);
        };
        p("Hello!");                                // вызов делегата
        // печатает
        // Message: Hello!
    }
}
```


Делегаты в C#

Но в такой форме записи возникает соблазн использовать локальные переменные контекста вызова при описании делегата (и язык C# позволяет это сделать):

```
class Sample {  
    private static void PrintMsg(string msg) {  
        Console.Out.WriteLine(msg);  
    }  
  
    delegate void PrintAction(string arg);    // описание типа делегата  
  
    public static void Main(string [] args) {  
        string messagePrefix = "User, ";  
        PrintAction p = (s) => {  
            Sample.PrintMsg(messagePrefix + s);  
        };  
        p("Hello!");                    // вызов делегата  
        // печатает  
        // User, Hello!  
    }  
}
```

Несложно догадаться, что информация из контекста вызова просто размещается в автоматически генерируемом классе делегата.

Делегаты в С#

Делегаты применяются в тех случаях, когда нужно написать полиморфный код, принимающий на вход метод определенной сигнатуры (код, параметризуемый методом):

```
class Logger {  
    public delegate void LogAction(string arg); // описание типа делегата  
  
    public static LogAction LogDelegate {get; set;}  
  
    static Logger() { // статический конструктор  
        LogDelegate = (s) => {}; // «пустая» реализация делегата  
    }  
  
    public static void LogError(string e) {  
        string message = e; // или сами сформировали сообщение  
        LogDelegate(message); // вызов делегата  
    }  
  
    public static void Main(string [] args) {  
        Logger.LogDelegate = (s) => Console.Out.WriteLine(s);  
        Logger.LogError("My message"); // -> печатает MyMessage  
    }  
}
```

Перегрузка операторов в С#

Язык С# позволяет выполнять переопределение операторов:

```
class IntValue {
    private int val;
    public int Value {get {return val;}}
    public IntValue(int v) {
        val = v;
    }

    public static IntValue operator+(IntValue arg1, IntValue arg2) {
        return new IntValue(arg1.Value + arg2.Value);
    }

    public static void Main(string [] args) {
        IntValue v1 = new IntValue(1);
        IntValue v2 = new IntValue(2);
        IntValue v3 = v1 + v2;
        Console.Out.WriteLine(v3.Value);
    }
}
```

Перегрузка операторов в C#

Переопределить можно следующие операторы:

1. Унарные `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
2. Бинарные `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`
3. Бинарные операции сравнения `==`, `!=`, `<`, `>`, `<=`, `>=`. При этом важно сохранять симметричность или антисимметричность отношений!

Во всех случаях переопределяемая операция размещается в классе, для которого выполняется ее переопределение. В качестве первого аргумента должен приниматься объект этого класса.

Операции сокращенных логических или арифметических операций будут использовать переопределенные унарные или бинарные операции.

Операцию индексации массива `[]` переопределить нельзя, но можно определить индексатор, который подменит семантику операции `[]`.

Операцию приведения типа переопределить нельзя, но можно определять свои правила приведения типов.

Перегрузка операторов в С#

Определение индексатора:

```
class StringValue {
    private string val;
    public string Value {get {return val;}}
    public StringValue(string v) {
        val = v;
    }

    // Так описывается индексатор (в данном случае только на чтение):
    public char this[int index] {
        get {
            return val[index];
        }
    }

    public static void Main(string [] args) {
        StringValue v1 = new StringValue("Hello!");
        // Так можно использовать индексатор:
        Console.WriteLine(v1[4]);           // -> напечатает 'o'
    }
}
```

Перегрузка операторов в С#

Переопределение правил приведения типов:

```
class StringValue {
    private string val;
    public string Value {get {return val;}}
    public StringValue(string v) {
        val = v;
    }
    // задание оператора неявного приведения типа
    public static implicit operator String(StringValue arg) {
        return arg.Value;
    }
    // задание оператора явного приведения типа
    public static explicit operator int(StringValue arg) {
        return Int32.Parse(arg.Value);
    }

    public static void Main(string [] args) {
        StringValue v1 = new StringValue("123");
        string s = v1;                // неявное приведение типа
        int i = (int)v1;               // явное приведение типа
        Console.Out.WriteLine(s);      // -> напечатает '123'
        Console.Out.WriteLine(i);      // -> напечатает '123'
    }
}
```

Особенности класса **Object** в **C#**

В классе `Object` в языке `C#` присутствуют лишь 5 методов: `ToString()`, `Equals()`, `GetHashCode()`, `GetType()` и `MemberwiseClone()`. Семантика, соответственно, эквивалентна методам `toString()`, `equals()`, `hashCode()`, `getClass()` и `clone()` в языке `Java`.

Поскольку операторы `==` и `!=` в `C#` могут быть переопределены, класс `Object` содержит статический метод `Object.ReferenceEquals(ref1, ref2)`, который позволяет сравнить две ссылки на равенство.

Метод `Object.Equals(obj1, obj2)` позволяет сравнить два объекта даже с учетом того, что какая-либо из ссылок (или обе) равны `null`.

Для реализации клонирования класс должен реализовать интерфейс `ICloneable` и, как следствие, метод `clone()`. При этом он может пользоваться унаследованным защищенным методом `MemberwiseClone()` для быстрого поверхностного клонирования своих полей.

Частичные классы

Описание класса может быть разделено на несколько файлов:

```
// Код находится в одном файле
partial class StringValue {
    private string val;
    public string Value {get {return val;}}
    public StringValue(string v) {
        val = v;
    }
}

// Код находится в другом файле
partial class StringValue
    // задание оператора неявного приведения типа
    public static implicit operator String(StringValue arg) {
        return arg.Value;
    }
    // задание оператора явного приведения типа
    public static explicit operator int(StringValue arg) {
        return Int32.Parse(arg.Value);
    }
}
```


Частичные классы

В частичных классах можно использовать все возможности языка (по сути, части класса при компиляции просто собираются в один класс).

Легко понять, что части класса не должны конфликтовать между собой.

Например, нельзя сделать одну часть открытой (public), а другую – закрытой (private). Некоторые модификаторы (например, sealed), будучи примененными к одной из частей, оказывают влияние и на остальные.

Все части класса должны быть описаны в одной сборке и должны попасть в один исполнимый модуль.

Методы-расширения

Язык C# позволяет (с точки зрения видимого синтаксиса) вызывать статические методы одного класса как методы другого. Эта возможность удобна в ситуациях, в которых используется шаблон «интерфейс и абстрактный класс», поскольку позволяет собрать «общие» методы в один класс. Но в этом случае классы задействуют такие методы не через наследование, а через механизм методов-расширений.

Методы-расширения должны быть статическими и находиться в статическом классе. Первый аргумент метода-расширения должен быть помечен ключевым словом `this` (метод-расширение будет применяться к типу первого параметра).

Методы-расширения

Пример определения операции Reverse для строк через метод-расширение:

```
static class StringExtensions {
    public static string Reverse(this string source) {
        System.Text.StringBuilder result =
            new System.Text.StringBuilder(source.Length);
        for (int idx = source.Length - 1; idx >= 0; idx--)
            result.Append(source[idx]);
        return result.ToString();
    }
}

public class EntryPoint {
    public static void Main(string [] args) {
        Console.Out.WriteLine("Hello!".Reverse());
        // -> напечатает '!olleH'
    }
}
```

Автоматический вывод типа переменной

Язык C# позволяет не указывать явно тип локальной переменной в тех случаях, когда его можно вывести из контекста. В таких случаях вместо типа переменной пишется ключевое слово **var**:

```
public class EntryPoint {  
    public static void Main(string [] args) {  
        var argLength = args.Length; // тип int  
        var param1 = args[0];         // тип string  
    }  
}
```

Языки и технологии программирования.

Объектно-ориентированное программирование

*Библиотеки стандартных контейнеров
Реализация на базе полиморфизма (Java)*

Стандартные библиотеки контейнеров.

Рассмотренная абстракция Stack представляет собой так называемую контейнерную структуру данных. То есть объект-контейнер хранит какие-то другие объекты (в рассмотренном примере хранятся целые числа).

Наблюдение: логика работы стека не зависит от того, какие элементы в нем хранятся.

Наблюдение: реализации стека целых чисел и стека вещественных чисел будут идентичны с точностью до замены типа данных `int` на тип данных `double`.

Вопрос: каким образом создать «универсальную» реализацию стека, чтобы его можно было использовать для разных типов данных?

Стандартные библиотеки контейнеров.

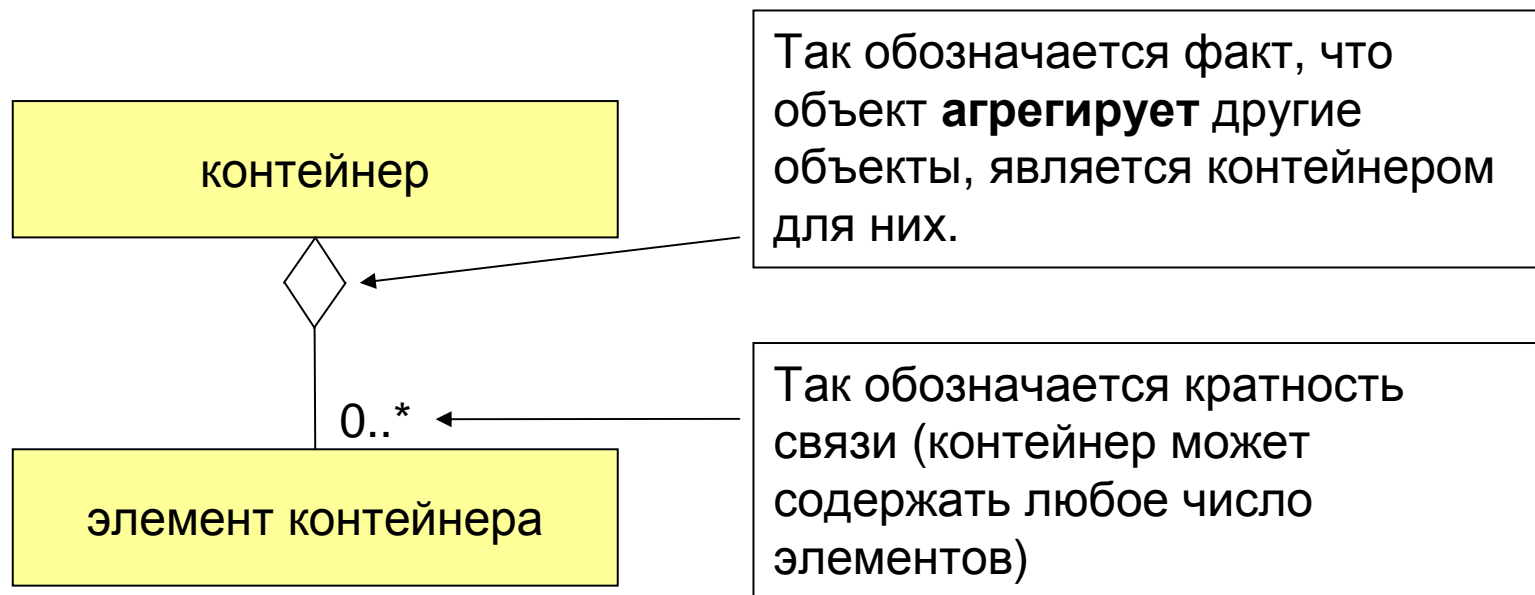
Вопрос: каким образом создать «универсальную» реализацию стека, чтобы его можно было использовать для разных типов данных?

Есть два подхода:

1. использовать абстракции и полиморфизм (годится для Java и C#)
2. использовать шаблонизацию кода (применяется в C#)

Контейнеры на базе полиморфизма.

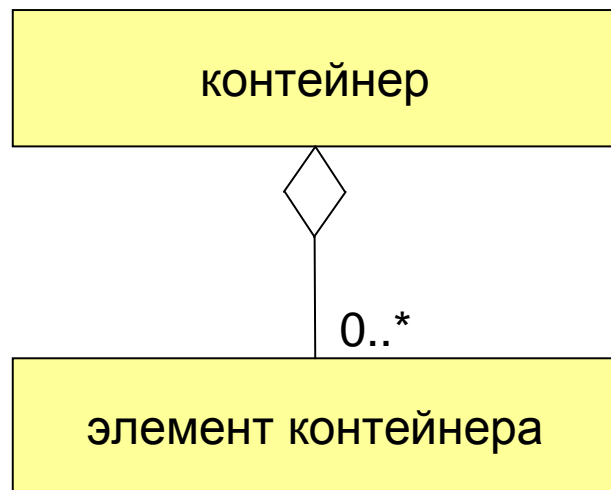
При реализации любого контейнера выделяются два уровня абстракции: уровень контейнера и уровень хранимого элемента.



Контейнеры на базе полиморфизма.

Наблюдение: стратегия работы контейнера как правило не зависит от конкретного типа элемента контейнера.

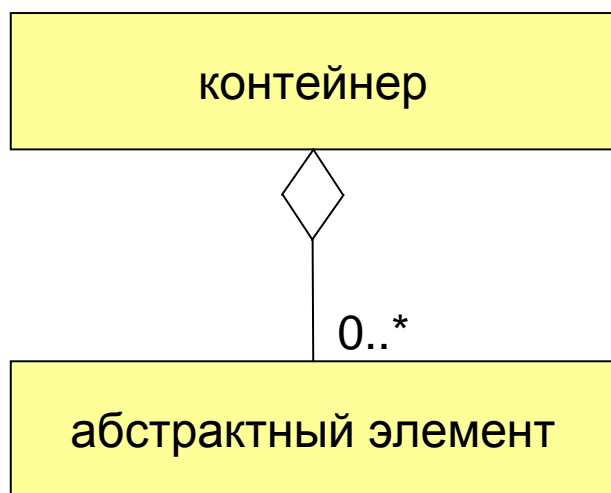
Вопрос: что это значит с точки зрения проектирования?



Контейнеры на базе полиморфизма.

Наблюдение: стратегия работы контейнера как правило не зависит от конкретного типа элемента контейнера.

Вопрос: что это значит с точки зрения проектирования?



Ответ: для контейнера его элемент является абстракцией.

Следовательно, можно описать стратегию работы контейнера в терминах абстракций и полиморфного кода.

Вопрос: что выбрать в качестве абстракции для элемента контейнера?

В Java нужная абстракция есть изначально: это тип `Object`, от которого наследуются все объекты в Java.

Наблюдение: если написать класс, представляющий стек, с использованием типа `Object` в качестве типа элемента стека, то получится универсальный стек, в который можно класть любые элементы.

Универсальный стек в Java

```
public class StackException
    extends Exception
{
    // Набор допустимых кодов ошибок
    public final static int
        STACK_OVERFLOW = 0;
    public final static int
        STACK_EMPTY = 1;

    // Код ошибки
    private int m_code;

    public StackException(int code)
    {
        m_code = code;
    }
}
```

```
// Печатает сообщение об ошибке
public String getMessage()
{
    switch (m_code)
    {
        case STACK_OVERFLOW:
        {
            return "Error: Stack is full!!!";
        }
        case STACK_EMPTY:
        {
            return "Error: Stack is empty!!!";
        }
        default:
        {
            return "Error: Unknown error!!!";
        }
    }
}
}
```

Важно: исключение является частью контракта контейнера, и его нужно проектировать вместе с классом контейнера.

Универсальный стек в Java

```
class Stack
{
    // массив с элементами стека
    private Object [] m_data;
    // число элементов в стеке
    private int m_size;

    public Stack(int maxSize)
    {
        m_data = new Object[0];
        m_size = 0;
        growTo(maxSize);
        System.out.println("Stack created");
    }

    // добавляет элемент в стек
    public void push(Object value)
    {
        if (m_size >= m_data.length)
        {
            growTo(m_data.length * 2);
        }
        m_data[m_size] = value;
        m_size++;
    }

    // извлекает элемент из стека
    public Object pop()
        throws StackException
    {
        if (m_size > 0)
        {
            m_size--;
            return m_data[m_size];
        }
        else
        {
            throw new StackException(
                StackException.STACK_EMPTY);
        }
    }
}
```

Универсальный стек в Java

```
// при необходимости увеличивает размер
// стека до нужной величины
public void growTo(int maxSize)
{
    // Проверяем необходимость
    // расширения стека
    if (maxSize > m_data.length)
    {
        // Удвоение размера приводит к низким
        // распределенным затратам
        // на увеличение размера
        int iNewDataLength = 2 * m_data.length;
        if (maxSize > iNewDataLength)
        {
            iNewDataLength = maxSize;
        }

        Object [] newData =
            new Object[iNewDataLength];

        // Если нужно, переносим уже
        // накопленные данные
        if (m_data != null)
        {
            System.arraycopy(m_data, 0,
                             newData, 0, m_data.length);
        }
        // Старый блок памяти освобожден
        // Можно заменить указатели
        m_data = newData;

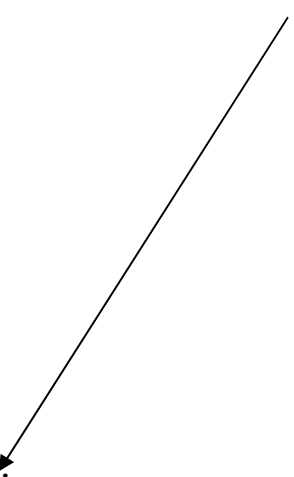
        System.out.println(
            "growTo: new size="
            + m_data.length);
    }

    // возвращает размер стека
    int getSize()
    {
        return m_size;
    }
}
```

Универсальный стек в Java

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest1(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest1()
        throws StackException
    {
        Stack stack = new Stack(2);
        stack.push(new Integer(15));
        stack.push(new Integer(7));
        stack.push(new Integer(23));
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```

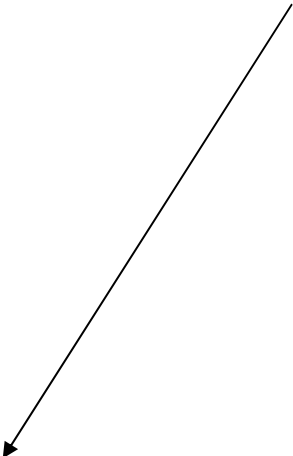


Наблюдение: значения примитивных типов объектами не являются, поэтому для их размещения в полиморфных контейнерах примитивные типы нужно «оборачивать» в парные объектные типы.

Универсальный стек в Java

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest1(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest1()
        throws StackException
    {
        Stack stack = new Stack(2);
        stack.push(new Integer(15));
        stack.push(new Integer(7));
        stack.push(new Integer(23));
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```



Наблюдение: значения примитивных типов объектами не являются, поэтому их нужно «оборачивать» в парные объектные типы.

Результат работы программы:

```
growTo: new size=2
Stack created
growTo: new size=4
Pop: 23
Pop: 7
Pop: 15
StackException: Error: Stack is
empty!!!
    at Stack.pop(Stack.java:73)
    at Test.main(Test.java:14)
```


Универсальный стек в Java

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest2(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest2()
        throws StackException
    {
        Stack stack = new Stack(2);
        stack.push("One");
        stack.push("Two");
        stack.push("Three");
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```

Наблюдение: строковые константы в Java тоже являются объектами.

Результат работы программы:

```
growTo: new size=2
Stack created
growTo: new size=4
Pop: Three
Pop: Two
Pop: One
```

Универсальный стек в Java

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest3(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest3()
        throws StackException
    {
        Stack stack = new Stack(2);
        stack.push(new Integer(15));
        stack.push("Two");
        stack.push(Boolean.FALSE);
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```

Полиморфность позволяет использовать получившийся стек с любыми объектными элементами!

Результат работы программы:

growTo: new size=2
Stack created
growTo: new size=4
Pop: false
Pop: Two
Pop: 15

Универсальный стек в Java

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest4(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest4()
        throws StackException
    {
        Stack stack = new Stack(2);
        stack.push(15);
        stack.push("Two");
        stack.push(Boolean.FALSE);

        while (stack.getSize() > 0)
        {
            Object obj = stack.pop();
            System.out.println("Pop: "+obj);
            System.out.println("CN: "+
                obj.getClass().getName());
        }
    }
}
```

Начиная с JDK 1.5 (Java 5),
обертки для примитивных
типов могут создаваться
компилятором автоматически.

Попросим Java показать имена
классов объектов,
извлекаемых из стека

Результат работы программы:

```
growTo: new size=2
Stack created
growTo: new size=4
Pop: false
CN: java.lang.Boolean
Pop: Two
CN: java.lang.String
Pop: 15
CN: java.lang.Integer
```

Достоинства и недостатки полиморфизма при проектировании контейнеров:

+ полиморфизм позволяет создать одну копию кода, способную обслуживать объекты разных типов. Как следствие, уменьшается дублирование кода, упрощается его отладка и поддержка.

- полиморфизм требует создания объектных оберток для примитивных типов (небольшой проигрыш в быстродействии).

- контейнер на основе полиморфизма обычно не способен контролировать тип своих элементов. Например, в рассмотренном выше примере универсального стека невозможно обеспечить контроль за тем, что в стеке лежат только целые числа, без создания подкласса класса Stack.

Как следствие, даже если программист точно знает, что в стеке будут лежать лишь целые числа, он вынужден писать примерно следующий код:

```
Stack stack = new Stack(2);  
stack.push(new Integer(15));  
Integer intValue = (Integer)stack.pop();
```

То есть при извлечении элемента из стека приходится делать downcasting, понижающее (в иерархии классов) приведение типа, что может быть чревато ошибками, поскольку сама конструкция класса Stack никак не гарантирует, что возвращаемый элемент будет являться допустимым для подобного преобразования.

Проблема: при реализации контейнеров на основе полиморфизма всегда возникает потенциальная опасность некорректного использования понижающего преобразования типов при извлечении элементов из контейнера.

В JDK 1.5 (Java 5) было введено расширение языка, позволяющее при использовании контейнерных типов автоматически контролировать тип сохраняемых в контейнере данных.

В JDK 1.5 (Java 5) было введено расширение языка, позволяющее при использовании контейнерных типов автоматически контролировать тип сохраняемых в контейнере данных.

Тем самым, компилятор получил возможность выполнять проверку совместимости типов элементов контейнера еще на этапе компиляции, сохраняя при этом преимущества реализации контейнеров на основе полиморфизма.

Поэтому с использованием стандартных контейнеров Java 5 получаем возможность оформлять код следующим образом:

```
java.util.Stack<Integer> stack = new java.util.Stack<Integer>();  
stack.push(new Integer(15));  
Integer intValue = stack.pop();
```

В этом случае компилятор сам следит за тем, чтобы в стек могли попасть лишь элементы класса Integer или его подклассов. Следовательно, понижающее преобразование к типу Integer при получении элемента из контейнера становится ненужным.

В составе стандартных библиотек контейнеров обычно выделяются следующие абстракции:

1. Набор элементов (список с прямым или с последовательным доступом)
2. Множество элементов (набор элементов с эффективными операциями объединения, пересечения, взятия дополнения и проверки принадлежности)
3. Отображение (сопоставление элементов одного типа элементам другого типа)
4. Итератор (класс, обеспечивающий просмотр набора объектов)

Рассмотрим диаграмму стандартной библиотеки контейнеров Java.

