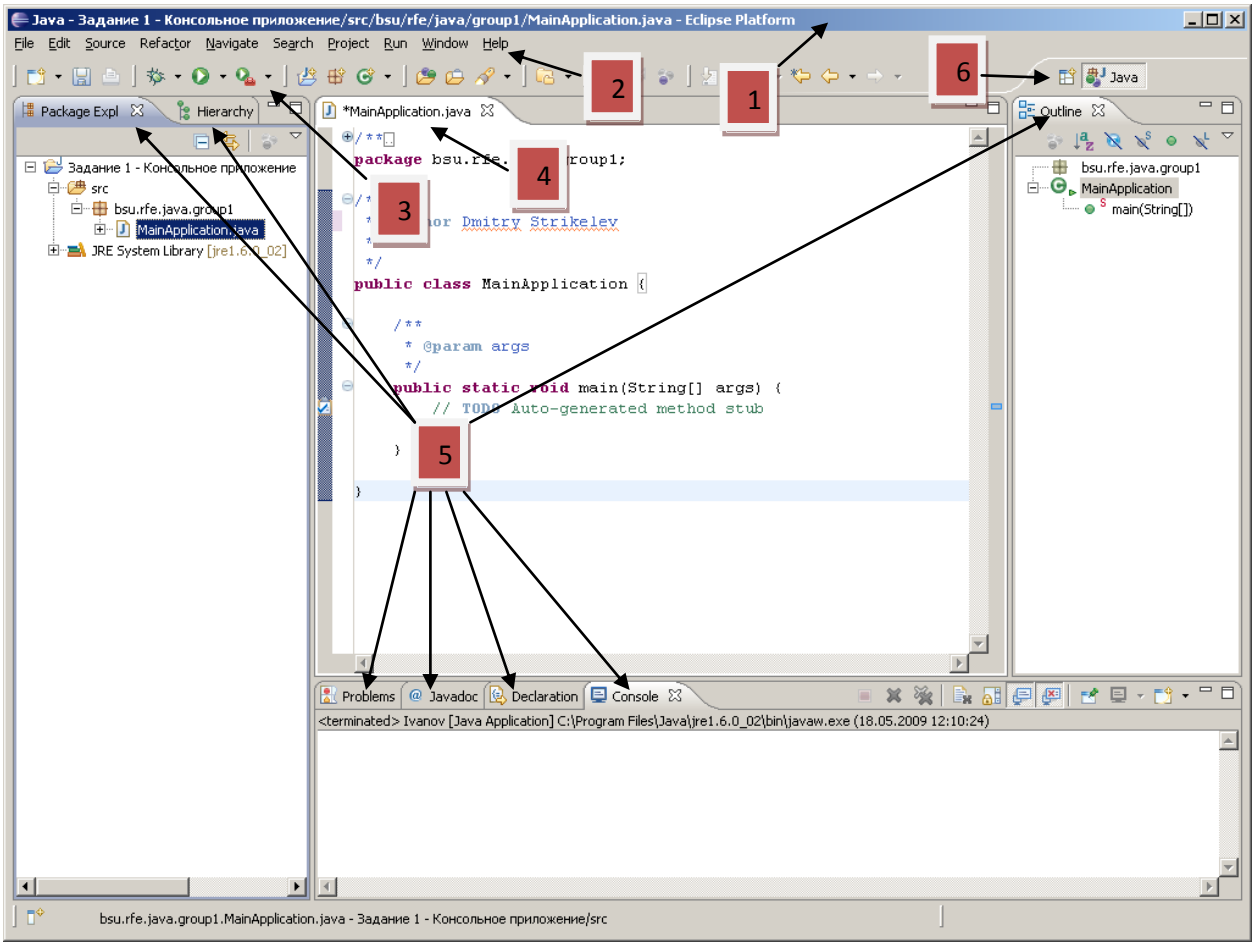
ТЕМА 1
Разработка простейшего консольного приложения

Цель лабораторной работы

Освоить основные принципы написания консольных приложений на языке Java в среде Eclipse.

1 Среда Eclipse

Среда Eclipse является платформой для разработки приложений на различных языках программирования, в том числе и на Java. Она визуально представлена в виде нескольких одновременно открытых на экране видов (окон). Количество, расположение и размер видов могут изменяться программистом в зависимости от его текущих нужд. При запуске Eclipse можно увидеть на экране картинку, схожую с рисунком 1.1.



1 – главное окно; 2 – основное меню; 3 – пиктограммы основного меню;
4 – окно редактора; 5 – виды; 6 – пиктограммы переключения перспектив

Рисунок 1.1 – Внешний вид среды Eclipse в режиме Java-перспективы

Главное окно всегда присутствует на экране и предназначено для управления процессом создания программы. **Основное меню** содержит все необходимые средства для управления проектом. **Пиктограммы** основного меню облегчают доступ к наиболее часто применяемым его командам.

Окно редактора предназначено для просмотра, написания и редактирования текста программы и является частью инструментария разработчика (Developer Toolkit – DT). Использованием альтернативных DT позволяет использовать среду Eclipse для создания программ на различных языках программирования. В рамках данного курса мы будем работать с языком Java и инструментарием разработчика Java (Java Developer Toolkit – JDT). Среди прочих, в задачи редактора входит автоматическая проверка синтаксиса текста программы, облегчение перемещения по исходному коду с помощью расставленных закладок, управление отладкой с помощью расстановки точек прерывания и др. При создании нового проекта окно редактора будет скрыто.

Виды представляют собой отдельные окна, специализированные для отображения информации определённого рода. На экране может быть открыто сразу несколько видов, переключение между которыми производится щелчком на заголовке вида. Наиболее часто используемыми и необходимыми являются виды, приведенные в таблице 1.1.

Таблица 1.1 – Наиболее часто используемые виды среды Eclipse

Название	Предназначение
Console	Отображает вывод результатов работы приложения
Problems	Показывает предупреждения и ошибки компиляции приложения
Navigator	Показывает иерархию проекта на системном уровне (уровне папок и файлов)
Outline	Показывает структуру файла, открытого в данный момент в редакторе
Package Explorer	Показывает все проекты, пакеты и файлы рабочего окружения в виде дерева
Declaration	Показывает исходный код, где определён выделенный объект
Javadoc	Показывает описание (из специальных комментариев) выделенного объекта
Hierarchy	Показывает отношение выделенного объекта к классам и интерфейсам

Наборы редакторов и видов организованы в группы, называемые **перспективами** (с возможностью их изменения программистом). При выборе какой-либо перспективы на экране автоматически появляется набор связанных с ней видов и редакторов. Каждая перспектива специализирована для выполнения определённой задачи и содержит необходимые для этого компоненты. На данном этапе нами чаще всего будут использоваться Java-перспектива, перспектива Java-обозревателя, перспектива отладки. Для быстрого переключения между перспективами (не через команду меню «*Window*» → *Open Perspective*») можно использовать блок пиктограмм 6 (рисунок 1.1).

Java-перспектива (рисунок 1.1) предназначена для работы с Java-проектами и по умолчанию состоит из редактора и таких видов как: *Package Explorer*, *Hierarchy*, *Outline*, *Problems*, *Javadoc*, *Declaration*, *Console*.

Перспектива Java-обозревателя (рисунок 1.2) включает редактор и виды: *Projects* (отображает проекты), *Packages* (пакеты), *Types* (типы, определённые в пакете), *Members* (члены классов). Выбор элемента в одном виде приводит к отображению содержания этого элемента в другом виде. Выбор элемента в виде *Members* приводит к его показу в окне редактора.

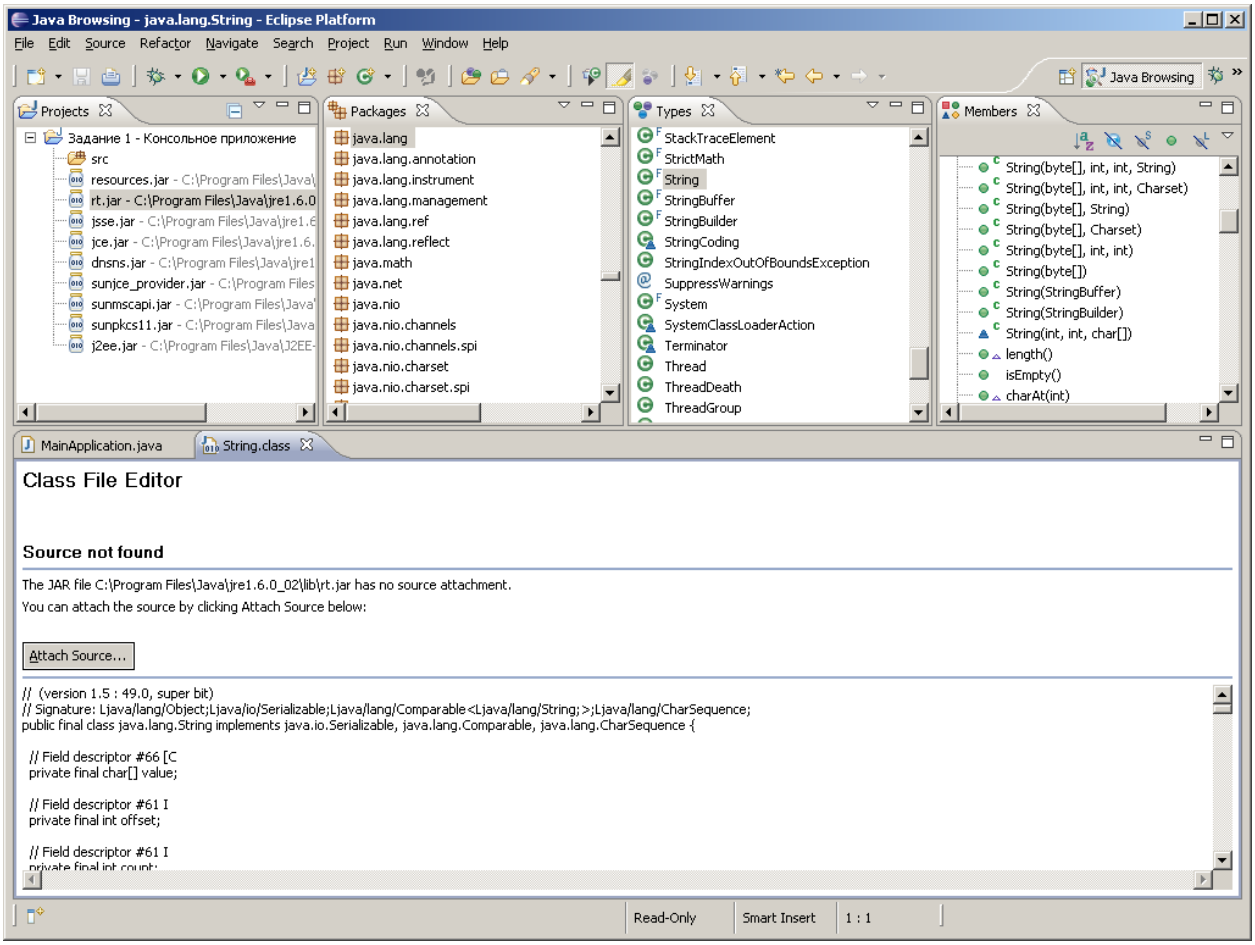


Рисунок 1.2 – Внешний вид среды Eclipse в режиме перспективы Java-обозревателя

Перспектива отладки (рисунок 1.3) предназначена для отладки Java-программ и включает окно редактора, и, как правило, виды: *Debug* (указывает местоположение выполняющегося в данный момент кода), *Breakpoints* (показывает точки прерывания), *Variables* (значения переменных), *Outline*, *Console*.

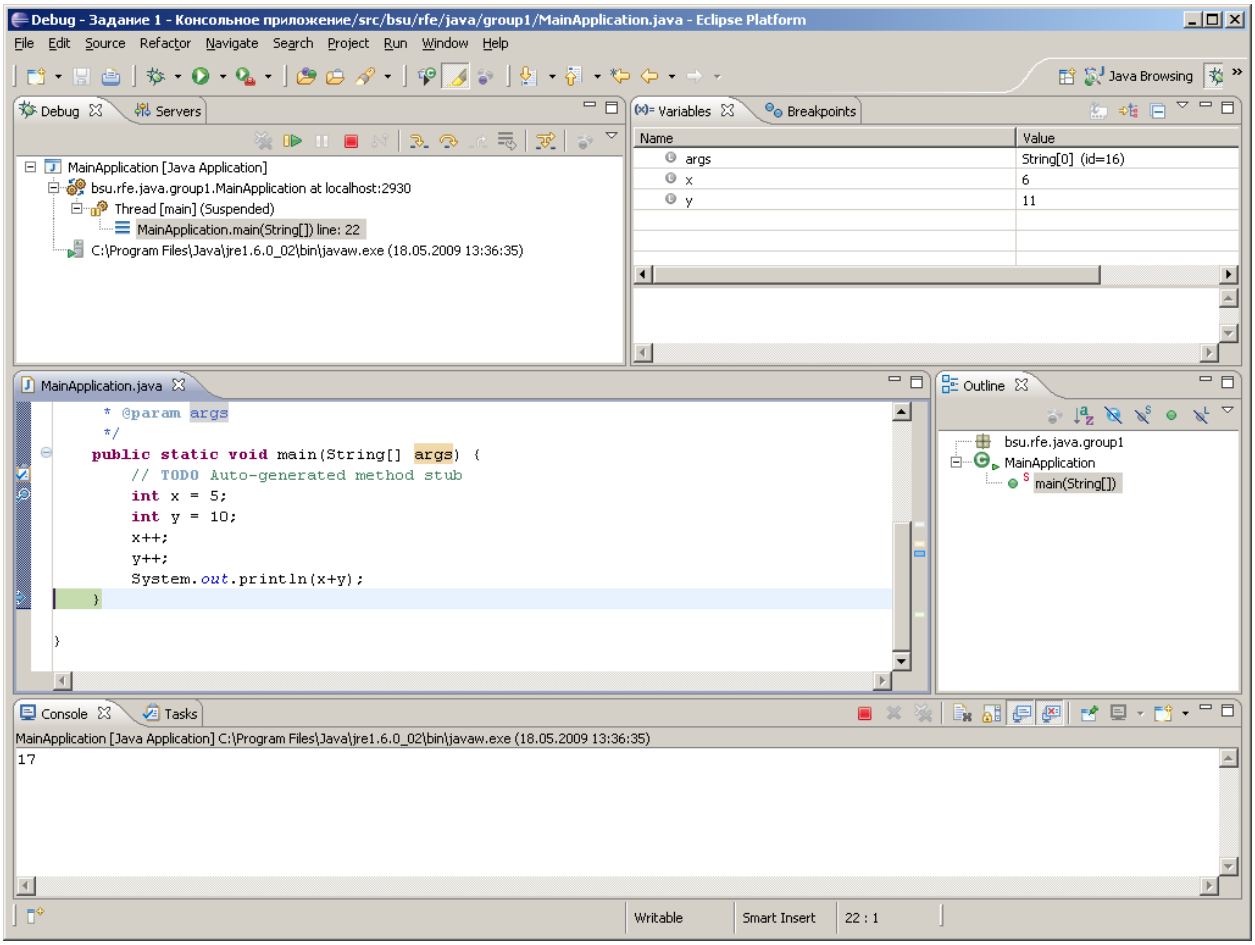


Рисунок 1.3 – Внешний вид среды Eclipse в режиме перспективы отладки

Среда Eclipse позволяет ускорить процесс разработки приложений с помощью трёх основных методов:

