

# Лабораторная работа № 1

## Создание консольного приложения, использующего разработанную иерархию классов

### Оглавление

Цель лабораторной работы.....	2
1 Среда Eclipse .....	2
2 Краткая памятка по языку Java .....	5
2.1 Java-приложение .....	5
2.2 Документирование программы .....	7
2.3 Объекты как главные элементы языка Java .....	8
2.3.1 Определение классов .....	8
2.3.2 Определение интерфейсов .....	12
2.3.3 Класс Object.....	13
2.3.4 Создание и удаление объектов в Java .....	15
2.3.5 Представление данных простых типов в виде объектов .....	16
2.3.6 Строковые объекты .....	18
2.4 Организация циклов.....	20
2.5 Стандартный ввод и вывод.....	21
3 Базовое приложение для лабораторной работы .....	22
3.1 Структура приложения.....	23
3.2 Создание проекта программы в Eclipse.....	25
3.3 Создание пакета для классов .....	27
3.4 Создание общего интерфейса <i>Consumable</i> , используемого в иерархии классов .....	29
3.5 Реализация базового класса иерархии продуктов <i>Food</i> .....	30
3.6 Реализация классов <i>Apple</i> и <i>Cheese</i> .....	36
3.7 Реализация главного класса приложения.....	39
3.7.1 Создание экземпляров классов с помощью сравнения строк .....	41
3.7.2 Создание экземпляров классов с помощью Java Reflection .....	41
3.7.3 Сортировка массива .....	42
3.7.4 Выполнение операций над элементами массива .....	44
3.8 Запуск приложения .....	44
4 Задания.....	47
4.1 Вариант сложности А.....	47
4.2 Вариант сложности В.....	47
4.3 Вариант сложности С.....	48
Приложение 1. Исходный код базового приложения .....	50

## Цель лабораторной работы

Получить практические навыки создания консольных приложений на языке Java в среде Eclipse с использованием объектно-ориентированного подхода к программированию.

### 1 Среда Eclipse

Eclipse является свободно распространяемой интегрированной средой разработки (Integrated Development Environment – IDE) для создания модульных кроссплатформенных приложений, разработку которой координирует Eclipse Foundation. Гибкость данной IDE обеспечивается за счет возможности ее расширения и дополнения подключаемыми модулями. Благодаря этому с использованием этой IDE можно вести разработку не только на Java (который поддерживается изначально), но и на других языках программирования, среди которых C/C++, Perl, Ruby, Python, PHP и др.

Внешний вид основного окна среды разработки Eclipse и его основные элементы представлены на рисунке 1.

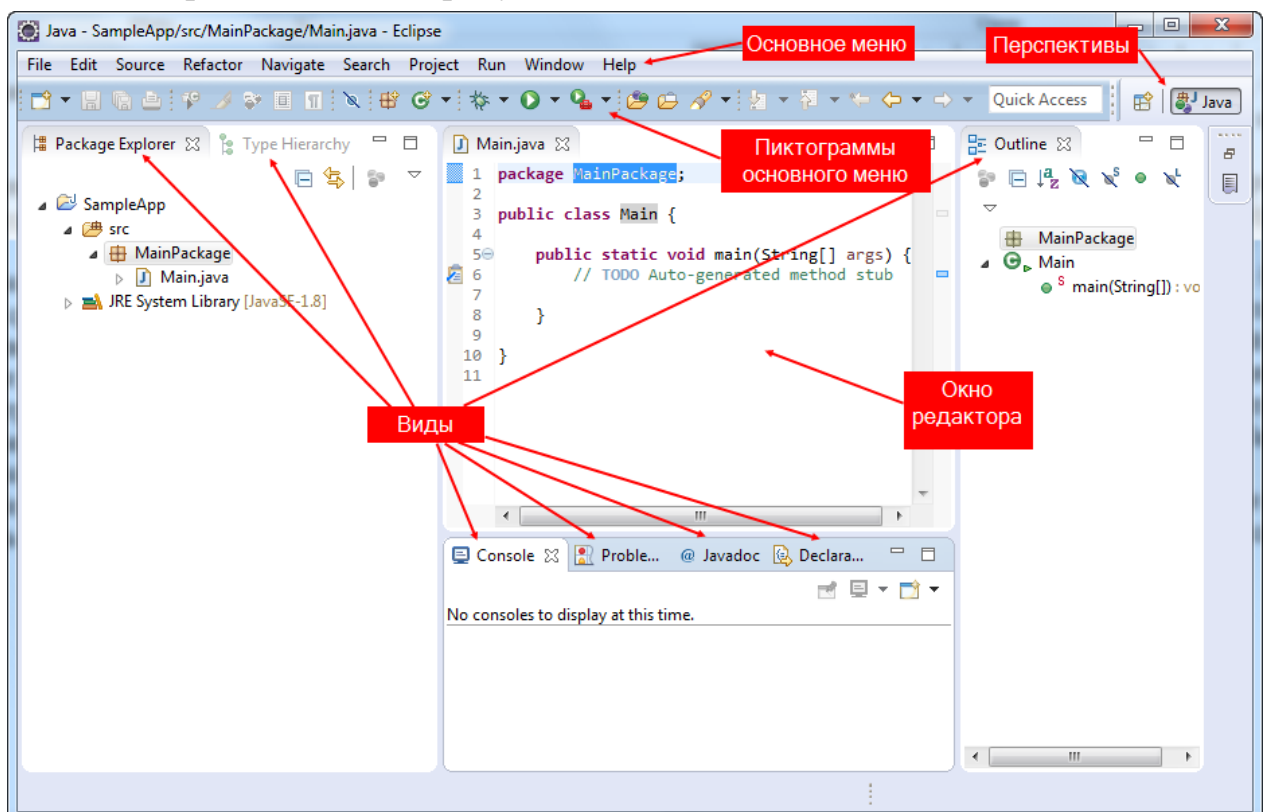


Рис 1. Основное окно среды разработки Eclipse и его основные элементы

**Основное меню** среды Eclipse включает все необходимые средства для управления проектом и самой средой разработки.

**Пиктограммы основного меню** позволяют получить быстрый доступ к набору наиболее часто используемых команд основного меню.

**Встроенный редактор исходного кода** программ позволяет просматривать и редактировать исходный код. Данный редактор является частью инструментария разработчика Java (Java Developer Toolkit). В среде Eclipse так же возможно использование альтернативных инструментов разработчика для написания программ на других языках программирования. Встроенный редактор исходного кода содержит ряд средств, предназначенных для облегчения и ускорения разработки приложений. К таким средствам можно отнести автоматическую проверку синтаксиса текста программы, организацию закладок для ускорения перемещения по исходному коду, всплывающие окна с информацией о классах, их методах и полях, автоматическую подстановку кода, возможность расстановки точек прерывания для управления отладкой приложения и т.д. Окна встроенного редактора можно отобразить только при наличии файлов с расширением .java, содержащих исходный код.

**Виды** представляют собой отдельные подчиненные окна в рамках среды Eclipse, которые могут как встраиваться в главное окно, так и отображаться в виде отдельных окон. Каждый вид предназначен для отображения информации определенного рода. Основные виды, обычно используемые при разработке Java-приложений, а также их основное назначение представлены в таблице 1.

Таблица 1 Виды, наиболее часто используемые при разработке Java-приложений

Название	Назначение
Project Explorer	Отображает в виде древообразной структуры все проекты, пакеты и файлы рабочего окружения
Type Hierarchy	Позволяет просматривать содержимое отображаемых классов, а также их положение в иерархии
Outline	Отображает структуру файла, открытого в данный момент в редакторе
Navigator	Показывает содержимое папки проекта (подчиненные папки и файлы)
Declaration	Указывает на часть исходного кода, где определен выделенный объект
Javadoc	Показывает описание выделенного в исходном коде элемента, взятое из комментариев, оформленных с использованием символов <code>/** */</code>
Problems	Отображает предупреждения и ошибки компиляции приложения
Console	Отображает результаты, если таковые выводятся приложением в консоль

**Перспективы** представляют собой наборы видов и редакторов, которые необходимы для наиболее удобного выполнения некоторой задачи в рамках среды Eclipse. В качестве примеров задач, для которых существуют отдельные перспективы, можно привести такие как написание исходного кода приложения на Java (Java-перспектива) и отладка программы (перспектива отладки). Содержимое каждой перспективы может настраиваться программистом. Для переключения между перспективами можно использовать либо меню «*Window* → *Open Perspective*», либо соответствующий блок пиктограмм (смотри рисунок 1). При выборе определенной перспективы на экране автоматически появляются связанные с ней виды и редакторы.