Схема интерфейсов стандартной библиотеки контейнеров Java

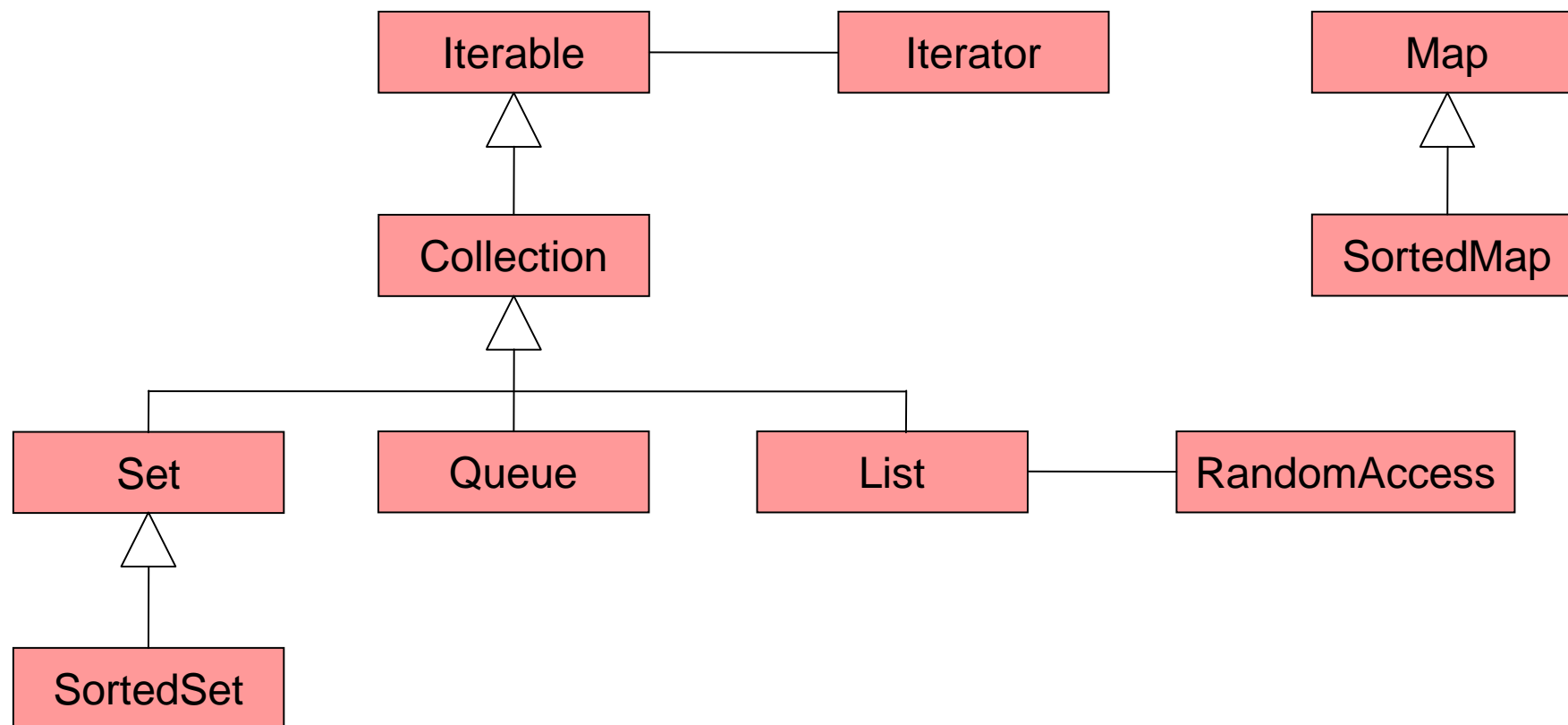
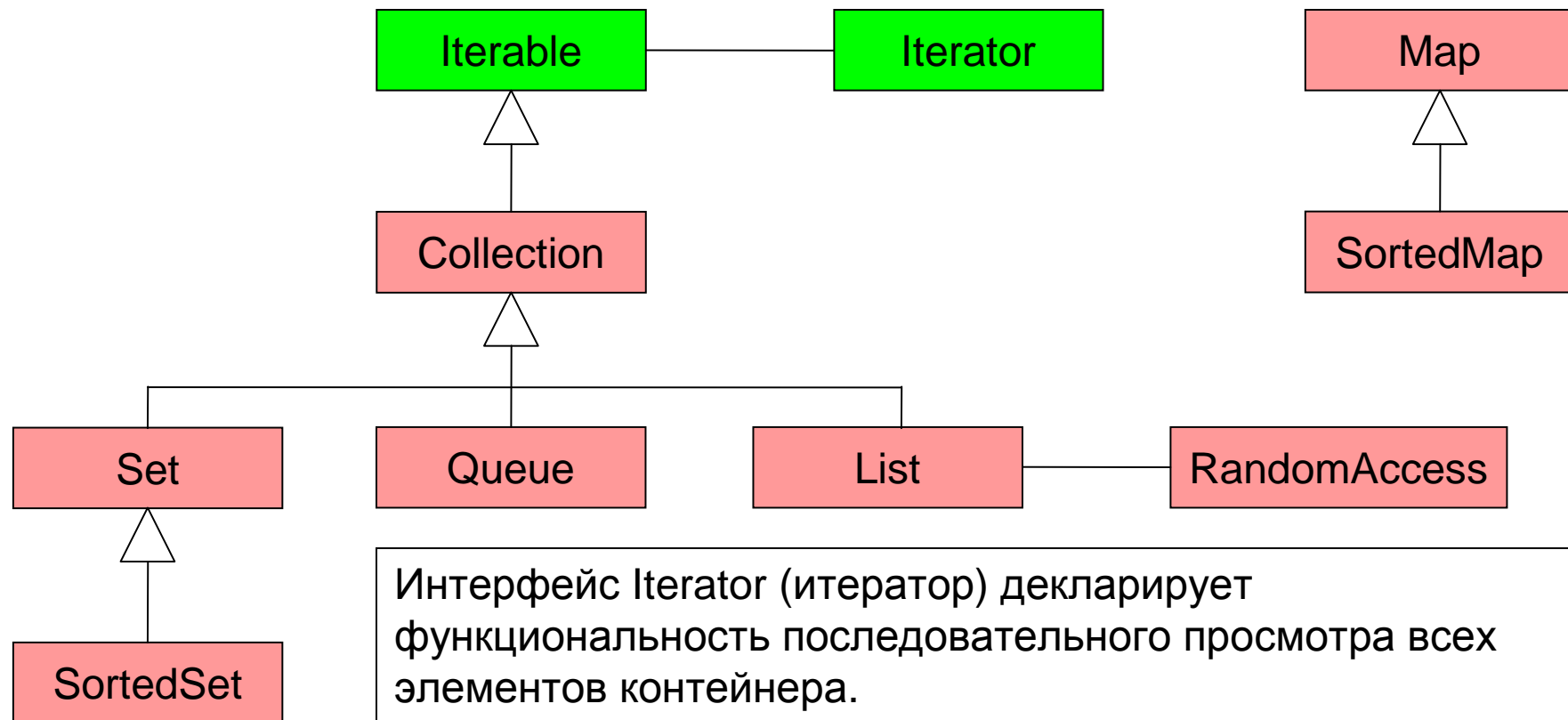


Схема интерфейсов стандартной библиотеки контейнеров Java



Интерфейс `Iterator` (итератор) декларирует функциональность последовательного просмотра всех элементов контейнера.

Интерфейс `Iterable` декларирует возможность получения итератора у контейнера.

Пример использования итераторов

```
import java.util.*;

public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest1(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest1()
    {
        Stack stack = new Stack();
        stack.push(new Integer(15));
        stack.push(new Integer(7));
        stack.push(new Integer(23));
        Iterator iter = stack.iterator();
        while (iter.hasNext())
        {
            Object obj = iter.next();

            System.out.println("Element: "+obj);
        }
    }
}
```

В этом примере используется класс Stack, входящий в состав Java

Так используются итераторы

Результат работы программы:

Element: 15
Element: 7
Element: 23

Наблюдение: контейнер может быть устроен просто или сложно, но наличие итератора позволяет выделить функциональность просмотра контейнера в виде абстракции, скрыть особенности устройства конкретного контейнера от использующего его кода.

Последовательный перебор (итерирование) элементов контейнера предполагает использование цикла. Для упрощения итерирования в Java 5 введен специальный синтаксис цикла `for` для применения с итерируемыми (реализующими интерфейс `Iterable`) объектами:

```
Iterable<Type> collection = ...;

for (Type element : collection)
{
    // обработать element
}
```

Эквивалентный код без использования специальной версии цикла `for`:

```
Iterable collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext())
{
    Type element = (Type)iterator.next();
    // обработать element
}
```

Схема интерфейсов стандартной библиотеки контейнеров Java

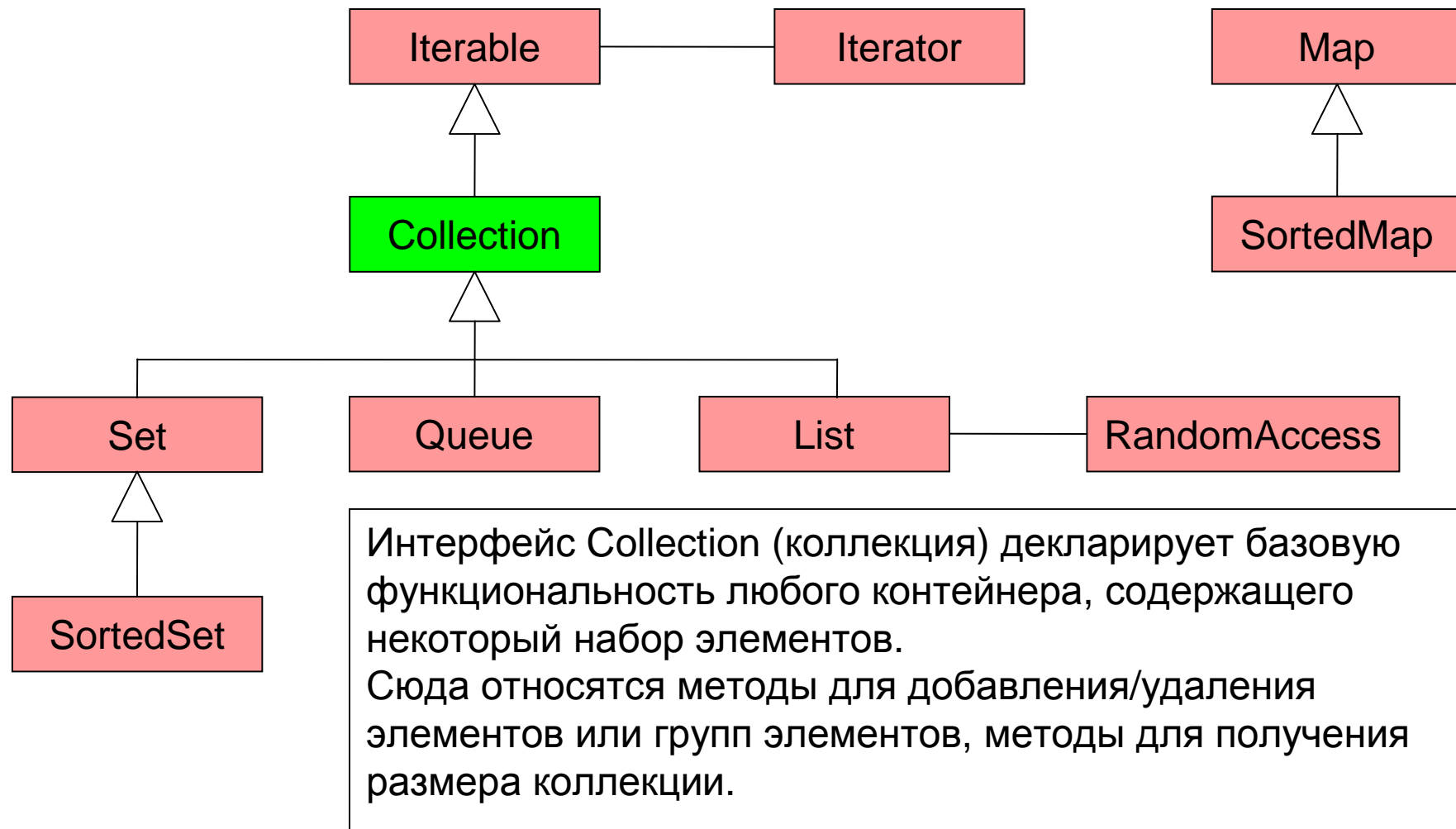


Схема интерфейсов стандартной библиотеки контейнеров Java

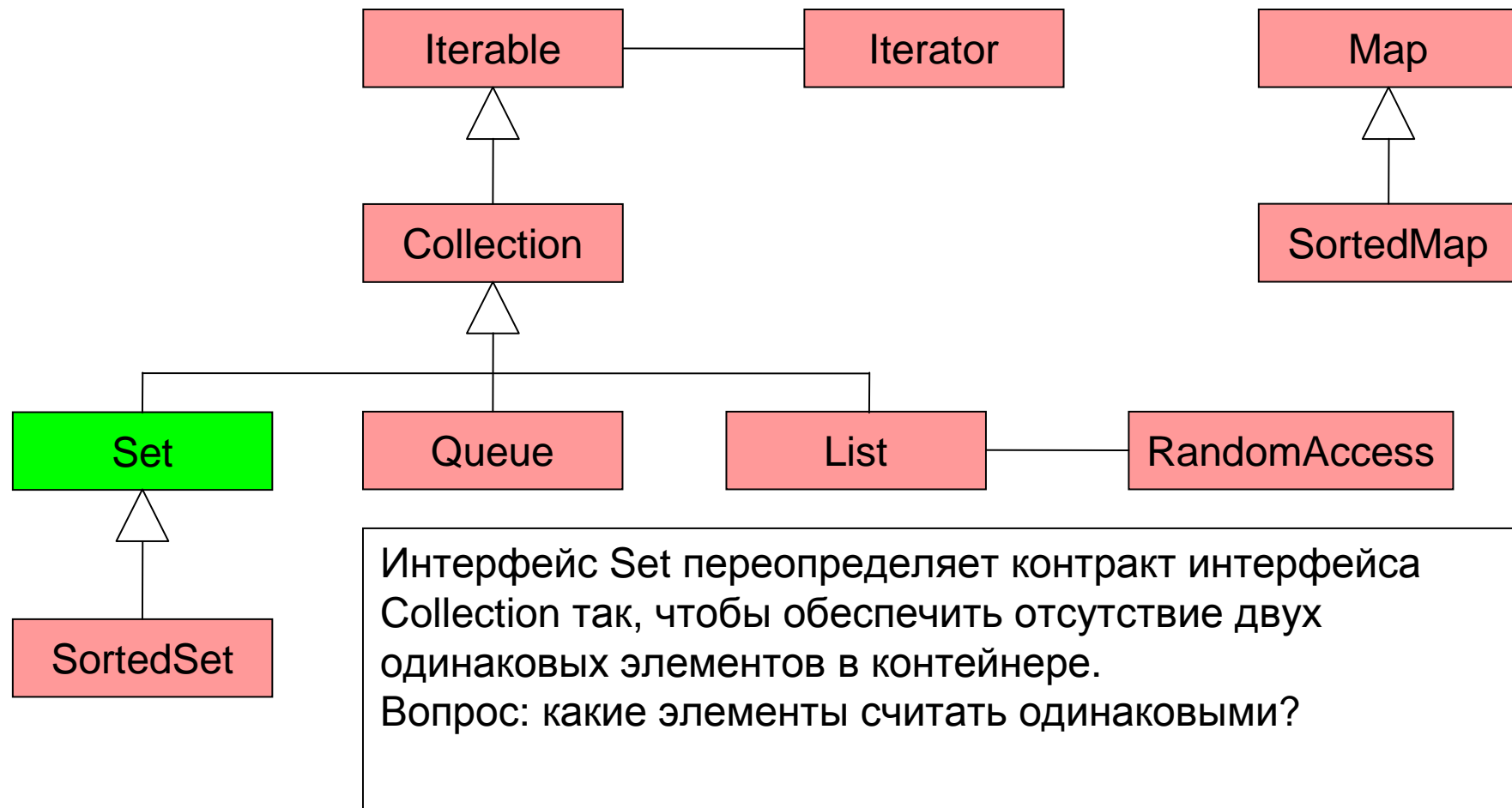
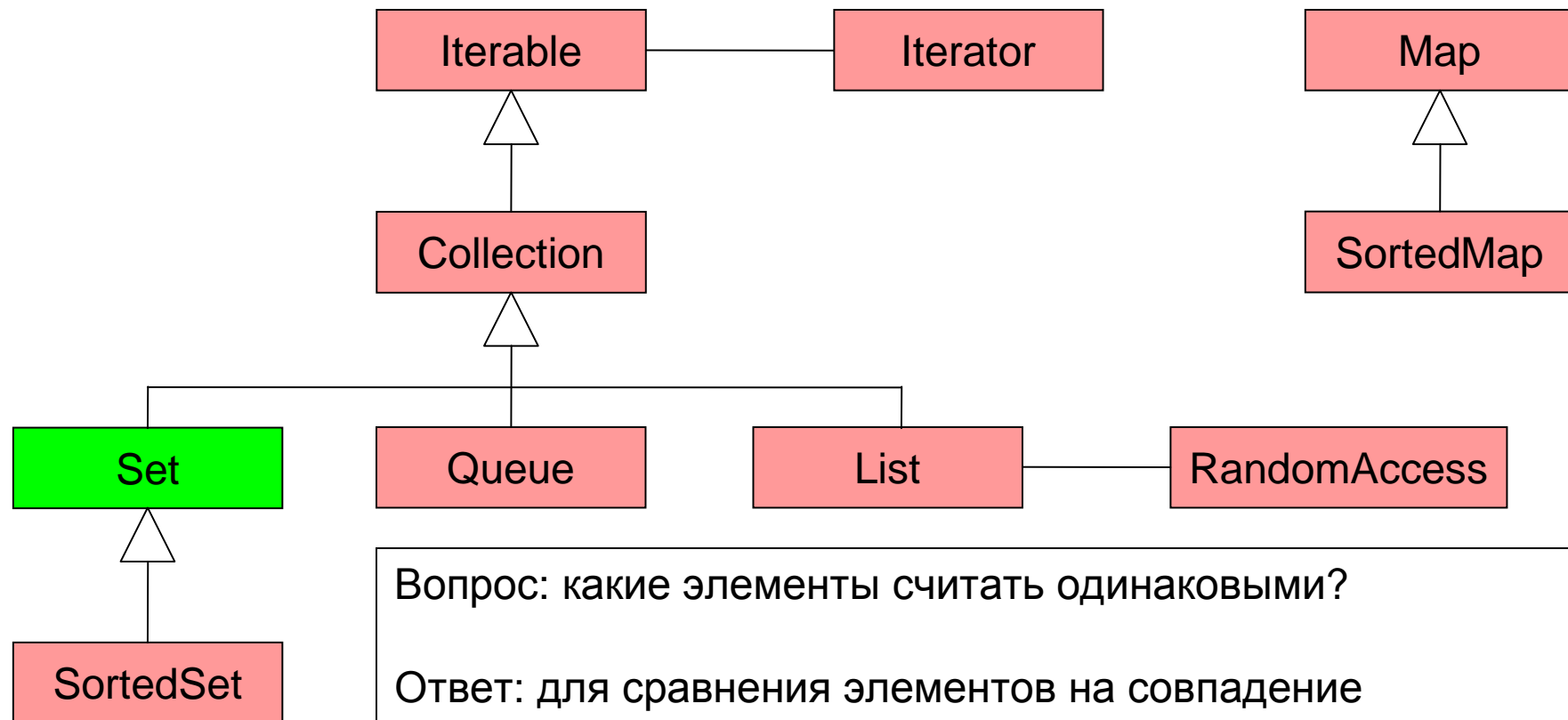


Схема интерфейсов стандартной библиотеки контейнеров Java



Вопрос: какие элементы считать одинаковыми?

Ответ: для сравнения элементов на совпадение используются методы `Object.hashCode()` и `Object.equals()`. Путем переопределения этих методов в классах элементов можно влиять на поведение контейнеров.

Схема интерфейсов стандартной библиотеки контейнеров Java

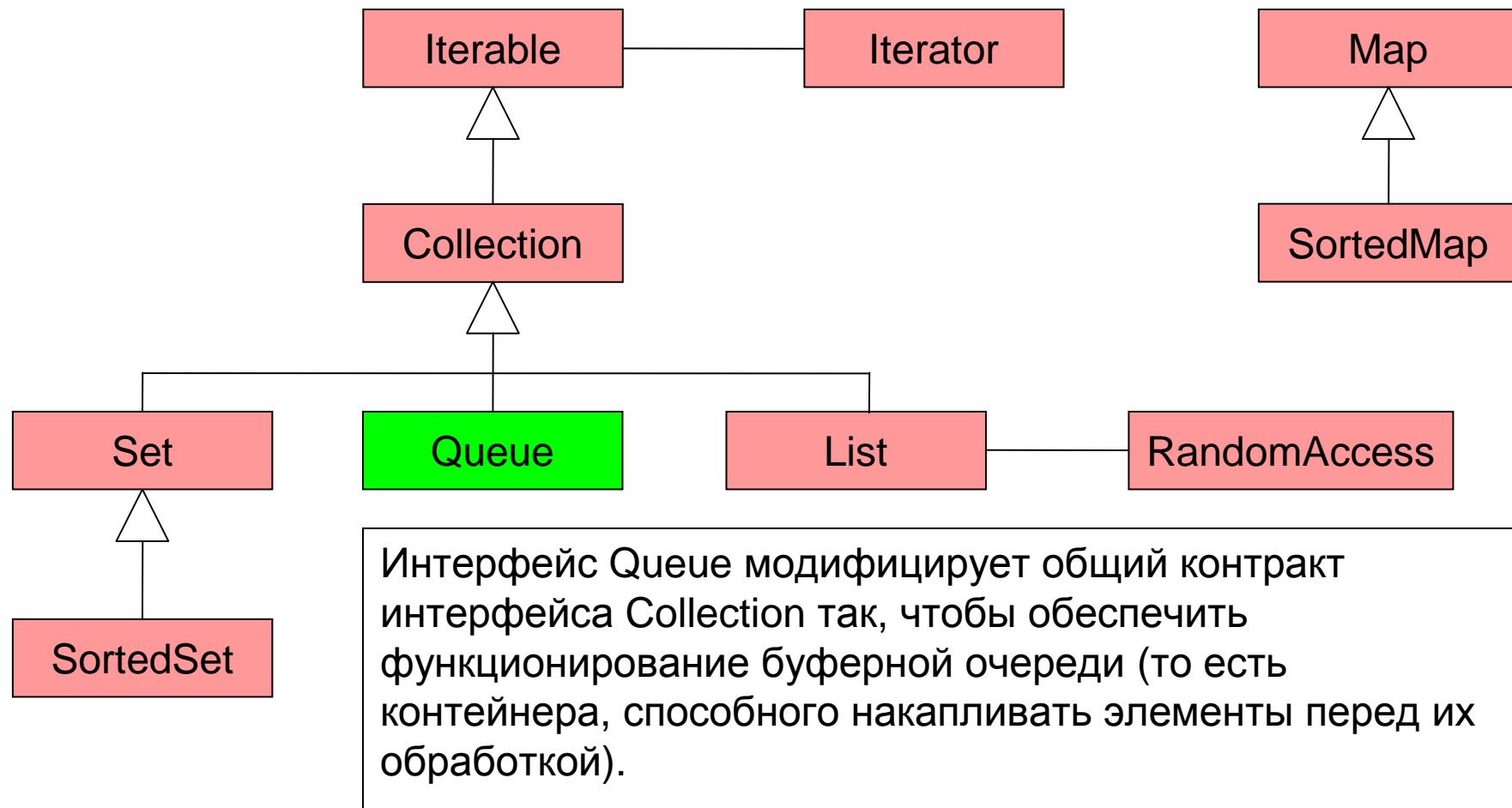


Схема интерфейсов стандартной библиотеки контейнеров Java

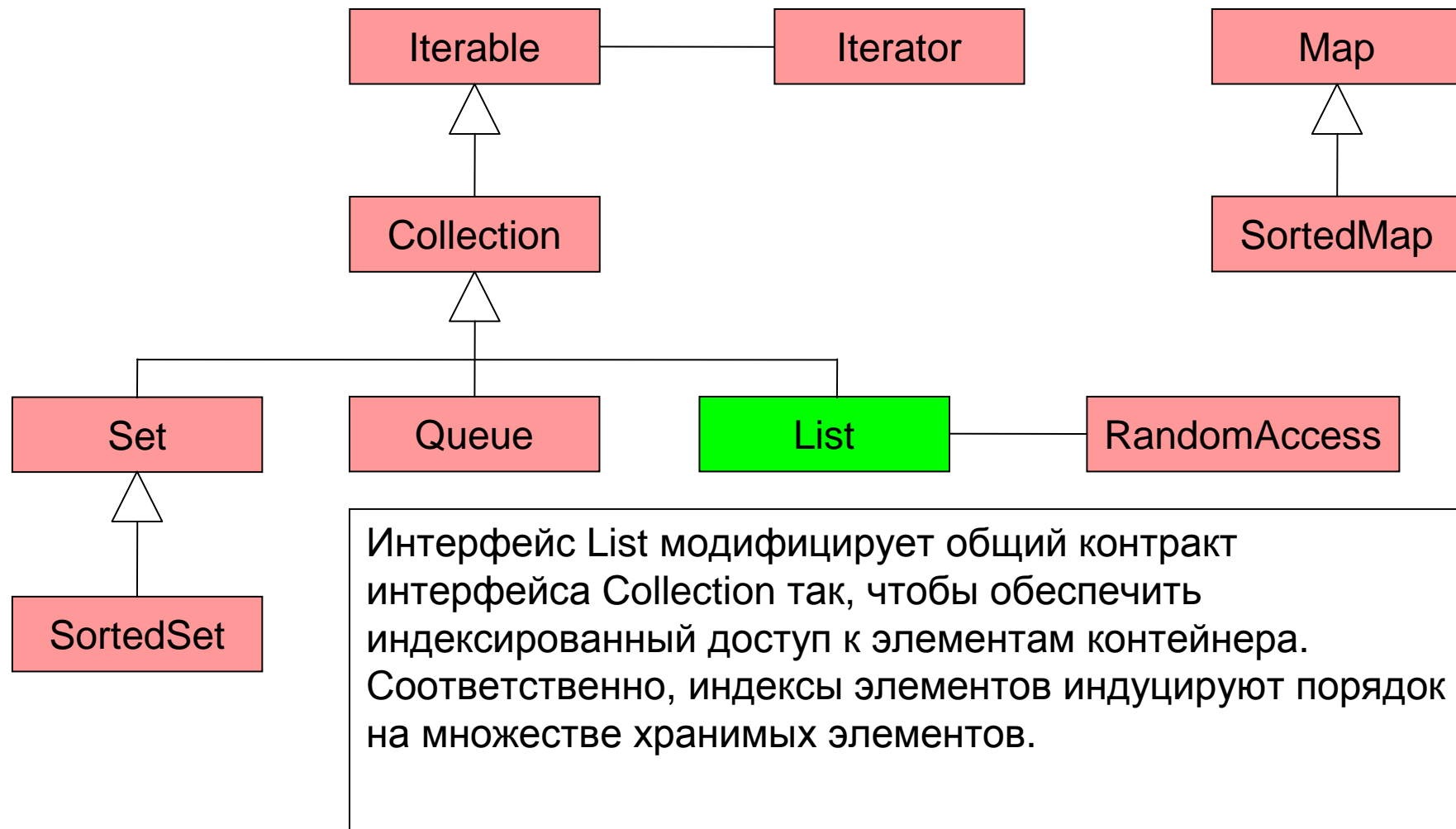
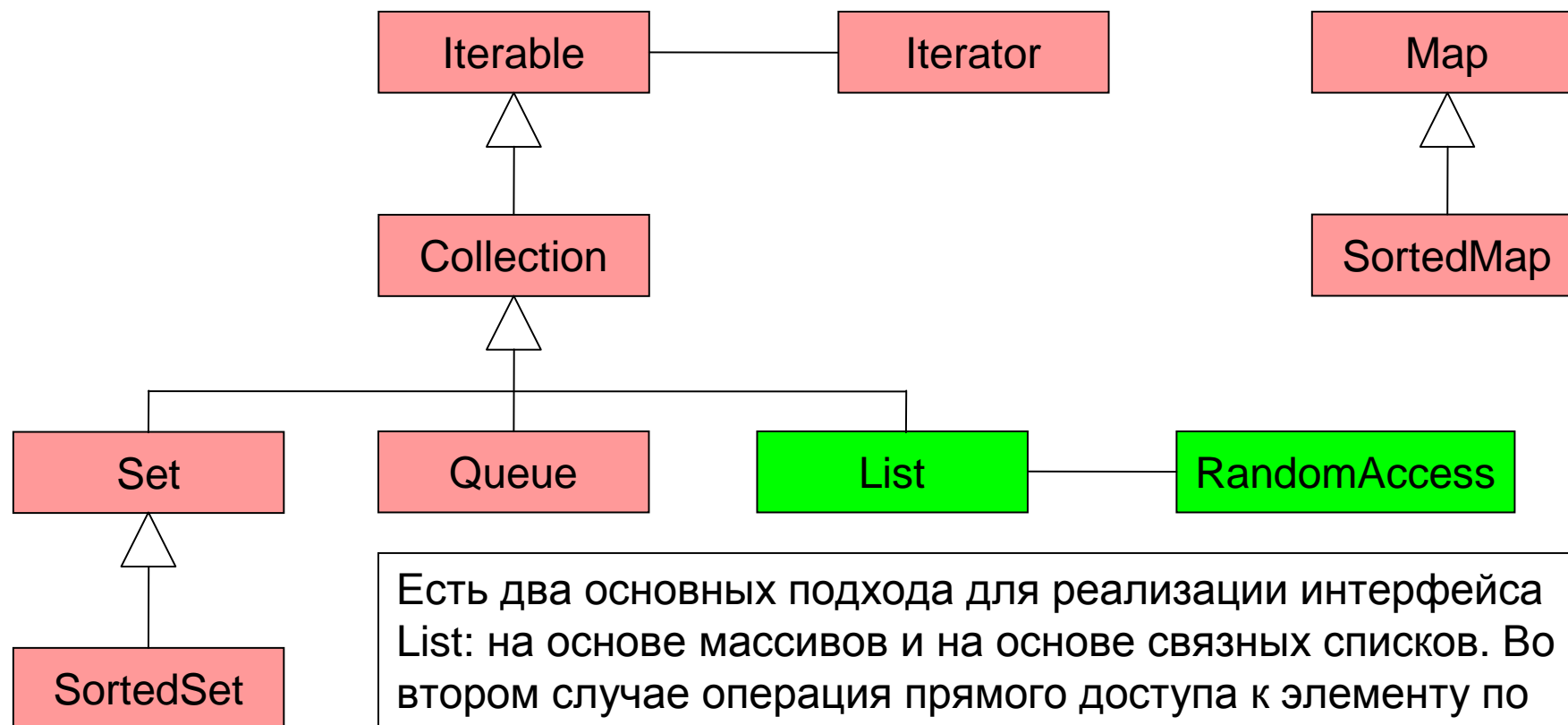


Схема интерфейсов стандартной библиотеки контейнеров Java



Есть два основных подхода для реализации интерфейса **List**: на основе массивов и на основе связанных списков. Во втором случае операция прямого доступа к элементу по индексу может быть дорогой. Если коллекция реализует маркирующий интерфейс **RandomAccess**, это означает, что операция прямого доступа является дешевой.