Eclipse использует **автоматическую подстановку кода**, позволяя выбрать из списка подходящий метод, объект или поле данных и завершить выражение. Для этого необходимо ввести символ «.», подождать некоторое время (0.1 с) и Eclipse отобразит возможные варианты продолжения выражения. Нажатие «*Ctrl + Space*» позволяет устранить временную задержку перед показом списка альтернатив.

Eclipse использует **шаблоны генерации кода**, что позволяет подставлять вместо некоторых условных обозначений (шаблонов) заданные пользователем фрагменты кода. В Eclipse существует ряд встроенных шаблонов, список которых можно изучить в диалоговом окне, открываемом из пункта меню «*Window → Preferences → Java → Editor → Templates*». В этом же окне определяются и новые шаблоны, при этом можно использовать специальные обозначения для позиционирования курсора после подстановки выражения, вставки метаданных о документе или методе, с которым ведётся работа, и т.д. Для инициирования события подстановки необходимо напечатать аббревиатуру шаблона и нажать «*Ctrl + Space*». Одним из

наиболее часто используемых примеров шаблонов генерации кода является аббревиатура «`sop`», соответствующая инструкции печати строки в консоль `System.out.println("")`.

Eclipse предоставляет **возможность рефакторинга кода** (изменения текста программы без изменения её функциональности, например, переименование классов, переменных, пакетов; перемещение классов из пакета в пакет и т.д.).

В Eclipse используется ряд «горячих клавиш», упрощающих работу в среде (таблица 1.2):

Таблица 1.2 – Основные «горячие клавиши» среды Eclipse

Функция	Клавиши
Контекстный помощник	Ctrl + Space
Включить/выключить комментарии одной строки – //	Ctrl + /
Добавить блоковые комментарии	Ctrl + Shift + /
Убрать блоковые комментарии	Ctrl + Shift + \
Сдвинуть блок кода вправо	Tab
Сдвинуть блок кода влево	Shift + Tab
Сдвинуть код в положение с правильным отступом	Ctrl + I

2 Краткая памятка по языку Java

2.1 Java-приложение

Java-приложение представляет собой некоторую композицию классов. Код Java-приложения содержится в классах Java, каждый из которых определён **в едином файле (**в файл включаются и определение, и реализация класса) **исходного кода** с расширением **.java**.

В результате компиляции для каждого класса из файла исходного кода создается **класс-файл с расширением .class**, содержащий байт-код (ряд бинарных инструкций) для абстрактной платформы (виртуальной машины Java). Класс-файлы можно использовать в любой системе, в которой есть экземпляр виртуальной машины Java, реализованной в виде интерпретатора.

В Java все файлы с расширением **.class** определенным образом организованы в тесно связанные группы классов, или **пакеты** (packages). Это означает размещение класс-файла в локальной системе файлов, а также **привилегированный доступ к переменным и методам класса из того пакета, к которому класс принадлежит**. Для объявления класса частью пакета применяется ключевое слово `package`.

Например, объявим класс `MyDemoClass` частью пакета, названного **`bsu.rfe.java.group1.lab1.Ivanov.varA12`**:

```
package bsu.rfe.java.group1.lab1.Ivanov.varA12;

class MyDemoClass {

...

}
```

Имя пакета определяет, где должен находиться класс-файл. Каждое слово в имени пакета интерпретируется как имя подкатолога. Приведенный в примере файл `MyDemoClass.class` будет находиться в файловой системе в подкаталоге с именем *bsu/rfe/java/group1/lab1/Ivanov/varA12*. Если файл там отсутствует, виртуальная машина не сможет его загрузить. Для использования класса из другого пакета необходимо указывать его полное имя. Например, полное имя класса `MainApplication`, объявленного частью пакета `bsu.rfe.java.group1.lab1.Ivanov.varA12` выглядит так:
`bsu.rfe.java.group1.lab1.Ivanov.varA12.MainApplication`.

Чтобы не набирать полные имена классов при обращении к классу из другого пакета, в Java используется оператор `import`, который позволяет осуществить доступ из текущего файла к классам из других пакетов и целым пакетам:

```
import (packageName).(classNames);
import (packageName).*;
```

При наличии такой директивы (при поиске идентификаторов классов, методов и переменных, используемых в программе) просматриваются названные пакеты и/или классы. Импорт целого пакета указывается следующим образом:

Как мы отметили, Java-приложение представляет собой композицию классов. При этом основная часть классов приложения не являются исполняемыми и предназначены для использования другими классами (т.е. не могут выступать в качестве точек входа в программу). В то же время, класс, содержащий метод с именем `main()`,может быть запущен на выполнение. Аргументы, передаваемые при запуске приложения в метод `main` (как параметр `args`) перечисляются в строке вызова.

Как правило, в приложении имеется только один запускаемый (главный) класс. Тем не менее, в ряде случаев оправдано наличие нескольких альтернативных точек входа в приложение (нескольких классов с методом `main()`). Например, для обеспечения работы приложения в различных режимах (класс `StartServer` является точкой входа в случае

функционирования в роли сервера, а класс `StartClient` – в роли клиента); для обеспечения возможности тестирования (класс `MainApplication` является точкой входа в нормальном режиме работы, класс `TestApplication` – в режиме тестирования после сбоя).

Определение метода `main()`:

```
// Задание пакета, которому принадлежит класс
package bsu.rfe.java.group1.lab0.Ivanor.varA1;

public class Lab1 {   //объявление главного класса Lab1 приложения
    public Lab1() { // конструктор класса Lab1 вызывается из метода main
          //в простых программах может не определяться
    }
    public static void main(String[] args) {
        // main() - главный метод, автоматически вызывается при запуске,
        // обязательно должен быть определён как public static void,
        // из него вызываются и управляются все классы и объекты приложения,
        // параметры командной строки находятся в массиве строк args,
        // после выполнения метода main приложение завершает свою работу
        System.out.println("Лабораторная работа 1");
    }
}
```

2.2 Документирование программы

Утилита *javadoc* обеспечивает возможность использования специальным образом оформленных комментариев в исходном коде для автоматического создания документации. Комментарии вносятся в исходные файлы с помощью специальных меток (тегов) внесены комментарии, и утилита создаёт документы формата HTML, описывающие классы, методы, переменные и константы из этих файлов.

Комментарий для утилиты *javadoc* представляет собой блок, который начинается символами «/**» и заканчивается символами «*/». Блок начинается с описательного предложения, после которого следует пустая строка, а затем специальные строки, начинающиеся с тегов *javadoc*. Блок комментария, указанный перед определением класса, метода или переменной, обрабатывается утилитой *javadoc* и преобразуется в соответствующий блок документации.

Основные javadoc-теги:

@author текст //указывает автора класса по одному в строке

@exception имя исключение // указывает причины ошибок, // сообщаемые методом

@param имя параметра //указывает параметр, передаваемый в метод

@return описание //описывает тип и диапазон значений, // возвращаемых методом

2.3 Объекты как главные элементы языка Java

2.3.1 Определение классов

Главными элементами языка Java являются **объекты**. Тип объекта определяется классом, к которому принадлежит объект. **Определение классов в Java** осуществляется следующим образом:

```
[<class_modifiers>] class <class_name>  [extends <superclass_name>]
            [implements <interface_1>, <interface_2>, ...] {
// здесь приводятся описания методов и переменных экземпляров класса
}
```

Секция `[extends <superclass_name>]` используется при определении данного класса наследником некоторого суперкласса (фактически заменяет оператор наследования C++ '**:**').

Секция `[implements <interface_1>, <interface_2>, ...]` появляется в определении класса, если он реализует один или несколько интерфейсов. Так как в Java, в отличие от C++, не допускается множественное наследование, **то общность поведения различных классов определяется в интерфейсах, содержащих набор обязательных для реализации методов.**

Модификаторы классов (секция `[<class_modifiers>]`) являются дополнительными ключевыми словами, которые записываютс перед словом `class`. Они указывают на область видимости и статус класса в программе:

- Модификатор `abstract` описывает класс, содержащий абстрактные методы. Абстрактный метод – пустой метод, без реализации. Перед описанием абстрактного метода ставится слово `abstract`. Класс, содержащий только абстрактные методы и константы, называют интерфейсом. Как правило, абстрактный класс содержит сочетание абстрактных и действительных методов.
- Модификатор `final` показывает, что данный класс не содержит подклассов.
- Модификатор `public` описывает открытый класс, который может быть переопределен или расширен с помощью любого другого элемента из того же пакета или импортируемого в класс из другого пакета.
- Класс **без модификатора** `public` (принимается по умолчанию) является «дружественным», то есть может быть использован и переопределен всеми классами того же пакета.

В классе определяются три общих для всех объектов класса понятия:

- **Набор данных** (состояние объекта).

- **Переменные** (поля) **экземпляра класса** (Java-объекта) содержат его данные. Они могут быть базового типа или содержать ссылки на объекты других классов. В классе объявляют некоторое количество (может быть и нулевое) переменных экземпляра класса.
- **Переменная класса** является общей для всех объектов класса и объявляется с модификатором `static`. Модификаторы `public`, `private`, `protected` предназначены для управления доступом к переменным класса. Модификатор `final` помечает окончательное поле, которому должно быть присвоено первоначальное значение, после чего новое значение ему уже не может быть присвоено. Значения констант, связанных с классом, объявляются с помощью модификаторов `static` и `final`.

Идентификаторы в Java начинаются с буквы и представляют собой строку букв, цифр и символов подчеркивания (причем могут быть использованы буквы и цифры любого языка, имеющегося в кодировке Unicode). В качестве идентификаторов нельзя использовать только ключевые слова языка.

Для объявления переменных экземпляра класса используются следующие базовые типы, **не являющиеся объектами**:

`boolean` – логическое значение: `true` или `false`;
`char` – 16-битовое значение в кодировке Unicode (символьное);
`byte` – 8-битовое (байтовое) целое число со знаком;
`short` – 16-битовое короткое целое число со знаком;
`int` – 32-битовое обычное целое число со знаком;
`long` – 64-битовое длинное целое число со знаком;
`float` – 32-битовое вещественное число;
`double` – 64-битовое вещественное число.

- **Методы класса.** Операции, содержащие «сообщение», на которое реагирует объект, и выполняемые над данными, называются **методами.** Методы определяют поведение объектов данного класса. Описание метода состоит из двух частей: **сигнатуры**, определяющей имя, число и типы параметров метода, и **тела метода**, в котором описываются выполняемые действия. Чтобы получить из функции несколько значений, можно объединить их в один сложный объект, поля которого содержат все возвращаемые значения, и возвратить в методе ссылку на этот сложный объект. Второй способ – изменить внутреннее состояние объекта, обрабатываемого этим методом.

Переданные в метод параметры могут быть данными базового типа или ссылками на объект. Все параметры передаются в метод значением, то есть при передаче параметра создается его копия, которая и используется внутри метода. Модификаторы метода: `public` – открытые методы, доступны отовсюду; `private` – закрытые методы, могут вызываться только методами данного класса; `protected` – защищенные методы, вызываются из класса того же пакета или их подклассов.

При отсутствии модификаторов `public`**,** `private` **и** `protected` **устанавливаются привилегии доступа по умолчанию: метод считается дружественным.** Дружественные методы вызываются для объектов **всех классов одного пакета.**

Модификатор `abstract` объявляет абстрактный метод, не имеющий кода (после списка параметров абстрактного метода ставится точка с запятой, и тело метода не описывается). Абстрактные методы используются только в абстрактных классах.

Модификатор `static` описывает метод, относящийся ко всему классу, а не к определенному его экземпляру. Статические методы используются при работе со статическими переменными класса (если они не помечены модификатором `final`).

Модификатор `final` указывает, что метод не может быть переопределен в подклассах.

- **Родительские классы.** В случае если классы порождаются от других классов, в иерархии классов при каждом вызове конструктора порожденного класса вызывается и конструктор родительского класса. Для вызова конструктора предка следует использовать конструкцию вида `super(СПИСОК_АРГУМЕНТОВ)`, а для обращения к методам предка – `super.ИМЯ_МЕТОДА(СПИСОК_АРГУМЕНТОВ)`.