Среда Eclipse позволяет ускорить процесс разработки приложений предоставляя следующие возможности.

Eclipse использует **автоматическую подстановку кода**, позволяя выбрать из списка подходящий метод, объект или поле данных и завершить выражение. Для этого в коде программы после имени класса или объекта класса 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 ведется 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 ее функциональности, например, переименование классов, переменных, пакетов; перемещение классов из пакета в пакет и т.д.). Для этого можно воспользоваться пунктами, содержащимися в меню «*Refactor*».

В Eclipse используется ряд «горячих клавиш», упрощающих работу в среде (таблица 2).

Таблица 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

Программы на языке Java полностью основаны на методологии объектно-ориентированного программирования (ООП) и представляют собой комбинацию классов, взаимодействующих друг с другом. Создание методов, не принадлежащих какому-либо классу, невозможно.

Исходный код класса на Java располагается в отдельном файле с расширением **.java**, причем в данный файл включается как определение класса, так и его реализация.

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-программах классы, составляющие приложение, группируются по **пакетам**, которые физически представляют собой папки на диске. Пакеты обеспечивают также **привилегированный доступ к переменным и методам класса из того пакета, к которому класс принадлежит**. Для объявления класса частью пакета применяется ключевое слово ***package***.

Так класс *SampleClass*, принадлежащий пакету *by.bsu.rfkt*, объявляется следующим образом:

```
package by.bsu.rfkt;

public class SampleClass {
    ...
}
```

При этом физический путь к файлу данного класса относительно корневого каталога проекта, где располагаются файлы с исходным кодом, будет выглядеть как «by\bsu\rfkt\SampleClass.java».

Поскольку классы, используемые в рамках приложения на Java, разделены по пакетам, то для обращения к классам из другого пакета необходимо использовать кроме имени класса так же и полное имя пакета, которому он принадлежит. Например, для класса `Date`, принадлежащего пакету `java.util`:

```
class MyDate extends java.util.Date{
    ...
}
```

Для того, чтобы не использовать длинные имена пакетов каждый раз, для обеспечения видимости классов из другого пакета используется оператор ***import***. Общая форма записи данного оператора выглядит следующим образом:

```
import pkg1[.pkg2].(classname | *);
```

где *pkg1* – имя пакета верхнего уровня, *pkg2* – имя подчиненного пакета внутри *pkg1*, *classname* или *\** обозначают либо явное имя класса, либо все классы пакета, соответственно. Если используется символ *\**, то импортируются все классы только из непосредственно указанного пакета. Из вложенных пакетов импорт не выполняется. При наличии оператора ***import*** можно использовать имена проимпортированных классов без указания имен пакетов:

```
import java.util.*
class MyDate extends Date{
    ...
}
```

Основная часть классов Java-приложения 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.

Определение метода *main*:

```

// Задание пакета, которому принадлежит класс
package by.bsu.rfkt;

//объявление главного класса Lab1 приложения
public class Lab1 {
    // конструктор класса Lab1 вызывается из метода main
    //в простых программах может не определяться
    public Lab1() {
        }

    //Главный метод, автоматически вызывается при запуске,
    //обязательно должен быть определен 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
    public static void main(String[] args) {
        System.out.println("Лабораторная работа 1");
    }
}

```

## 2.2 Документирование программы

Утилита *javadoc* обеспечивает возможность использования специальным образом оформленных **комментариев разработчика** в исходном коде для автоматического создания документации. Комментарии вносятся в файлы исходного кода и утилита создает 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 представлены в таблице 3.

Таблица 3 Основные специальные тэги в комментариях разработчика

Тег	Описание	Применим к
@author	Автор	класс, интерфейс
@version	Версия. Не более одного тега на класс	класс, интерфейс
@param	Входной параметр метода	метод
@return	Описание возвращаемого значения	метод
{@value}	Описание значения переменной	статическое поле



## 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. **Объявление класса в Java** в общем виде может быть представлено следующим образом:

```
[<Модификаторы_класса>] class <Имя_класса> [extends <Имя_суперкласса>]
                                     [implements <Интерфейс_1>, <Интерфейс_2>, ...]
{
    // Описания полей и методов класса
}
```

При определении класса можно использовать следующие **модификаторы**:

- *public* — задает общедоступный (публичный) уровень видимости. Если он отсутствует, класс доступен только в рамках того же пакета, в котором он определен.
- *abstract* — указывает на то, что класс является абстрактным и не может быть использован для создания объектов (экземпляров). Данный модификатор обязательно необходимо использовать если хотя бы один метод класса является абстрактным (не имеет реализации).
- *final* — указывает на то, что класс является окончательным и не может иметь подклассов.

Ключевое слово *extends* используется для указания **суперкласса**, от которого определяемый класс наследует поля и методы. В Java после ключевого слова *extends* можно указать лишь один суперкласс. В случае, если ключевое слово *extends* явно не используется при определении класса, то по умолчанию его суперклассом считается класс библиотеки Java *Object*.

Дополнительно к суперклассу определяемый класс может реализовывать более одного **интерфейса**, имена которых указываются через запятую после ключевого слова *interface*. Интерфейс в Java представляет собой полностью абстрактный класс, определение которого может содержать только константы и абстрактные методы (не имеющие реализации).

Основными компонентами, которые может содержать класс являются *поля* класса, *блоки инициализации*, *конструкторы* и *методы* класса. Кроме этого класс может включать *внутренние классы*.

**Набор данных класса представлен его полями.** Общий вид определения поля класса можно представить в виде следующего шаблона.

```
[public|protected|private] [static] [final] <Тип_поля> <Имя_поля>
    [ = <Начальное значение> | = <Имя_конструктора>(<Аргументы_конструктора>) ] ;
```

При определении поля класса можно использовать следующие модификаторы доступа:

- *public* — задает общедоступный уровень видимости поля;



- *protected* – позволяет получать доступ к полю в рамках самого определяемого класса, из всех его подклассов, а так же из классов того же пакета;
- *private* – указывает, что доступ к данному полю класса из других классов закрыт. Указание данного модификатора для полей класса реализует принцип инкапсуляции.

Если модификатор доступа отсутствует, то поле будет доступно в рамках того же пакета, в котором определен включающий его класс.

Модификатор *static* указывает на определение поля класса, а не объекта класса. Такое поле имеет общее значение для всех объектов класса и доступ к нему осуществляется с использованием имени класса, а не объекта.

Модификатор *final* позволяет создать константу, которой должно быть присвоено начальное значение, изменение которого в дальнейшем невозможно.

При определении поля класса после знака = ему сразу же может быть присвоено начальное значение. В зависимости от типа поля, это может быть или значение, соответствующее одному из простых типов (*int*, *double* и т.д.), или ссылка на объект класса.

**Конструкторы** – особого вида методы, которые по имени автоматически вызываются при создании объекта класса с помощью оператора *new*. Для конструктора не указывается тип возвращаемого значения. Основной задачей конструктора является создание объекта класса и инициализация его полей.

Для вызова конструктора суперкласса при конструировании объекта используется ключевое слово *super*, после которого в круглых скобках через «*,*» перечисляются параметры, необходимые для конструктора предка. Если для класса определено несколько конструкторов с различными наборами параметров, то из одного конструктора можно вызвать другой используя ключевое слово *this*, после которого следует необходимый набор параметров.

<pre>public class A {     private String info;      public A(String info) {         this.info = info;     } }</pre>	<pre>public class B extends A {     private int data;      public B(String info) {         //Вызов конструктора класса B с 2-мя параметрами         this(info, 0);     }      public B(String info, int data) {         //Вызов конструктора суперкласса A         super(info);         //Запись значения аргумента конструктора в поле data         //this.data - обращение к полю data         this.data = data;     } }</pre>
---	--

При наличии нескольких конструкторов оператор *new* будет вызывать тот из них, который имеет соответствующий вызову список параметров. Если конструктор в классе не определен явно, то компилятором будет автоматически добавлен конструктор по умолчанию без параметров. Если в классе определен конструктор с параметрами, то конструктор по умолчанию автоматически не добавляется и если он необходим, то он должен быть определен в явном виде.