Сравнение коллекций

```
public static
void testCollection(Collection coll)
{
    System.out.println("Testing: "
        +coll.getClass().getName());
    coll.add(new Integer(15));
    coll.add(new Integer(5));
    coll.add(new Integer(10));
    coll.add(new Integer(5));
    coll.add(new Integer(1));
    Iterator iter = coll.iterator();
    while (iter.hasNext())
    {
        Object obj = iter.next();
        System.out.println("Element: "+obj);
    }
}

public static void doTest2()
{
    testCollection(new ArrayList(10));
    testCollection(new LinkedList());
    testCollection(new HashSet(10));
    testCollection(new PriorityQueue(10));
}
```

Результат работы программы:

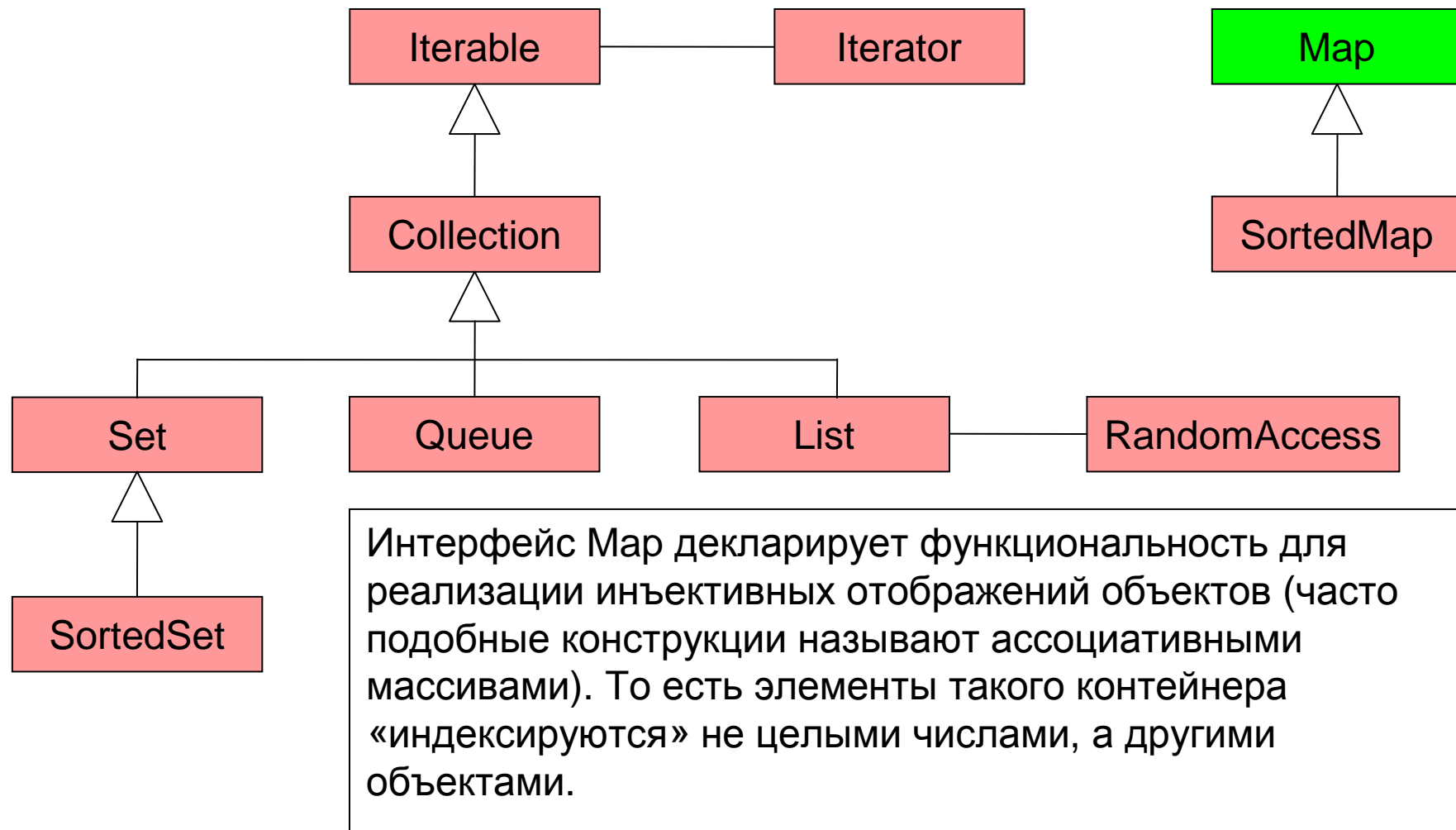
```
Testing: java.util.ArrayList
Element: 15
Element: 5
Element: 10
Element: 5
Element: 1
Testing: java.util.LinkedList
Element: 15
Element: 5
Element: 10
Element: 5
Element: 1
Testing: java.util.HashSet
Element: 15
Element: 1
Element: 10
Element: 5
Testing: java.util.PriorityQueue
Element: 1
Element: 5
Element: 10
Element: 15
Element: 5
```

Важно: при обходе контейнера итератором вообще говоря не гарантируется наличие какого-то конкретного порядка обхода. Часто порядок обхода индуцируется внутренней организацией контейнера (у ArrayList и LinkedList порядок обхода естественный, у PriorityQueue порядок индуцирован устройством пирамиды (heap), используемой для упорядочения элементов по приоритету).

Сортирующие контейнеры явно определяют порядок обхода элементов итератором.

Если же контейнер не определяет порядок обхода, то программист не в праве полагаться на знание порядка обхода, реализованного какой-либо конкретной реализацией контейнера (например, реализации HashSet и TreeSet контейнера Set выполняют обход элементов в разном порядке).

Схема интерфейсов стандартной библиотеки контейнеров Java



Пример использования отображений

```
public static void doTest3()
{
    System.out.println("Testing HashMap");
    Map<String, String> map =
        new HashMap<String, String>(10);
    map.put("day", "monday");
    map.put("month", "November");
    map.put("year", "2006");
    System.out.println(map.get("day"));
    System.out.println(map.get("month"));
    System.out.println(map.get("year"));

    System.out.println("Keys are:");
    for (String key : map.keySet())
    {
        System.out.println(key);
    }
}
```

Результат работы программы:

```
Testing HashMap
monday
November
2006
Keys are:
month
day
year
```

Итераторы настолько часто встречаются в программах, что для них был специально доработан синтаксис цикла for.

В таком виде цикл for может применяться к любому классу, реализующему интерфейс Iterable.

Пример использования отображений

```
public static void doTest3()
{
    System.out.println("Testing HashMap");
    Map<String, String> map =
        new HashMap<String, String>(10);
    map.put("day", "monday");
    map.put("month", "November");
    map.put("year", "2006");
    System.out.println(map.get("day"));
    System.out.println(map.get("month"));
    System.out.println(map.get("year"));

    System.out.println("Keys are:");
    for (String key : map.keySet())
    {
        System.out.println(key);
    }

    // эквивалентный цикл
    Iterator<String> iter =
        map.keySet().iterator();
    while (iter.hasNext())
    {
        String key = iter.next();
        System.out.println(key);
    }
}
```

Результат работы программы:

```
Testing HashMap
monday
November
2006
Keys are:
month
day
year
```

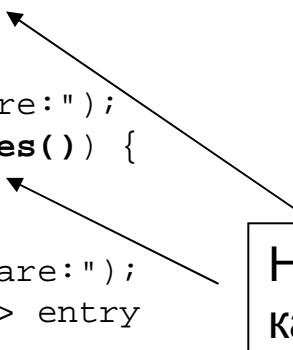
Эти два цикла эквивалентны



Пример использования отображений

```
public static void doTest3()
{
    System.out.println("Testing HashMap");
    Map<String, String> map =
        new HashMap<String, String>(10);
    map.put("day", "monday");
    map.put("month", "November");
    map.put("year", "2006");
    System.out.println(map.get("day"));
    System.out.println(map.get("month"));
    System.out.println(map.get("year"));

    System.out.println("Keys are:");
    for (String key : map.keySet()) {
        System.out.println(key);
    }
    System.out.println("Values are:");
    for (String value : map.values()) {
        System.out.println(value);
    }
    System.out.println("Entries are:");
    for (Map.Entry<String, String> entry
        : map.entrySet()) {
        System.out.println(entry.getKey()
            + " -> " + entry.getValue());
    }
}
```

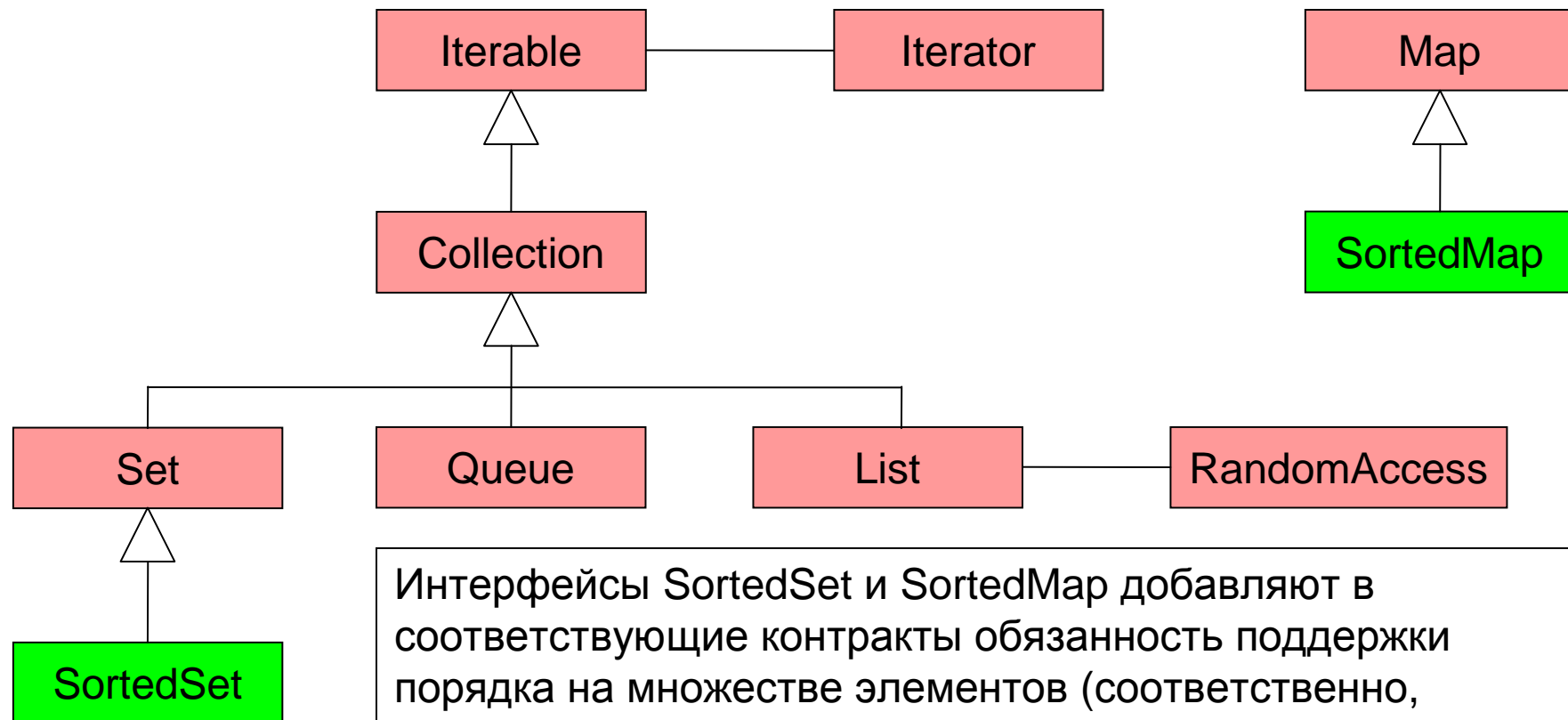


Результат работы программы:

```
Testing HashMap
monday
November
2006
Keys are:
month
day
year
Values are:
November
2006
monday
Entries are:
month -> November
year -> 2006
day -> monday
```

На отображение можно смотреть как на множество ключей, коллекцию значений и множество пар (ключ, значение)

Схема интерфейсов стандартной библиотеки контейнеров Java



Интерфейсы **SortedSet** и **SortedMap** добавляют в соответствующие контракты обязанность поддержки порядка на множестве элементов (соответственно, итераторы возвращают элементы в указанном порядке). Для указания правила сравнения элементов используются интерфейсы **Comparable** и **Comparator**.

В случае, если требуется обеспечить возможность сравнения каких-либо объектов друг с другом, есть два подхода.

1. Реализовать в классе, элементы которого должны уметь сравниваться друг с другом, интерфейс Comparable.

Для этого достаточно определить метод:

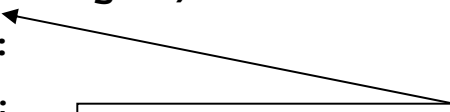
```
public int compareTo(Object o);
```

Данный метод должен вернуть положительное (соответственно отрицательное) число, если объект, у которого вызван метод «больше» (соответственно, «меньше»), чем объект-аргумент и ноль, если объекты «совпадают».

Важно: если на вход в метод compareTo передается класс, сравнение с которым невозможно, метод может завершить работу с пробросом исключения ClassCastException.

Например, для целых чисел (класс `Integer`) этот метод выглядит так:

```
public int compareTo(Object anotherInteger)
{
    int thisVal = this.value;
    int anotherVal = ((Integer)anotherInteger).value;
    return (thisVal < anotherVal ? -1 :
        (thisVal == anotherVal ? 0 : 1));
}
```



Допустимость понижающего преобразования входит в контракт метода

Соответственно, так выглядит работа этого метода:

```
Integer a = new Integer(5); Integer b = new Integer(11);
Integer c = new Integer(5);
a.compareTo(b)    =>   -1
b.compareTo(a)    =>    1
a.compareTo(c)    =>    0
```

Наблюдение: контракт интерфейса Comparable не регламентирует величину возвращаемых значений, фиксируется лишь знак. Поэтому такая реализация метода compareTo также была бы верной:

```
public int compareTo(Object anotherInteger)
{
    int thisVal = this.value;
    int anotherVal = ((Integer)anotherInteger).value;
    return (thisVal < anotherVal ? -3 :
        (thisVal == anotherVal ? 0 : 17));
}
```

Следствие: будет ошибкой использование метода compareTo способом, подобным следующему:

```
if (a.compareTo(b) == 1)
{
    // считаем, что a больше b
}
```

Легко понять, что подход с использованием интерфейса Comparable годится для классов, проектируемых как тип данных. В этом случае, как правило, на множестве элементов изначально имеется естественное упорядочение, и именно его можно определить в методе compareTo().

Иногда правила сравнения объектов могут изменяться во времени, либо таких правил может быть несколько. Например, при работе со строками всегда есть несколько правил сравнения:

- лексикографическое сравнение, символы сравниваются по кодам символов в кодировке Unicode (в этом порядке буква 'Ё' меньше буквы 'А', но буква 'ё' больше буквы 'а');
- лексикографическое сравнение, символы сравниваются с учетом упорядочения букв в алфавите пользователя (в этом порядке буква 'Ё' всегда больше буквы 'А', причем без учета регистра).

Поэтому в Java есть еще один способ определения правил сравнения объектов внешним по отношению к сравниваемым объектам образом:

2. Реализовать в некотором классе (этот класс будет отвечать за сравнение поступающих на вход элементов) интерфейс `Comparator`, состоящий из единственного метода:

```
int compare(Object o1, Object o2);
```

Как и в случае с реализацией интерфейса `Comparable`, данный метод должен вернуть:

- положительное число, если $o1 > o2$;
- отрицательное число, если $o1 < o2$;
- ноль, если $o1 == o2$.

Для строк есть готовая реализация интерфейса `Comparator`, которая позволяет сравнивать строки с учетом особенностей конкретного человеческого разговорного языка. Для этого служит класс `Collator` (от англ. *collation sequence*).

Пример сравнения строк

```
import java.util.*;
import java.text.*;

public class Test
{
    public static void main(String [] args)
    {
        try { doTest1(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest1()
    {
        int cmpResult1 = "Ё".compareTo("A");
        int cmpResult2 = "ё".compareTo("a");
        System.out.println("Ё vs A : "+cmpResult1+ "   ё vs a : "+cmpResult2);
        Locale loc = new Locale("ru", "RU");
        Collator collator = Collator.getInstance(loc);
        cmpResult1 = collator.compare("Ё","A");
        cmpResult2 = collator.compare("ё","a");
        System.out.println("Ё vs A : "+cmpResult1+ "   ё vs a : "+cmpResult2);
        cmpResult1 = collator.compare("Ё","a");
        cmpResult2 = collator.compare("ё","A");
        System.out.println("Ё vs a : "+cmpResult1+ "   ё vs A : "+cmpResult2);
    }
}
```

Результат работы программы:

```
Ё vs A : -15   ё vs a : 33
Ё vs A : 1    ё vs a : 1
Ё vs a : 1    ё vs A : 1
```

Общий контракт контейнеров, обладающих способностью к сортировке, выглядит так:

- либо при инициализации контейнера следует снабдить его экземпляром класса, реализующего интерфейс `Comparator` (тогда сравнение элементов контейнера будет проводиться при помощи этого класса)
- либо контейнер будет предполагать, что все хранимые в нем элементы реализуют интерфейс `Comparable` (сравнение элементов делается при помощи метода этого интерфейса)

Важно помнить, что в любом случае реализация алгоритма сравнения должна быть антисимметричной, т.е. должны выполняться соотношения:

$$(a.compareTo(b) > 0) \iff (b.compareTo(a) < 0)$$

и

$$(comparator.compare(a,b) > 0) \iff (comparator.compare(b,a) < 0)$$

В противном случае сортирующие алгоритмы контейнеров не гарантируют правильной работы.

Пример использования компараторов для сортировки массивов

```
public static void doTest8()
{
    Integer [] data =
        new Integer []{1,5,3,7,6,9,2,6,8};

    Arrays.sort(data);

    for (int idx = 0; idx < data.length; idx++)
    {
        System.out.println(""+data[idx]);
    }
}
```

Алгоритм сортировки
полагается на реализацию
элементами массива
интерфейса Comparable

Результат работы программы:

1
2
3
5
6
6
7
8
9

Пример использования компараторов для сортировки массивов

```
static class MyComparator
    implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Integer i1 = (Integer)o1;
        Integer i2 = (Integer)o2;
        if ((i1 % 2) != (i2 % 2))
        {
            return new Integer(i1 % 2)
                .compareTo(new Integer(i2 % 2));
        }
        return i1.compareTo(i2);
    }
}

public static void doTest8()
{
    Integer [] data =
        new Integer []{1,5,3,7,6,9,2,6,8};
    Arrays.sort(data, new MyComparator());

    for (int idx = 0; idx < data.length; idx++)
    {
        System.out.println(""+data[idx]);
    }
}
```

Используется явно заданный компаратор, который считает, что четные числа «меньше» нечетных, а внутри групп четных/нечетных чисел сохраняет естественный порядок.

Результат работы программы:

2
6
6
8
1
3
5
7
9

Пример использования компараторов с контейнерами

```
public static void doTest5()
{
    Integer [] data =
        new Integer []{1,5,3,7,6,9,2,6,8};

    Set<Integer> set =
        new HashSet(Arrays.asList(data));

    for (Integer value : set)
    {
        System.out.println(
            String.valueOf(value));
    }

    /*
    // Эквивалентный код с явным
    // использованием итераторов
    Iterator<Integer> iter = set.iterator();
    while (iter.hasNext())
    {
        Integer value = iter.next();
        System.out.println(
            String.valueOf(value));
    }
    */
}
```

Трансформация массива
элементов в множество.

Множество на базе HashSet не
гарантирует порядок просмотра
элементов итератором.

Результат работы программы:

2
8
9
6
1
3
7
5

Пример использования компараторов с контейнерами

```
public static void doTest5()
{
    Integer [] data =
        new Integer []{1,5,3,7,6,9,2,6,8};

    SortedSet<Integer> set =
        new TreeSet(Arrays.asList(data));

    for (Integer value : set)
    {
        System.out.println(
            String.valueOf(value));
    }
}
```

Трансформация массива
элементов в множество.

Множество на базе TreeSet
гарантирует порядок просмотра
элементов итератором.

Результат работы программы:

1
2
3
5
6
7
8
9

Пример использования компараторов с контейнерами

```
static class MyComparator
    implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Integer i1 = (Integer)o1;
        Integer i2 = (Integer)o2;
        if ((i1 % 2) != (i2 % 2))
        {
            return new Integer(i1 % 2)
                .compareTo(new Integer(i2 % 2));
        }
        return i1.compareTo(i2);
    }
}

public static void doTest7()
{
    Integer [] data =
        new Integer []{1,5,3,7,6,9,2,6,8};
    SortedSet<Integer> set =
        new TreeSet(new MyComparator());
    set.addAll(Arrays.asList(data));
    for (Integer value : set)
    { System.out.println(""+value); }
}
```

Множество на базе TreeSet
гарантирует порядок просмотра
элементов итератором.
Порядок определяется
внешним компаратором.

Результат работы программы:

2
6
8
1
3
5
7
9

Применение хэш-таблиц для хранения разреженных массивов

```
static public class CellID
{
    private int m_iRow;
    private int m_iCol;

    public CellID(int iRow, int iCol)
    { m_iRow = iRow; m_iCol = iCol; }

    public boolean equals(Object obj)
    {
        if (obj instanceof CellID)
        {
            CellID other = (CellID)obj;
            return m_iRow == other.m_iRow
                && m_iCol == other.m_iCol;
        }
        return false;
    }

    public int hashCode()
    { return (m_iRow << 15) ^ m_iCol; }

    public String toString()
    {
        return "(" + m_iRow + ", " + m_iCol + ")";
    }
}
```

Известно, что если в графе мало ребер (разреженный граф), то хранить матрицу смежности неэффективно.

Наблюдение: можно отказаться от прямого использования матрицы и эмулировать ее при помощи хэш-таблицы.

В этом случае хэш-таблица работает как отображение из множества векторов индексов в множество значений элементов матрицы.

Класс CellID будет выполнять роль ключа в хэш-таблице.