  Ключевое слово `this` описывает ссылку на экземпляр того объекта, чей код выполняется. **Конструктор** `this` можно использовать для вызова конструктора этого же класса с другим списком аргументов.

2.3.2 Класс Object

Фактически все классы Java порождены от класса `Object`. Когда определяется класс, не являющийся наследником какого-либо иного класса, в этом определении неявно присутствует оператор `extends Object`. Следующие два определения класса эквивалентны:

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

```
class Point extends Object{
   int x;
   int y;
```

`}`	`}`

Набора данных в классе `Object` нет. Методы `Object` наследуются всеми классами Java. Методы класса `Object` (они наследуются всеми классами Java и могут быть перекрыты) приведены в таблице 2.1:

Таблица 2.1 – Методы класса `Object`

Метод	Описание
`getClass()`	Основной идентификатор типа на этапе выполнения; идентифицирует класс любого произвольного объекта.
`hashCode()`	Вычисляет хэш-код экземпляра класса, при использовании в хэш-таблицах возвращает псевдоуникальное значение любого объекта.
`equals(Object obj)`	Используется для проверки равенства двух объектов. Сущность метода различна для разных классов. Обычно он возвращает `true`, если имеются две ссылки на один объект. Класс `String` возвращает `true` и в том случае, если символы двух строк совпадают.
`clone()`	Создает новый объект, являющийся копией исходного.
`toString()`	Преобразует состояние объекта в строковое представление.
`notify(), notifyAll(), wait()`	Используется для синхронизации программных потоков.
`finalize()`	Реализует деструктор, вызывается при разрушении экземпляра класса.

2.3.3 Создание и удаление объектов в Java

В Java отсутствуют указатели и используются только ссылки. Оператор `new` создаёт новый объект и возвращает **ссылку на него.** При этом выполняется:

- Новый объект динамически размещается в памяти; для всех его полей устанавливаются **по умолчанию стандартные значения** (переменные экземпляра класса, содержащие ссылки, получают значение `null`, поля базовых типов получают значение 0, логические поля получают значение `false`).
- Выбирается версия конструктора, соответствующая переданному набору аргументов, и выполняется его тело.
- Возвращается ссылка на созданный объект.

Если выражение имеет форму операции присваивания, то созданный объект приписывается объекту-переменной того же типа:

```
<variable_name> = new <class_type> ([param, param, …]);
```

где:

<variable_name> – идентификатор, определяющий объект-переменную, т.е. **ссылку** на созданный объект;
<class_type> описывает тип класса и одновременно обозначает вызов конструктора, инициализирующего создание объекта.

Пользователю в Java никогда не приходится уничтожать объект или освобождать память. Когда всем переменным, хранящим ссылки на объект, присваиваются другие значения или когда они выходят из области определения, объект помечается виртуальной машиной Java как неиспользуемый, и система асинхронно ликвидирует все такие объекты. Этот процесс называется **сбором мусора**.

Когда сборщик мусора принимается за удаление объекта, вызывается деструктор, содержащий код, запускаемый непосредственно перед удалением объекта из памяти. У всех деструкторов в Java одно имя – `finalize()` (метод наследуется из класса `Object`).

2.3.4 Числовые объекты

В Java для каждого числового типа используются специальные классы для каждого числового типа – **числовые классы**:

Таблица 2.2 – Классы-оболочки числовых типов

Базовый тип	Имя класса	Пример создания	Пример доступа
`byte`	`Byte`	`Byte n=new Byte((byte)34)`	`n.byteValue()`
`short`	`Short`	`Short n=new Short ((short)100)`	`n.shortValue()`
`int`	`Integer`	`Integer n=new Integer ((integer)1045)`	`n.intValue()`
`long`	`Long`	`Long n=new Long ((long)10849L)`	`n.longValue()`
`float`	`Float`	`Float n=new Float((float)3.934F)`	`n.floatValue()`
`double`	`Double`	`Double n=new Double((double)3.934)`	`n.doubleValue()`

В любом числовом классе существует статический метод `toString()`, который принимает в качестве аргумента данные базового типа соответствующего числового класса и возвращает его представление в виде строки. Например, метод `Integer.toString(i)` возвращает строковое представление целого числа `i`, а метод `Double.toString(d)` возвращает строковое представление вещественного числа `d`. Аналогичные действия выполняет метод `valueOf()` класса `String`, который принимает в качестве аргумента значение переменной некоторого базового типа и преобразует его в строковое представление (например, `String.valueOf(i)` – возвращает строковое представление целого числа `i`, а `String.valueOf(d)` – возвращает строковое представление вещественного числа `d`).

Для преобразования строки в соответствующий базовый тип, каждый числовой класс Java содержит метод `valueOf()`, который принимает в качестве аргумента строку и преобразует ее **в объект** соответствующего числового класса. Например, `Integer.valueOf("12")` возвращает **объект** класса-оболочки `Integer`, соответствующий целому числу 12, а `Double.valueOf("3.141592")` возвращает **объект** класса-оболочки `Double`, соответствующий вещественному числу 3.141592. Для преобразования полученного экземпляра класса-оболочки в базовый тип необходимо дополнительно вызвать соответствующий метод: `intValue()` – для целого (`Integer.valueOf("12").intValue()`), `doubleValue()` – для вещественного (`Double.valueOf("3.141592").doubleValue()`).

Класс Math, находящийся в пакете `java.lang`, содержит основные математические функции и константы:

Таблица 2.3 – Математические функции и константы класса `Math`

Функция	Назначение
`double E`	Число типа double, ближайшее к e, основанию натурального логарифма
`double PI`	Число типа double, ближайшее к числу π
`<number> abs (<number> x)`	Абсолютное значение x, при условии, что x не является наименьшим отрицательным целым числом
`double acos (double x)`	Арккосинус значения x, выраженный в радианах
`double asin (double x)`	Арксинус значения x, выраженный в радианах
`double atan (double x)`	Арктангенс значения x, выраженный в радианах
`double cos (double a)`	Косинус угла a
`double exp(double x)`	Экспоненциальная функция
`double log (double x)`	Натуральный логарифм по основанию e
`<number> max (<number> x, <number> y)`	Большее из x и y
`<number> min (<number> x, <number> y)`	Меньшее из x и y
`double pow (double x, double y)`	Возведение x в степень y
`double random ()`	Псевдослучайное число из интервала [0.0, 1.0]
`double round (double x)`	Целое число типа long, ближайшее к x
`double sin (double a)`	Синус угла a
`double sqrt (double x)`	Значение квадратного корня
`double tan (double a)`	Тангенс угла a

В Java встроены два числовых класса `java.math.BigDecimal` и `java.math.BigInteger`, позволяющие работать с числами, превосходящими стандартную размерность числовых типов.

2.3.5 Строковые объекты

В Java для представления строк используются классы `String` (класс объектов с неизменяемой строкой) и `StringBuffer` (класс объектов с изменяемой строкой). Соответственно, строковые операции могут быть двух видов: для модификации строк и для получения информации о строке без её изменения.

В качестве алфавита, используемого для описания строк, выступает международный алфавит Unicode с 16-битовой кодировкой.

Набор **основных операций**, выполняемых над неизменяемой строкой (в качестве примера рассмотрим объект-строку `S`, проинициализированную значением `"abcdefghijklmnop"`):

Таблица 2.4 – Основные методы класса `String`

Операция	Действие	Результат
`length()`	Возвращает длину строки S	**16**
`charAt(i)`	Возвращает символ, имеющий индекс i из строки S	При i =5: ‘**f**’
`startsWith(Q)`	Определяет, является ли Q префиксом строки S	При Q =”abcd”: **true**
`endsWith(Q)`	Определяет, является ли Q суффиксом строки S	При Q =”javapop”: **false**
`substring(i, j)`	Возвращает подстроку S[i,j]	При i=4, j=9 : ”**efghij**”
`concat(Q)`	Возвращает объединение S + Q	При Q =”qrs”: ”**abcdefghijklmnopqrs**”
`equals(Q)`	Определяет равенство Q и S	При Q = ”abcdefghijklmnop”: **true**
`indexOf(Q)`	Если Q является подстрокой строки S, то возвращает индекс начального символа первого появления Q в S, в противном случае возвращает – -1	При Q =”ghi”: **6**

Таблица 2.5 – Основные методы класса `StringBuffer`

Операция	Действие	Результат
`append(Q)`	Возвращает S+Q, замещая S на S+Q	При Q =”qrs”: ”**abcdefghijklmnopqrs**”:
`insert(i, Q)`	Возвращает S с введенной в нее, начиная с индекса i, подстрокой Q	При i=3, Q =”xyz”: ”**abcxyzdefghijklmnopqrs**”:

`reverse()`	Выстраивает строку S в обратном порядке и возвращает ее	“**srqponmlkjihgfedzyxcba**”
`setCharAt(i,ch)`	Устанавливает символу с индексом i в строке S значение ch	При i=7, Q =”W”: ” **srqponmWlkjihgfedzyxcba** ”:
`charAt(i)`	Возвращает символ с индексом i из строки S	

Большинство методов класса `String` не всегда сразу доступны объекту `S` класса `StringBuffer`. Метод `toString()` класса `StringBuffer` позволяет из экземпляра своего класса получить экземпляр класса `String`, а затем уже для него вызывать требуемый метод. Например:

```
sb.toString().length();
```

2.4 Организация циклов

В Java циклы организуются аналогично языку C++ (цикл-итератор `for`, цикл с предусловием `while`, цикл с постусловием `do while`). В то же время, для ряда случаев (в частности, перебора элементов массива) используется новая конструкция:

```
for (<ИМЯ_КЛАССА> <ИМЯ_ЛОКАЛЬНОЙ_ПЕРЕМЕННОЙ> : <ИМЯ_МАССИВА>) {
      ...
}
```

Ниже представлены различные способы записи конструкций циклов:

```
// Вычисление суммы элементов массива
int[] numbers = new int[] {1, 2, 3, 4, 5};
int sum = 0;
// Классическая конструкция цикла
for (int i=0; i<numbers.length; i++) {
      sum += i;
}
sum = 0;
// Новая конструкция цикла
for (int i: numbers) {
      sum += i;
}
```

2.5 Стандартный ввод и вывод

Стандартный вывод в Java осуществляется через консоль с помощью статического объекта `System.out`, являющегося экземпляром класса `java.io.PrintStream`. Данный класс определяет методы, осуществляющие буферизованный вывод.

Метод `System.out.print()` выводит текст в режиме консоли. Метод `System.out.println()` отличается от него автоматическим переходом на

новую строку после вывода. Метод `System.out.flush()` выводит содержимое буфера. Например, фрагмент кода:

```
System.out.print (“Java values:”);
System.out.print (3.1415);
System.out.print (“,  ”);
System.out.print (15);
System.out.println (“(double, char, int).”);
```

отображает на консоли выходной поток:

```
Java values: 3.1415,15 (double, char, int).
```

Для ввода информации с консоли используется объект `System.in`, являющийся экземпляром класса `java.io.InputStream`, обеспечивающим байтовый ввод (возможность чтения данных из потока отдельными байтами или их группами). Ввод символьной информации обеспечивает класс `java.io.InputStreamReader`, но экземпляр этого класса может одновременно считывать только по одному символу, что весьма неудобно. Класс `java.io.BufferedReader` использует буферизованный ввод и позволяет вводить строку. Он определяет методы:

- `int read()` – возвращает один символ из исходного потока;
- `String readLine()` – возвращает строку текста, заканчивающуюся символом перевода строки, который обрабатывается, но не включается в возвращаемую строку.

Пример фрагмента кода, с входным и выходным потоками:

```
java.io.BufferedReader  stndin;
String line;
double sum, d=0.0;
int i=0;
// Выполняется последовательное обёртывание байтового потока чтения
// сначала в символьный поток чтения, а затем в буферизованный символьный
// поток чтения для чтения данных сразу строками
stndin=new java.io.BufferedReader(new java.io.InputStreamReader(System.in));
System.out.print(“Input a double: “);
System.out.flush();
if ((line = stndin.readLine() !=null)
      d=Double.valueOf(line).doubleValue();
System.out.print(“Input an int: “);
System.out.flush();
if ((line = stndin.readLine() !=null)
      i=Integer.valueOf(line).intValue();
sum=d+i;
System.out.println(“Sum is“ + sum +”.”);
```

После выполнения кода имеем:

```
Input a double:  6.1078
```

```
Input an int: 209
Sum is 215.1078.
```

3 Пример приложения

Задание: составить программу завтрака на основе списка продуктов, передаваемых в качестве параметров в командной строке. Завтрак может включать не более 20 наименований продуктов. Включение продуктов в завтрак осуществляется путём добавления имени соответствующего класса в параметры командной строки (например, *Apple*, если класс яблока называется `Apple`, *Tea*, если класс чая называется `Tea`). Каждый из продуктов может иметь (но не обязательно) дополнительные параметры (например, для яблока это величина), которые должны указываться после имени класса через наклонную черту (например, *Apple/большое*, *Tea/зелёный*, *Sandwich/сыр/ветчина*). Употребление каждого из продуктов должно сопровождаться выводом соответствующего сообщения на экран.

Например, для параметров командной строки: *"Apple/малое" "Apple/большое" "Cola/диетическая" "Sandwich/сыр/ветчина"* приложение должно отобразить текст:

```
Яблоко размера 'МАЛОЕ' съедено
Яблоко размера 'БОЛЬШОЕ' съедено
Кока-кола типа 'ДИЕТИЧЕСКАЯ' выпита
Бутерброд с СЫР и ВЕТЧИНА съеден
Всего хорошего!
```

3.1 Структура приложения

Структура разрабатываемого приложения включает:

- главный класс приложения (класс, содержащий метод `main()`), обрабатывающий входные аргументы программы; определяющий набор продуктов завтрака; реализующий процедуру их употребления;
- иерархию классов продуктов, используемых главным классом.