Помимо конструктора для инициализации полей класса и объекта могут использоваться специальные **блоки инициализации**, оформление которых в общем виде можно представить следующим образом.

```
[<Модификаторы>] class <Имя_класса> [extends <Имя_суперкласса>]
{
    // Задание полей

    static
    {
        // тело блока инициализации полей класса
    }

    {
        // тело блока инициализации объекта
    }

    // Задание конструкторов, методов класса, методов объекта
}
```

Класс может содержать несколько блоков инициализации, которые в этом случае вызываются последовательно в порядке размещения. Порядок выполнения кода блоков инициализации и конструктора следующий:

1. выполнение блоков инициализации класса;
2. выполнение блоков инициализации объекта;
3. выполнение конструктора.

Кроме задания значений полей блоки инициализации так же могут включать и вызовы методов.

**Методы**, определенные в классе, задают поведение его объектов и обеспечивают возможность их взаимодействия с другими объектами в рамках приложения. В общем виде определение метода можно представить следующим образом.

```
[<Модификаторы>] <Тип_возвращаемого_значения> <Имя_метода>(<Аргументы_метода>)
{
    // Тело метода
}
```

Набор модификаторов метода может одновременно включать:

1. один из модификаторов доступа (*public*, *private*, *protected*) использование которых аналогично их использованию при определении полей класса;

2. модификатор *static*, указывающий на то, что метод является общим методом класса;
3. модификатор *final*, указывающий на невозможность переопределения данного метода в подклассах;
4. модификатор *abstract*, который указывает на то, что метод не имеет реализации (тела);
5. модификатор *synchronized*, указывающий на то, что метод является синхронизированным и при работе многопоточного приложения различные потоки могут вызывать данный метод только поочередно;
6. модификатор *native*, который указывает, что метод задан во внешней библиотеке (DLL), написанной на другом языке программирования.

Переданные в метод параметры могут быть данными простого типа (*int*, *double* и т.д.) или ссылками на объект. Все параметры передаются в метод по значению, то есть при передаче параметра создается его копия, которая и используется внутри метода.

В случае, если метод возвращает значение некоторого типа, то данный тип указывается в определении метода, в противном случае вместо типа возвращаемого значения указывается зарезервированное слово *void*.

При вызове метода его поиск происходит сначала в классе, объект которого был использован для организации вызова метода, затем в иерархии его суперклассов, пока метод не будет найден или не достигнут класс *Object* – суперкласс для всех классов.

Если в рамках построенной иерархии классов определено два метода с одинаковыми именами и возвращаемыми значениями, но разными списками параметров, то такие методы называются *перегружаемыми* (*overloading*). При обращении вызывается метод, список параметров которого совпадает со списком параметров вызова.

Если объявление метода подкласса полностью, включая параметры, совпадает с объявлением метода суперкласса, то метод подкласса *переопределяет* (*overriding*) метод суперкласса.

**Обращение к полям и методам текущего класса и суперклассов.** Для обращения к полям, определенным в рамках некоторого класса, из кода методов этого же класса используется ссылка на текущий объект *this*. При использовании *this* для обращения к полям, после данной ссылки указывается через точку имя поля. Если в коде некоторого метода создана локальная переменная, имя которой совпадает с именем поля того же класса, где определен данный метод, то для обращения к полю необходимо обязательно использовать *this*, в противном случае использование *this* возможно, но не обязательно.

```

public class A {
    //Определение поля info
    private String info;

    public A(String info) {
        //Запись значения аргумента конструктора в поле info
        //this.info - обращение к полю info
        this.info = info;
    }
}

```

Для обращения к полям и методам суперкласса из методов его подкласса используется ссылка *super*, после которой ставится точка, а затем указывается имя поля, либо метода суперкласса.

```

public class A {
    private String info;
    ...
    //Определение метода getInfo
    public String getInfo() {
        return info;
    }
}

public class B extends A {
    ...
    public void ViewInfo() {
        //Обращение к методу getInfo() класса A
        System.out.println("Info: " + super.getInfo());
    }
}

```

### 2.3.2 Определение интерфейсов

Отдельным структурным элементом программ на Java являются *интерфейсы*, которые позволяют определить набор действий (методов), которые могут выполнять реализующие его классы. Интерфейсы определяются в соответствии со следующим шаблоном.

```

[<Модификатор>] interface <Имя> [extends <Интерфейс_1>,<Интерфейс_2>,...]
{
    // Описание полей
    // Описание методов
}

```

Как видно из представленного шаблона, для определения интерфейсов вместо ключевого слова *class* используется ключевое слово *interface*. В качестве модификатора можно использовать *public*, что делает интерфейс общедоступным. Если модификатор отсутствует, то интерфейс доступен в рамках пакета. При определении интерфейсов после ключевого слова *extends* допускается указание более одного родительского интерфейса. Если ключевое слово *extends* отсутствует, то интерфейс считается не имеющим предка. Поля,

определяемые в рамках интерфейса, по умолчанию (без явного указания соответствующих модификаторов) считаются общедоступными (*public*), не изменяемыми (*final*) и принадлежащими классу (*static*). Другие модификаторы для задания полей интерфейсов использоваться не могут. Определяемые в интерфейсе методы могут быть разделены на четыре категории:

1. **общедоступные абстрактные** – методы без реализации, по умолчанию (даже без явного указания) имеющие модификаторы *public* и *abstract*;
2. **статические методы интерфейса (начиная с JDK 8)** – решают служебные задачи, связанные с интерфейсом, и могут иметь модификаторы доступа *public* (явно или при отсутствии модификатора доступа) или *private* (явно);
3. **общедоступные по умолчанию с реализацией (начиная с JDK 8)** – предназначены для расширения функционала ранее созданных интерфейсов и реализации методов, которые могут быть необязательными для классов, реализующих интерфейс; всегда имеют модификаторы *default* (указывается явно) и *public* (даже без явного указания);
4. **закрытые с реализацией (начиная с JDK 9)** – предназначены для решения вспомогательных задач и устранения дублирования кода в методах по умолчанию.

Интерфейсы не могут напрямую использоваться для создания объектов. Они могут быть унаследованы классами или другими интерфейсами. Класс не может реализовывать несколько интерфейсов, у которых есть методы, определение которых отличается только типом возвращаемого значения.

### 2.3.3 Класс *Object*

Класс *Object* представляет особенный интерес среди других классов библиотеки Java поскольку является суперклассом для всех остальных классов. Когда определяется класс, не являющийся наследником какого-либо иного класса, в этом определении неявно присутствует оператор *extends Object*. Следующие два определения класса эквивалентны:

<pre>class Point {     int x;     int y; }</pre>	<pre>class Point extends Object {     int x;     int y; }</pre>
--	---

Класс *Object* определяет ряд методов, которые наследуются всеми классами Java (смотри таблицу 4).

Таблица 4 – Методы класса *Object*

Метод	Описание
<i>Class getClass()</i>	возвращает ссылку на объект типа <i>Class</i> , который: <ul style="list-style-type: none"> <li>содержит описание класса, задействованного при создании объекта (сведения о его модификаторах, конструкторах, полях, методах и т.д.);</li> <li>позволяет обращаться к конструкторам описываемого класса для создания его объектов (с помощью библиотеки Reflection API)</li> </ul>
<i>int hashCode()</i>	возвращает целочисленный идентификатор объекта (хэш-код). Хэш-коды объектов могут использоваться для быстрой адресации при их хранении в коллекциях. Предполагается, что данный метод возвращает одинаковое значение для объектов, сравнение которых методом <i>equals</i> возвращает <i>true</i> .
<i>Boolean equals(Object obj)</i>	сравнивает вызывающий объект и объект <i>obj</i> и возвращает <i>true</i> , если они равны, иначе <i>false</i> . Метод, реализованный в классе <i>Object</i> , сравнивает объекты по их ссылкам и эквивалентен оператору <i>==</i> . Для реализации сравнения по содержимому объектов метод необходимо переопределять в подклассах.
<i>Object clone()</i>	возвращает ссылку на созданный дубликат (клон) объекта. Перед выполнением клонирования метод <i>clone</i> проверяет, реализован ли в классе, использовавшемся для создания объекта, интерфейс <i>Cloneable</i> , и если это не так, то генерируется исключение <i>CloneNotSupportedException</i> . Сам класс <i>Object</i> не реализует <i>Cloneable</i> , поэтому для поддержки клонирования данный интерфейс необходимо реализовать в подклассах. Если <i>Cloneable</i> реализован, то создается новый объект такого же класса, поля которого инициализируются значениями полей исходного объекта. Если объект содержит поля ссылочного типа, то они не копируются, а инициализируются ссылками их исходного объекта, т.е. выполняется так называемое "мелкое копирование" ("shallow copy"). Для реализации "глубокого копирования" ("deep copy"), когда для полей ссылочного типа создаются новые объекты, необходимо переопределять метод <i>clone</i> в подклассах.
<i>String toString()</i>	возвращает строковое представление объекта в следующем виде: <b>&lt;имя_пакета&gt;.&lt;имя_класса_при_создании&gt;@&lt;хэш-код объекта&gt;</b> Для изменения выдаваемой информации, метод необходимо переопределять в подклассах.
<i>void notifyAll()</i> <i>void notify()</i> <i>void wait()</i>	используются для синхронизации программных потоков в многопоточном приложении.
<i>void finalize()</i>	вызывается перед уничтожением объекта и должен быть переопределен, если в этом случае необходимо выполнить какие-либо действия.