Применение хэш-таблиц для хранения разреженных массивов

```
public static class Matrix
{
    private
        HashMap <CellID, Integer> m_map;
    private int m_iDefaultValue;

    public Matrix(int iDefaultValue)
    {
        m_iDefaultValue = iDefaultValue;
        m_map = new HashMap<CellID, Integer>();
    }

    void putValue(
        int iRow, int iCol, int iValue)
    {
        if (iValue == m_iDefaultValue)
        {
            m_map.remove(new CellID(iRow, iCol));
        }
        else
        {
            m_map.put(new CellID(iRow, iCol),
                new Integer(iValue));
        }
    }
}
```

Класс Matrix моделирует поведение обычной матрицы путем трансформации операций доступа к элементам матрицы в операции с хэш-таблицей.

Применение хэш-таблиц для хранения разреженных массивов

```
public static class Matrix
{
    int getValue(int iRow, int iCol)
    {
        Integer value =
            m_map.get(new CellID(iRow, iCol));
        if (value != null)
        {
            return value.intValue();
        }
        else
        {
            return m_iDefaultValue;
        }
    }

    void printState()
    {
        System.out.println("Holding:");
        for (CellID cell : m_map.keySet())
        {
            Integer value = m_map.get(cell);
            System.out.println(
                "+" + cell + " -> " + value);
        }
    }
}
```

Класс Matrix моделирует поведение обычной матрицы путем трансформации операций доступа к элементам матрицы в операции с хэш-таблицей.

Применение хэш-таблиц для хранения разреженных массивов

```
public static void doTest1()
{
    Matrix matrix = new Matrix(0);

    matrix.putValue(1,2, 15);
    matrix.putValue(3,1, 7);

    System.out.println(
        ""+matrix.getValue(0,0));
    System.out.println(
        ""+matrix.getValue(1,2));
    System.out.println(
        ""+matrix.getValue(3,1));

    matrix.printState();
}
```

Тест для класса Matrix

Результат работы программы:

```
0
15
7
Holding:
(3,1) -> 7
(1,2) -> 15
```


Вопрос: какой недостаток у рассмотренной реализации класса Matrix?

Вопрос: какой недостаток у рассмотренной реализации класса Matrix?

Ответ: нигде не запоминается размер матрицы. Формально для реализации он не нужен, но отсутствие этой информации в классе не позволяет проводить проверки корректности использования матрицы (например, допускаются отрицательные индексы).

С использованием стандартных контейнеров можно легко строить хранилища данных произвольной сложности. При этом обычно достаточно использовать predetermined классы из стандартной библиотеки контейнеров.

Если же требуется создать свой собственный контейнер, то в стандартной библиотеке контейнеров присутствуют следующие абстрактные классы с базовой реализацией контейнерных интерфейсов:

AbstractCollection

AbstractSet

AbstractList

AbstractQueue

AbstractSequentialList

AbstractMap

Языки и технологии программирования.

Объектно-ориентированное программирование

Наследование и контейнерные типы
Обобщенные типы

Обобщенное программирование.

Методология программирования, когда программист явно отделяет алгоритм от данных, используемых в алгоритме, называется обобщенным программированием (generic programming).

Разные языки предлагают разные подходы к реализации обобщенного программирования. В частности, Java предлагает обобщенное программирование, основанное на полиморфизме и стирании типов (type erasure), а C# - на комбинации полиморфизма и синтеза кода по шаблону.

Контейнерные типы и наследование.

Разберемся, что происходит с отношением наследования при переходе от элемента к контейнеру.

Пусть классы A и B связаны отношением наследования (B является потомком A). И пусть есть возможность для контейнера указать тип элемента.

Вопрос: будут ли связаны отношением наследования контейнеры для A и для B (например, `ArrayList<A>` и `ArrayList`)?

Контейнерные типы и наследование.

Разберемся, что происходит с отношением наследования при переходе от элемента к контейнеру.

Пусть классы A и B связаны отношением наследования (B является потомком A). И пусть есть возможность для контейнера указать тип элемента.

Вопрос: будут ли связаны отношением наследования контейнеры для A и для B (например, `ArrayList<A>` и `ArrayList`)?

Все зависит от реализации «конструкции» в угловых скобках!

Контейнерные типы и наследование.

Легко понять, что если контейнеры `ArrayList<A>` и `ArrayList` превращаются в независимые классы (в том числе с разным внутренним устройством), то эти два класса окажутся несравнимыми с точки зрения отношения наследования (именно так строит код C#).

В Java для управления типом элементов в контейнере используется так называемый подход со стиранием типов (type erasure). Этот подход подразумевает, что при указании типа элемента контейнера новой реализации класса контейнера не возникает, а информация о типе элемента используется лишь на стадии компиляции для проверки корректности использования «параметризованного» класса. При таком подходе во время исполнения программы информация о типе элемента конкретной коллекции оказывается недоступной (отсюда термин «стирание типов»). Целью указания типа является не синтез конкретной версии кода, а лишь контроль согласованности типов.

Техника стирания типов позволяет в определенных случаях переносить наследование с элементов на контейнеры.

Контейнерные типы и наследование.

Разберемся, что происходит с отношением наследования при переходе от элемента к контейнеру.

Пусть классы A и B связаны отношением наследования (B является потомком A). И пусть есть возможность для контейнера указать тип элемента.

Вопрос: будут ли связаны отношением наследования контейнеры для A и для B (например, `ArrayList<A>` и `ArrayList`)?

Контейнерные типы и наследование.

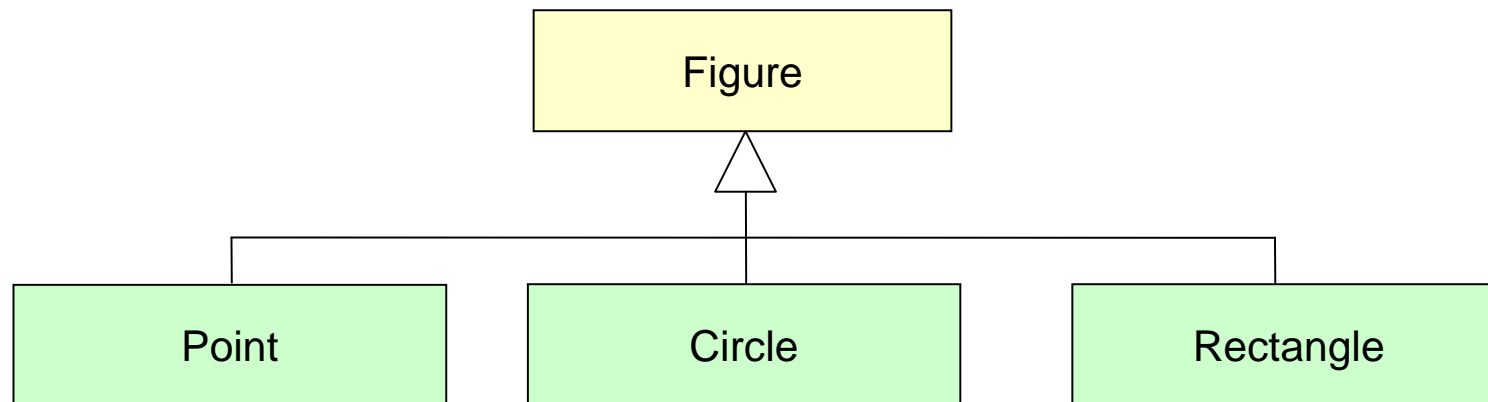
Разберемся, что происходит с отношением наследования при переходе от элемента к контейнеру.

Пусть классы A и B связаны отношением наследования (B является потомком A). И пусть есть возможность для контейнера указать тип элемента.

Вопрос: будут ли связаны отношением наследования контейнеры для A и для B (например, `ArrayList<A>` и `ArrayList`)?

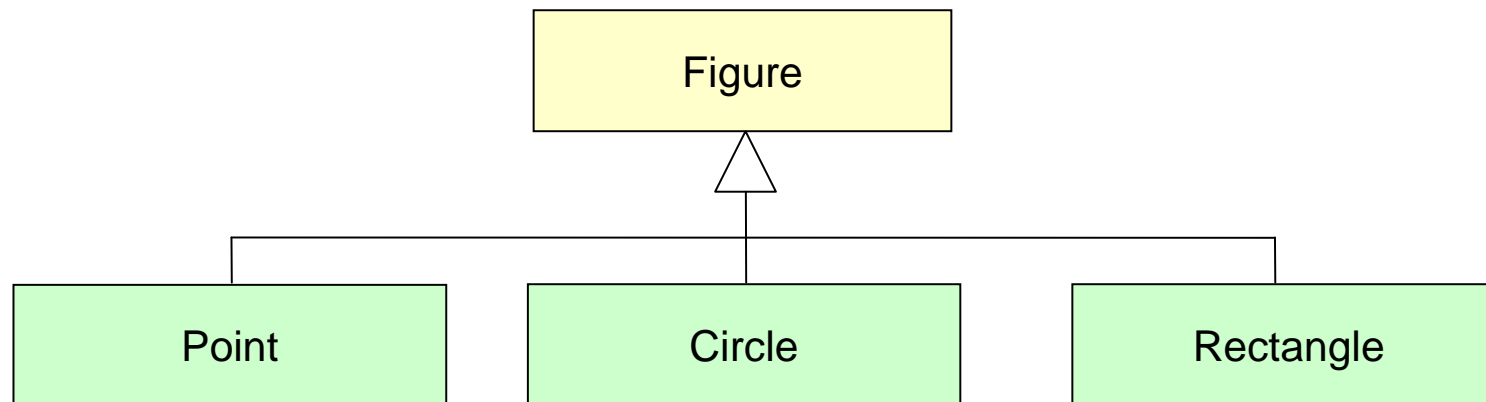
Ответ: если контейнер работает только на чтение, то можно; если контейнер разрешает добавление новых элементов, то нельзя!

Почему так?



```
ArrayList<Circle> circles =  
    new ArrayList<Circle>();  
circles.add(new Circle(...));  
circles.add(new Circle(...));  
  
ArrayList<Figure> figures = circles;
```

В этой строке Java выдаст
ошибку компиляции
Почему?



```
ArrayList<Circle> circles =  
    new ArrayList<Circle>();  
circles.add(new Circle(...));  
circles.add(new Circle(...));  
  
ArrayList<Figure> figures = circles;
```

В этой строке Java выдаст
ошибку компиляции

Почему?

```
figures.add(new Point(...));
```

Причина в том, что в противном
случае посредством
переменной figures в коллекцию
circles (она должна хранить
только круги) можно было бы
положить точку!

Java обеспечивает строгий контроль приведения типов (и на этапе компиляции, и на этапе выполнения программы). Для создания контейнерных типов с возможностью контроля соответствия типов хранимых элементов в Java 5 были введены так называемые обобщенные типы (англ. Generics).

Механизм работы обобщенных типов в Java следующий:

1. в основе лежит полиморфизм (то есть всегда используется одна копия откомпилированного кода).
2. в синтаксис языка введен набор инструкций, позволяющий в процессе компиляции проверить соответствие типа данных элементов контейнера желаемому:

```
ArrayList<Circle> circles =      // Список может содержать только объекты
    new ArrayList<Circle>();      // класса Circle (или его подклассов)

circles.add(new Circle(...));    // Эта строка допустима

circles.add(new Point(...));     // А эта приведет к ошибке компиляции

// Эта строка также приводит к ошибке компиляции,
// поскольку позволяет обойти контроль соответствия типов
ArrayList<Figure> figures = circles;
figures.add(new Point(...));
```

Вопрос: а как теперь организовать код так, чтобы полиморфно использовать метод `move()`, которым обладают все потомки класса `Figure`?
Что на самом деле требуется передать на вход методу `moveAll()`?

```
ArrayList<Circle> circles = new ArrayList<Circle>();

// Эта строчка допустима
circles.add(new Circle(...));

// Эта строка также приводит к ошибке компиляции (поскольку при передаче
// параметра в подпрограмму выполняем присваивание, позволяющее обойти
// контроль соответствия типов)
moveAll(circles);

// Но нам хочется всего лишь написать полиморфный код:
public static void moveAll(ArrayList<Figure> figures)
{
    for (int iIdx = 0; iIdx < figures.size(); iIdx++)
    {
        Figure figure = figures.get(iIdx);
        figure.move(...);
    }
}
```

Вопрос: а как теперь организовать код так, чтобы полиморфно использовать метод `move()`, которым обладают все потомки класса `Figure`?
Метод `moveAll()` ожидает получить на вход любую коллекцию с элементами расширяющими `Figure` (чтобы можно было добраться до метода `move()`):

```
ArrayList<Circle> circles = new ArrayList<Circle>();

// Эта строка допустима
circles.add(new Circle(...));

// Теперь эта строка допустима
moveAll(circles);

// Используем маску для описания типа элемента коллекции:
public static void moveAll(ArrayList<? extends Figure> figures)
{
    for (int iIdx = 0; iIdx < figures.size(); iIdx++)
    {
        // Это присваивание оказывается по-прежнему корректным:
        Figure figure = figures.get(iIdx);
        figure.move(...);
    }
}
```

Наблюдение: попытка использовать вызов `figures.add(...)` теперь будет приводить к ошибке компиляции, поскольку компилятор теперь не знает тип элементов контейнера `figures`! И это вполне корректно, поскольку контейнер, разрешающий добавление элементов, не обеспечивает сохранение отношения наследования.

```
ArrayList<Circle> circles = new ArrayList<Circle>();

// Эта строка допустима
circles.add(new Circle(...));

// Теперь эта строка допустима
moveAll(circles);

// Используем маску для описания типа элемента коллекции:
public static void moveAll(ArrayList<? extends Figure> figures)
{
    for (int iIdx = 0; iIdx < figures.size(); iIdx++)
    {
        // Это присваивание оказывается по-прежнему корректным:
        Figure figure = figures.get(iIdx);
        figure.move(...);
    }
    figures.add(new Figure(...)); // <- это некорректно
}
```


Для использования обобщенного программирования следует действовать так:

- реализуется и отлаживается обычный класс или метод, решающий задачу для параметров конкретных типов;
- в получившемся коде реальные типы параметров заменяются на специальные маркеры типов-параметров.

Пример обобщенного типа Stack на Java

```
class Stack<Type>
{
    // массив с элементами стека
    private Type [] m_data;
    // число элементов в стеке
    private int m_size;

    public Stack(int maxSize)
    {
        m_data = (Type[])new Object[maxSize];
        m_size = 0;
        System.out.println("Stack created");
    }

    // добавляет элемент в стек
    public void push(Type value)
        throws StackException
    {
        if (m_size >= m_data.length)
        {
            throw new StackException(StackException.STACK_OVERFLOW);
        }
        m_data[m_size] = value;
        m_size++;
    }
}
```

Параметр обобщенного типа

Описание массива в терминах обобщенного типа.
Вопрос: какой реально тип элемента у массива m_data?

Пример обобщенного типа Stack на Java

```
class Stack<Type>
{
    // массив с элементами стека
    private Type [] m_data;
    // число элементов в стеке
    private int m_size;

    public Stack(int maxSize)
    {
        m_data = (Type[])new Object[maxSize];
        m_size = 0;
        System.out.println("Stack created");
    }

    // добавляет элемент в стек
    public void push(Type value)
        throws StackException
    {
        if (m_size >= m_data.length)
        {
            throw new StackException(
                StackException.STACK_OVERFLOW);
        }
        m_data[m_size] = value;
        m_size++;
    }
}
```

Параметр обобщенного типа

Описание массива в терминах обобщенного типа.
Вопрос: какой реально тип элемента у массива m_data?

На стадии компиляции обобщенный тип можно ограничить сверху только типом Object. Поэтому и массив надо создавать для элементов типа Object с последующим приведением к типу параметру.

Согласованность типов проверяется автоматически.

Пример обобщенного типа Stack на Java

```
// извлекает элемент из стека
public Type pop()
    throws StackException
{
    if (m_size > 0)
    {
        m_size--;
        return m_data[m_size];
    }
    else
    {
        throw new StackException(
            StackException.STACK_EMPTY);
    }
}

// возвращает размер стека
int getSize()
{
    return m_size;
}
}
```

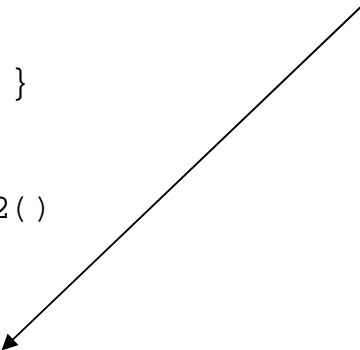
Согласованность типов
проверяется автоматически.

Тест обобщенного типа Stack

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest2(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest2()
        throws StackException
    {
        Stack<String> stack =
            new Stack<String>(10);
        stack.push("One");
        stack.push("Two");
        stack.push("Three");
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```

Создаем стек строк (способен
хранить объекты типа String)



Тест обобщенного типа Stack

```
public class Test
{
    public static void main(String [] args)
    {
        try
        { doTest3(); }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    public static void doTest3()
        throws StackException
    {
        Stack<Number> stack =
            new Stack<Number>(10);
        stack.push(new Integer(15));
        stack.push(new Double(7.5));
        stack.push(new BigDecimal(
            "123456789.123456789123456789"));
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
        System.out.println("Pop: "+stack.pop());
    }
}
```

Создаем стек чисел (способен хранить любые подклассы класса Number)

Результат работы программы:

```
Stack created
Pop:
123456789.123456789123456789
Pop: 7.5
Pop: 15
```

Помимо обобщенных типов в Java можно создавать обобщенные методы. В этом случае появляется возможность согласования типов параметров.

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: какие типы следует согласовать в этой ситуации?

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
void copy(List<Object> source, List<Object> target)
```

Правильно ли это?

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
void copy(List<Object> source, List<Object> target)
```

Плохой вариант, поскольку разрешает подавать на вход лишь контейнеры, у которых тип элемента явно выставлен в Object.

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
void copy(List<?> source, List<?> target)
```

Правильно ли это?

Наблюдение:

маска <?> - это компактная форма записи для <? extends Object>

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
void copy(List<?> source, List<?> target)
```

Плохой вариант, поскольку не позволит вызвать метод `target.add()`.

Чтобы связать типы входа и выхода следует использовать обобщенный метод (то есть ввести маркер типа).

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
static <T> void copy(List<T> source, List<T> target)
{
    for (T value : source)
    {
        target.add(value);
    }
}
```

Этот вариант уже будет компилироваться, но с использованием будут проблемы. Какие?

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
static <T> void copy(  
    List<T> source, List<T> target)  
{  
    for (T value : source)  
    {  
        target.add(value);  
    }  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    copy(arrOfInt, arrOfNum);  
}
```

Сигнатура метода позволяет выполнить копирование из реализации интерфейса List в другую реализацию List при условии, что они обслуживают один и тот же тип элемента! Это делает невозможным копирование данных между контейнерами разных типов.

Можно ли исправить ситуацию?

Задача: реализовать метод, копирующий информацию из одного контейнера в другой.

Вопрос: как правильно описать сигнатуру метода в терминах обобщенных типов?

```
static <T> void copy(  
    List<? extends T> source, ←  
    List<T> target)  
{  
    for (T value : source)  
    {  
        target.add(value);  
    }  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    copy(arrOfInt, arrOfNum);  
}
```

Чтобы создать наиболее универсальную версию алгоритма копирования следует выполнить указанное согласование конкретизаций классов при помощи обобщенного метода.

Маска обеспечивает полиморфное использование параметра для чтения данных. Точное указание типа позволяет добавлять элементы в коллекцию.

Правильная версия кода

```
static <T, S extends T> void copy(  
    List<S> source,  
    List<T> target)  
{  
    for (T value : source)  
    {  
        target.add(value);  
    }  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    copy(arrOfInt, arrOfNum);  
}
```

Наблюдение: предыдущий пример мог бы быть переписан таким образом (два маркера типа с указанием их взаимосвязи).

Параметр типа S обобщенного метода copy используется лишь в конкретизации типа для source. Это верный признак того, что более правильно использовать маску типа, а не выносить тип в параметры обобщенного метода.

Задача: реализовать метод, который объединяет данные из двух коллекций

```
static List combine(  
    List source1,  
    List source2)  
{  
    ArrayList result = new ArrayList(  
        source1.size() + source2.size());  
    result.addAll(source1);  
    result.addAll(source2);  
    return result;  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    List combined =  
        combine(arrOfInt, arrOfNum);  
}
```

Так выглядит версия кода без использования обобщенных типов.

Вопрос: как трансформировать код, чтобы использовать возможности обобщенных типов?

Задача: реализовать метод, который объединяет данные из двух коллекций

```
static <T> List<T> combine(  
    List<? extends T> source1,  
    List<? extends T> source2)  
{  
    ArrayList<T> result = new ArrayList<T>(  
        source1.size() + source2.size());  
    result.addAll(source1);  
    result.addAll(source2);  
    return result;  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    List<Number> combined =  
        combine(arrOfInt, arrOfNum);  
}
```

Для решения задачи используется комбинация обобщенного метода и масок при определении типа.

Параметр метода обеспечивает согласование типов, а маски создают возможность для полиморфизма.

Правильная версия кода

Задача: реализовать метод, который получает на вход произвольную коллекцию, копирует ее поэлементно в другую коллекцию и возвращает последний скопированный элемент.

```
static Object copy(
    List source,
    List target)
{
    Object last = null;
    for (Object value : source)
    {
        last = value;
        target.add(value);
    }
    return last;
}

static void test()
{
    ArrayList<Integer> arrOfInt =
        new ArrayList<Integer>();
    ArrayList<Number> arrOfNum =
        new ArrayList<Number>();

    Integer number =
        (Integer)copy(arrOfInt, arrOfNum);
}
```

Так выглядит версия кода без использования обобщенных типов.

Вопрос: как трансформировать код, чтобы использовать возможности обобщенных типов?

Задача: реализовать метод, который получает на вход произвольную коллекцию, копирует ее поэлементно в другую коллекцию и возвращает последний скопированный элемент.

```
static <T, R extends T, S extends R>
    R copy(List <S> source, List <T> target)
{
    R last = null;
    for (S value : source)
    {
        last = value;
        target.add(value);
    }
    return last;
}

static void test()
{
    ArrayList<Integer> arrOfInt =
        new ArrayList<Integer>();
    ArrayList<Number> arrOfNum =
        new ArrayList<Number>();

    Integer number =
        (Integer)copy(arrOfInt, arrOfNum);
}
```

Параметры обобщенного метода служат для согласования возвращаемого значения с типами параметров, передаваемых в метод.

Здесь для согласования типов отталкиваемся от типа списка-приемника.

Правильная версия кода.

Задача: реализовать метод, который получает на вход произвольную коллекцию, копирует ее поэлементно в другую коллекцию и возвращает последний скопированный элемент.

```
static <T> T copy(  
    List <? extends T> source,  
    List <? super T> target)  
{  
    T last = null;  
    for (T value : source)  
    {  
        last = value;  
        target.add(value);  
    }  
    return last;  
}  
  
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
    ArrayList<Number> arrOfNum =  
        new ArrayList<Number>();  
  
    Integer number =  
        (Integer)copy(arrOfInt, arrOfNum);  
}
```

Здесь для согласования типов отталкиваемся от типа результата.

Важно: ключевое слово `super` можно использовать в маске типа, но нельзя в описании именованных типов!

Тоже правильная версия кода.

Задача: реализовать метод, который находит максимальный элемент в данной коллекции.

```
static Object max(
    Collection data)
{
    Object max = null;
    Iterator iter = data.iterator();
    while (iter.hasNext())
    {
        Comparable value =
            (Comparable)iter.next();
        if (value == null)
        { continue; }
        if (value.compareTo(max) > 0)
        {
            max = value;
        }
    }
    return max;
}

static void test()
{
    ArrayList<Integer> arrOfInt =
        new ArrayList<Integer>();
    Integer value = (Integer)max(arrOfInt);
}
```

Так выглядит версия кода без использования обобщенных типов.

Вопрос: как трансформировать код, чтобы использовать возможности обобщенных типов?

Наблюдение: за счет обобщенных типов можно попытаться наложить условие, что все элементы коллекции должны быть взаимно сравнимыми!

Задача: реализовать метод, который находит максимальный элемент в данной коллекции.

```
static <T extends Comparable<T>> T max(
    Collection<T> data)
{
    T max = null;
    Iterator<T> iter = data.iterator();
    while (iter.hasNext())
    {
        T value = iter.next();
        if (value == null)
        { continue; }
        if (value.compareTo(max) > 0)
        {
            max = value;
        }
    }
    return max;
}
```

```
static void test()
{
    ArrayList<Integer> arrOfInt =
        new ArrayList<Integer>();
    Integer value = max(arrOfInt);
}
```

Первая попытка исправить код.
Все правильно, но наложены
слишком жесткие ограничения.
Например, она не сработает
для класса

```
class Complex
    extends Number
    implements Comparable<Number>
{
    ...
}
```

Задача: реализовать метод, который находит максимальный элемент в данной коллекции.

```
static
    <T extends Comparable<? super T>>
    T max(Collection<T> data)
{
    T max = null;
    Iterator<T> iter = data.iterator();
    while (iter.hasNext())
    {
        T value = iter.next();
        if (value == null)
        { continue; }
        if (value.compareTo(max) > 0)
        {
            max = value;
        }
    }
    return max;
}

static void test()
{
    ArrayList<Integer> arrOfInt =
        new ArrayList<Integer>();
    Integer value = max(arrOfInt);
}
```

Наблюдение: если хотим обеспечить максимальную гибкость в коде, извлекающем информацию из объектов, то используем маску, ограничивающую сверху (extends), а если хотим обеспечить максимальную гибкость в коде, передающем информацию в объект, то используем маску, ограничивающую снизу (super).