Заметим, что все продукты имеют в основе нечто общее – они относятся к классу «Еда» и могут быть употреблены в пищу. В Java существуют два способа выделения подобной общности поведения – использование механизма наследования от базового класса, и реализация интерфейсов, задающих некоторые черты поведения. Это позволяет спроектировать иерархию классов тремя различными способами:

Способ 1 (рисунок 3.1): использует только механизм наследования, без объявления и реализации интерфейсов.

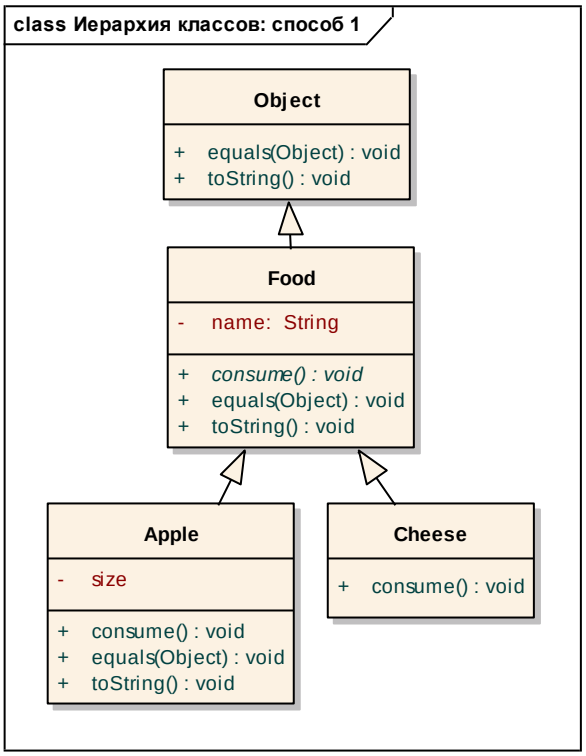


Рисунок 3.1 – Иерархия классов на основе наследования

В основе иерархии находится абстрактный (создание его экземпляров невозможно) класс `Food` (Еда), являющийся (неявно) потомком стандартного класса `Object`. В `Food` объявлено внутреннее поле данных `name` типа `String` (содержит название продукта) и абстрактный метод `consume()` (отвечает за употребление продукта), который в классе только объявляется (на рисунке абстрактность метода показана курсивным начертанием его названия). Задача реализации `consume()` возложена на потомков класса `Food`. Унаследованные от класса `Object` методы `toString()` и `equals()`в `Food` должны быть переопределены (для обработки поля `name`).

Класс `Cheese` (Сыр) является потомком `Food`. Он не определяет новых полей данных, поэтому переопределение методов `equals()` и `toString()` не требуется. Таким образом, в классе `Cheese` необходимо только реализовать метод употребления продукта в пищу `consume()`.

Класс `Apple` (Яблоко) также является потомком `Food`, но определяет дополнительное поле данных `size` (размер) типа `String`. Добавление нового поля вынуждает переопределить методы `equals()` и `toString()` и реализовать метод употребления яблока в пищу – `consume()`.

Способ 2 (рисунок 3.2): использует только механизм интерфейсов, без механизма наследования.

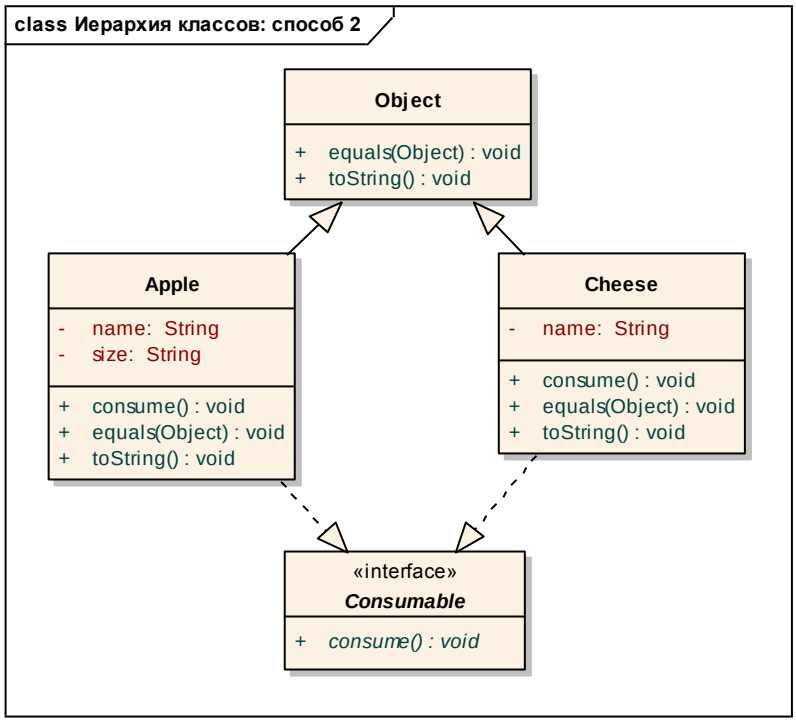


Рисунок 3.2 – Иерархия классов на основе интерфейсов

В этом случае общая качественная характеристика «употребляемость» выделена в понятие интерфейса `Consumable` (Употребляемый), который будет использоваться классами и который содержит абстрактный метод `consume()`. Классы `Apple` и `Cheese` не имеют явного общего предка (неявно их предком является `Object`). Так как интерфейс не содержит содержать полей экземпляров класса, то в оба класса необходимо независимо добавить поле данных `name` (вместе с селекторами и модификаторами), т.е. вместо наследования его от предка имеет место двойное объявление (повторение кода). Кроме этого, в обоих классах необходимо:

- переопределить методы `equals()` и `toString()`, унаследованные от `Object` для учёта внутренних полей данных;
- реализовать метод `consume()`, наличие которого предопределено использованием классами интерфейса `Consumable`.

Очевидно, что необходимость дублирования одинаковых полей в различных классах, а также увеличенный объём кода, который необходимо написать, делает этот проект иерархии неудачным.

Способ 3 (рисунок 3.3): предполагает использование, как наследования, так и интерфейсов.

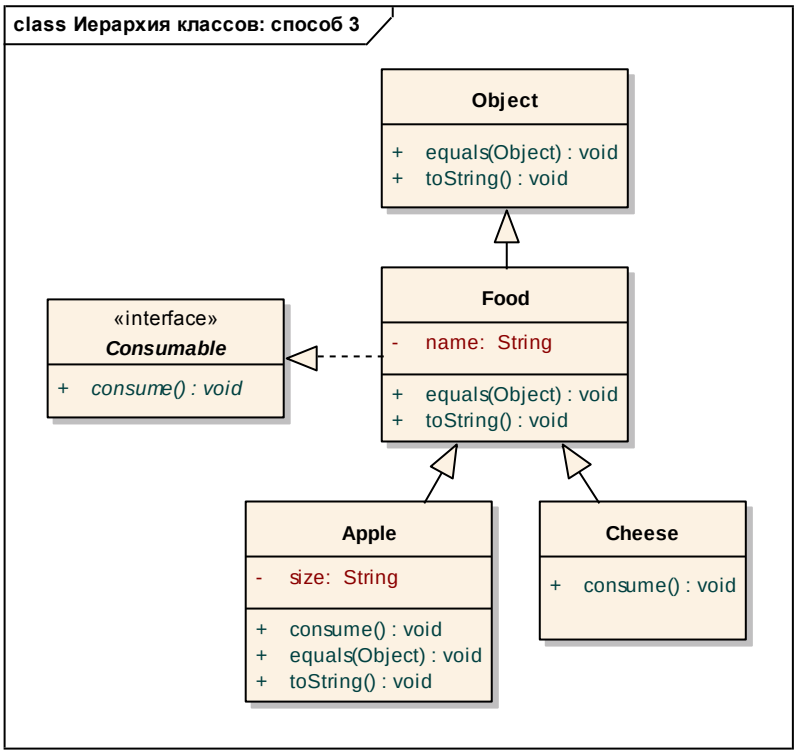


Рисунок 3.3 – Иерархия классов на основе наследования и интерфейсов

В основе иерархии в этом случае находится класс `Food`, который объявляет поле данных `name`, а также переопределяет унаследованные от класса `Object` методы `equals()` и `toString()` для учёта нового поля `name`. Так как качество «употребляемость» характерно не только для еды (можно употребить и одежду, и косметику и т.д.), то оно вынесено в интерфейс `Consumable`, реализуемый классом `Food`. Абстрактный метод `consume()`, определённый в интерфейсе `Consumable`, может быть реализован как классом `Food`, так и его потомками (если `Food` является абстрактным классом, как в данном случае).

Класс `Cheese` не добавляет новых полей данных, то ему нет необходимости переопределять методы `equals()` и `toString()`, а нужно только реализовать метод `consume()`.

Класс `Apple` добавляет поле `size`, поэтому должен реализовать метод `consume()` и переопределить методы `equals()` и `toString()`.

Данный способ построения иерархии классов является предпочтительным, так как, с одной стороны, обеспечивает совместное использование общих полей данных (с помощью механизма наследования), а, с другой стороны, выделяет качество «употребляемость» в абстракцию, которая может быть использована и в иерархии других классов (например, косметики или одежды). Именно этот способ и будет выбран для реализации.

3.2 Создание проекта

Для создания нового проекта в Eclipse необходимо активировать пункт меню «*File* → *New* → *Project*» (рисунок 3.4). Из списка возможных типов проектов в разделе «*Java*» следует выбрать «*Java Project*» (рисунок 3.5). В появившемся диалоговом окне (рисунок 3.6) указать название проекта, например «Задание 1 – Консольное приложение». В группе «*Contents*» оставить включенной радио-кнопку «*Create new project in workspace*» (создание нового проекта в директории, указанной как рабочее место). В группе «*JRE*» оставить включенной радио-кнопку «*Use default JRE*» (в качестве рабочего окружения Java использовать рабочее окружение по умолчанию). В группе «*Project layout*» оставить включенной радио-кнопку «*Create separate folders…*» (для хранения исходных файлов и откомпилированных классов будут использоваться отдельные папки). Нажать «*Finish*».

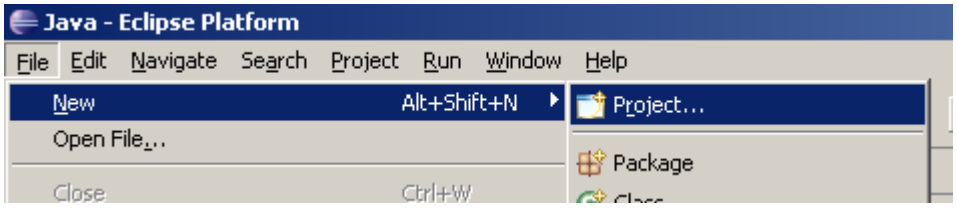


Рисунок 3.4 – Создание нового проекта

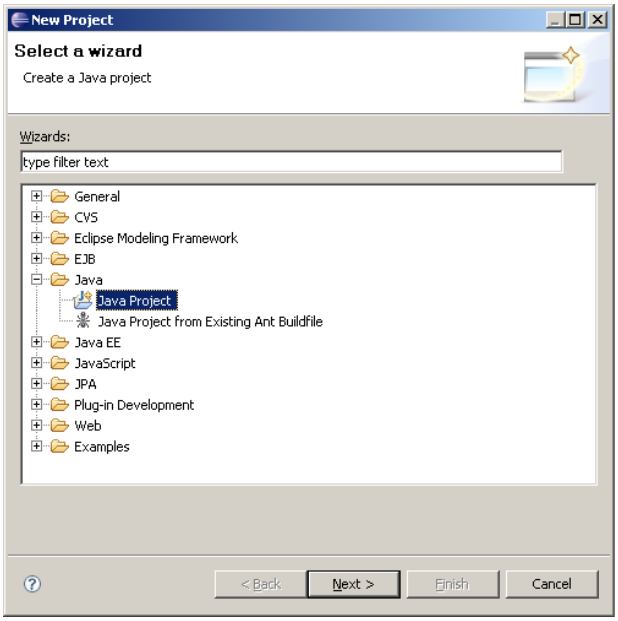


Рисунок 3.5 – Указание типа создаваемого проекта

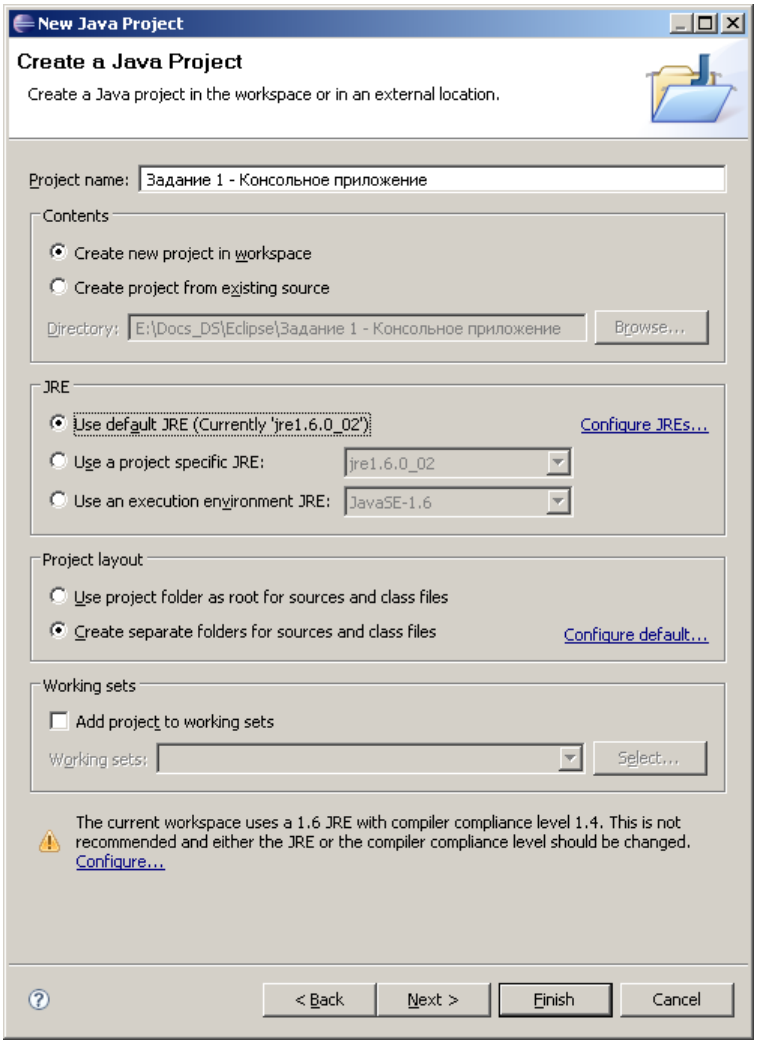


Рисунок 3.6 – Диалоговое окно создания нового проекта

В результате описанных действий будет создан пустой проект. Его содержание в виде *Package Explorer* представлено на рисунке 3.7.