### 2.3.4 Создание и удаление объектов в Java

Для создания объектов (экземпляров) класса в Java используется оператор *new*, который создает объект класса при помощи одного из его конструкторов и возвращает **ссылку на созданный объект**.

При создании объекта класса выполняется следующий ряд действий:

1. выполняется создание объекта, поля данных которого имеют по умолчанию нулевые значения (переменные тип которых задан именем класса, содержащие ссылки, получают значение *null*, поля простых типов, таких как *int*, *double* и т.д., получают значение 0, логические поля получают значение *false*);
2. вызывается конструктор класса, для которого создается объект, соответствующий переданному набору аргументов, который иницирует вверх по иерархии вызов конструкторов всех суперклассов вплоть до класса *Object*;
3. в каждом классе, начиная с *Object* и далее вниз по иерархии вплоть до класса, для которого создается объект, выполняются следующие действия:
  - инициализируются поля, определенные в данном классе, значениями, заданными в определении полей;
  - выполняются блоки инициализации данного класса;
  - выполняется тело конструктора данного класса.
4. возвращается ссылка на созданный объект.

Для сохранения адреса созданного объекта используются *переменные ссылочного типа*. В следующем примере такой переменной будет переменная *cls* типа *SampleClass*:

```
//Создание объекта 'cls' класса 'SampleClass'  
SampleClass cls = new SampleClass();
```

Отличие переменных ссылочного типа от переменных примитивного типа:

- хранят не сами значения данных (объектов), а адреса, по которым эти данные (объекты) расположены;
- несколько переменных могут ссылаться на один и тот же набор данных в памяти, где расположен некоторый объект;
- операция присвоения для ссылочных переменных меняет адреса, а не сами данные, расположенные по этим адресам;
- размер памяти резервируемый для всех ссылочных переменных одинаков не зависимо от того, на объект какого типа они ссылаются.

После создания объекта, ссылочная переменная, содержащая его адрес, может использоваться для обращения к доступным полям и методам объекта. Для этого указывается имя переменной и через точку имя поля или метода, к которому необходимо обратиться. Для обращения к статическим полям и



методам обычно используется имя класса, при этом создание объекта не требуется.

```
public class SampleClass {
    private int fld;

    public static void SampleMethod() {
        System.out.println("This is static method call");
    }

    public void PrintInfo() {
        System.out.println("This is sample class");
    }
}

//Пример вызова статического метода класса
SampleClass.SampleMethod();

//Создание объекта 'cls' класса 'SampleClass'
SampleClass cls = new SampleClass();
//Пример вызова метода с использованием объекта
cls.PrintInfo();
```

Классы в Java не содержат деструкторов, поскольку ранее созданные объекты классов не приходится уничтожать напрямую из кода программы. 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**.

Для выполнения действий перед уничтожением объекта можно использовать метод *finalize()*, который изначально определен в классе *Object* и вызывается каждый раз, когда сборщик мусора будет уничтожать объект класса. При необходимости данный метод может быть переопределен в подклассе класса *Object*, и в него может быть добавлен код, который необходимо вызывать при уничтожении объектов.

### 2.3.5 Представление данных простых типов в виде объектов

В Java для представления чисел, символов и логических значений можно использовать как переменные простых типов, так и специальные классы-оболочки, которые соответствуют каждому простому типу и представлены в таблице 5.

Таблица 5 – Классы-оболочки для простых типов

Простой тип	Класс-оболочка	Назначение
byte	Byte	8-битовое (байтовое) целое число со знаком
short	Short	16-битовое короткое целое число со знаком

int	Integer	32-битовое обычное целое число со знаком
long	Long	64-битовое длинное целое число со знаком
char	Character	16-битовое символьное значение в кодировке <i>Unicode</i>
float	Float	32-битовое вещественное число
double	Double	64-битовое вещественное число
boolean	Boolean	логическое значение: <i>true</i> или <i>false</i>

Классы-оболочки позволяют:

- построить объект на основе значения простого типа, для чего явно, либо неявно используется конструктор соответствующего класса-оболочки:

```
//Создание объекта типа Integer
//без явного указания конструктора
Integer intObject = 10;
//Создание объекта типа Integer
//с использованием конструктора в явном виде
Integer intObject = new Integer(10);
```

- получить значение простого типа из ранее созданного объекта класса-оболочки (метод с именем, заканчивающимся словом **Value**):

```
//Создание объекта типа Integer
Integer intObject = 10;
//Получение значения типа int
int value = intObject.intValue();
```

- сконvertировать значение соответствующего типа в строку (метод **toString**):

```
//Статический вариант метода toString
String str = Integer.toString(10);

//Обычный вариант метода toString
Integer value = 10;
String str = value.toString();
```

- сконvertировать строковое представление значения соответствующего типа в значение простого типа (метод начинается со слова **parse**):

```
int value = Integer.parseInt("10");
```

- создать объект с типом, соответствующим классу-оболочке, и содержащим значение, полученное из его строкового представления (метод **valueOf**):

```
Integer value = Integer.valueOf("10");
```

Для работы с числовыми данными библиотека Java предоставляет следующие классы:

- Класс *java.lang.Math* – содержит основные математические функции и константы;

- Классы *java.math.BigDecimal* и *java.math.BigInteger*, позволяющие работать с числами, превосходящими стандартную размерность числовых типов.

### 2.3.6 Строковые объекты

Отдельные символы в рамках строки в Java кодируются с использованием международного алфавита Unicode с 16-битовой кодировкой.

Для представления строк в Java используется класс *String*, использование которого сопряжено со следующими особенностями:

- объекты класса *String* можно создавать без использования оператора *new*. Так, при задании строкового литерала, автоматически порождается объект класса *String*:

```
//Классический вариант создания объекта
String str = new String("Строка");

//Неявное создание объекта без использования new
String str = "Строка";
```

- содержимое объектов класса *String* не может быть изменено после их создания, и для изменения хранящейся в них строки необходимо создавать новый объект;
- для класса *String* переопределен оператор *+*, который позволяет объединять строки. Если к строке прибавляется значение другого типа, то данное значение автоматически преобразуется в строку:

```
//выводит в консоль строку 'Result: 123true13.65'
System.out.print("Result: "+ 123 + true + 13.65);
```

Некоторые, наиболее востребованные на практике, методы класса *String* приведены в таблице 6.

Таблица 6 Некоторые методы класса *String*

Метод	Назначение
<i>char charAt(int i)</i>	возвращает символ с индексом <i>i</i> (индекс первого элемента в строке равен 0)
<i>boolean endsWith(String str)</i>	возвращает <i>true</i> в случае, когда строка заканчивается последовательностью символов, содержащихся в строке <i>str</i>
<i>boolean equals(Object str)</i>	возвращает <i>true</i> в случае, если последовательность символов строки совпадает с последовательностью символов в <i>str</i>
<i>boolean equalsIgnoreCase(String str)</i>	аналогичен <i>equals(str)</i> , но без учета регистра
<i>int indexOf(String str)</i>	возвращает индекс позиции, где в строке первый раз встретилась последовательность символов <i>str</i>
<i>int length()</i>	возвращает длину строки (для пустой строки – 0)