Правильная версия кода

Согласование обобщенного и обычного кода

```
static void test()  
{  
    ArrayList<Integer> arrOfInt =  
        new ArrayList<Integer>();  
  
    List list = arrOfInt; // Корректно  
  
    List<Integer> listInt1 =  
        list; // Ошибка компиляции!  
  
    List<Integer> listInt2 =  
        (List<Integer>)list; // Предупреждение  
}
```

Информация о параметрах обобщенного типа после компиляции «забывается» (т.н. механизм стирания типов - type erasure), то есть `ArrayList<Integer>` и `ArrayList<String>` - это один и тот же класс `ArrayList`. Поэтому любая конкретизация совместима по присваиванию с неконкретизированным классом. Обратное присваивание формально (с точки зрения Java-машины) корректно, но без явного приведения типов вызовет ошибку компиляции, а с приведением типов – предупреждение о возможном разрушении данных.

Выводы:

1. Проверки соответствия типов для обобщенных классов проводятся лишь на этапе компиляции программы (это упрощает разработку программы). В процессе компиляции, информация о параметрах обобщенных классов стирается. Но встроенный в Java-машину механизм контроля типов при присваивании продолжает работу, обеспечивая целостность данных.
2. Маска типа вида "? extends TypeA" обеспечивает создание полиморфного кода.
3. Маска типа вида "? super TypeB" обеспечивает совместимость по присваиванию для типов параметров обобщенного типа, возможность модификации объекта обобщенного типа.
4. Обобщенные методы позволяют согласовать типы, использованные в разных параметрах метода.

Ковариантность и контравариантность в обобщенных типах:

В зависимости от того, сохраняет ли обобщенный тип отношение наследования, все обобщенные типы делятся на следующие группы:

1. **Ковариантные** – сохраняют отношение наследования. То есть описание типа разрешает присваивание более точного (производного) типа.

Например: `ArrayList<? extends Number> list = new ArrayList<Number>();`

Ковариантные параметры обычно используются «на чтение».

2. **Контравариантные** – инвертируют отношение наследования. То есть описание типа разрешает присваивание более общего (родительского, базового) типа.

Например: `ArrayList<? super Integer> list = new ArrayList<Number>();`

Контравариантные параметры обычно используются «на запись».

3. **Инвариантные** – не сохраняют отношение наследования.

Например: `ArrayList<String>`

Ковариантность и контравариантность в обобщенных типах:

1. Массивы в Java (и в C#) ковариантны.

```
Object [] test = new String[5]; // верное присваивание
```

Ковариантность массивов является причиной того, что ключевое слово `super` запрещено в описании именованных параметров типов:

```
// если бы такое описание было допустимо:
```

```
void <S, T super S> copy(S[] source, T[] target) {...}
```

```
// этот вызов был бы корректным:
```

```
copy(new Integer[] {1,2,3}, new String [3]);
```

Дело в том, что в силу ковариантности массивов массив `String[]` приводится к `Object[]`, который, оказался бы совместим с generic-описанием метода, но давал бы некорректный с точки зрения контроля типов код.

Ковариантность и контравариантность в обобщенных типах:

2. Полиморфизм (правила совместимости по присваиванию для объектных типов) обеспечивает ковариантность параметров методов, имеющих **объектный тип**:

```
void printNumber(Number n) {...}  
printNumber(new Integer(5)); // верный вызов
```

3. В Java есть ковариантность результата метода при наследовании:

```
class Parent {  
    Object doSomething() {...};  
}  
class Child{  
    String doSomething() {...}; // <- здесь переопределение,  
                                // а не перегрузка  
}
```

Ковариантность и контравариантность в обобщенных типах:

4. При наследовании входные параметры методов ведут себя как инвариантные:

```
class Parent {  
    String doSomething(String s) {...};  
}  
  
class Child{  
    String doSomething(Object o) {...}; // <- здесь перегрузка, а не  
                                         // переопределение  
}
```

Наблюдение: при наследовании входные типы могли бы вести себя как контравариантные.

Ковариантность и контравариантность в обобщенных типах:

5. Контравариантность возникает при описании типов, ведущих себя как функции. В следующем примере класс `NumberPrinter`, принимающий на вход тип `Number` используется (контравариантно) для печати типа `Integer`.

```
interface Printer <T> {  
    public String print(T value);  
}
```

```
class NumberPrinter implements Printer<Number> {  
    public String print(Number v) { return v.toString(); }  
}
```

```
Printer <? super Integer> p = new NumberPrinter();  
String result = p.print(new Integer(10));
```

Обобщенные типы в порождающих конструкциях

Пусть есть несколько реализаций некоторого интерфейса `IActionPerformer` и каждой реализации соответствует свой фабричный класс, порождающий соответствующую реализацию с использованием соответствующих параметров. Тогда эту взаимосвязь можно описать так:

```
public class ActionPerformerParams
{
    // базовый класс для параметров
}

public interface IActionPerformer
{
    void doAction();
}

public interface IFactory<TResult extends IActionPerformer,
                        TParams extends ActionPerformerParams>
{
    TResult createInstance(TParams params);
}
```

И реализация этих интерфейсов может быть такой:

```
public class TestClassParams extends ActionPerformerParams {
    public int someValue;
}

public class TestClassFactory
    implements IFactory<TestClass, TestClassParams>
{
    public static TestClassFactory instance = new TestClassFactory();
    public TestClass createInstance(TestClassParams params) {
        return new TestClass(params);
    }
}

public class TestClass implements IActionPerformer {
    public TestClass(TestClassParams params) {
        // some code
    }

    public void doAction() {
        // perform action
    }
}
```

Отметим, что исходное описание интерфейса IFactory заставляет, например, сделать так, чтобы TestClass реализовывал IActionPerformer.

Обобщенные типы в порождающих конструкциях

И теперь хотим описать метод, который позволял бы порождать объект с использованием указанной фабрики и с использованием корректных параметров:

```
public static class Test
{
    public <TParams extends ActionPerformerParams,
        TResult extends IActionPerformer,
        TFactory extends IFactory<TResult, TParams>>
        TResult CreateInstance(TFactory factory, TParams params) {
        return factory.createInstance(params);
    }

    public void TestCode() {
        TestClass c =
            CreateInstance(TestClassFactory.instance, new TestClassParams());
        c.doAction();
    }
}
```

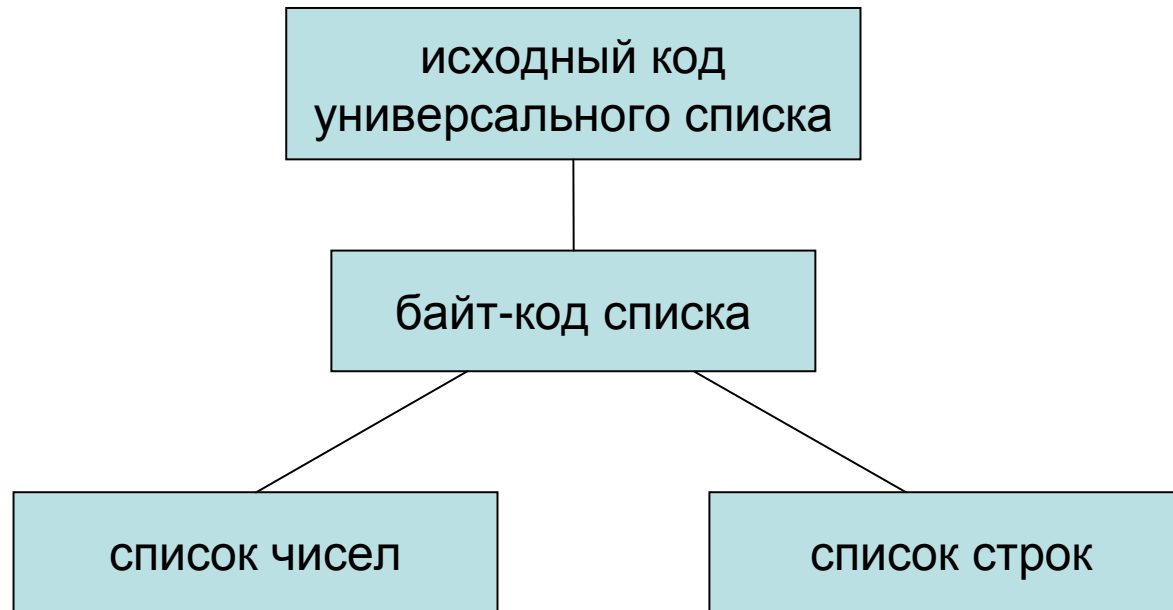
Обобщенные типы позволяют согласовать типы фабрики и параметров!

Языки и технологии программирования.

Объектно-ориентированное программирование

*Обобщенное программирование
в языке C#*

В языке Java в основу обобщенного программирования положена техника стирания типов (type erasure). Эта техника подразумевает создание по исходному коду единственного класса, полиморфно обеспечивающего работу в сценариях с ограничениями. Сами ограничения проверяются компилятором.



В частности, при указании типа элементов контейнера лишь включаются дополнительные проверки уровня компиляции. На уровне откомпилированного кода информация о типе элементов контейнера стирается, «забывается»:

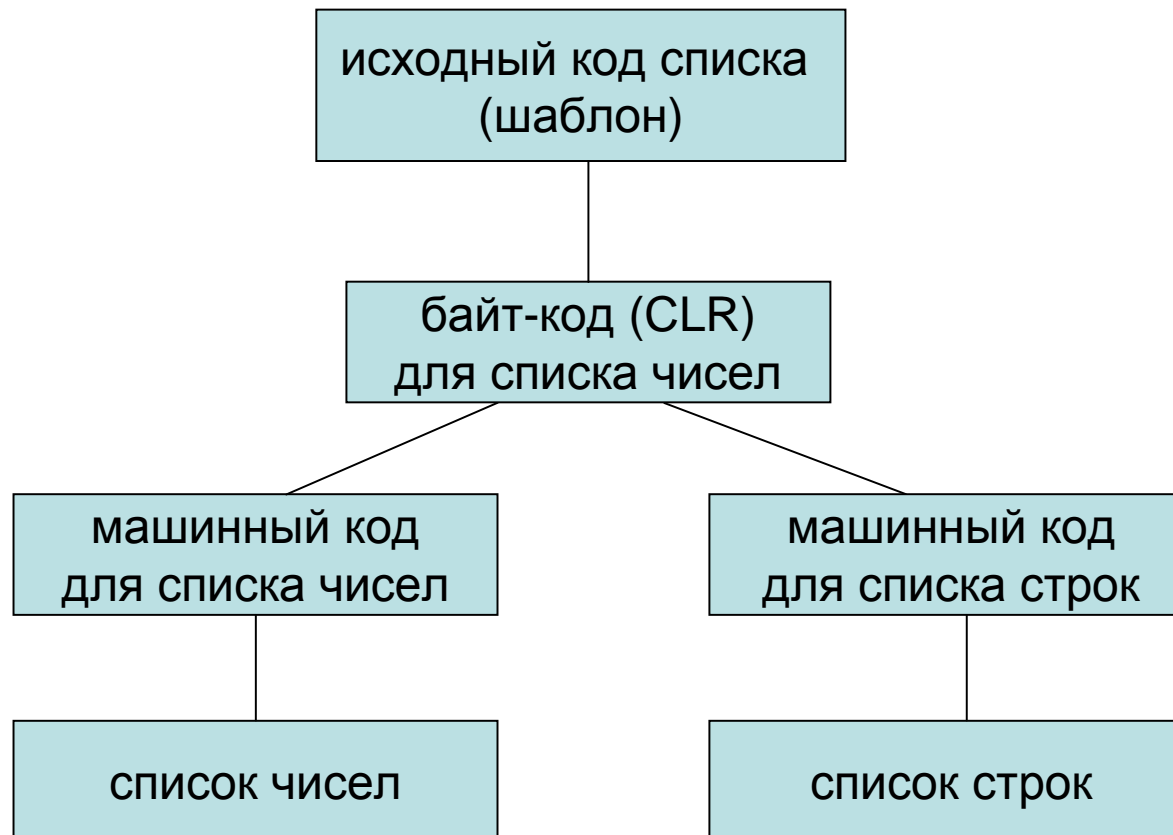
```
ArrayList<Integer> arrOfInt =  
    new ArrayList<Integer>();  
  
ArrayList<String> arrOfString =  
    new ArrayList<String>();  
  
// Получаем имена классов контейнеров  
System.out.println(  
    arrOfInt.getClass().getName());  
System.out.println(  
    arrOfString.getClass().getName());  
  
// сравниваем классы контейнеров  
System.out.println(" "+  
    arrOfInt.getClass().equals(  
        arrOfString.getClass()));
```

Реально используется одна и та же реализация класса `ArrayList`, позволяющая хранить объекты типа `Object`.

Результат работы программы:

```
java.util.ArrayList  
java.util.ArrayList  
true
```

В языке C# в основу техники обобщенного программирования положена возможность шаблонизации кода, когда программист формирует заготовку кода (шаблон), из которого компилятор и среда исполнения автоматически сгенерируют реальный код (конкретизацию шаблона).



Техника синтеза кода по шаблону позволяет получить код, более эффективный для примитивных типов за счет того, что можно отказаться от операций оборачивания (boxing). Это возможно за счет того, что в момент синтеза кода известен точный тип данных для конкретизации шаблона и, как следствие, примитивные типы можно хранить по значению без использования ссылок.

Наблюдение: конкретизация шаблонов ведет к размножению реализаций (для каждого варианта параметризации шаблона имеем свой код).

Наблюдение: для всех конкретизаций шаблона объектными параметрами можно использовать один и тот же код, поскольку реально в реализации будут храниться ссылки (они имеют один и тот же формат хранения вне зависимости от целевого типа).

Таким образом, размножение кода возникает лишь при параметризациях примитивными типами.

Наблюдение: техника стирания типов именно за счет отказа от примитивных типов позволяет обходиться одной полиморфной копией кода. За счет этого поддержку техники стирания типов на уровне виртуальной машины (среды исполнения) обеспечить ощутимо проще, чем поддержку шаблонизации.

Отличие техник стирания типов и шаблонизации ведут к первому существенному отличию: отличию в поведении статических полей классов.

Статические поля в обобщенных типах в Java

```
public class StaticTest <T>
{
    static String value;

}

void doTest()
{
    StaticTest<String> a =
        new StaticTest<String>();
    a.value = "value A";

    StaticTest<Number> b =
        new StaticTest<Number>();
    b.value = "value B";

    System.out.println(a.value);
    System.out.println(b.value);
}
```

Обобщенный тип

Поскольку стирание типов оставляет единственную копию класса StaticTest, поле value будет общим для всех «специализаций» обобщенного класса.

В обоих случаях будет напечатано:

value B

поскольку в памяти это одно и то же поле.

Статические поля в обобщенных типах в Java

```
public class StaticTest <T>
{
    static String value;
    static StaticTest <T> instance;
}

void doTest()
{
    StaticTest<String> a =
        new StaticTest<String>();
    a.value = "value A";

    StaticTest<Number> b =
        new StaticTest<Number>();
    b.value = "value B";

    System.out.println(a.value);
    System.out.println(b.value);
}
```

Обобщенный тип

Поскольку стирание типов оставляет единственную копию класса StaticTest, поле value будет общим для всех «специализаций» обобщенного класса.

Так написать нельзя.

Почему?

Статические поля в обобщенных типах в Java

```
public class StaticTest <T>
{
    static String value;

    static StaticTest <T> instance;
}

void doTest()
{
    StaticTest<String> a =
        new StaticTest<String>();
    a.value = "value A";

    StaticTest<Number> b =
        new StaticTest<Number>();
    b.value = "value B";

    System.out.println(a.value);
    System.out.println(b.value);
}
```

Обобщенный тип

Поскольку стирание типов оставляет единственную копию класса StaticTest, поле value будет общим для всех «специализаций» обобщенного класса.

Так написать нельзя. Это поле не привязано к какой-либо конкретной «специализации» обобщенного класса. То есть у него невозможно зафиксировать тип-параметр.

Статические поля в обобщенных типах в C#

```
public class StaticTest <T>
{
    static string value;
    static StaticTest <T> instance;
}

void doTest()
{
    StaticTest<String> a =
        new StaticTest<String>();
    a.value = "value A";

    StaticTest<Number> b =
        new StaticTest<Number>();
    b.value = "value B";

    System.out.println(a.value);
    System.out.println(b.value);
}
```

Обобщенный тип

Поскольку применяется шаблонизация, для каждой конкретизации будет создаваться свой класс, каждый такой класс будет иметь свою копию поля value.

Так написать можно. Тип T при конкретизации известен.

Теперь поля разные, и напечатано будет:
value A
value B

Вариантность типов в C#

Обобщенные классы в C# всегда инвариантны. То есть переменную типа `Type<T>` можно инициализировать только объектом класса `Type<T>` с тем же самым `T`.

Делегаты по умолчанию предполагают контравариантность по входным аргументам инвариантность по выходным аргументам и ковариантность по результату.

Обобщенные интерфейсы и делегаты могут быть описаны так, чтобы допускать ковариантность или контравариантность параметров типов.

Вариантность типов в C#

```
public class Test
{
    private delegate object
        SampleAction(string actionArg);

    private static
        string Action1(string actionArg)
    {
        return actionArg;
    }

    private static
        object Action2(object actionArg)
    {
        return actionArg;
    }

    private SampleAction d1 = Action1;
    private SampleAction d2 = Action2;
}
```

Описание делегата

Присваивание корректно
(ковариантность по результату)

Присваивание корректно
(контравариантность по
входным аргументам)

Вариантность типов в C#

```
public interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

Описание обобщенного интерфейса с ковариантным параметром типа

```
public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

Описание обобщенного интерфейса с контравариантным параметром типа

```
class Sample : IComparable<object>
{
    public int CompareTo(object other)
    {
        return ...; // как-то сравниваем
    }
}
```

Присваивание корректно за счет ковариантности параметра типа

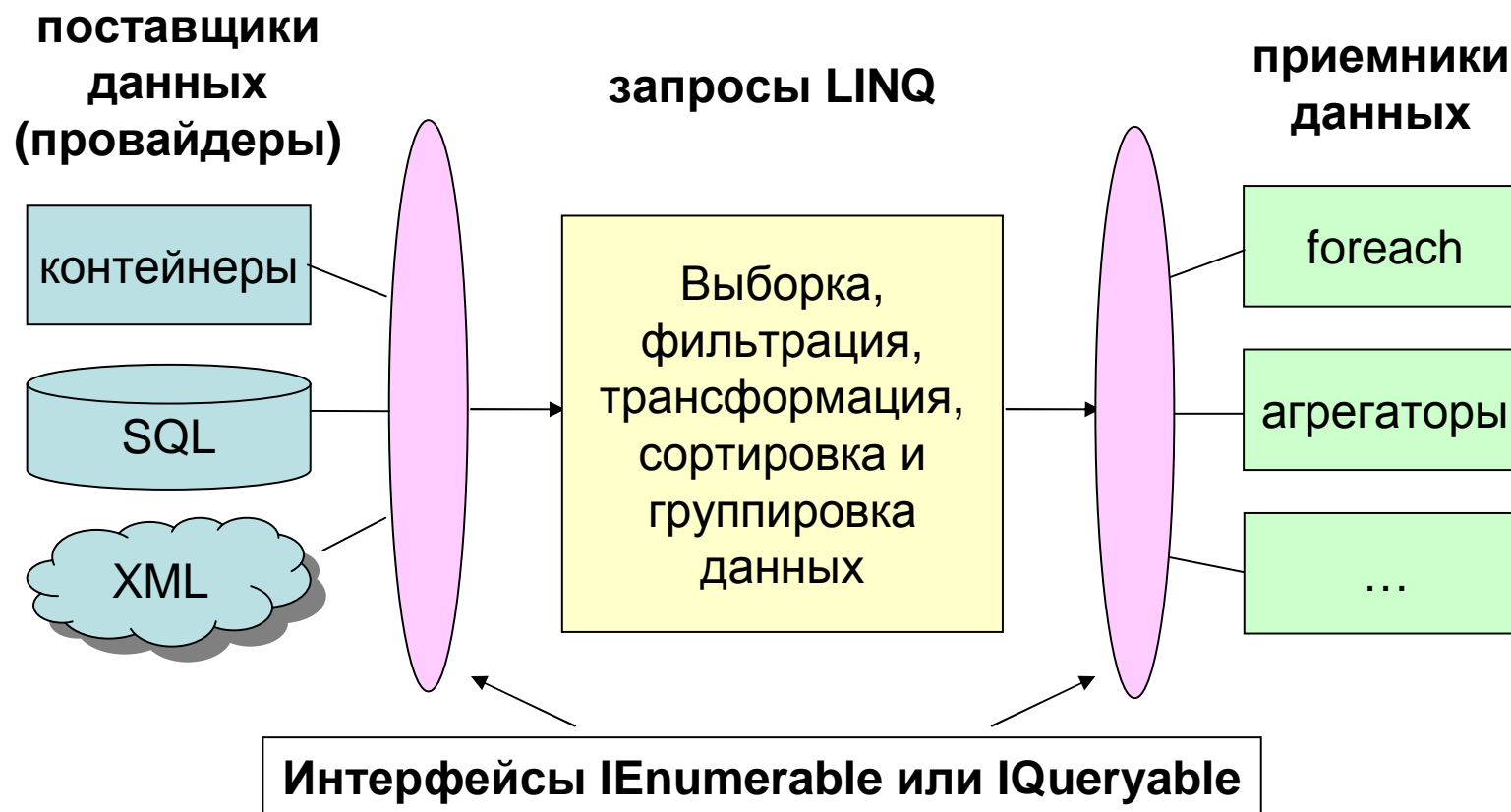
```
public class Test
{
    private IEnumerable<object> e =
        new List<string>();

    private IComparable<string> c =
        new Sample();
}
```

Присваивание корректно за счет контравариантности параметра типа (см. описание класса Sample)

Интегрированные в язык запросы (LINQ) в C#

Язык C# предоставляет специальный набор инструкций, позволяющий описывать запросы к различным источникам данных:



Простейший LINQ запрос

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 }; ←
```

Источник данных (неявно реализует IEnumerable<int>)

```
// LINQ запрос определяет правило  
// извлечения и обработки данных  
var query =
```

```
    from num in data ←
```

Выбор источника данных и фиксация имени диапазонной переменной (num) для доступа к извлекаемым данным

```
    where num % 2 == 0 ←
```

Фильтр извлекаемых данных

```
    select num * num; ←
```

Правило формирования выходных данных

```
foreach (var value in query) ←  
{  
    Console.WriteLine("{0}", value);  
}
```

Потребитель данных (выполнение запроса происходит только в момент потребления данных)

```
// В результате печатаются  
// квадраты четных чисел  
// 16  
// 4  
// 36
```


Простейший LINQ запрос

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 };
```

LINQ обеспечивает строгий контроль типов.

```
// LINQ запрос определяет правило  
// извлечения и обработки данных
```

```
var query =  
  
    from num in data  
  
    where num % 2 == 0  
  
    select num * num;
```

Какой тип у query?

```
foreach (var value in query)  
{  
    Console.WriteLine("{0}", value);  
}
```

Какой тип у value?

```
// В результате печатаются  
// квадраты четных чисел  
// 16  
// 4  
// 36
```

Простейший LINQ запрос

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 };
```

```
// LINQ запрос определяет правило  
// извлечения и обработки данных
```

```
IEnumerable<int> query = ←
```

```
    from num in data
```

```
    where num % 2 == 0
```

```
    select num * num;
```

```
foreach (int value in query) ←  
{  
    Console.WriteLine("{0}", value);  
}
```

```
// В результате печатаются  
// квадраты четных чисел  
// 16  
// 4  
// 36
```

У переменной `query` тип `IEnumerable <int>`

Важно: не `IEnumerator<int>`, а `IEnumerable<int>`, то есть на момент описания запроса данные еще не извлечены!

То есть имеем отложенное (deferred) выполнение кода (код выполняется не в точке описания, а позже).

В типе `IEnumerable<int>` переменной `query` параметр `int` определяется типом результата в фразе `select`.

Следовательно, и у переменной `value` тип `int`.

Простейший LINQ запрос

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 };  
  
// LINQ запрос определяет правило  
// извлечения и обработки данных  
var query =  
    from num in data  
    where num % 2 == 0  
    select new {Value = num,  
                Square = num * num};  
  
foreach (var value in query)  
{  
    Console.WriteLine("{0} -> {1}",  
        value.Value, value.Square);  
}  
  
// В результате печатается:  
// 4 -> 16  
// 2 -> 4  
// 6 -> 36
```

В LINQ запросе тип результата (и промежуточных данных) может быть неизвестен (LINQ допускает использование анонимных классов). Поэтому обычно при описании LINQ запросов пользуются способностью компилятора выводить актуальные типы из контекста.