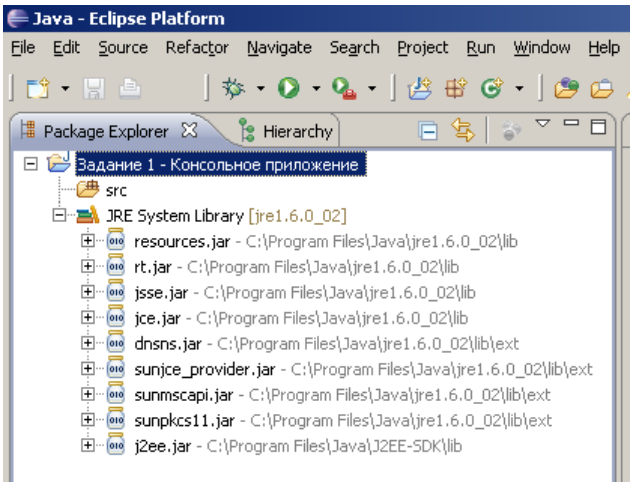


Рисунок 3.7 – Представление нового проекта в виде *Package Explorer*

3.3 Создание пакета для классов

Как уже отмечалось, все приложения Java представлены набором взаимодействующих классов, принадлежащих определённым пакетам. Хорошим стилем считается явное указание названий пакетов. Если имя пакета в явном виде не задано, используется безымянный «пакет по умолчанию».

Для создания в проекте нового пакета необходимо щёлкнуть правой кнопкой мыши на пиктограмме папки с исходными файлами «*src*» и в контекстном меню (рисунок 3.8) выбрать «*New* → *Package*».

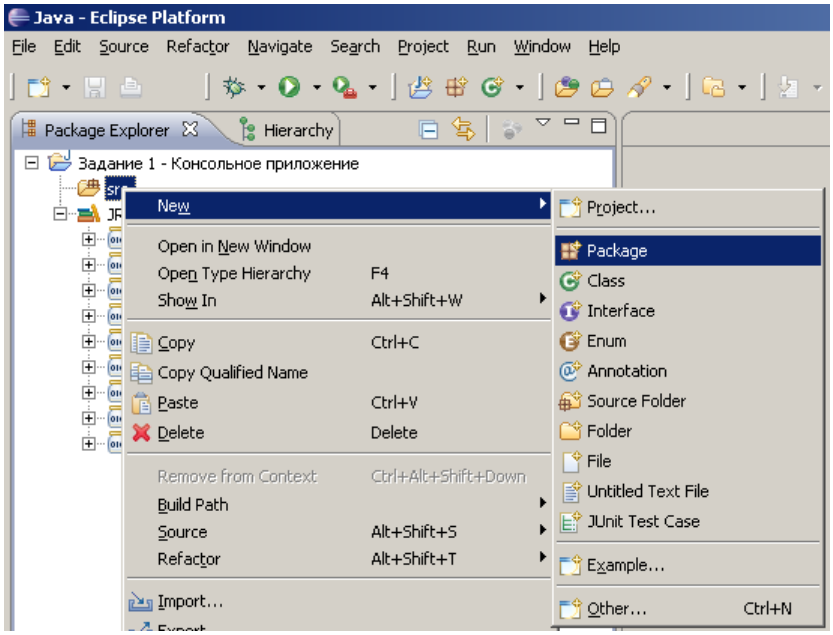


Рисунок 3.8 – Создание в проекте нового пакета

В появившемся диалоговом окне (рисунок 3.9) необходимо указать имя создаваемого пакета. Пакеты должны именоваться способом, облегчающим осознание их принадлежности и уровня общности. Предлагается следующая схема именования:

`bsu.rfe.java.group`*X*`.lab`*Y*`.`*NAME*`.var`*Z*,

где *X* – номер группы студента,

Y – номер лабораторной работы,

NAME – фамилия студента в английской транскрипции,

Z – номер варианта задания.

Например:

```
bsu.rfe.java.group7.lab1.Ivanov.varB4
```

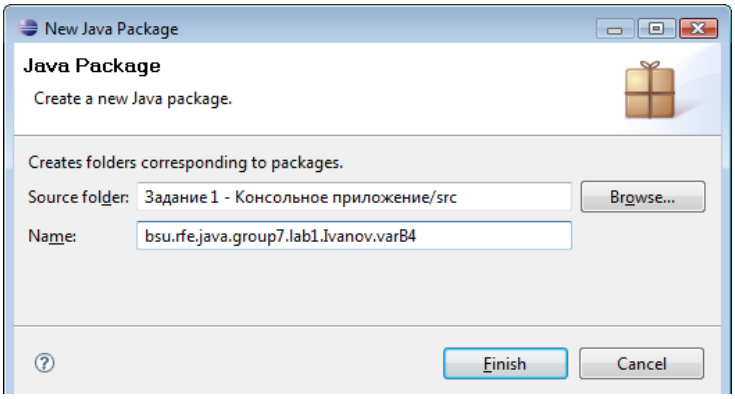


Рисунок 3.9 – Диалоговое окно создания нового пакета

3.4 Создание каркаса главного класса приложения

Напомним, что в Java главный (исполняемый) класс содержит статический общедоступный метод `main()`, получающий в качестве аргумента массив строк (соответствующий параметрам командной строки). Для создания главного класса приложения следует щёлкнуть правой кнопкой мыши на имени пакета, в котором необходимо создать класс, и в контекстном меню выбрать «*New* → *Class*» (рисунок 3.10).

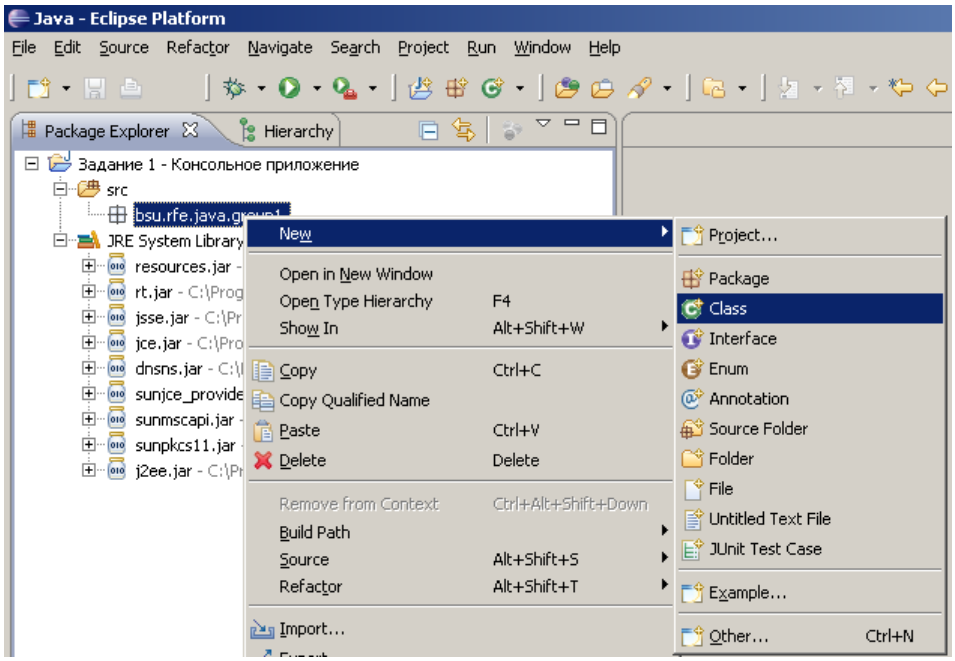


Рисунок 3.10 – Добавление в пакет нового класса

В появившемся диалоговом окне (рисунок 3.11) необходимо указать:

- имя создаваемого класса (в поле «*Name*»);
- тип доступа к классу (*public*, *default*, *private*, *protected*);
- дополнительные характеристики класса (*abstract*, *final*, *static*);
- класс-предок (поле «*Superclass*»);

- реализуемые классом интерфейсы (в поле «*Interfaces*»);
- необходимость создания метода main (соответствующий флажок под «*Interfaces*»);
- необходимость автоматической генерации прототипов унаследованных от потомка конструкторов;
- необходимость автоматической генерации прототипов унаследованных абстрактных методов;
- необходимость генерации комментариев.

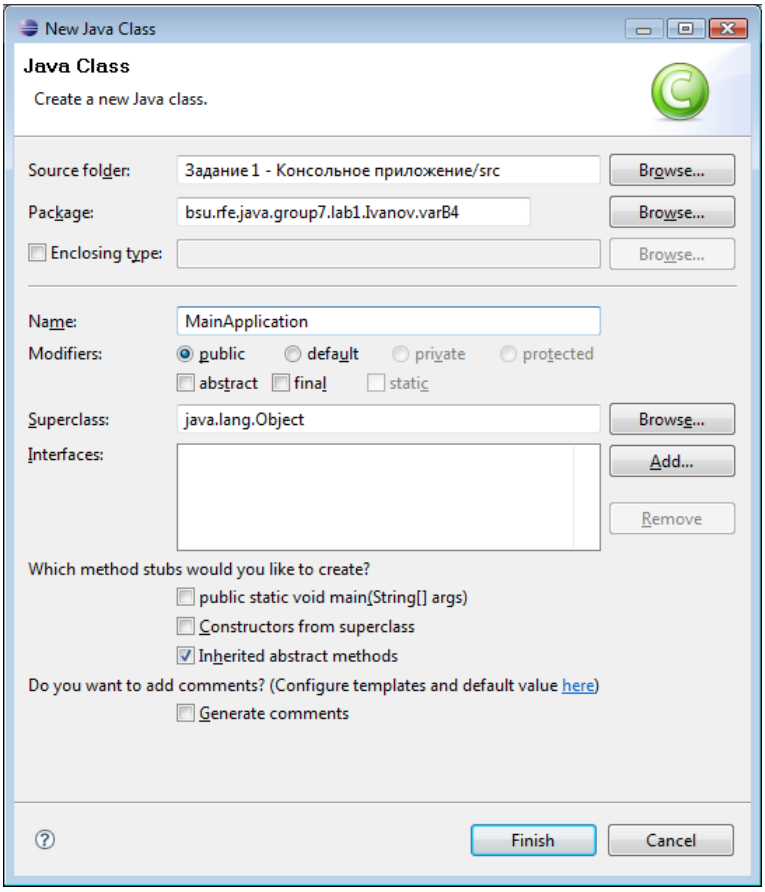


Рисунок 3.11 – Диалоговое окно создания нового класса

После нажатия кнопки «*Finish*» в папке, соответствующей выбранному пакету (в случае рисунка 3.11 – *bsu/rfe/java/group7/lab1/Ivanov/varB4*), будет создан новый файл с именем «*ИМЯ_КЛАССА*.java» (в случае рисунка 3.11 – *MainApplication.java*), и экран примет вид схожий с приведенным на рисунке 3.12. В качестве имени главного класса можно, например, выбрать `Breakfast`.