<i>String</i> <b>replaceFirst</b> ( <i>String</i> <i>str1</i> , <i>String</i> <i>str2</i> )	возвращает строку, в которой первое вхождение последовательности символов <i>str1</i> заменено на символы строки <i>str2</i>
<i>String</i> <b>replaceAll</b> ( <i>String</i> <i>str1</i> , <i>String</i> <i>str2</i> )	возвращает строку, в которой все вхождения символов <i>str1</i> заменены на символы <i>str2</i>
<i>String</i> [] <b>split</b> ( <i>String</i> <i>str</i> )	возвращает массив строк <i>String</i> [], полученный разделением строки на части по местам вхождения разделителя, задаваемого строкой <i>str</i> . При этом символы, содержащиеся в <i>str</i> , в получившиеся строки не входят. Пустые строки из конца получившегося массива удаляются.
<i>boolean</i> <b>startsWith</b> ( <i>String</i> <i>str</i> )	возвращает <i>true</i> , если строка начинается с символов строки <i>str</i>
<i>String</i> <b>substring</b> ( <i>int</i> <i>index1</i> )	возвращает строку с символами, скопированными из исходной строки начиная с позиции <i>index1</i>
<i>String</i> <b>substring</b> ( <i>int</i> <i>index1</i> , <i>int</i> <i>index2</i> )	возвращает строку, скопированную из исходной строки начиная с позиции <i>index1</i> и заканчивая позицией <i>index2</i>
<i>char</i> [] <b>toCharArray</b> ()	возвращает массив символов строки
<i>String</i> <b>toLowerCase</b> ()	возвращает копию строки с преобразованными к нижнему регистру символами
<i>String</i> <b>toUpperCase</b> ()	возвращает копию строки с преобразованными к верхнему регистру символами
<i>String</i> <b>trim</b> ()	возвращает копию строки, из которой удалены пробелы, идущие в начале и в конце

Поскольку при изменении строк, представленных объектами класса *String*, каждый раз создаются новые объекты, это приводит к неэффективному расходу памяти. В таких случаях при выполнении частых изменений содержимого строки имеет смысл использовать класс ***StringBuffer***, созданный для представления изменяемых строк.

Для создания объектов типа *StringBuffer*, используются его конструкторы, принимающие в качестве параметра строковый литерал, или ссылку на объект типа *String*:

```
//создание объекта класса StringBuffer
//на базе строкового литерала
StringBuffer sb = new StringBuffer("Строка");
//создание объекта класса StringBuffer
//на базе объекта класса String
String str = new String("Строка");
StringBuffer sb = new StringBuffer(str);
```

Для получения объектов типа *String* на базе объекта класса *StringBuffer*, используется метод *toString*:

```
StringBuffer sb = new StringBuffer("Строка");
String str = sb.toString();
```

Класс *StringBuffer* содержит ряд методов для изменения содержимого строки, которые представлены в таблице 7.

Таблица 7 Некоторые методы класса *StringBuffer*

Метод	Назначение
<i>StringBuffer</i> <b>append</b> ( <i>String str</i> )	добавляет к текущему содержимому строку <i>str</i>
<i>StringBuffer</i> <b>insert</b> ( <i>int i</i> , <i>String str</i> )	вставляет строку <i>str</i> в текущее содержимое буфера начиная с позиции <i>i</i>
<i>StringBuffer</i> <b>reverse</b> ()	выстраивает содержимое буфера в обратном порядке
<i>void</i> <b>setCharAt</b> ( <i>int i</i> , <i>char ch</i> )	заменяет символ на позиции <i>i</i> на <i>ch</i>
<i>char</i> <b>charAt</b> ( <i>int i</i> )	возвращает символ, находящийся в позиции <i>i</i>

## 2.4 Организация циклов

Как и в C++ в Java циклы организуются с использованием следующих операторов:

- **for** – преимущественно используется для создания циклов с заранее известным числом итераций;
- **while** и **do-while** – используются для организации циклов, количество итераций которых заранее не известно, и выход из цикла выполняется по некоторому условию.

Для цикла *for* кроме классической формы записи, аналогичной C++, существует специальная форма записи, используемая при работе с массивами и реализуемая в соответствии со следующим шаблоном.

```
for (<Тип> <Имя_локальной_переменной> : <Имя_массива>) {
    <блок кода>;
}
```

В данном случае количество итераций цикла определяется числом элементов массива, используемого в секции <Имя\_массива>. На каждой итерации цикла в локальную переменную, которая определяется при задании цикла, помещается очередной элемент из используемого массива, который выбирается в соответствии с номером итерации.

Пример использования обоих вариантов цикла *for* для подсчета суммы элементов массива приведен ниже. Оба приведенных цикла дают одинаковый результат.

<pre>int[] numbers = {3, 7, 2, 11, 1}; int sum = 0;</pre>	
<pre>//Классическая конструкция цикла for (int i=0; i&lt;numbers.length; i++) {     sum += numbers[i]; }</pre>	<pre>//Специальная конструкция цикла for (int value: numbers) {     sum += value; }</pre>

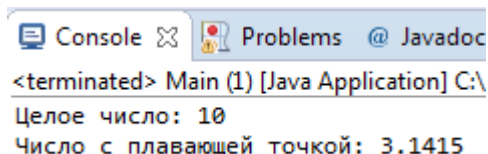
## 2.5 Стандартный ввод и вывод

Для выполнения стандартного ввода/вывода информации в Java используются статические объекты класса *System* *in* и *out*. Оба этих объекта являются потоками, которые по умолчанию работают с консолью.

Объект *System.out* типа *java.io.PrintStream* предоставляет методы *print* и *println*, которые наиболее часто используются для вывода информации в текстовом виде. Метод *println* отличается от метода *print* тем, что после вывода информации выполняет переход на следующую строку. Использование данных методов можно продемонстрировать на следующем примере:

```
System.out.print ("Целое число: ");  
System.out.println (10);  
System.out.print ("Число с плавающей точкой: ");  
System.out.println (3.1415);
```

после выполнения которого в консоли будет отображен следующий результат:



```
<terminated> Main (1) [Java Application] C:\  
Целое число: 10  
Число с плавающей точкой: 3.1415
```

Для ввода информации из консоли используется статический объект *System.in* типа *java.io.InputStream*. Он представляет собой поток байт, который обеспечивает возможность чтения данных из потока отдельными байтами или их группами. Поскольку чтение данных в виде последовательности байт во многих случаях является неудобным с точки зрения их последующего представления в программе в виде чисел определенного типа или в виде строк, то поток *System.in* предварительно “оборачивается” потоком другого, более подходящего типа. Смысл “обертывания” заключается в том, что ссылка на *System.in* используется не напрямую, а передается в конструктор потока другого, более подходящего класса, объект которого затем используется для чтения данных из консоли.

Для демонстрации совместной работы потоков стандартного ввода/вывода, а также использования классов, более удобных для чтения вводимой из консоли информации, приведем пример, который читает из консоли два числа, складывает их значения и выводит результат обратно в консоль. При чтении введенной информации в качестве обертки для *System.in* используется объект класса *java.io.BufferedReader*, позволяющий читать сразу строки символов, для чего используется его метод *readLine*, который срабатывает при нажатии пользователем в консоли клавиши <Enter>.



```

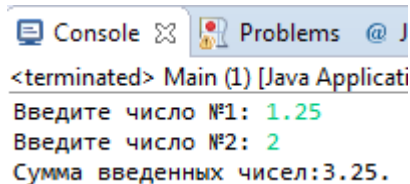
java.io.BufferedReader stdin;
String line;
double sum=0;
// Выполняется последовательное обертывание байтового потока чтения
// сначала в символьный поток чтения java.io.InputStreamReader,
// а затем в буферизованный символьный поток чтения java.io.BufferedReader
// для чтения данных сразу строками
stdin=new java.io.BufferedReader(new java.io.InputStreamReader(System.in));

for(int i=0; i<2; i++) {
    //Выводим запрос на чтение очередного числа
    System.out.print("Введите число №" + (i+1) + ": ");
    //Читаем введенную строку
    line = stdin.readLine();
    //Получаем число из введенной строки и обновляем сумму
    if (line != null)
        sum += Double.valueOf(line).doubleValue();
}

//Выводим сумму введенных чисел
System.out.println("Сумма введенных чисел:" + sum + ".");

```

Результат работы данного кода выглядит следующим образом:



```

Console Problems @ J
<terminated> Main (1) [Java Applicati
Введите число №1: 1.25
Введите число №2: 2
Сумма введенных чисел:3.25.

```

### 3 Базовое приложение для лабораторной работы

**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 путем 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

Базовое приложение в данной лабораторной работе включает:

- главный класс приложения **Main**, содержащий метод *main*, в котором обрабатываются входные аргументы командной строки программы, определяется набор продуктов завтрака и реализуется процедура их употребления;
- иерархию классов продуктов, используемых главным классом.

При создании классов, представляющих отдельные продукты, стоит учитывать, что все продукты имеют общие черты:

- все они относятся к еде, т.е. имеют некоторые общие свойства, характеризующие абстрактный продукт питания, среди которых можно выделить наличие имени;
- все они имеют общие черты поведения, среди которых в частности можно выделить их способность быть съеденными.