Создание анонимного класса с данными

Как работает LINQ

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 };
```

```
// LINQ запрос определяет правило  
// извлечения и обработки данных
```

```
var query =  
    from num in data  
    where num % 2 == 0  
    select new {Value = num,  
                Square = num * num};
```

```
foreach (var value in query)  
{  
    Console.Out.WriteLine("{0} -> {1}",  
        value.Value, value.Square);  
}
```

```
var data =  
    new int[] { 1, 3, 4, 2, 6, 5, 3 };
```

```
// LINQ запрос определяет правило  
// извлечения и обработки данных
```

```
var query =  
    data  
    .Where(num => num % 2 == 0)  
    .Select(num => new {Value = num,  
                        Square = num*num});
```

```
foreach (var value in query)  
{  
    Console.Out.WriteLine("{0} -> {1}",  
        value.Value, value.Square);  
}
```

В процессе компиляции инструкции LINQ превращаются в цепочку вызовов готовых методов, предоставляемых библиотекой LINQ.

Как работает LINQ

```
var data =  
    new int[] { 1, -3, 4, -2, 6, -5 };  
  
var query =  
    from num in data  
    where num % 2 == 0  
    group num by num >= 0 into numGroup  
    orderby numGroup.Key  
    select numGroup;  
  
foreach (var group in query)  
{  
    Console.WriteLine(  
        "Group key: {0}", group.Key);  
    foreach(var value in group)  
        Console.WriteLine(  
            "value: {0}", value);  
}
```

```
var data =  
    new int[] { 1, -3, 4, -2, 6, -5 };  
  
var query =  
    data  
    .Where(num => num%2 == 0)  
    .GroupBy(num => num >= 0)  
    .OrderBy(numGroup => numGroup.Key);  
  
foreach (var group in query)  
{  
    Console.WriteLine(  
        "Group key: {0}", group.Key);  
    foreach(var value in group)  
        Console.WriteLine(  
            "value: {0}", value);  
}
```

Все инструкции LINQ (from, where, select, group, orderby, join) могут быть записаны и в виде методов. Но не для каждого метода есть соответствующая инструкция LINQ (например, Union()).

Как работает LINQ

1. Для использования LINQ нужен источник данных. То есть какой-то класс, реализующий интерфейс `IEnumerable<type>` или `IQueryable<type>`. Такой класс называется источником (провайдером, от англ. provider) данных.

Наблюдение: класс, трансформирующий один интерфейс в другой, называется **адаптером**. То есть источником данных для LINQ всегда является какой-то класс-адаптер, возвращающий исходные (сырые) данные в рамках контракта `IEnumerable<type>` или `IQueryable<type>`

Как работает LINQ

1. Для использования LINQ нужен источник данных. То есть какой-то класс, реализующий интерфейс `IEnumerable<type>` или `IQueryable<type>`. Такой класс называется источником (провайдером, от англ. provider) данных.

2. К данным, полученным от адаптера можно применить один или несколько методов-расширений, принимающих на вход `IEnumerable<typeA>` или `IQueryable<typeA>`, и возвращающих на выходе `IEnumerable<typeB>` или `IQueryable<typeB>`

Наблюдение 1: если `typeB` совпадает с `typeA`, то имеем метод-**фильтр**, а если не совпадает, то – метод-**адаптер**. Например, методы `Where` и `OrderBy` – это фильтры, а методы `Select` или `GroupBy` – адаптеры.

Наблюдение 2: все методы LINQ реализованы как методы-расширения обобщенных интерфейсов `IEnumerable<type>` или `IQueryable<type>`. Это позволяет применять их к любым реализациям этих интерфейсов.

Как работает LINQ

1. Для использования LINQ нужен источник данных. То есть какой-то класс, реализующий интерфейс `IEnumerable<type>` или `IQueryable<type>`. Такой класс называется источником (провайдером, от англ. provider) данных.
2. К данным, полученным от адаптера можно применить один или несколько методов-расширений, принимающих на вход `IEnumerable<typeA>` или `IQueryable<typeA>`, и возвращающих на выходе `IEnumerable<typeB>` или `IQueryable<typeB>`
3. Получившийся запрос описывает цепочку извлечения и обработки данных. За счет того, что правила обработки могут быть внедрены в запрос как дерево выражений (т.е. без превращения в код CLR), они доступны провайдеру данных и, следовательно, провайдер может исполнить их непосредственно на уровне источника данных (так, например, инструкции LINQ `where`, `orderby`, `group by`, если применяются к источнику на основе SQL, могут быть исполнены непосредственно на сервере БД, а не на клиенте).

Как работает LINQ

1. Для использования LINQ нужен источник данных. То есть какой-то класс, реализующий интерфейс `IEnumerable<type>` или `IQueryable<type>`. Такой класс называется источником (провайдером, от англ. provider) данных.
2. К данным, полученным от адаптера можно применить один или несколько методов-расширений, принимающих на вход `IEnumerable<typeA>` или `IQueryable<typeA>`, и возвращающих на выходе `IEnumerable<typeB>` или `IQueryable<typeB>`
3. Получившийся запрос описывает цепочку извлечения и обработки данных.
4. Для активации запроса нужно вызвать его метод `GetEnumerator()`. Это можно сделать неявно (цикл `foreach`, агрегирующие операции `First()`, `Sum()` и т.п. или копирующие операции `ToList()`, `ToArray()`) или явно (в этом случае следует использовать оператор `using` или конструкцию `try/finally`, поскольку перечисления являются закрываемыми объектами).

Методы-генераторы

Язык C# предоставляет возможность простой реализации итерируемого объекта при помощи инструкций `yield return` (возвращает очередное значение) и `yield break` (завершает итерирование).

Операции `yield return` и `yield break` можно использовать только внутри методов, возвращающих `IEnumerable<type>`. При этом аргумент `yield return` обязан иметь тип `type`.

```
// метод реализует итератор по всем делителям переданного на вход числа
public IEnumerable<int> Divisors(int value)
{
    for (int i = 2; i <= value/2; i++)
    {
        if (value%i == 0)
            yield return i;
    }
}

public void TestMethod1()
{
    foreach (var divisor in Divisors(18394))
    {
        Console.WriteLine("{0}", divisor);
    }
}
```

LINQ и несколько источников

Данные можно запрашивать из нескольких источников.

Если источники независимы, то используется операция `join`, позволяющая указать очередной источник и условие связи.

```
var s1 = "abracadabra";
var s2 = "12345";

// Запрос строит все пары символов, первый из которых взят из строки s1,
// а второй – из s2
// условие связи специально указано так, чтобы быть всегда истинным
var pairs = from c1 in s1
            join c2 in s2 on 1 equals 1
            select " " + c1 + c2;

// Эквивалентная запись через методы:
// var pairs = s1.Join(s2, c1 => 1, c2 => 1, (c1, c2) => " " + c1 + c2);

foreach (var p in pairs)
{
    Console.WriteLine(p);
}
```

LINQ и несколько источников

Если же источники зависимы, то используется операция `from`, позволяющая построить очередной источник по уже имеющимся данным:

```
public IEnumerable<int> Divisors(int value) {
    for (int i = 2; i <= value/2; i++) {
        if (value%i == 0)
            yield return i;
    }
}

public void TestMethod() {
    var data = new int[] { 13645, 4235, 19364 };

    var query3 =
        from num in data
        from div in Divisors(num)
        select div;

    // Эквивалентная запись через методы
    // var query3 = data.SelectMany(num => Divisors(num));
}
```

Наблюдение: операция `Select` заменяет один элемент исходного перечисления на один элемент новых данных, а операция `SelectMany` заменяет один элемент исходного перечисления на последовательность новых элементов.

Языки и технологии программирования.

Объектно-ориентированное программирование

*Стандартные библиотеки
ввода-вывода*

Программа, работающая без взаимодействия с внешним миром, навряд ли представляет интерес, поскольку не может получать и отдавать информацию. Поэтому любой язык программирования и любая платформа предоставляют средства для взаимодействия программ с внешним миром.

Основные способы взаимодействия программ с внешним миром перечислены ниже:

1. файловый ввод-вывод (обмен данными производится с объектами операционной системы);
2. сетевой ввод-вывод (обмен данными производится между двумя или несколькими машинами посредством сетевых протоколов);
3. обмен данными с СУБД (обмен данными производится со специализированным хранилищем данных – СУБД посредством специальных программных интерфейсов)

Во всех названных случаях возникает переход информации из среды исполнения программы во внешний мир. Следовательно, в этот момент обязаны происходить трансформация данных и согласование форматов данных.

Файлы и файловые системы.

Минимальная единица информации, участвующей в операции ввода-вывода – байт. Соответственно, с точки зрения операций ввода-вывода нас будут интересовать наборы байтов.

При размещении в памяти такие наборы байтов естественным образом соответствуют массивам и идентифицируются адресом в памяти. Если же такой набор байтов находится вне программы (либо размещается на физическом или логическом устройстве хранения информации, либо моделируется средствами операционной системы или среды исполнения программы), то обычно для идентификации такого набора используется имя, сам набор байтов называется файлом.

Совокупность файлов, для которой определены правила именования и идентификации отдельных файлов, называется файловой системой. Часто в понятие файловой системы вкладывают также соглашения о формате хранения файлов на физическом носителе.

Файловые системы.

Файловые системы бывают очень разными. Рассмотрим две наиболее часто используемые их разновидности: файловую систему операционных систем Windows и файловую систему Unix.

Фрагмент дерева файловой системы Windows

```
C:\
|- WINDOWS\
|   |- System32\
|       |- wscript.exe
|
|- Documents and Settings\
|   |- All Users\
|   |- Default User\
|   |- student\
|       |- NTUSER.DAT
|- Program Files\
|   |- ICQ\
|       |- Icq.exe
|
D:\
|- ...
...
```

Фрагмент дерева файловой системы Unix

```
/
|- boot/
|- etc/
|   |- mail/
|       |- sendmail.cf
|
|- home/
|   |- ftp/
|   |   |- pub/
|   |   |- student/
|   |       |- .bashrc
|- media/
|   |- cdrecorder/
|- usr/
|   |- java/
|       |- j2sdk1.4.2_05/
|           |- src.zip
```


Важно: различия файловых систем могут быть весьма ощутимыми:

В Unix всегда один корень файловой системы (дополнительные файловые системы «встраиваются» в корневую файловую систему как подкаталоги.

В Windows каждое устройство хранения информации имеет собственный корень файловой системы (их может быть до 26)

Windows

```
c:\
|- WINDOWS\
|   |- System32\
|       |- wscript.exe
|
|- Documents and Settings\
|   |- All Users\
|   |- Default User\
|   |- student\
|       |- NTUSER.DAT
|- Program Files\
    |- ICQ\
        |- Icq.exe
```

```
D:\
|- ...
...
```

Unix

```
/
|- boot/
|- etc/
|   |- mail/
|       |- sendmail.cf
|
|- home/
|   |- ftp/
|   |   |- pub/
|   |   |- student/
|   |       |- .bashrc
|- media/
|   |- cdrecorder/
|- usr/
|   |- java/
|       |- j2sdk1.4.2_05/
|           |- src.zip
```

Важно: различия файловых систем могут быть весьма ощутимыми:
Для отделения компонент имени файла используются разные символы:
слэш в Unix и обратный слэш в Windows.

Windows

```
C:\
|- WINDOWS\
|   |- System32\
|       |- wscript.exe
|
|- Documents and Settings\
|   |- All Users\
|   |- Default User\
|   |- student\
|       |- NTUSER.DAT
|- Program Files\
|   |- ICQ\
|       |- Icq.exe
|
D:\
|- ...
...
```

Unix

```
/
|- boot/
|- etc/
|   |- mail/
|       |- sendmail.cf
|
|- home/
|   |- ftp/
|   |   |- pub/
|   |   |- student/
|   |       |- .bashrc
|- media/
|   |- cdrecorder/
|- usr/
|   |- java/
|       |- j2sdk1.4.2_05/
|           |- src.zip
```

Важно: различия файловых систем могут быть весьма ощутимыми:

В Windows имя файла обычно содержит расширение (расширение определяет ассоциированную с файлом программу). В Unix расширение не используется – ассоциация с программой делается по сигнатуре файла, внедренной непосредственно в файл). Ведущая точка в имени файла в Unix означает, что файл является скрытым.

Windows

```
C:\
|- WINDOWS\
|   |- System32\
|       |- wscript.exe
|
|- Documents and Settings\
|   |- All Users\
|   |- Default User\
|   |- student\
|       |- NTUSER.DAT
|- Program Files\
|   |- ICQ\
|       |- Icq.exe
|
D:\
|- ...
...
```

Unix

```
/
|- boot/
|- etc/
|   |- mail/
|       |- sendmail.cf
|
|- home/
|   |- ftp/
|   |   |- pub/
|   |   |- student/
|   |       |- .bashrc
|- media/
|   |- cdrecorder/
|- usr/
|   |- java/
|       |- j2sdk1.4.2_05/
|           |- src.zip
```

Java разработана так, чтобы не зависеть от используемой платформы, поэтому при грамотной организации кода все названные различия в устройствах файловых систем можно обойти.

Для идентификации файлов в Java применяется класс `File`.

```
String strUserHome =  
    System.getProperty("user.home"); // путь к домашнему каталогу пользователя  
System.out.println(strUserHome);      // --> C:\Documents and Settings\kav  
  
File fUserHome = new File(strUserHome); // идентификатор домашнего каталога  
System.out.println(fUserHome);         // --> C:\Documents and Settings\kav  
  
File fSubDir = new File(fUserHome, "SubDir"); // идентификатор подкаталога  
System.out.println(fSubDir);             // --> C:\Documents and Settings\kav\SubDir  
  
File fSubDir1 = new File(strUserHome + "/SubDir"); // нотация из Unix!  
System.out.println(fSubDir1);           // --> C:\Documents and Settings\kav\SubDir
```

Наблюдение: если использовать в качестве разделителя компонент имени файла слэш, то такая нотация будет работать не только в Unix (что очевидно), но и в Windows!

Функциональность класса File:

- построение идентификаторов файлов, обратное разложение строкового идентификатора на компоненты, построение канонического идентификатора файла (конструкторы, getName(), getPath(), getParentFile(), getAbsolutePath(), getCanonicalPath() и т.п.)
- получение атрибутов файла (canRead(), canWrite(), exists(), isDirectory(), isFile(), isHidden(), lastModified(), length())
- операции над файлами (createNewFile(), createTempFile(), delete(), renameTo())
- операции над каталогами (list(), listFiles(), mkdir(), mkdirs(), listRoots(), delete(), renameTo())
- предоставление информации о символах-разделителях используемой файловой системы (separator, pathSeparator)

Пример выдачи содержимого каталога:

```
File fUserHome = new File("C:/");
System.out.println("Files of "+fUserHome);
File [] files = fUserHome.listFiles();
for (int iIdx = 0; iIdx < files.length; iIdx++)
{
    File file = files[iIdx];
    String strLength = "";
    if (file.isDirectory())
    { strLength = "<DIR>"; }
    else if (file.isFile())
    { strLength = String.valueOf(
        file.length()); }
    else
    { strLength = "unknown"; }
    String strMsg = file.getPath();
    while (strMsg.length() < 30)
    {
        strMsg = strMsg + " ";
    }
    strMsg = strMsg + strLength;
    System.out.println(strMsg);
}
```

Пример выдачи содержимого каталога:

```
File fUserHome = new File("C:/");
System.out.println("Files of "+fUserHome);
File [] files = fUserHome.listFiles();
for (int iIdx = 0; iIdx < files.length; iIdx++)
{
    File file = files[iIdx];
    String strLength = "";
    if (file.isDirectory())
    { strLength = "<DIR>"; }
    else if (file.isFile())
    { strLength = String.valueOf(
        file.length()); }
    else
    { strLength = "unknown"; }
    String strMsg = file.getPath();
    while (strMsg.length() < 30)
    {
        strMsg = strMsg + " ";
    }
    strMsg = strMsg + strLength;
    System.out.println(strMsg);
}
```

Программа работает. Но в ней
есть неудачный код. Какой?

```
Files of C:\
C:\Arc                                     <DIR>
C:\AUTOEXEC.BAT                           0
C:\boot.ini                               193
C:\BORLANDC                               <DIR>
C:\CONFIG.SYS                             0
C:\cygwin                                 <DIR>
C:\Distr                                  <DIR>
C:\Documents and Settings                 <DIR>
C:\IO.SYS                                 0
C:\jdk1.5.0                               <DIR>
C:\MSDOS.SYS                             0
C:\NTDETECT.COM                           47580
C:\ntldr                                  233632
C:\pagefile.sys                           1610612736
C:\Program Files                          <DIR>
C:\RECYCLER                               <DIR>
C:\Sysprep                                <DIR>
C:\System Volume Information              <DIR>
C:\WINDOWS                                <DIR>
```

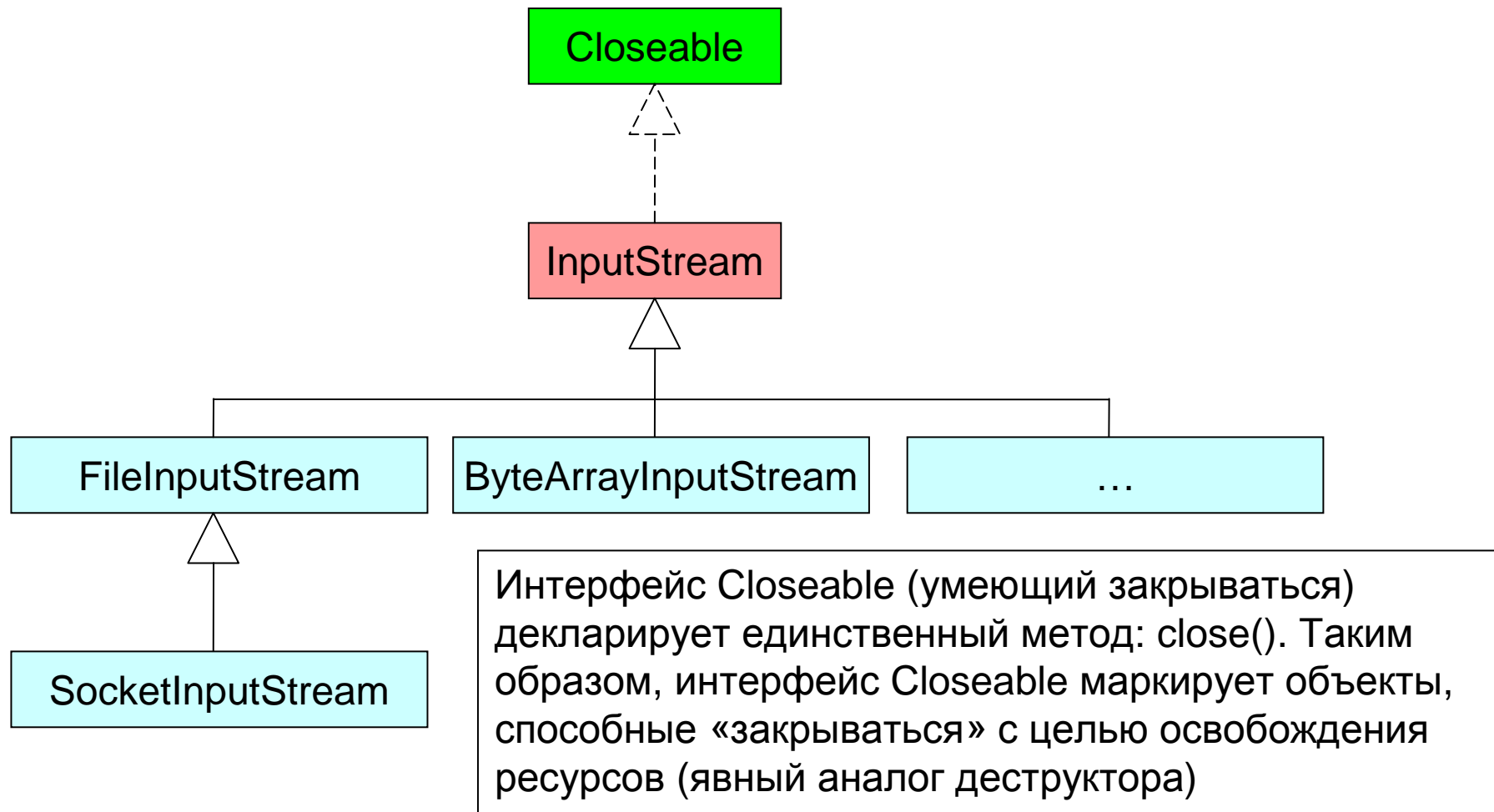
Пример выдачи содержимого каталога:

```
File fUserHome = new File("C:/");
System.out.println("Files of "+fUserHome);
File [] files = fUserHome.listFiles();
for (int iIdx = 0; iIdx < files.length; iIdx++)
{
    File file = files[iIdx];
    String strLength = "";
    if (file.isDirectory())
    { strLength = "<DIR>"; }
    else if (file.isFile())
    { strLength = String.valueOf(
        file.length()); }
    else
    { strLength = "unknown"; }
    String strMsg = file.getPath();
    while (strMsg.length() < 30)
    {
        strMsg = strMsg + " ";
    }
    strMsg = strMsg + strLength;
    System.out.println(strMsg);
}
```

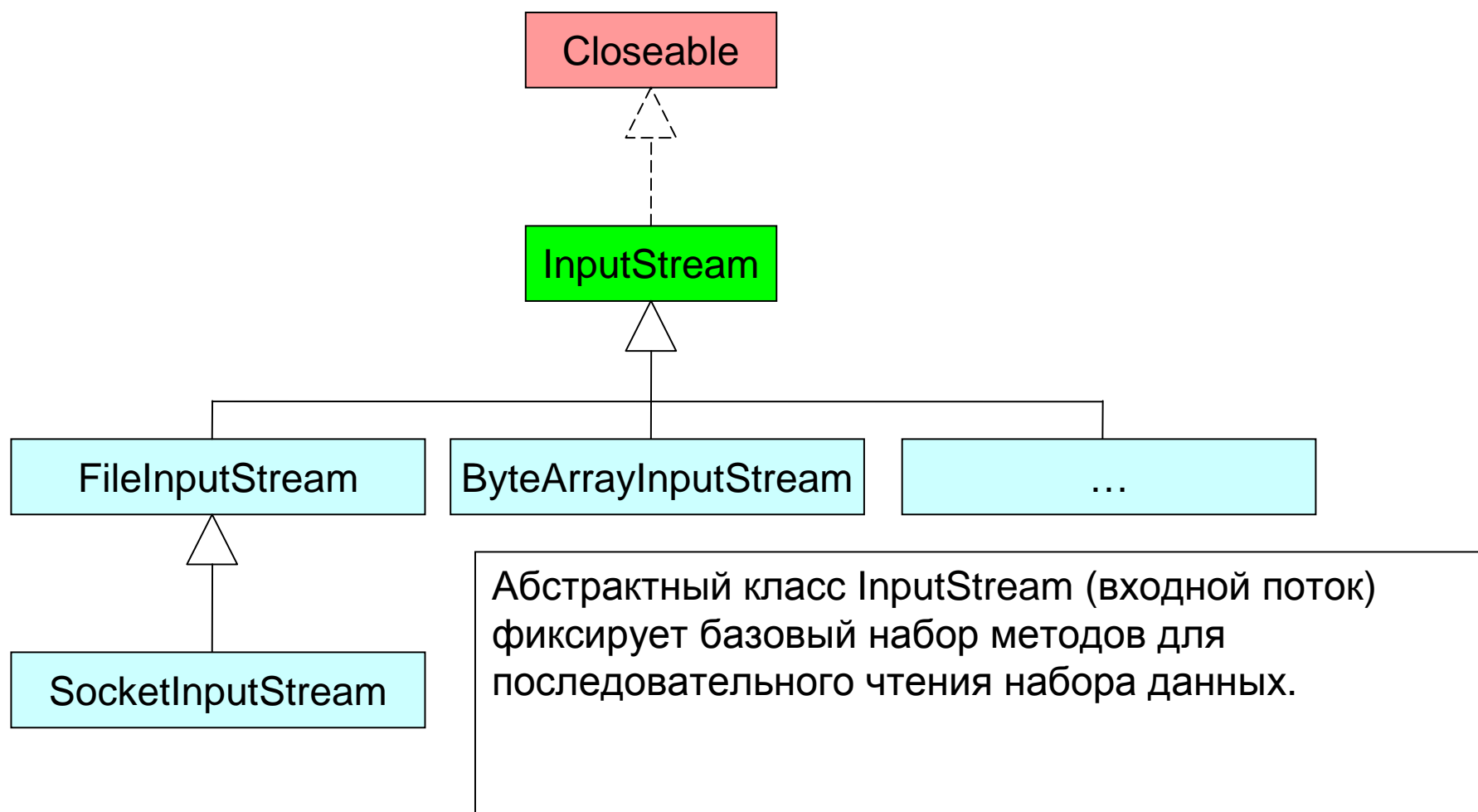
Длина имени файла может достигать 255 символов, что нарушит логику выравнивания.

Это очень неэффективный способ набирания необходимой длины строки

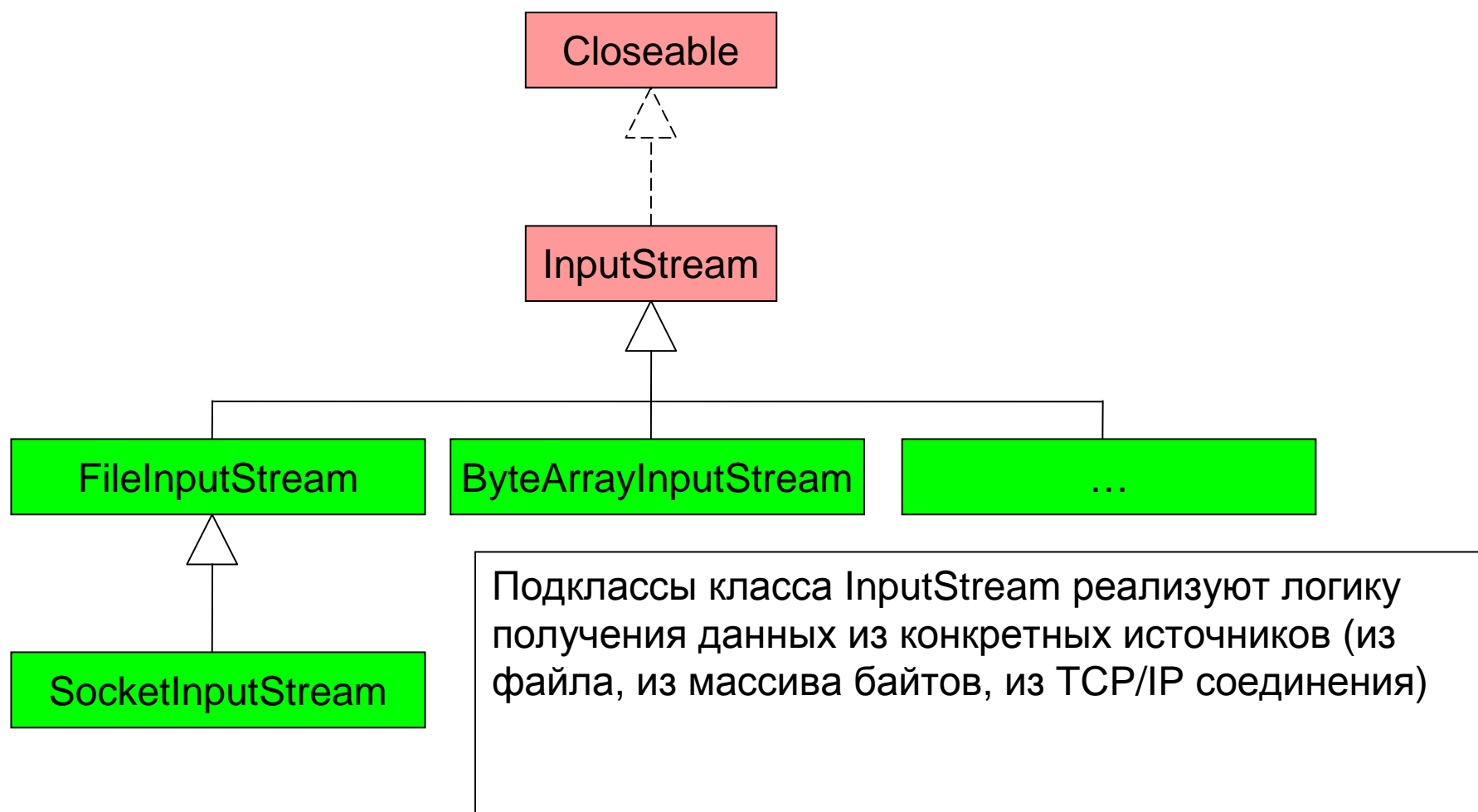
Для чтения данных из файлов в Java предусмотрена специальная система классов:



Для чтения данных из файлов в Java предусмотрена специальная система классов:



Для чтения данных из файлов в Java предусмотрена специальная система классов:



Пример выдачи содержимого файла:

```
public static void dumpStream(InputStream is)
    throws IOException
{
    int iValue;
    while ((iValue = is.read()) >= 0)
    {
        System.out.println(Integer.toHexString(iValue));
    }
}
```

Полиморфная реализация
метода, печатающего
содержимое любого потока
байтов.