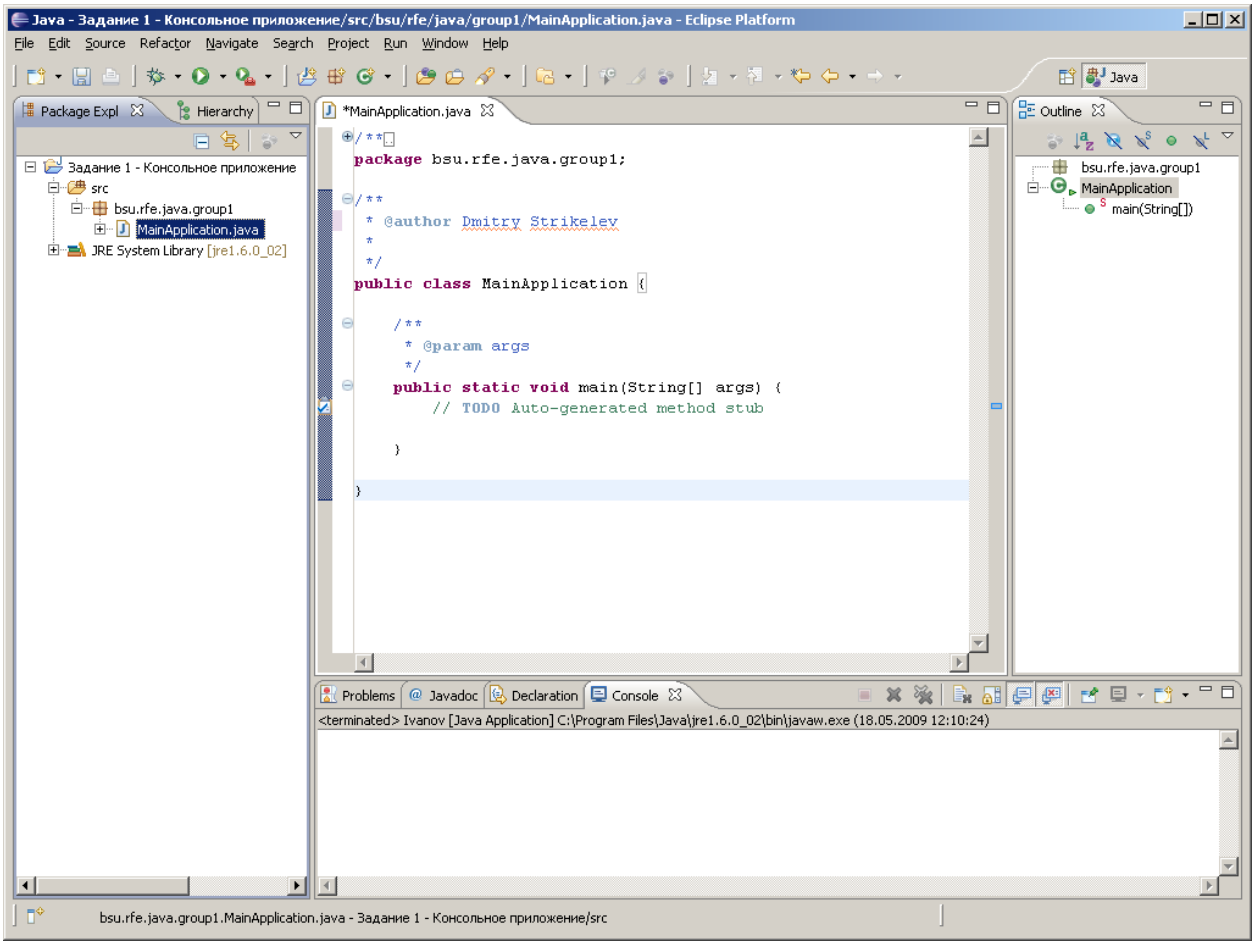


Рисунок 3.12 – Содержание файла созданного класса

3.5 Создание общего интерфейса, используемого в иерархии классов

Оставим на некоторое время главный класс нашего приложения и рассмотрим множество классов, представляющих продукты, входящие в завтрак. Все они обладают общим сходством – могут быть употреблены, но механизм их употребления различается: яблоко – съедается, чай – выпивается и т.д. Это сходство может быть выражено в свойстве «быть употреблённым», реализуемом посредством метода «употребить» (consume). Средством выделения общности поведения объектов в Java являются интерфейсы. Создадим интерфейс `Consumable`, обладающий методом `consume()`. Для этого необходимо щёлкнуть на имени пакета, в который добавляется интерфейс, и в контекстном меню выбрать «*New* → *Interface*». В появившемся диалоговом окне (рисунок 3.13) необходимо указать имя интерфейса, тип доступа к нему (*public*, *default*, *private*, *protected*), а также задать расширяемый им набор интерфейсов (базовые интерфейсы).

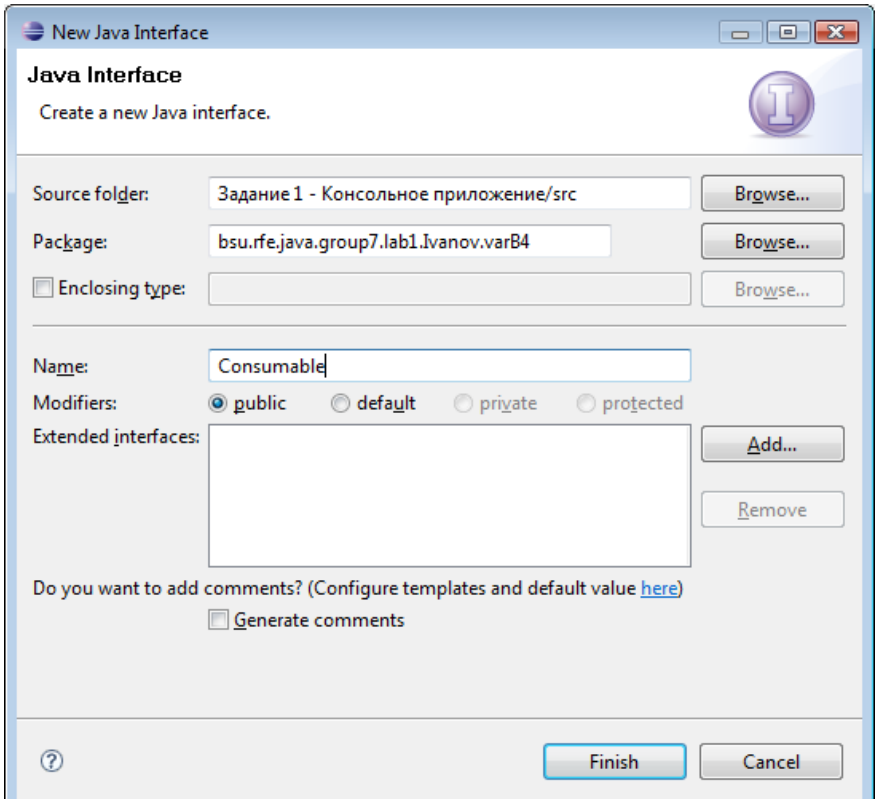


Рисунок 3.13 – Диалоговое окно создания нового интерфейса

После нажатия кнопки «*Finish*» в папке, соответствующей выбранному пакету (в случае рисунка 3.13 – *bsu/rfe/java/group7/lab1/Ivanov/varB4*), будет создан новый файл с именем «*Consumable.java*».

В созданном интерфейсе определим метод `consume()`, не имеющий параметров и не возвращающий никаких значений. Исходный код интерфейса весьма прост:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public interface Consumable {
      public abstract void consume();

}
```

3.6 Реализация базового класса иерархии

Реализацию базового класса иерархии начнём с создания его каркаса. Создадим базовый класс иерархии продуктов, которые могут быть включены в завтрак, аналогично главному классу приложения, но с некоторыми отличиями (рисунок 3.14): классу не требуется метод `main`; класс является абстрактным (создание его экземпляров недопустимо, т.к. включение в состав завтрака некоторой «еды в общем смысле» невозможно); класс реализует интерфейс `Consumable`.

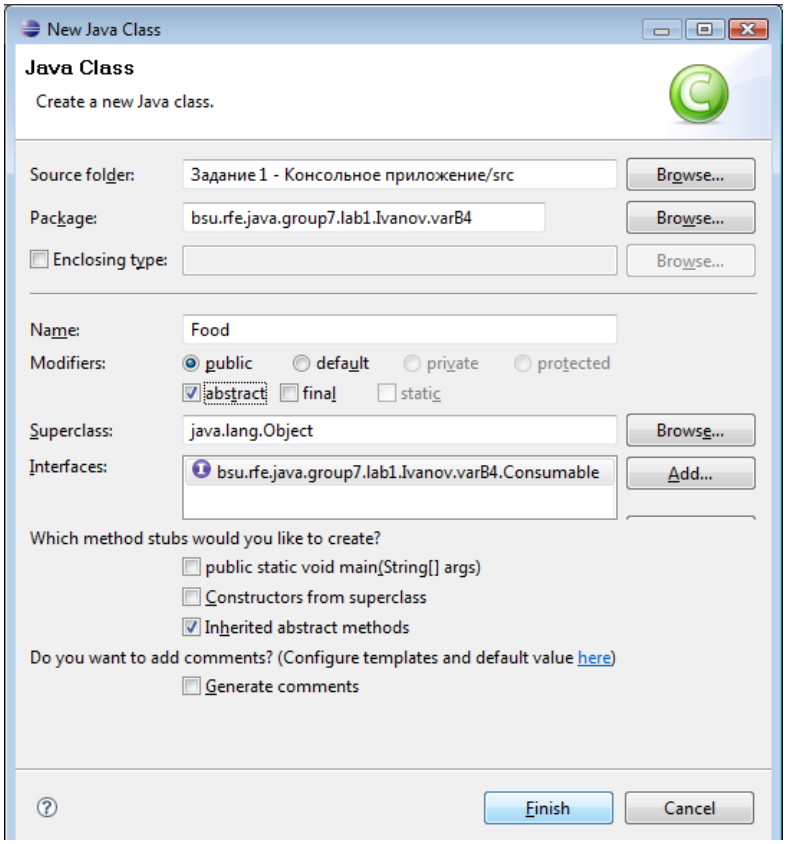


Рисунок 3.14 – Создание абстрактного класса, реализующего интерфейс

Созданный класс будет содержать следующий исходный код:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public abstract class Food implements Consumable {

	@Override
	public void consume() {
		// TODO Auto-generated method stub

	}

}
```

Так как класс `Food` реализует интерфейс `Consumable`, то в полученном коде для метода `consume()` уже присутствует «заглушка». Но поскольку класс является абстрактным, реализация этого метода в нём не является обязательной и может быть возложена на потомков (на данном этапе метод из класса можно удалить).

Исходные коды каркасов всех классов приложения, сгенерированные средой Eclipse автоматически, размещены в Приложении 1.

Теперь перейдём к реализации базового класса `Food`. Для представления названия продукта, включаемого в завтрак, добавим в класс внутреннее поле `name`, имеющее тип `String`.

Напомним, что свойство инкапсуляции требует сокрытия полей данных класса от прямого доступа извне и предоставления специальных методов для опосредованной работы с ними: метода чтения, называемого также «аксессором», «селектором», «геттером», и метода изменения, называемого также «модификатором», «мутатором», «сеттером». В среде Eclipse эти методы можно сгенерировать автоматически.

Выполним автоматическую генерацию кода с помощью средств Eclipse. Для этого активируем пункт меню «*Source* → *Generate Getters and Setters*». В появившемся диалоговом окне (рисунок 3.15) пометим флажком поля данных, для которых необходимо сгенерировать сеттеры и геттеры (в нашем случае – `name`); укажем, в каком месте описания класса разместить сгенерированный код (в списке «*Insertion point*»); укажем порядок сортировки (в списке «*Sort by*»), а также тип доступа к создаваемым методам и их дополнительные характеристики. С учётом сгенерированных методов исходный код класса примет вид:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;
public abstract class Food implements Consumable {

	String name = null;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

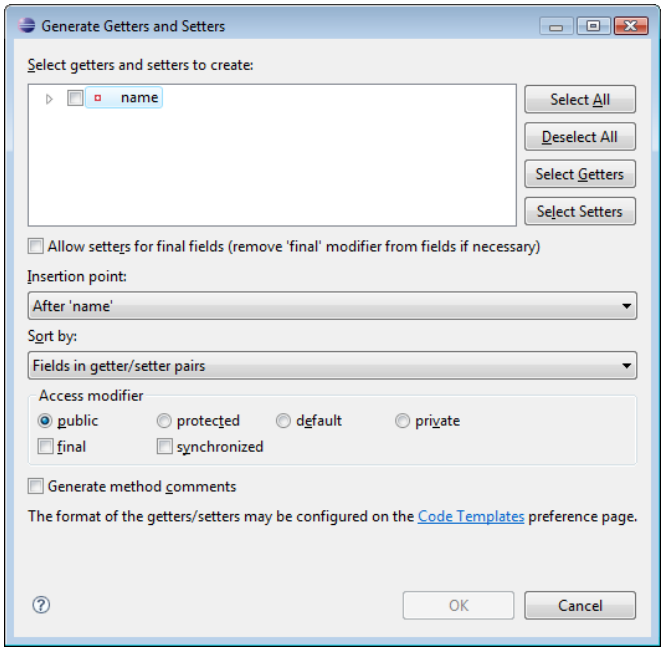


Рисунок 3.15 – Диалоговое окно добавления сеттеров и геттеров

Обратите внимание, что для различения внутреннего поля `name` данных класса и входного аргумента `name` метода `setName()`, при доступе к полю данных класса, используется ссылка `this`.

Замечание: В отличие от языка C++, `this` не указатель, а ссылка (как и все объектные типы в Java), и для доступа к членам объекта используется оператор . (точка).

Определим для класса `Food` конструктор инициализации, получающий в качестве аргумента строку с названием продукта:

```
public Food(String name) {
        this.name = name;

}
```

Переопределим в классе `Food` методы `toString()` и `equals(Object arg0)`, унаследованные от базового класса `Object`. Для этого активируем пункт меню «*Source* → *Override/Implement Methods*» и в появившемся диалоговом окне (рисунок 3.16) отметим флажками соответствующие методы, а также зададим место вставки нового кода.