Исходя из выделенных общих свойств продуктов питания, которые обычно реализуются в виде полей, размещаемых в едином суперклассе для всех продуктов, а также общности в их поведении, которая реализуется в виде отдельных интерфейсов, можно предложить иерархию классов продуктов питания, представленную на рисунке 2.

Интерфейс **Consumable**, создан для представления способности некоторой вещи (предмета одежды, продукта и т.д.) быть употребленной. В данном интерфейсе определен абстрактный метод *consume*, который выполняет действие, связанное с употреблением предмета. Данный метод должен быть переопределен в классах, реализующих интерфейс **Consumable** и представляющих конкретные предметы, поскольку детали употребления того или иного предмета становятся доступными именно в рамках этих классов.

В основе иерархии продуктов питания находится класс **Food**, который представляет абстрактный продукт питания и предназначен для размещения полей, общих для всех продуктов, а также для определения их общего поведения. Данный класс объявляет поле данных *name*, предназначенное для хранения имени продукта питания, а также переопределяет унаследованные от класса **Object** методы *equals* и *toString*, реализации которых используют значение данного поля. Кроме того, поскольку любой продукт питания может

быть употреблен, то класс *Food* реализует интерфейс *Consumable*. В то же время, поскольку детали употребления продукта питания пока остаются неизвестными, метод *consume* не переопределяется в классе *Food* и остается абстрактным. По этой причине сам класс *Food* также объявлен как абстрактный класс и не может быть использован непосредственно для создания объектов.

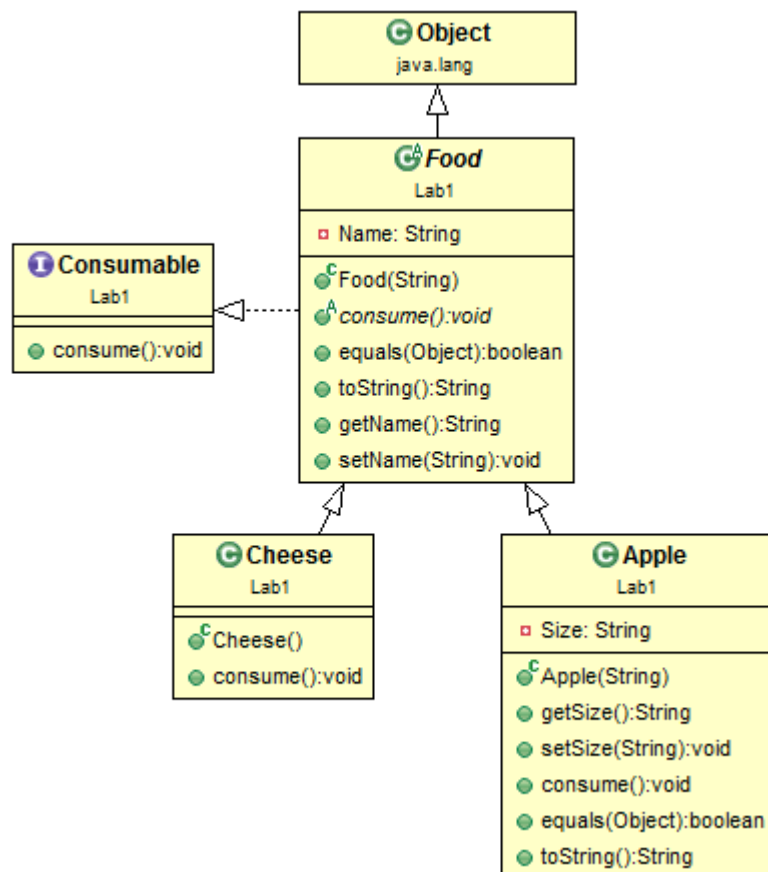


Рис 2. Иерархия классов и интерфейсов базового приложения для представления продуктов питания

Класс *Cheese* представляет продукт питания *сыр*. Поскольку данный класс не добавляет новых полей данных, то ему нет необходимости переопределять методы *equals* и *toString*, а нужно только реализовать метод *consume*.

Класс *Apple* представляет продукт питания *яблоко*. Данный продукт питания характеризуется дополнительным свойством – размером, что требует добавления нового поля *size*. Появление нового поля приводит к необходимости переопределения методов *equals* и *toString*. Для реализации способности яблока быть употребленным переопределяется метод *consume*.

### 3.2 Создание проекта программы в Eclipse

Для создания нового проекта необходимо использовать пункт «*File→New→Project*» главного меню приложения (рисунок 3), после активизации которого на экране появится окно выбора типа создаваемого проекта.

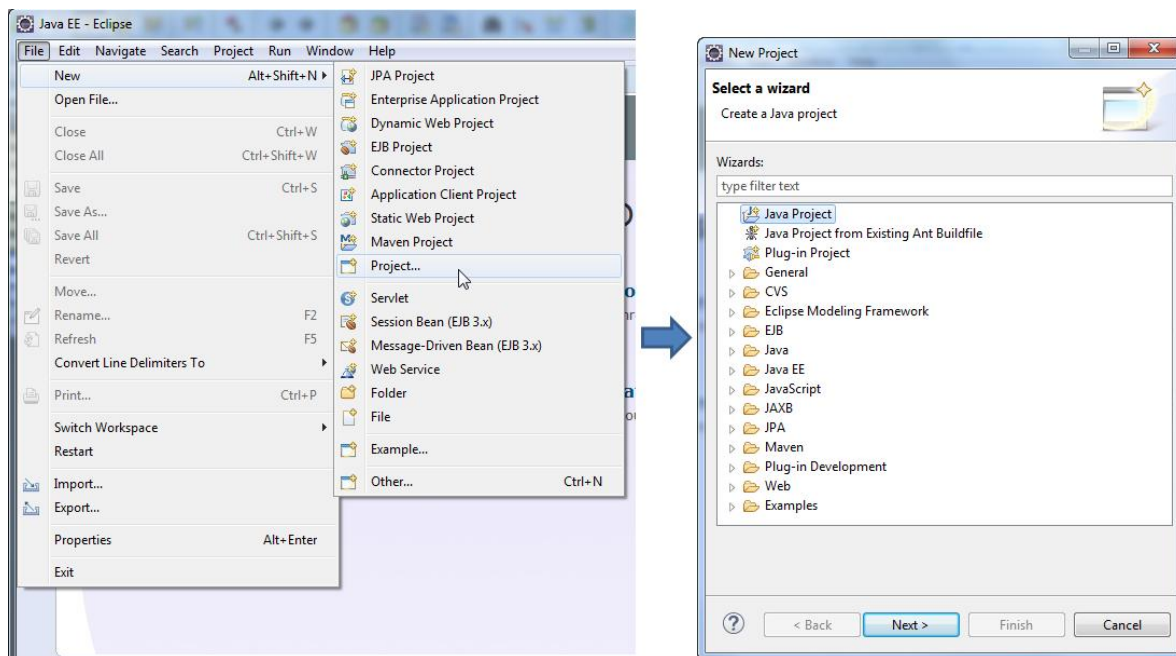


Рис 3. Получение окна выбора типа создаваемого проекта

В окне выбора типа проекта, необходимо выбрать «*Java Project*» и для продолжения нажать кнопку «*Next >*», после чего на экране появится окно настройки параметров, где необходимо указать название создаваемого проекта (рисунок 4), например, «Задание 1 – Консольное приложение».