```
public static void test3()
{
    File file = new File("input.txt");
    try
    {
        InputStream is = new FileInputStream(file);
        try
        {
            dumpStream(is);
        }
        finally
        {
            is.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Методы чтения сообщают о конце
потока данных путем возврата
значения -1.
Для проверки доступности данных для
чтения следует использовать метод
`InputStream.available()`

Пример выдачи содержимого файла:

```
public static void dumpStream(InputStream is)
    throws IOException
{
    int iValue;
    while ((iValue = is.read()) >= 0)
    {
        System.out.println(Integer.toHexString(iValue));
    }
}

public static void test3()
{
    File file = new File("input.txt");
    try
    {
        InputStream is = new FileInputStream(file);
        try
        {
            dumpStream(is);
        }
        finally
        {
            is.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Следует обратить внимание на использование комбинации блоков try/finally и try/catch.

Почему эти блоки разделены?

Пример выдачи содержимого файла:

```
public static void dumpStream(InputStream is)
    throws IOException
{
    int iValue;
    while ((iValue = is.read()) >= 0)
    {
        System.out.println(Integer.toHexString(iValue));
    }
}
```

```
public static void test3()
{
    File file = new File("input.txt");
    try
    {
        InputStream is = new FileInputStream(file);
        try
        {
            dumpStream(is);
        }
        finally
        {
            is.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Блок try/finally обеспечивает гарантированное закрытие объекта, связанного с внешними ресурсами. Блок try/catch обеспечивает перехват ошибок и выдачу диагностики на экран.

Важно: для обеспечения стабильной работы кода, использование объектов, ассоциированных с какими-либо внешними ресурсами всегда должно оформляться по следующей схеме (внешний блок try/catch может отсутствовать):

```
try // блок try/catch для ловушки ошибок
{
    <создание объекта>;
    try // блок try/finally для обеспечения закрытия объекта
    {
        <использование объекта>;
    }
    finally
    {
        <закрытие объекта и освобождение ассоциированных ресурсов>;
    }
}
catch (SomeException ex)
{
    <обработка ошибок>
}
```

Блочные операции с потоками.

Рассмотренный пример работы с файлом обладает низкой производительностью по причине того, что данные читаются последовательно по одному байту за операцию.

Для повышения эффективности кода следует использовать блочные операции, которые обеспечивают чтение целого блока данных:

```
public static void dumpStreamFast(InputStream is)
    throws IOException
{
    byte [] buffer = new byte[1000]; // буфер для данных
    int iCount;                      // число прочитанных байтов
    while ((iCount = is.read(buffer)) >= 0) // читаем блок данных в буфер
    {
        if (iCount > 0)
        {
            // выводим данные из буфера
            for (int iIdx = 0; iIdx < iCount; iIdx++)
            {
                System.out.println(Integer.toHexString(buffer[iIdx]));
            }
        }
    }
}
```


Наблюдение: на некоторых реализациях входных потоков рассмотренная модификация метода `dumpStreamFast` не даст увеличения производительности. Это связано с тем, что по умолчанию метод блочного чтения `read(byte [])`, наследуемый от класса `InputStream`, просто в цикле вызывает метод `read()`.

Для гарантированного повышения производительности при использовании входных потоков следует использовать класс `BufferedInputStream` (буферизующий входной поток), который умеет оптимизировать операции чтения посредством выполнения блочного чтения информации в буфер в памяти с последующей быстрой выдачей ее потребителю из буфера.

Так следует использовать буферизующий поток ввода:

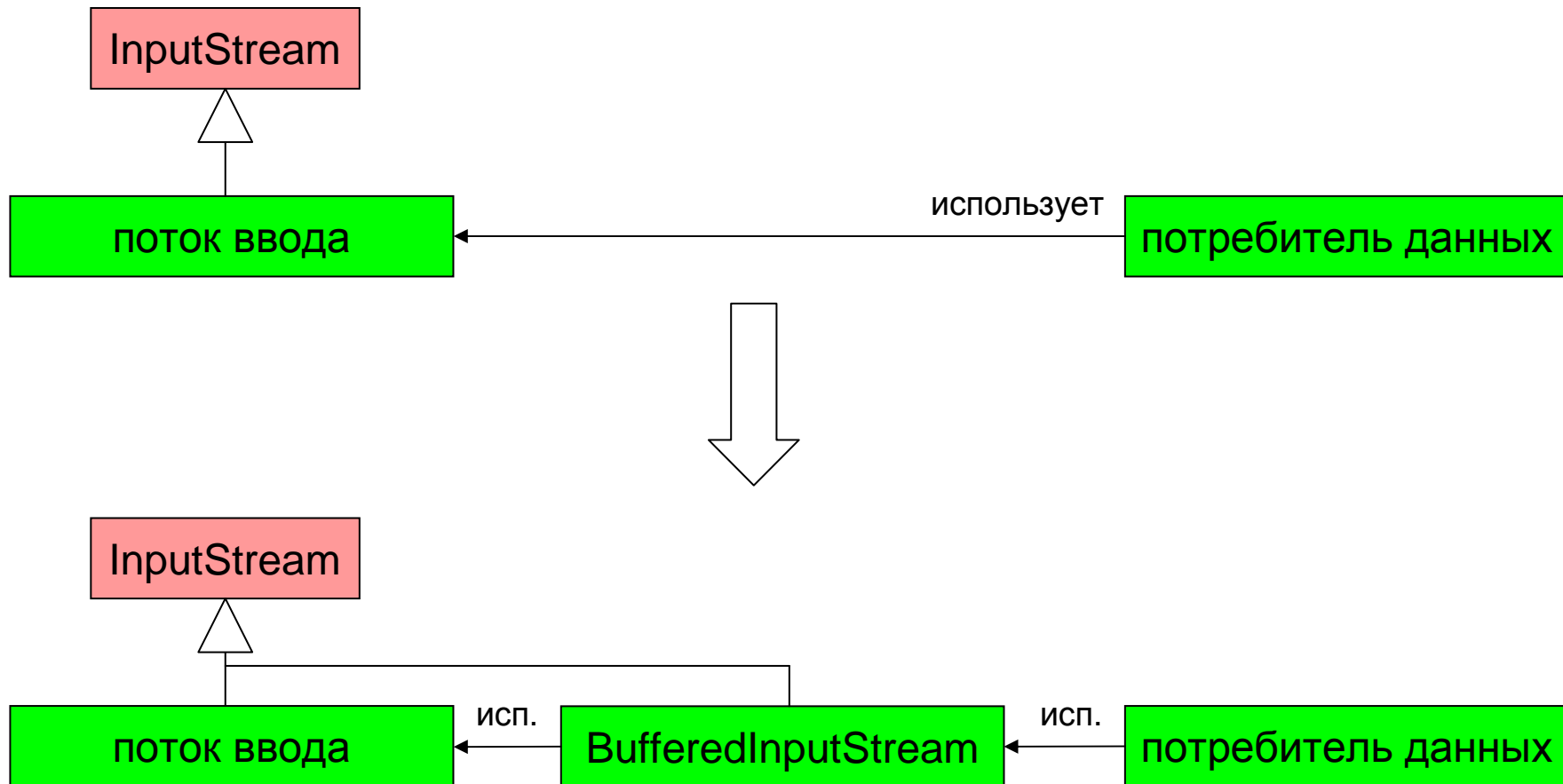
```
public static void test4()
{
    File file = new File("input.txt");
    try
    {
        InputStream is = new FileInputStream(file);
        try
        {
            BufferedInputStream bis = new BufferedInputStream(is);
            try
            {
                dumpStreamFast(bis);
            }
            finally
            {
                bis.close();
            }
        }
        finally
        {
            is.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Наблюдение: за счет полиморфизма код можно упростить:

```
public static void test4a()
{
    File file = new File("input.txt");
    try
    {
        InputStream is = new FileInputStream(file);
        try
        {
            is = new BufferedInputStream(is);
            dumpStreamFast(is);
        }
        finally
        {
            is.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

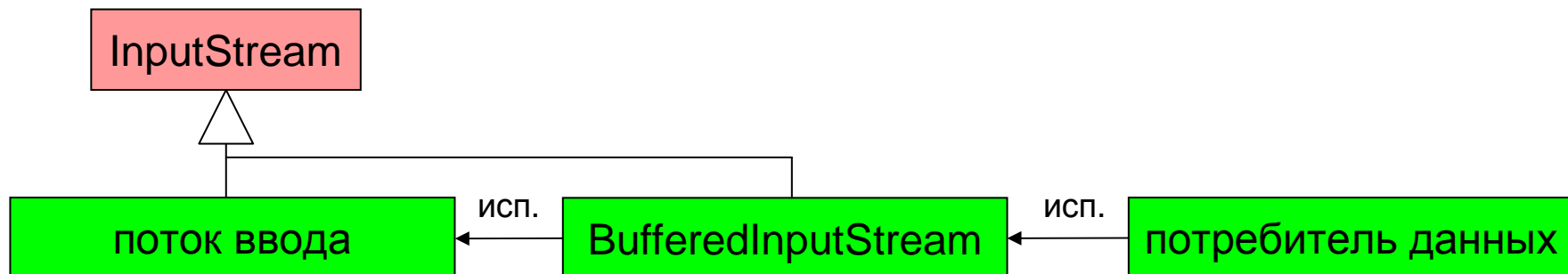
За счет того, что
BufferedInputStream является
подклассом абстрактного
класса InputStream, мы можем
заменить ссылку на
FileInputStream ссылкой на
BufferedInputStream, не
изменяя остального кода!

Наблюдение: класс `BufferedInputStream` принимает на вход любой подкласс класса `InputStream`; в то же время класс `BufferedInputStream` сам является подклассом класса `InputStream`. То есть его можно включать в произвольное место цепочки передачи данных:



Такая конструкция, когда класс реализует свою функциональность, определяемую некоторым интерфейсом, на основе класса делегата, реализующего тот же самый интерфейс, называется шаблоном «Фильтр».

Можно привести следующую аналогию. В трубу, по которой перекачивается нефть можно внедрить насос, поскольку входящая и выходящая магистрали насоса согласуются с трубой (интерфейсы согласованы). Тогда можно пользоваться трубой без насоса (прямое использование исходной реализации интерфейса), но при включении насоса скорость прохождения нефти возрастет (добавление элемента в цепочку изменяет функциональность). В то же время насос сам по себе работать не может – ему нужна подающая труба (базовая реализация интерфейса).



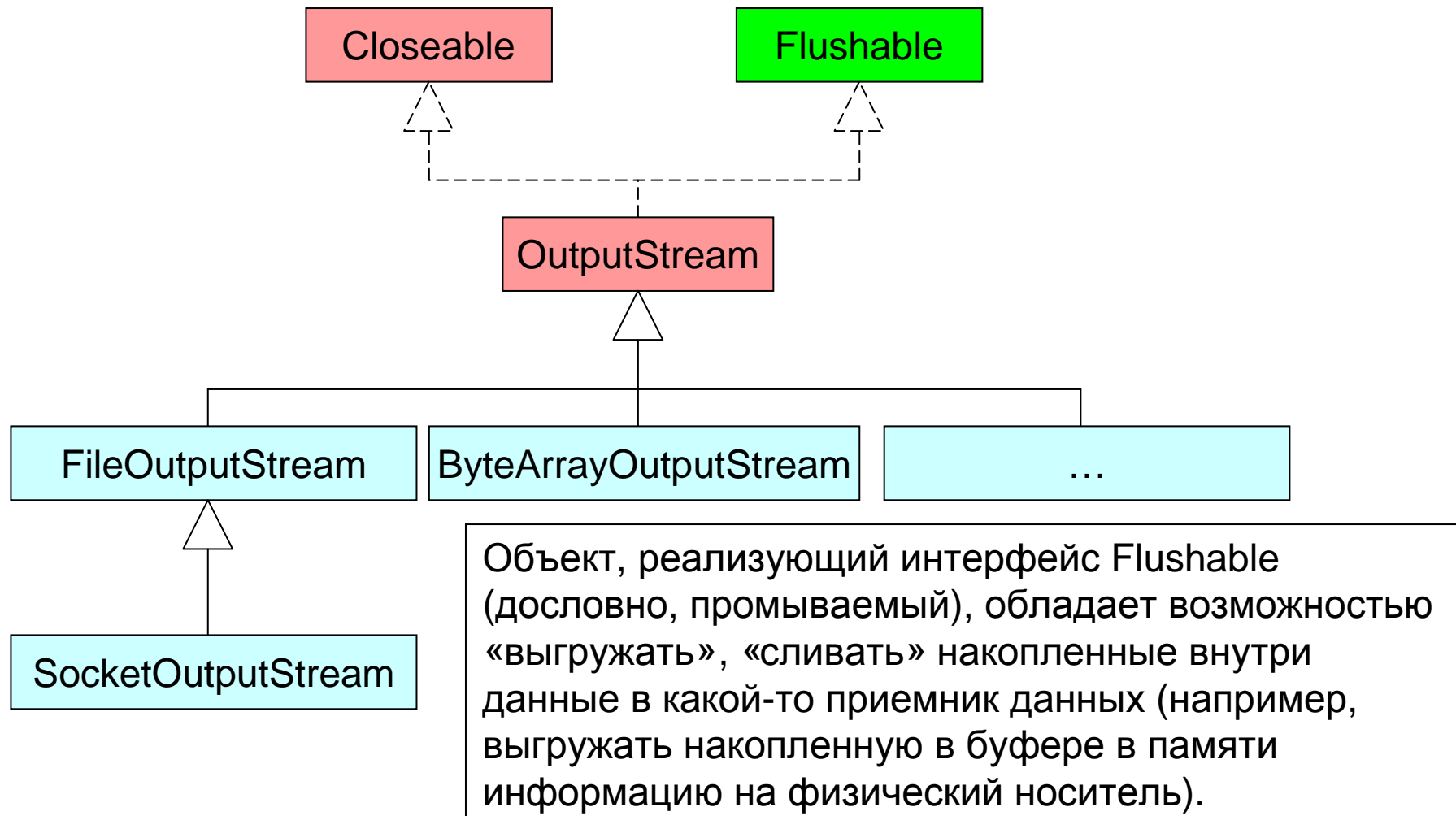
На основе шаблона «Фильтр» в Java реализована целая серия классов для обработки данных:

- **FilterInputStream** (базовый класс для создания входных потоков на основе шаблона «Фильтр»)
- **BufferedInputStream** (буферизует данные, тем самым повышая производительность)
- **DataInputStream** (добавляет методы для типизированных операций с данными)
- **LineNumberInputStream** (добавляет методы для определения номера строки файла, в которой находится текущая позиция ввода)
- **ObjectInputStream** (добавляет методы для чтения объектов)
- **PushbackInputStream** (добавляет методы, позволяющие возвращать уже прочитанные данные обратно в поток для повторного чтения)

Также для полноты картины следует упомянуть классы

- **PipedInputStream** (создает канал передачи данных между компонентами Java программы)
- **SequenceInputStream** (превращает несколько входных потоков в один, читая их последовательно)

По аналогии с тем, как реализованы в Java входные потоки, устроены и классы выходных потоков.



Система вспомогательных классов для организации выходных потоков также построена на основе шаблона «Фильтр». В нее входят следующие классы:

- **FilterOutputStream** (базовый класс для создания выходных потоков на основе шаблона «Фильтр»)
- **BufferedOutputStream** (буферизует выводимые данные, тем самым повышая производительность)
- **DataOutputStream** (предоставляет методы для типизированных операций с данными)
- **ObjectOutputStream** (добавляет методы для записи объектов)
- **PrintStream** (добавляет методы для форматированного вывода примитивных типов в текстовый файл; не рекомендуется использовать)

Также для полноты картины следует упомянуть класс **PipedOutputStream** (создает канал передачи данных между компонентами Java программы)

Пример работы с файлами (начало):

```
public static void test5()
{
    File file = new File("data.bin");
    try
    {
        OutputStream os = new FileOutputStream(file);
        try
        {
            DataOutputStream dos =
                new DataOutputStream(new BufferedOutputStream(os));
            try
            {
                dos.writeUTF("Here goes integer: ");
                dos.writeInt(2007);
                dos.flush();
            }
            finally
            {
                dos.close();
            }
        }
        finally
        {
            os.close();
        }
    }
}
```

Пример работы с файлами (конец):

```
InputStream is = new FileInputStream(file);
try
{
    DataInputStream dis = new DataInputStream(new BufferedInputStream(is));
    try
    {
        String strMsg = dis.readUTF() + dis.readInt();
        System.out.println(strMsg);
    }
    finally
    {
        dis.close();
    }
}
finally
{
    is.close();
}
}
catch (IOException ex)
{ ex.printStackTrace(); }
}
```

Результат работы:

Here goes integer: 2007

Если посмотреть на то, что оказалось записанным в файл data.bin в результате работы рассмотренного примера, то окажется, что файл не является текстовым! Вот так выглядит его содержимое в шестнадцатеричном виде:

```
00000000:  00 12 48 65 72 65 20 67 | 6F 65 73 20 69 6E 74 65    ↑Here goes inte
00000010:  67 65 72 3A 00 00 07 D7 |                                ger:  •‡
```

То есть рассмотренные системы классов, основанные на абстрактных классах `InputStream` и `OutputStream` решают задачу работы с бинарными файлами. В эти файлы можно помещать произвольную информацию, но для получения в файле текстовой информации требуются дополнительные действия.

Наблюдение: класс `DataOutputStream` сохранил строку в специальном внутреннем формате (был закодирован не только текст строки, но и ее длина).

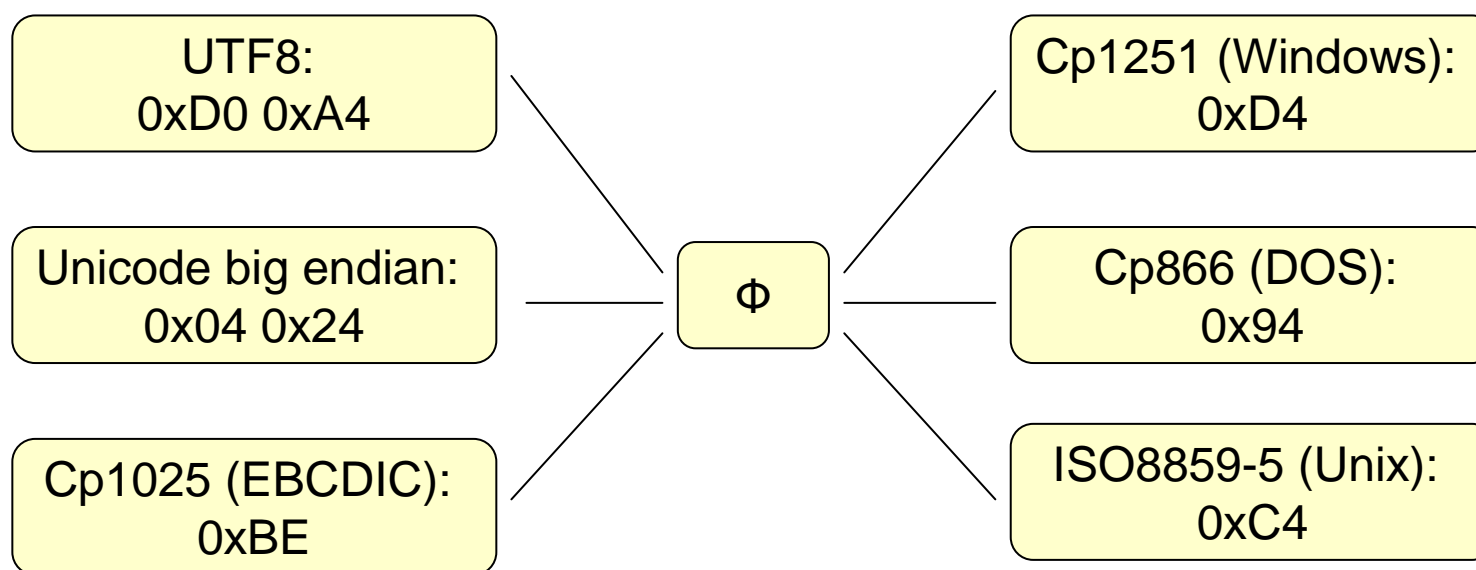
Для создания текстовых файлов следует прежде всего разобраться с тем, что такое «текстовый файл».

Что это такое?

Для создания текстовых файлов следует прежде всего разобраться с тем, что такое «текстовый файл».

Текстовый файл – это файл, бинарное наполнение которого состоит из кодов символов в заданной кодировке.

Наблюдение: одному символу в разных кодировках могут соответствовать разные комбинации байтов.



Чтобы у программиста не возникало путаницы относительно того, с чем он работает: с символом или с его кодом в какой-то кодировке, в Java типы данных для хранения байтов и для хранения символов разделены.

Для хранения символов в Java предназначается тип `char`. Этот тип хранит 16-битные целые значения, которые соответствуют кодам символов в кодировке Unicode. Данная кодировка поддерживает представление всех используемых в настоящее время на Земле символов (даже всех китайских иероглифов).

Переход от символов к их кодам и обратно может быть выполнен посредством использования методов класса String:

```
public static void test6()
{
    try
    {
        String strTest = "Φ";
        byte [] data = strTest.getBytes("ISO8859-5");
        String strMsg = "";
        for (int iIdx = 0; iIdx < data.length; iIdx++)
        {
            int value = (data[iIdx] + 256) % 256;
            strMsg += Integer.toHexString(value);
            strMsg += " ";
        }
        System.out.println(strMsg);
        System.out.println(new String(data, "ISO8859-5"));
    }
    catch (UnsupportedEncodingException ex)
    {
        ex.printStackTrace();
    }
}
```

Важно! В процессе работы всегда следует понимать, с чем ведется работа: с символами или с их кодами в некоторой кодировке. Иначе неизбежны ошибки при обработке информации.

Таким образом, для чтения текстового файла следует его открыть как бинарный, после чего его надо проинтерпретировать в заданной кодировке.

Если известна кодировка, то дальше можно мыслить себе файл не как набор байтов, а как набор символов. Соответственно, для дальнейшей работы с данными вполне можно применять технику, аналогичную рассмотренной применительно к бинарным данным. Только вместо абстракции `InputStream`, предоставляющей байты, следует рассматривать абстракцию `Reader`, предоставляющую символы, а вместо `OutputStream`, соответственно, `Writer`.

В настоящее время используется 7 существенно разных типов кодировок:

1. Однобайтовые кодировки (256 символов, однобайтовые коды)
 - а. семейство ASCII (коды 0-127 соответствуют таблице ASCII)
 - б. семейство EBCDIC (применяются на мэйнфреймах)
2. Мультибайтовые одноязычные кодировки (применяются для кодирования иероглифов)
 - а. префиксные (коды занимают 1 или 2 байта, разрядность кода определяется первым байтом)
 - б. с переключением (коды занимают 1 или 2 байта, переключение разрядности производится специальными кодами shift-in и shift-out)
3. Юникод (Unicode; 2 байтовый код, охватывает все действующие системы письменности)

Важно! При чтении кода Unicode надо знать порядок следования байтов в коде (little endian / big endian)
4. UTF-8 (алгоритмическая трансформация Unicode; префиксный код с переменной длиной; совместима с кодом ASCII; обладает свойством синхронизации)
5. GB-18030 (4-байтовая кодировка для представления китайских иероглифов)

UTF-8 (от англ. Unicode Transformation Format — формат преобразования Юникода) — кодировка, реализующая представление Юникода, совместимое с 8-битным кодированием текста (характерно для традиционных схем хранения строк в языках C/C++, Pascal и др.).