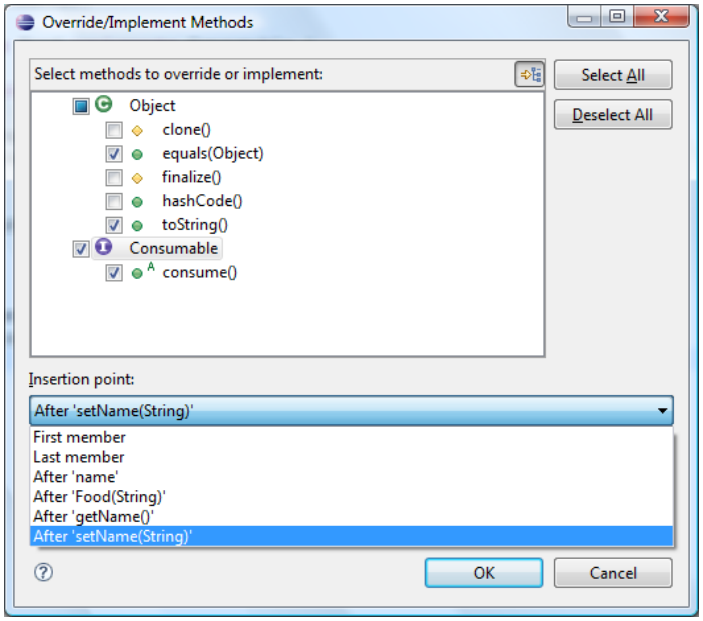


Рисунок 3.16 – Диалоговое окно перегрузки/реализации методов

Метод `toString()` возвращает значение внутреннего поля данных:

```
public String toString() {
        return name;
}
```

Код метода `equals()` выглядит так:

```
public boolean equals(Object arg0) {
      if (!(arg0 instanceof Food)) return false;      // Шаг 1
      if (name==null || ((Food)arg0).name==null) return false; // Шаг 2
```

```
            return name.equals(((Food)arg0).name); // Шаг 3
      }
```

Как видим, проверка равенства объектов является более сложным процессом и может быть реализована как последовательность шагов:

Шаг 1 позволяет с помощью оператора `instanceof` определить, является ли аргумент `arg0` экземпляром класса `Food` или какого-либо из его потомков (т.к. аргументом метода `equals()` является ссылка на самый общий тип `Object`). Этим самым проверяется совместимость классов объектов.

Шаг 2 проверкой равенства внутреннего поля `name` (являющегося ссылкой) константе `null` позволяет определить, полностью ли сконструированы наш объект (равенство которого проверяется) и объект-аргумент (равенство с которым проверяется).

При этом, так как для объекта класса `Object` внутреннее поле `name` не определено, а на шаге 1 мы убедились, что `arg0` есть экземпляр класса `Food` или какого-либо из его потомков, то необходимо привести его к типу `Food`.

Шаг 3 проверкой равенства поля `name` у обоих объектов определяет результат всей операции (равенство или неравенство объектов).

Окончательный код реализованного класса `Food`:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public abstract class Food implements Consumable {

      String name = null;

      public Food(String name) {
            this.name = name;
      }

      public boolean equals(Object arg0) {
            if (!(arg0 instanceof Food)) return false;
            if (name==null || ((Food)arg0).name==null) return false;
            return name.equals(((Food)arg0).name);
      }

      public String toString() {
            return name;
      }

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

}
```

3.7 Реализация класса потомка

На основе реализованного базового класса `Food` определим класс-потомок `Apple` (яблоко), представляющий один из продуктов, которые могут быть включены в завтрак. Яблоко отличается от абстрактного продукта (`Food`) тем, что:

1) могут создаваться его экземпляры (класс не является абстрактным);

2) содержит дополнительное поле данных, характеризующее его размер (например, малое, среднее, большое);

3) может быть употреблено (необходима реализация метода `consume()`);

4) объекты считаются равными, только если равны их размеры (т.е. необходимо переопределить метод `equals()`);

5) строковое представление объекта должно включать и информацию о его размере (необходимо переопределить метод `toString()`).

Для создания каркаса класса `Apple` повторим шаги, выполнявшиеся для класса `Food`: щёлкнуть правой кнопкой мыши на имени пакета; активировать пункт меню «*New → Class*»; указать имя класса-предка `Food`; отметить флажок «*Constructors from superclass*» (так как необходимо переопределить конструктор класса); оставить включенный флажок «*Inherit abstract methods*» (так как необходимо реализовать неопределённый ранее метод consume). Исходный код заготовки созданного класса будет иметь следующий вид:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Apple extends Food {

	public Apple(String name) {
		super(name);
		// TODO Auto-generated constructor stub
	}

	@Override
	public void consume() {
		// TODO Auto-generated method stub

	}

}
```

Выполним реализацию класса. Добавим в класс новое поле данных `size`. Для простоты сделаем его типа `String`, т.е. величина яблока будет не количественной характеристикой (1,2,3), а качественной (малое, среднее, большое):

```
	private String size;
```

С помощью уже известной процедуры создадим для этого поля данных сеттер и геттер:

```
public String getSize() {
	return size;
}

public void setSize(String size) {
	this.size = size;
}
```

В сгенерированной (на основе описания класса-предка) заготовке конструктор класса получает аргумент `name`. В реализации класса `Apple` конструктору нет необходимости получать в качестве аргумента имя продукта, т.к. оно априорно известно (яблоко). В то же время при конструировании объекта необходимо задать величину яблока, для чего будем использовать аргумент `size`, а имя продукта (передаваемое в качестве аргумента конструктору предка) жёстко зададим в коде класса:

```
public Apple(String size) {
	super("Яблоко");
	this.size = size;

}
```

Определим метод `consume()`, отвечающий за употребление яблока в процессе завтрака. Метод отображает на экране строковое представление объекта, после чего добавляет строку «*съедено*»:

```
public void consume() {
	System.out.println(this + " съедено");
}
```

В методе `consume()` ссылка `this` используется в строковом контексте (вывода в стандартный поток вывода `out`), поэтому автоматически будет вызван метод `toString()` преобразования состояния объекта в строку (унаследованный от класса `Food`). Результат вывода:

```
Яблоко съедено
```

Для вывода размера съеденного во время завтрака яблока, переопределим метод `toString()`:

```
public String toString() {
	return super.toString() + " размера '" + size.toUpperCase() + "'";
}
```

В результате вывода получим строку (например) вида:

```
Яблоко размера 'БОЛЬШОЕ' съедено
```

При реализации метода `equals()` необходимо проверить, принадлежат ли сравниваемые продукты к одной иерархии классов (являются ли потомками класса `Food`), являются ли определёнными (поле `name` не равно `null`), принадлежат ли одной группе (совпадают ли их поля `name`). Но эти проверки уже выполняются в реализации метода `equals()`, унаследованной от класса-предка `Food`, поэтому первым шагом переопределённой версии метода `equals()` является вызов метода предка:

```
if (super.equals(arg0)) {
      ...
} else
      return false;
```

Данная конструкция означает, что если унаследованный метод `equals()` сообщил о равенстве объектов (было возвращено значение `true`), то имеет смысл продолжать сравнение объектов, анализируя дополнительные поля. В противном случае, очевидно, что объекты не равны.

Если метод `equals()` предка сообщил о равенстве объектов, то это по-прежнему не означает, что два объекта принадлежат одному классу `Apple`. Любой объект может установить значение поля `name` равным «яблоко», но при этом не являться экземпляром `Apple`, а, следовательно, не иметь внутреннего поля `size`. Поэтому вторым шагом является дополнительная проверка, является ли аргумент `arg0` экземпляром или потомком `Apple`:

```
if (!(arg0 instanceof Apple)) return false;
```

Если сравниваемые объекты действительно являются экземплярами или потомками `Apple`, то сравниваем значения их полей `size`, и, тем самым, определяем факт равенства:

```
return size.equals(((Apple)arg0).size);
```

Окончательная версия метода `equals()` имеет вид:

```
public boolean equals(Object arg0) {
      if (super.equals(arg0)) {
            if (!(arg0 instanceof Apple)) return false;
            return size.equals(((Apple)arg0).size);
      } else
            return false;

}
```

3.8 Реализация главного класса приложения

Задачами класса `MainApplication` являются анализ аргументов командной строки (в качестве них передаются продукты, включаемые в завтрак); построение массива объектов для указанных продуктов; проведение некоторых операций над этим массивом; организация процедуры завтрака.

Так как максимальное количество продуктов, входящих в завтрак, ограничено числом 20, то зарезервируем необходимое место в памяти для хранения 20 ссылок на объекты класса `Food` (или его потомков):

```
Food[] breakfast = new Food[20];
```

Внимание! Выделение памяти для хранения данных самих объектов при этом не происходит!

Проанализируем теперь аргументы командной строки (передаваемые в виде массива строк `String[] args`) и создадим для них экземпляров соответствующих классов с использованием цикла `for`. Приведём обе формы записи цикла: классическую и новую, характерную для Java:

```
// Классическая запись
String arg;
for (int i=0; i<args.length; i++) {
      arg = args[i];
      if (arg.startsWith("-")) { ... }
      ...
}
// Новая запись
for (String arg: args) {
      if (arg.startsWith("-")) { ... }

      ...
}
```

В общем случае, `i`-ый аргумент может содержать как название класса, так и дополнительные параметры его инициализации. Например: “*Cheese*” (экземпляр класса «`Cheese`» без параметров), “*Apple/малое*” (экземпляр класса «`Apple`» с одним параметром «малое»), “*Sandwich/сыр/ветчина*” (экземпляр класса «`Sandwich`» с двумя параметрами «сыр» и «ветчина»).

Замечание: каждый аргумент командной строки рекомендуется помещать в двойные кавычки, в противном случае, при наличии пробелов, он будет интерпретирован как несколько различных аргументов. Пример: аргумент *IceCream/co сливками и шоколадом* будет передан методу `main()` в виде массива `{“IceCream/co”, “сливками”, “и”, “шоколадом”}`, в то время как “*IceCream/co сливками и шоколадом*” будет передан как `{“IceCream/co сливками и шоколадом”}`.

При обработке `i`-го аргумента необходимо сначала разбить его на компоненты, используя в качестве разделителя символ «/»:

```
String[] parts = arg.split("/");
```

Дальнейшая обработка требует для каждого аргумента анализа имени класса и количества параметров, а затем создания экземпляра указанного класса с его занесением в массив `breakfast`. Эта операция может выполняться по-разному. Способ её реализации для заданий уровней

сложности A и B описан в разделе 3.8.1, а заданий уровня сложности C – в разделе 3.8.2. Сортировка массива (необходимая на уровнях сложности B, C) описана в разделе 3.8.3. Обработка объектов массива описана в разделе 3.8.4.

3.8.1 Создание экземпляров классов с помощью сравнения строк

Данный способ предполагает, что перечень реализованных классов продуктов известен, и выбор экземпляра класса осуществляется с помощью цепочки связанных блоков `if-else`:

```
if (parts[0].equals("Cheese")) {
    // мы знаем, что у Cheese дополнительных параметров нет
    breakfast[i] = new Cheese();
} else
if (parts[0].equals("Apple")) {
    // мы знаем, что у Apple один параметр, который находится в parts[1]
    breakfast[i] = new Apple(parts[1]);
} else

...
```

В этом случае, для каждого класса должен быть реализован соответствующий блок `if`.

3.8.2 Создание экземпляров классов с помощью Java Reflection

Java Reflection является средством, позволяющим Java-программам проводить самоанализ: получать сведения о конструкторах, методах, переменных, реализованных классами. Эти возможности предоставлены пакетом `java.lang.reflect` и классом `Class`. В нашем случае это позволяет автоматически создавать экземпляры классов, полагая, что имена классов указываются в начале каждого аргумента командной строки.

Первым шагом является получение экземпляра класса `Class`, соответствующего описанию класса, имя которого задано в аргументе командной строки (например, `Apple`):

```
Class myClass = Class.forName("bsu.rfe.java.teacher." + parts[0]);
```

Замечание: в качестве аргумента методу `forName()` необходимо передавать полное имя класса, включающее имя пакета.

Замечание: в общем случае, попытка обнаружения класса с заданным именем может закончиться неудачей – исключительной ситуацией типа `ClassNotFoundException`. Для учёта этого необходимо либо окружить рассматриваемый код блоком

```
try {
      ...
} catch (ClassNotFoundException e) {
      ...
}
```

либо добавить после списка аргументов метода `main()` сведения о том, что при выполнении метода может возникнуть исключительная ситуация:

```
public static void main(String[] args) throws ClassNotFoundException { ... }
```

Вторым шагом является получение доступа к конструктору, позволяющему создавать экземпляры класса. Для этого применяется метод Reflection API `getConstructor()`, которому необходимо указать количество и тип аргументов, принимаемых искомым конструктором. Для простоты все классы, представляющие продукты завтрака, будут иметь параметры типа `String`, а их число можно определить исходя из длины массива `parts`.

```
if (parts.length==1) {
      // дополнительных параметров нет
      Constructor constructor = myClass.getConstructor();
      Breakfast[i] = (Food)constructor.newInstance();
} else
if (parts.length==2) {
      // параметр в parts[1]
      Constructor constructor = myClass.getConstructor(String.class);
      breakfast[i] = (Food)constructor.newInstance(parts[1]);
}
```

Замечание: в общем случае, попытка обнаружения конструктора с заданным числом и типом аргументов может закончиться неудачей – исключительной ситуацией типа `NoSuchMethodException`. Для учёта этого необходимо либо окружить рассматриваемый код блоком `try-catch`, либо добавить после списка аргументов метода `main()` сведения о том, что при выполнении метода может возникнуть исключительная ситуация.

3.8.3 Сортировка массива

Для операций над массивами в Java предложен специальный класс `Arrays`, содержащий ряд статических методов обработки. Для сортировки массива по заданному критерию мы будем использовать метод `sort(T[], Comparator c)`. Интерфейс `Comparator` определяет метод `compare(Object o1, Object o2)`, возвращающий отрицательное целое, если объект `o1` меньше `o2`; нуль, если они равны; положительное целое, если `o1` больше `o2`. Существует два способа определения класса-компаратора: в виде отдельного класса (для варианта B) или в виде анонимного класса (для варианта C).