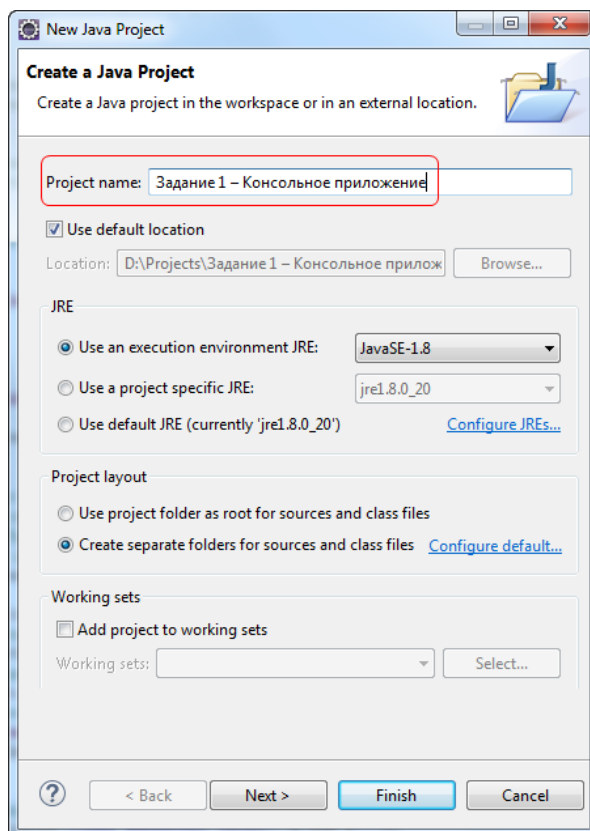


Рис 4. Окно настройки параметров проекта

Путь в папке, где будут располагаться файлы создаваемого проекта, отображается в поле «*Location*». В группе «*JRE*» указывается рабочее окружение Java, которое будет в дальнейшем использоваться для запуска разрабатываемого приложения. В группе «*Project layout*» указывается где именно будут храниться файлы, содержащие исходный код приложения, и файлы откомпилированных классов. В данной группе можно оставить включенной радио-кнопку «*Create separate folders for sources and class files*», которая указывает, что необходимо хранить исходные файлы и откомпилированные классы в отдельных папках. Нажатие кнопки «*Finish*», позволяет завершить процесс настройки и создания нового проекта. Для того, чтобы увидеть содержимое вновь созданного проекта в основном окне среды разработки Eclipse, следует закрыть стартовую страницу «*Welcome*», которая отображается по умолчанию после запуска “eclipse.exe”.

Содержимое созданного пока пустого проекта будет отображаться в виде «*Project explorer*» («*Project explorer*» view), пример которого представлен на рисунке 5.

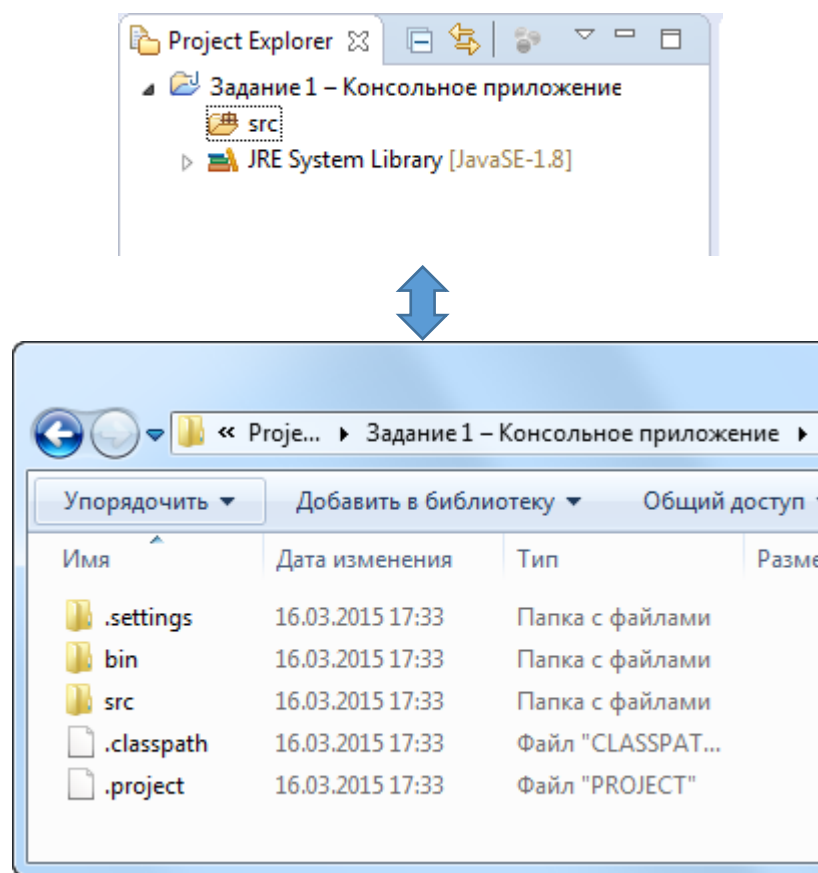


Рис 5. Представление содержимого Java-проекта в среде Eclipse и файловой системе

В папке «*Projects*», выбранной ранее для размещения проектов, была создана подчиненная папка «*Задание 1 – Консольное приложение*», в которой будут размещаться файлы созданного Java-проекта (рисунок 5). Файлы с исходными кодами будут размещаться в папке «*Src*», а файлы, содержащие откомпилированный код классов, в папке «*Bin*».

### 3.3 Создание пакета для классов

Любая программа на Java представляет собой набор взаимодействующих классов, принадлежащих различным пакетам. В случае если пакет не указан явно, то класс считается принадлежащим безымянному пакету по умолчанию. Тем не менее хорошим стилем считается явное указание пакета для каждого создаваемого класса. Для создания нового пакета классов необходимо щелкнуть правой кнопкой мыши на папке «*Src*» в виде «*Project explorer*» и в появившемся контекстном меню выбрать пункт «*New→Package*» (рисунок 6).

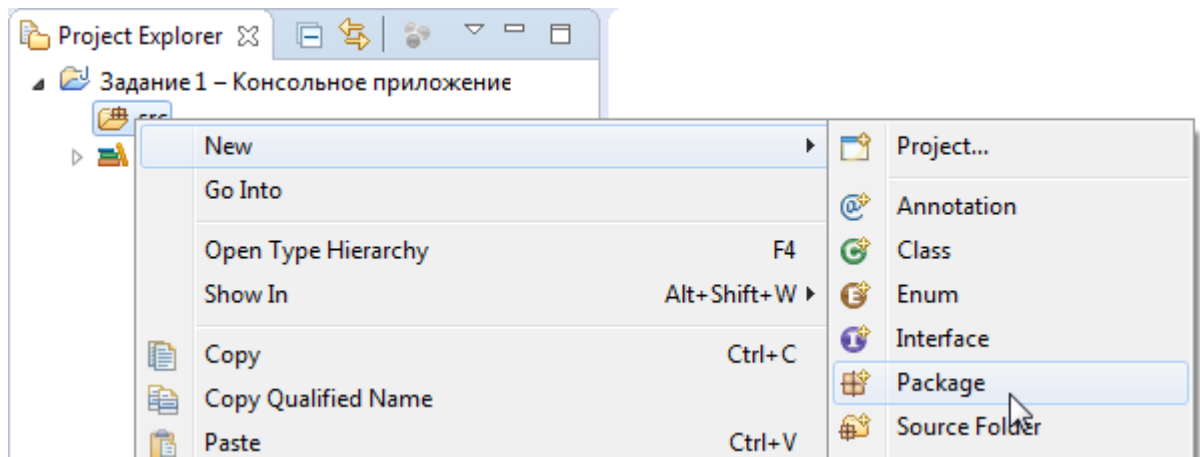


Рис 6. Создание в проекте нового пакета

Активизация данного пункта меню отобразит окно создания нового пакета, в котором необходимо ввести его имя (рисунок 7).

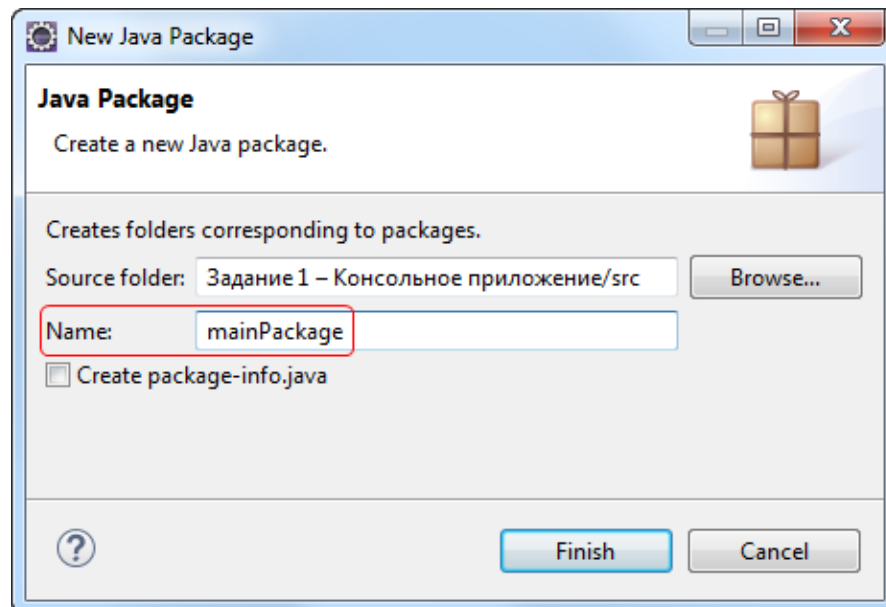


Рис 7. Окно создания нового пакета

Далее, после нажатия кнопки «*Finish*», будет создан новый пакет с введенным именем, который можно увидеть в виде «*Project explorer*». В папках «*Src*» и «*Bin*» проекта созданному пакету будут соответствовать одноименные подчиненные папки (рисунок 8).