Символы UTF-8 получаются из Unicode следующим образом:

Unicode	UTF-8
0x00000000–0x0000007F:	0xxxxxxx
0x00000080–0x000007FF:	110xxxxx 10xxxxxx
0x00000800–0x0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
0x00010000–0x001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
(теоретически возможны, но не включены в стандарт также:)	
0x00200000–0x03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0x04000000–0x7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

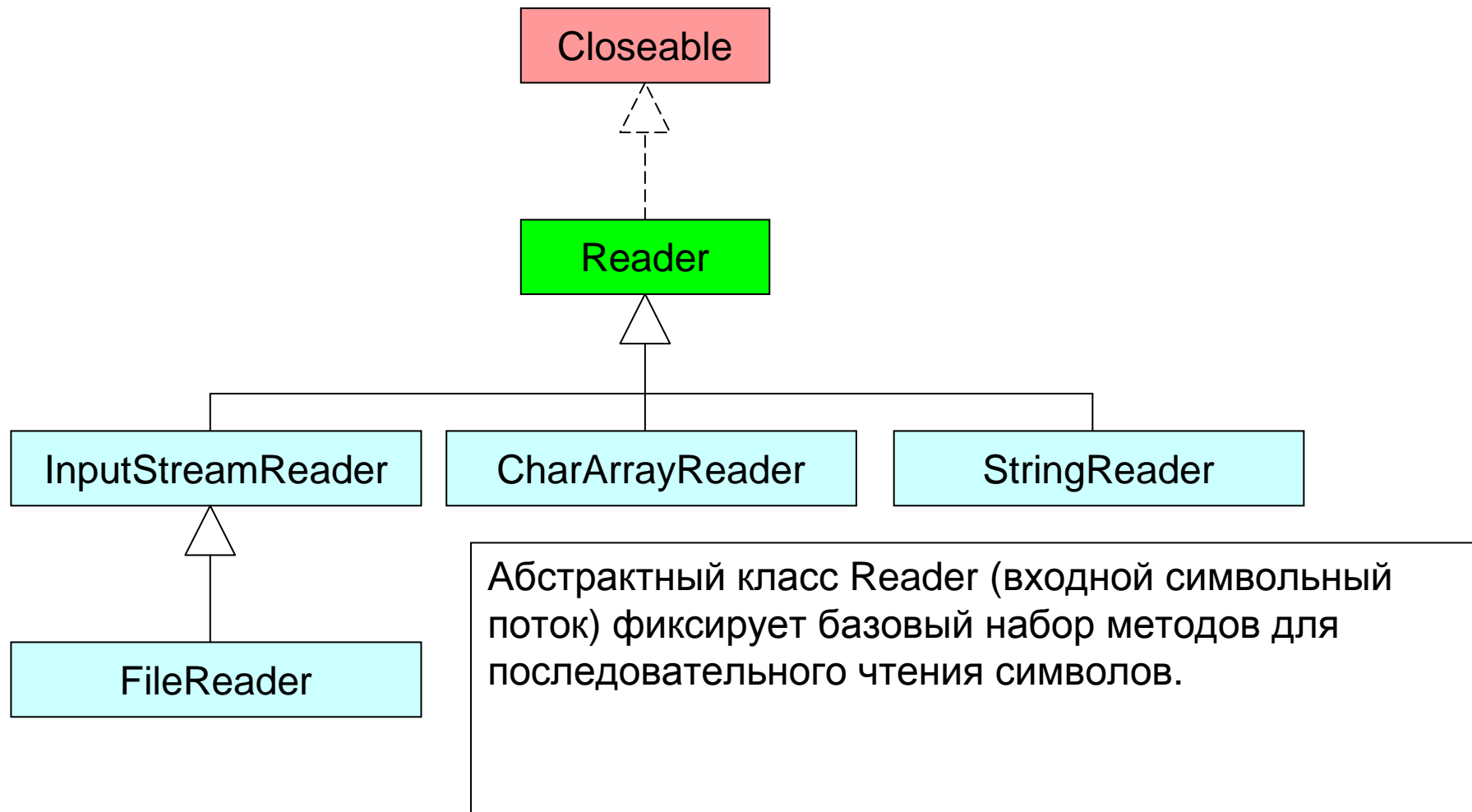
Достоинства UTF-8:

1. Байт со значением 0 используется только для представления символа с кодом 0.
2. Кодовая таблица ASCII является подмножеством UTF-8 (в частности, латиница в UTF-8 представляется однобайтовыми кодами).
3. UTF-8 – это префиксный синхронизируемый код (по структуре байтов можно обнаружить начало очередного символа, даже если произошла рассинхронизация при декодировании)

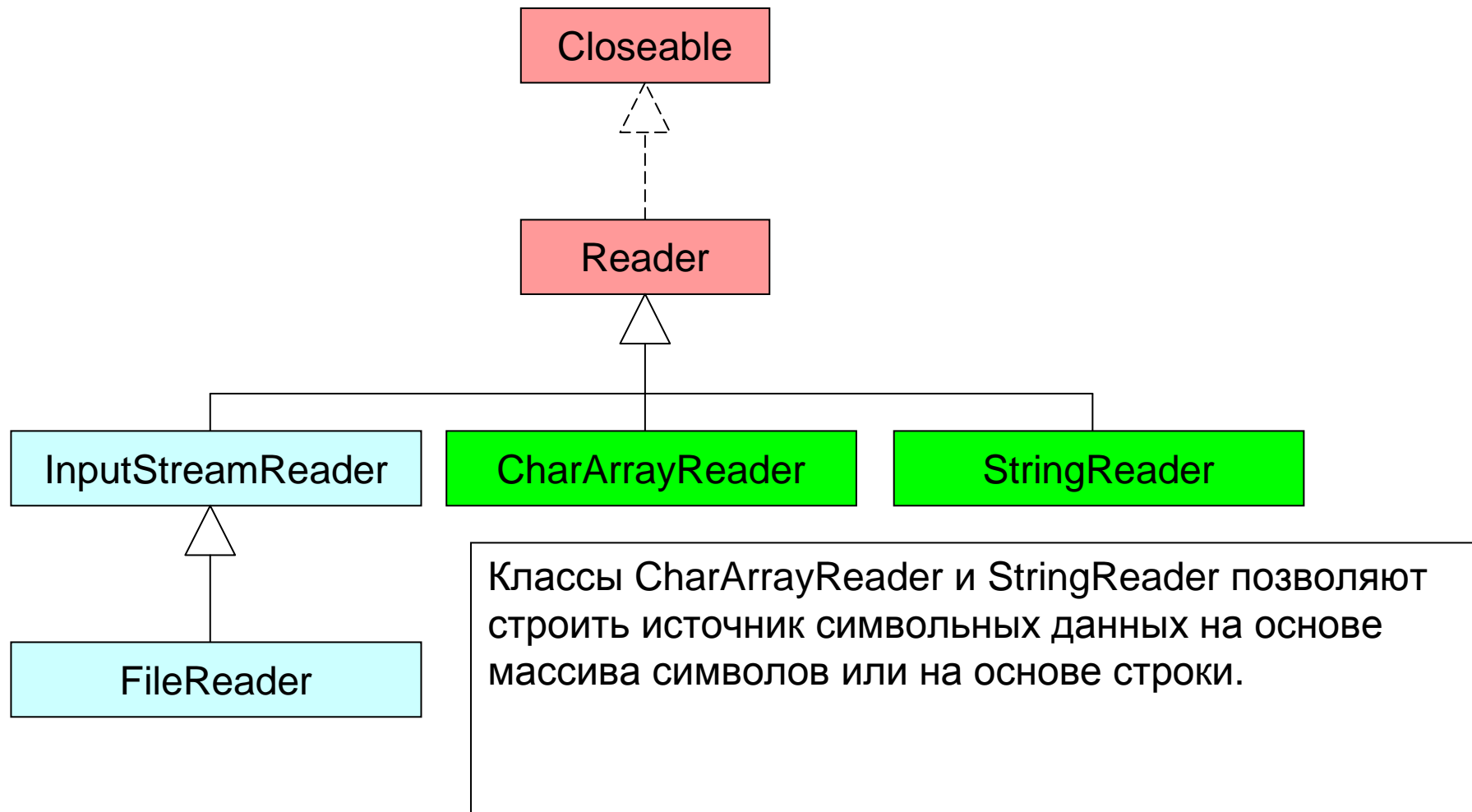
Недостатки UTF-8:

1. Переменная длина кода (от 1 до 3 байт на один символ Unicode, по 6 байт на составные символы).
2. Неэффективное представление символов за пределами латиницы (3 байта на символ для иероглифов, 2 байта на символ кириллицы).

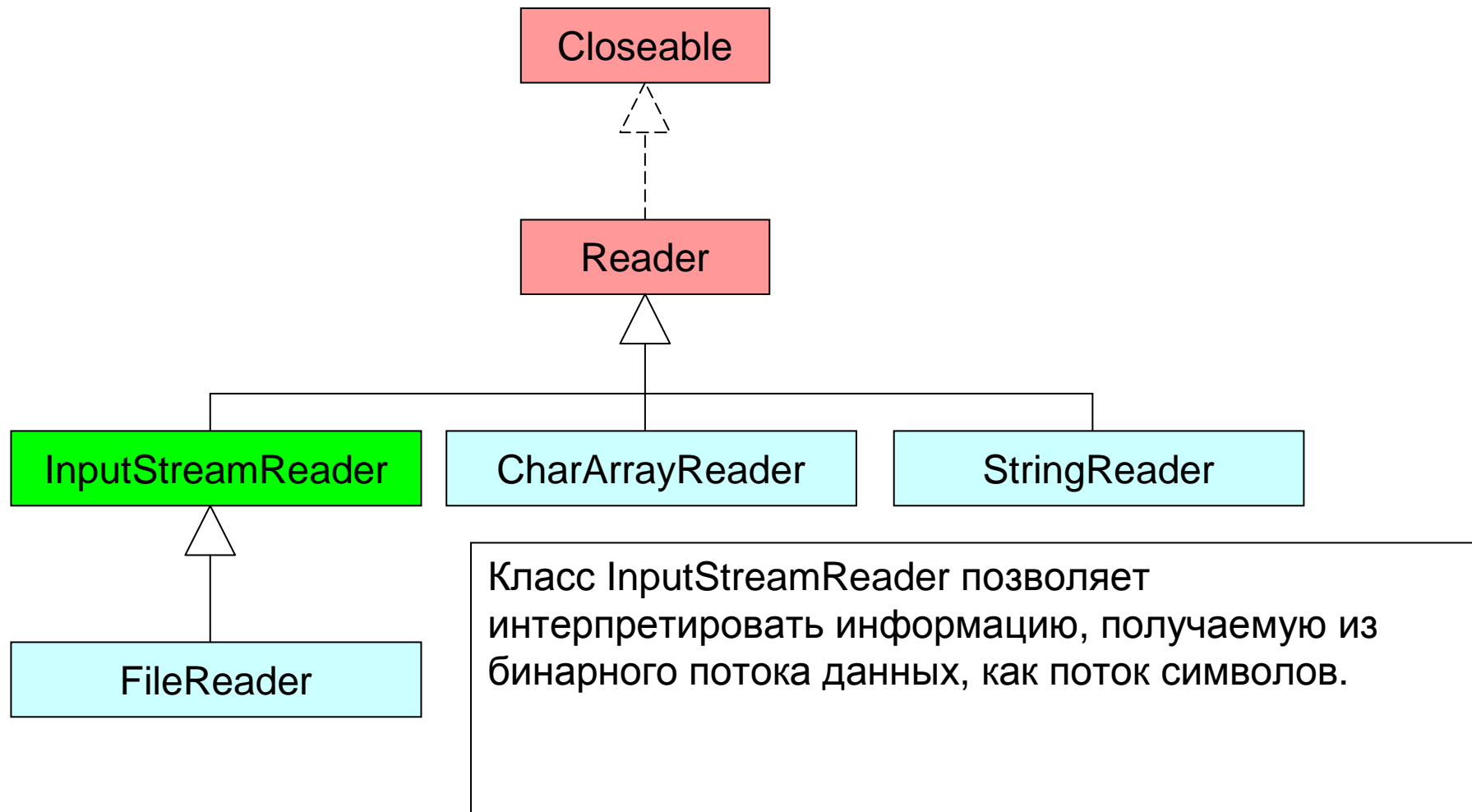
Так выглядит система классов для работы с текстовыми потоками:



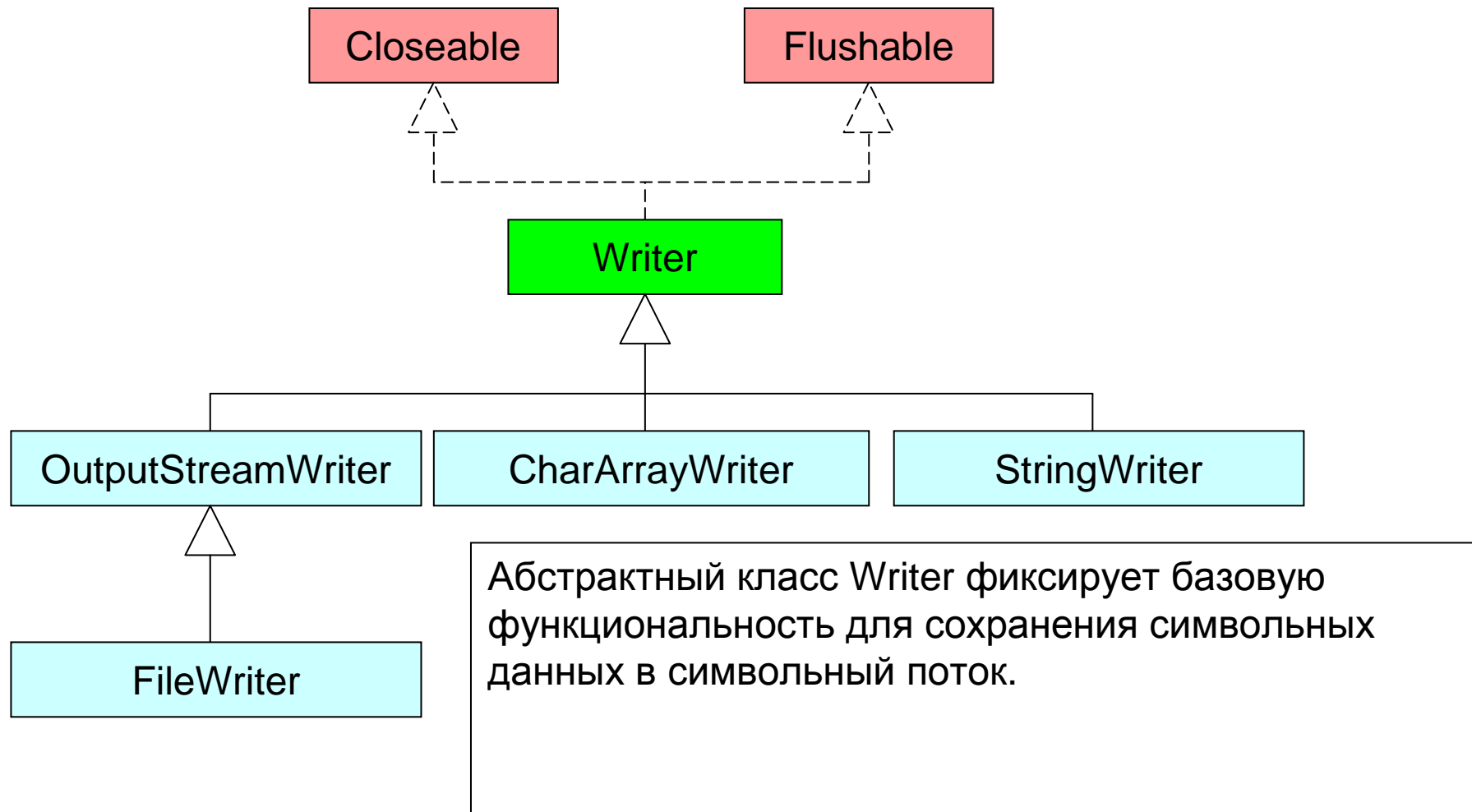
Так выглядит система классов для работы с текстовыми потоками:



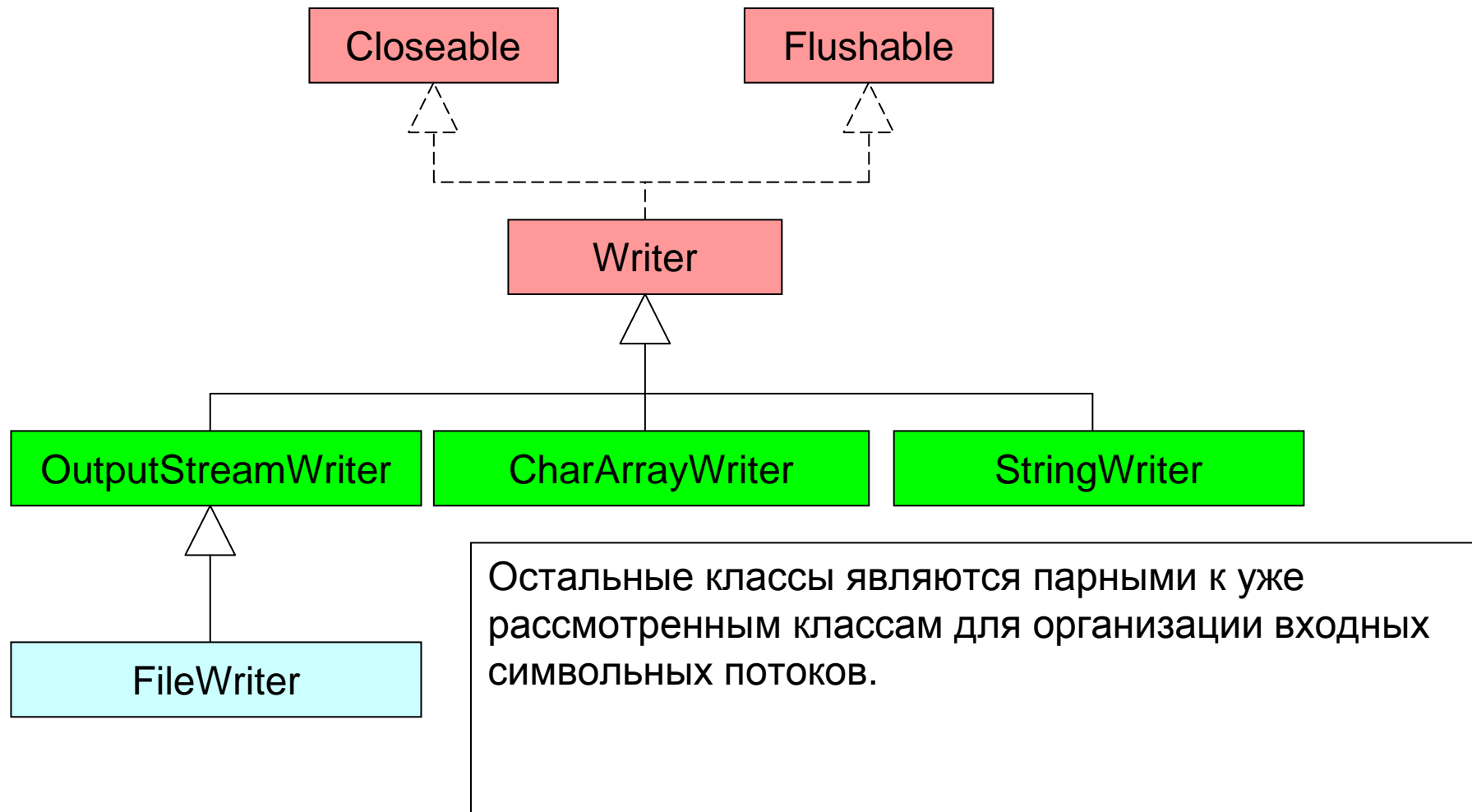
Так выглядит система классов для работы с текстовыми потоками:



По аналогии с тем, как реализованы в Java входные потоки, устроены и классы выходных потоков.



По аналогии с тем, как реализованы в Java входные потоки, устроены и классы выходных потоков.



Числовые данные в текстовых файлах

Для того, чтобы напечатать в текстовый поток числовую информацию, следует использовать класс `PrintWriter`. Этот класс спроектирован в рамках шаблона «Фильтр» и, следовательно, может работать поверх любой реализации абстракции `Writer`.

Класс `PrintWriter` предоставляет серии перегруженных методов `print` и `println`, позволяющих напечатать в текстовый поток данные любого примитивного типа.

Вопрос: что будет напечатано в результате следующего вызова `println()`?

```
Object obj = ...;  
PrintWriter pw = ...;  
pw.println(obj);
```

Числовые данные в текстовых файлах

Для того, чтобы напечатать в текстовый поток числовую информацию, следует использовать класс `PrintWriter`. Этот класс спроектирован в рамках шаблона «Фильтр» и, следовательно, может работать поверх любой реализации абстракции `Writer`.

Класс `PrintWriter` предоставляет серии перегруженных методов `print` и `println`, позволяющих напечатать в текстовый поток данные любого примитивного типа.

Вопрос: что будет напечатано в результате следующего вызова `println()`?

Ответ: следующие вызовы эквивалентны:

```
Object obj = ...;
PrintWriter pw = ...;
pw.println(obj);

pw.println(obj.toString());

pw.println(""+obj);
```

Числовые данные в текстовых файлах

Для того, чтобы прочитать числовые данные из текстового потока, следует использовать класс `Scanner (java.util.Scanner)`. Этот класс содержит готовые примитивы для синтаксического разбора информации во входном потоке и для преобразования ее в числовые данные.

Важно! При сохранении числовых данных в текстовый файл и при чтении числовых данных из текстового файла следует учитывать языковые настройки пользователя (локаль). Так, например, в разных локалях отличаются способы обозначения десятичного разделителя (точка в американской локали и запятая в русской).

Пример работы с текстовыми файлами:

```
public static void test7()
{
    File file = new File("output.txt");
    try
    {
        OutputStream os = new FileOutputStream(file);
        try
        {
            PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(new BufferedOutputStream(os), "Cp1251"));
            try
            {
                writer.println("Hello, World!");
            }
            finally
            {
                writer.close();
            }
        }
        finally
        {
            os.close();
        }
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```


Пример анализа входного потока:

```
public static void test7a()
{
    try
    {
        String strEncoding = System.getProperty("file.encoding");
        System.out.println("Input encoding is: "+strEncoding);

        System.out.print("Enter simple numeric expression: ");
        Scanner scanner = new Scanner(
            new InputStreamReader(System.in, strEncoding));
        scanner.useDelimiter("");
        String strVal1 = scanner.next("\\d+");
        String strOp = scanner.next("[\\+\\-\\*\\/]");
        String strVal2 = scanner.next("\\d+");

        PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(
                new BufferedOutputStream(System.out), strEncoding));
        writer.println(
            MessageFormat.format("Parsed expression: {0} {1} {2}",
                new Object[]{strVal1, strOp, strVal2}));
        writer.flush();
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Это один из способов для
определения кодировки,
используемой по умолчанию.



Пример анализа входного потока:

```
public static void test7a()
{
    try
    {
        String strEncoding = System.getProperty("file.encoding");
        System.out.println("Input encoding is: "+strEncoding);

        System.out.print("Enter simple numeric expression: ");
        Scanner scanner = new Scanner(
            new InputStreamReader(System.in, strEncoding));
        scanner.useDelimiter("");
        String strVal1 = scanner.next("\\d+");
        String strOp = scanner.next("[\\+\\-\\*\\/]");
        String strVal2 = scanner.next("\\d+");

        PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(
                new BufferedOutputStream(System.out), strEncoding));
        writer.println(
            MessageFormat.format("Parsed expression: {0} {1} {2}",
                new Object[]{strVal1, strOp, strVal2}));
        writer.flush();
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Так инициализируется сканнер для
синтаксического разбора входных данных

Пример анализа входного потока:

```
public static void test7a()
{
    try
    {
        String strEncoding = System.getProperty("file.encoding");
        System.out.println("Input encoding is: "+strEncoding);

        System.out.print("Enter simple numeric expression: ");
        Scanner scanner = new Scanner(
            new InputStreamReader(System.in, strEncoding));
        scanner.useDelimiter("");
        String strVal1 = scanner.next("\\d+");
        String strOp = scanner.next("[\\+\\-\\*\\/]");
        String strVal2 = scanner.next("\\d+");

        PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(
                new BufferedOutputStream(System.out), strEncoding));
        writer.println(
            MessageFormat.format("Parsed expression: {0} {1} {2}",
                new Object[]{strVal1, strOp, strVal2}));
        writer.flush();
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Так можно выделить во входном потоке лексемы по регулярному выражению

Пример анализа входного потока:

```
public static void test7a()
{
    try
    {
        String strEncoding = System.getProperty("file.encoding");
        System.out.println("Input encoding is: "+strEncoding);

        System.out.print("Enter simple numeric expression: ");
        Scanner scanner = new Scanner(
            new InputStreamReader(System.in, strEncoding));
        scanner.useDelimiter("");
        String strVal1 = scanner.next("\\d+");
        String strOp = scanner.next("[\\+\\-\\*\\/]");
        String strVal2 = scanner.next("\\d+");

        PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(
                new BufferedOutputStream(System.out), strEncoding));
        writer.println(
            MessageFormat.format("Parsed expression: {0} {1} {2}",
                new Object[]{strVal1, strOp, strVal2}));
        writer.flush();
    }
    catch (IOException ex)
    { ex.printStackTrace(); }
}
```

Так следует строить выходные
сообщения программы