Способ 1: для определения компаратора в виде отдельного класса следует добавить в проект новый файл, реализующий интерфейс `Comparator`, и реализовать его метод `compare()`:

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

import java.util.Comparator;

public class FoodComparator implements Comparator<Food> {

    public int compare(Food arg0, Food arg1) {
        // если 1-ый объект = null, то он всегда "больше", т.е. перемещается
        // в конец массива
        if (arg0==null) return 1;
        // если 2-ой объект = null, а 1-ый - нет (не сработала предыдущая
        // строчка), то 1-ый всегда меньше, т.е. перемешается в начало массива
        if (arg1==null) return -1;
        // если оба объекта не null, то результат сравнения определяется
        // сравнением их name
        return arg0.getName().compareTo(arg1.getName());
    }
}
```

Сортировка массива `breakfast` с помощью определённого компаратора выполняется как:

```
Arrays.sort(breakfast, new FoodComparator());
```

Способ 2: использование анонимного класса, определяемого непосредственно в месте использования. Его определение и применение выполняются следующим образом:

```
Arrays.sort(breakfast, new Comparator() {
    public int compare(Object f1, Object f2) {
        if (f1==null) return 1;
        if (f2==null) return -1;
        return ((Food)f1).getName().compareTo(((Food)f2).getName());
    }

});
```

3.8.4 Выполнение операций над элементами массива

Простейшей операцией над элементами массива продуктов, включенных в завтрак, является реализация их «употребления».

Способ 1: с использованием цикла `for`:

```
for (int i=0; i<breakfast.length; i++)
    if (breakfast[i]!=null)
        breakfast[i].consume();
    else break;
```

или

```
for (Food item: breakfast)
    if (item!=null)
        item.consume();
    else
        break;
```

Способ 2: с использованием цикла `while`:

```
int j = 0;
while (breakfast[j]!=null)
    breakfast[j++].consume();
```

3.9 Запуск приложения

Полные версии исходных кодов классов приложения (без реализации дополнительных заданий, предложенных в пункте 4) размещены в Приложении 2.

Для задания аргументов командной строки необходимо определить конфигурацию запуска приложения. Для этого следует активировать пункт меню «*Run → Run Configuration*», в открывшемся диалоговом окне (рисунок 3.17) перейти на вкладку «*Arguments*», и в поле ввода «*Program arguments*» указать перечень аргументов. После этого следует нажать кнопки «*Apply*» и «*Run*».

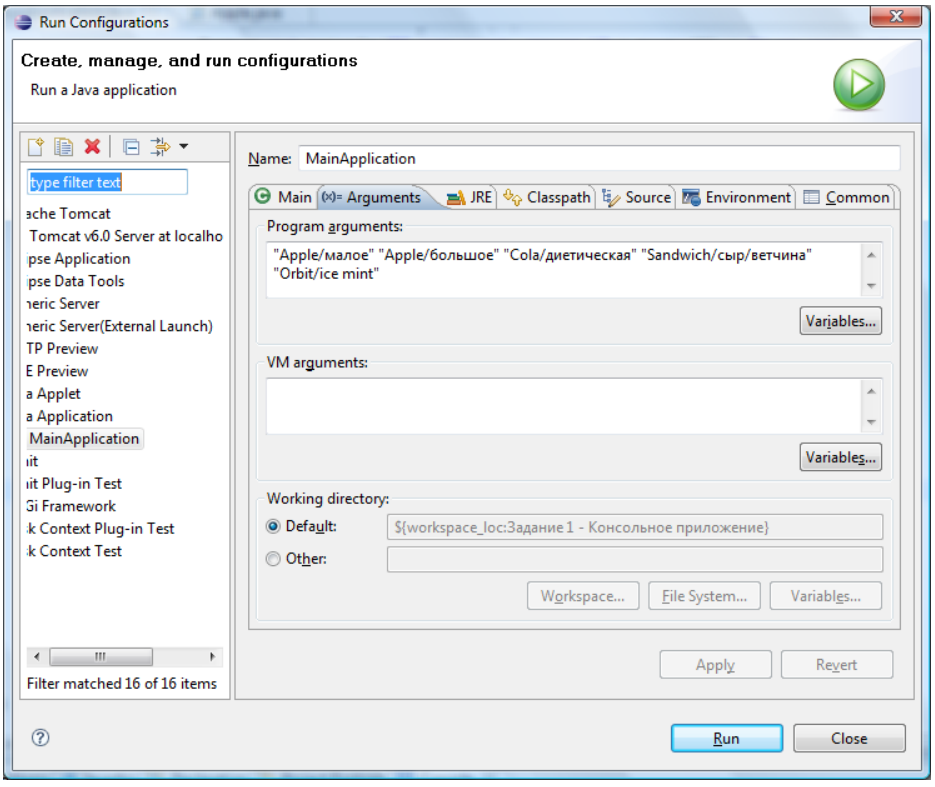


Рисунок 3.17 – Задание аргументов командной строки в диалоговом окне настройки конфигурации запуска

Результаты вывода приложения будут показаны в виде «*Console*».

4 Задания

4.1 Вариант А

а) Реализовать (по вариантам) класс продукта с одним параметром, подключить его в основную программу.

б) Реализовать процедуру подсчёта в завтраке продуктов заданного типа без сравнения по внутренним полям (использовать метод `equals()`, определённый в предке `Food`, без его переопределения в классах-потомках).

№ п/п	Класс продукта	Параметр	Значение параметра
1	Tea (чай)	color (цвет)	чёрный, зелёный
2	Pie (пирог)	filling (начинка)	вишнёвая, клубничная, яблочная
3	Milk (молоко)	fat (жирность)	1.5%, 2.5%, 5%
4	Potatoes (картошка)	type (тип)	жареная, вареная, фри
5	Burger (гамбургер)	size (размер)	малый, большой, средний
6	Coffee (кофе)	aroma (аромат)	насыщенный, горький, восточный
7	IceCream (мороженное)	sirup (сироп)	карамель, шоколад
8	ChewingGum (жевательная резинка)	flavour (привкус)	мята, арбуз, вишня
9	Eggs (яйца)	number (число)	одно, два, три
10	Lemonade (лимонад)	taste (вкус)	лимон, апельсин, клубника
11	Cake (пирожное)	icing (глазурь)	шоколадная, сливочная, карамель
12	Beef (мясо)	preparedness (готовность)	с кровью, норма, прожаренное

4.2 Вариант В

а) Реализовать (по вариантам) класс продукта с одним параметром, подключить его в основную программу.

б) Реализовать процедуру подсчёта в завтраке продуктов заданного типа со сравнением по внутренним полям (использовать метод `equals()`, переопределив его в классах-потомках).

в) Реализовать интерфейс `Nutritious` (Питательный), содержащий метод `calculateCalories()`. Для каждого из классов в зависимости от значений параметров вычислять калорийность.

г) Реализовать обработку специальных параметров (начинающихся с дефиса). При заданном параметре *-calories* вычислить и напечатать общую калорийность завтрака.

д) Реализовать класс-компаратор (как отдельный класс) и обработку специального параметра *-sort*. При заданном параметре *-sort* отсортировать завтрак указанным образом (по вариантам).

№ п/п	Класс продукта	Параметр	Значение параметра	Сортировка
1	Tea (чай)	color (цвет)	чёрный, зелёный	По длине названия в прямом порядке
2	Pie (пирог)	filling (начинка)	вишнёвая, клубничная, яблочная	По калорийности в прямом порядке
3	Milk (молоко)	fat (жирность)	1.5%, 2.5%, 5%	По длине названия в прямом порядке
4	Potatoes (картошка)	type (тип)	жареная, вареная, фри	По алфавиту дополнительных параметров в прямом порядке
5	Burger (гамбургер)	size (размер)	малый, большой, средний	По алфавиту дополнительных параметров в прямом порядке
6	Coffee (кофе)	aroma (аромат)	насыщенный, горький, восточный	По калорийности в прямом порядке
7	IceCream (мороженное)	sirup (сироп)	карамель, шоколад	По длине названия в обратном порядке
8	ChewingGum (жевательная резинка)	flavour (привкус)	мята, арбуз, вишня	По калорийности в обратном порядке
9	Eggs (яйца)	number (число)	одно, два, три	По алфавиту дополнительных параметров в обратном порядке
10	Lemonade (лимонад)	taste (вкус)	лимон, апельсин, клубника	По алфавиту дополнительных параметров в обратном порядке
11	Cake (пирожное)	icing (глазурь)	шоколадная, сливочная, карамель	По длине названия в прямом порядке
12	Beef (мясо)	preparedness (готовность)	с кровью, норма, прожаренное	По калорийности в обратном порядке

4.3 Вариант С

а) Реализовать (по вариантам) класс продукта с двумя параметрами, подключить его в основную программу (через Reflection API).

б) Реализовать процедуру подсчёта в завтраке продуктов заданного типа со сравнением по внутренним полям (использовать метод `equals()`, переопределив его в классах-потомках).

в) Реализовать интерфейс `Nutritious` (Питательный), содержащий метод `calculateCalories()`. Для каждого из классов в зависимости от значений параметров вычислять калорийность.
г) Реализовать обработку специальных параметров (начинающихся с дефиса). При заданном параметре *-calories* вычислить и напечатать общую калорийность завтрака.
д) Реализовать обработку исключений типа «класс не найден» (`ClassNotFoundException`) и «метод не найден» (`NoSuchMethodException`), которые могут возникнуть, если включить в завтрак продукт неопределённого класса. В этом случае сообщить пользователю о том, что продукт не может быть включен в завтрак, и пропустить его.
е) Реализовать класс-компаратор (как анонимный класс) и обработку специального параметра *-sort*. При заданном параметре *-sort* отсортировать завтрак указанным образом (по вариантам).

№ п/п	Класс продукта	Параметры	Порядок сортировки
1	Sandwich (бутерброд)	filling1(начинка №1), filling2 (начинка №2)	По длине названия в прямом порядке
2	Cocktail (коктейль)	drink (напиток), fruit (фрукт)	По калорийности в обратном порядке
3	Dessert (десерт)	component1(компонент №1), component2 (компонент №2)	По количеству дополнительных параметров в обратном порядке

Приложение 1. Исходный код каркаса приложения

Каркас главного класса приложения

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class MainApplication {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
	}

}
```

Каркас интерфейса Consumable

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public interface Consumable {
	public abstract void consume();
}
```

Каркас базового класса Food

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Food implements Consumable {

	@Override
	public void consume() {
		// TODO Auto-generated method stub

	}
}
```

Каркас базового класса Cheese

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Cheese extends Food {

}
```

Каркас базового класса Apple

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Apple extends Food {

}
```

Приложение 2. Окончательный исходный код приложения

Главный класс приложения

```
// Объявление класса частью пакета
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class MainApplication {

	// Конструктор класса отсутствует!!!

	// Главный метод главного класса
	@SuppressWarnings("unchecked")
	public static void main(String[] args) throws Exception {
		// Определение ссылок на продукты завтрака
		Food[] breakfast = new Food[20];
		// Анализ аргументов командной строки и создание для них
		// экземпляров соответствующих классов для завтрака
		int itemsSoFar = 0;
		for (String arg: args) {
			String[] parts = arg.split("/");
			if (parts[0].equals("Cheese")) {
				// У сыра дополнительных параметров нет
				breakfast[itemsSoFar] = new Cheese();
			} else
			if (parts[0].equals("Apple")) {
				// У яблока - 1 параметр, который находится в parts[1]
				breakfast[itemsSoFar] = new Apple(parts[1]);
			}
			// ... Продолжается анализ других продуктов для завтрака
			itemsSoFar++;
		}
		// Перебор всех элементов массива
		for (Food item: breakfast)
			if (item!=null)
				// Если элемент не null - употребить продукт
				item.consume();
			else
				// Если дошли до элемента null - значит достигли конца
				// списка продуктов, ведь 20 элементов в массиве было
				// выделено с запасом, и они могут быть не
				// использованы все
				break;
		System.out.println("Всего хорошего!");
	}
}
```

Интерфейс Consumable

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public interface Consumable {
	public abstract void consume();
}
```

Базовый класс Food

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;
```

```
public abstract class Food implements Consumable {

	String name = null;

	public Food(String name) {
		this.name = name;
	}

	public boolean equals(Object arg0) {
		if (!(arg0 instanceof Food)) return false;	// Шаг 1
		if (name==null || ((Food)arg0).name==null) return false; // Шаг 2
		return name.equals(((Food)arg0).name); // Шаг 3
	}

	public String toString() {
		return name;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	// Реализация метода consume() удалена из базового класса Food
	// Это можно сделать, потому что сам Food - абстрактный

}
```

Класс-потомок Cheese

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Cheese extends Food {

	public Cheese() {
		super("Сыр");
	}

	public void consume() {
		System.out.println(this + " съеден");
	}

	// Переопределять метод equals() в данном классе не нужно, т.к. он
	// не добавляет новых полей данных, а сравнение по внутреннему полю name
	// уже реализовано в базовом классе

	// Переопределять метод toString() в данном классе не нужно, т.к. он
	// не добавляет внутренних полей данных, а возврат поля name уже
	// реализован в версии toString() базового класса
}
```

Класс-потомок Apple

```
package bsu.rfe.java.group7.lab1.Ivanov.varB4;

public class Apple extends Food {

	// Новое внутреннее поле данных РАЗМЕР
	private String size;
```

```
	public Apple(String size) {
		// Вызвать конструктор предка, передав ему имя класса
		super("Яблоко");
		// Инициализировать размер яблока
		this.size = size;
	}

	// Переопределить способ употребления яблока
	public void consume() {
		System.out.println(this + " съедено");
	}

	// Селектор для доступа к полю данных РАЗМЕР
	public String getSize() {
		return size;
	}

	// Модификатор для изменения поля данных РАЗМЕР
	public void setSize(String size) {
		this.size = size;
	}

	// Переопределённая версия метода equals(), которая при сравнении
	// учитывает не только поле name (Шаг 1), но и проверяет совместимость
	// типов (Шаг 2) и совпадение размеров (Шаг 3)
	public boolean equals(Object arg0) {
		if (super.equals(arg0)) { // Шаг 1
			if (!(arg0 instanceof Apple)) return false;     // Шаг 2
			return size.equals(((Apple)arg0).size);   // Шаг 3
		} else
			return false;
	}

	// Переопределённая версия метода toString(), возвращающая не только
	// название продукта, но и его размер
	public String toString() {
		return super.toString() + " размера '" + size.toUpperCase() + "'";
	}
}
```