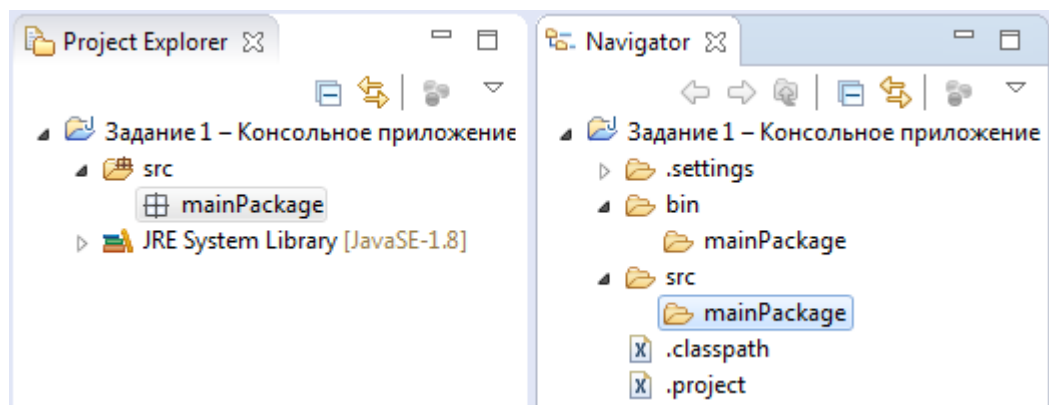


Рис 8. Созданный пакет «*mainPackage*» и соответствующие ему подчиненные папки

### 3.4 Создание общего интерфейса *Consumable*, используемого в иерархии классов

Для создания интерфейса *Consumable*, представляющего общую для продуктов способность быть употребленными, необходимо щелкнуть на имени пакета, в который добавляется интерфейс, и в контекстном меню выбрать «*New → Interface*». В появившемся диалоговом окне (рисунок 9) необходимо указать имя интерфейса («*Consumable*»), тип доступа к нему (*public*), а также задать расширяемый им набор базовых интерфейсов (нет).

После нажатия кнопки «*Finish*» в папке, соответствующей выбранному пакету (в случае рисунка 9 – *mainPackage*), будет создан новый файл с именем «*Consumable.java*».

Для добавления в созданный интерфейс метода *consume* необходимо открыть файл «*Consumable.java*» в редакторе исходного кода и дописать определение необходимого метода. В результате должен получиться код вида:

```
package mainPackage;

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



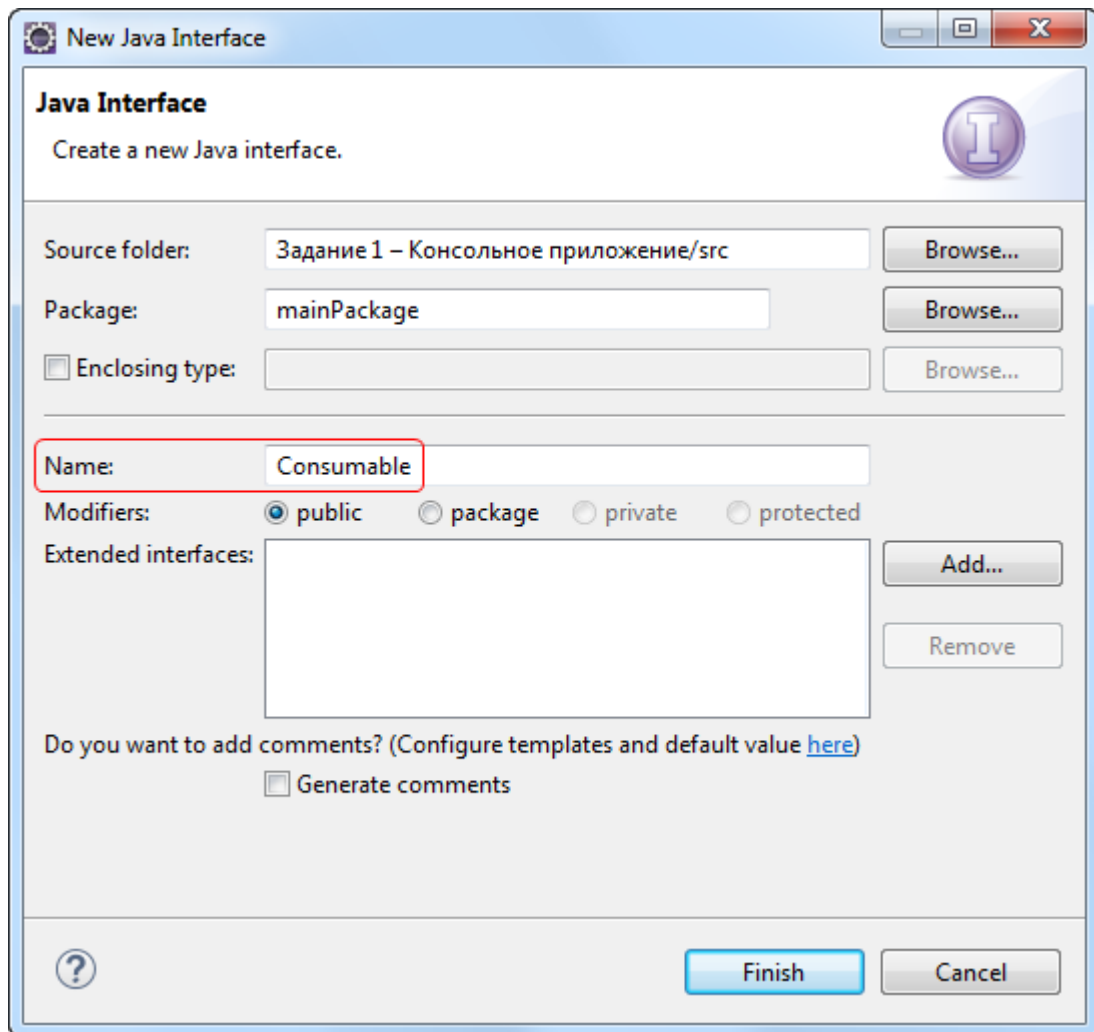


Рис 9. Окно создания нового интерфейса

### 3.5 Реализация базового класса иерархии продуктов *Food*

Для создания каркаса класса *Food* необходимо щелкнуть правой кнопкой мыши на имени пакета в виде «*Project explorer*» и в появившемся контекстном меню выбрать пункт «*New→Class*». (рисунок 10).

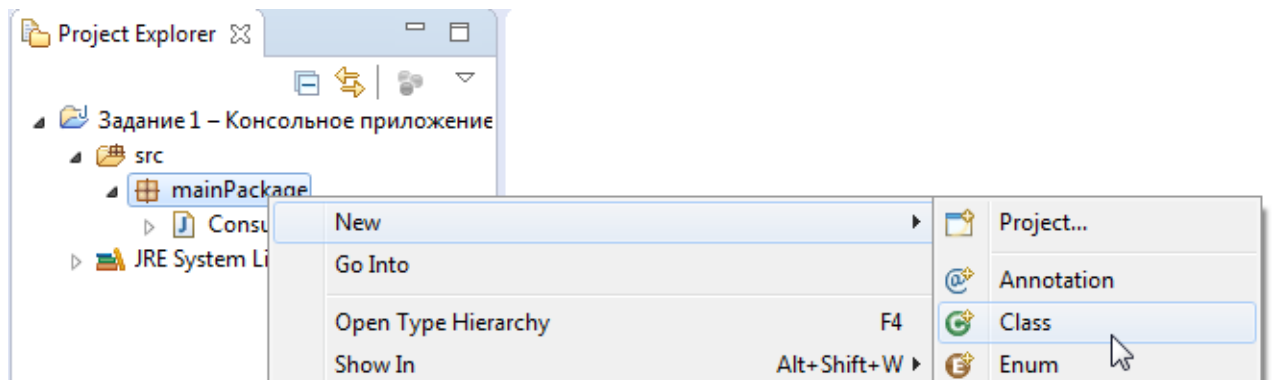


Рис 10. Добавление в пакет нового класса

Активизация данного пункта меню позволяет отобразить окно создания нового класса (рисунок 11).

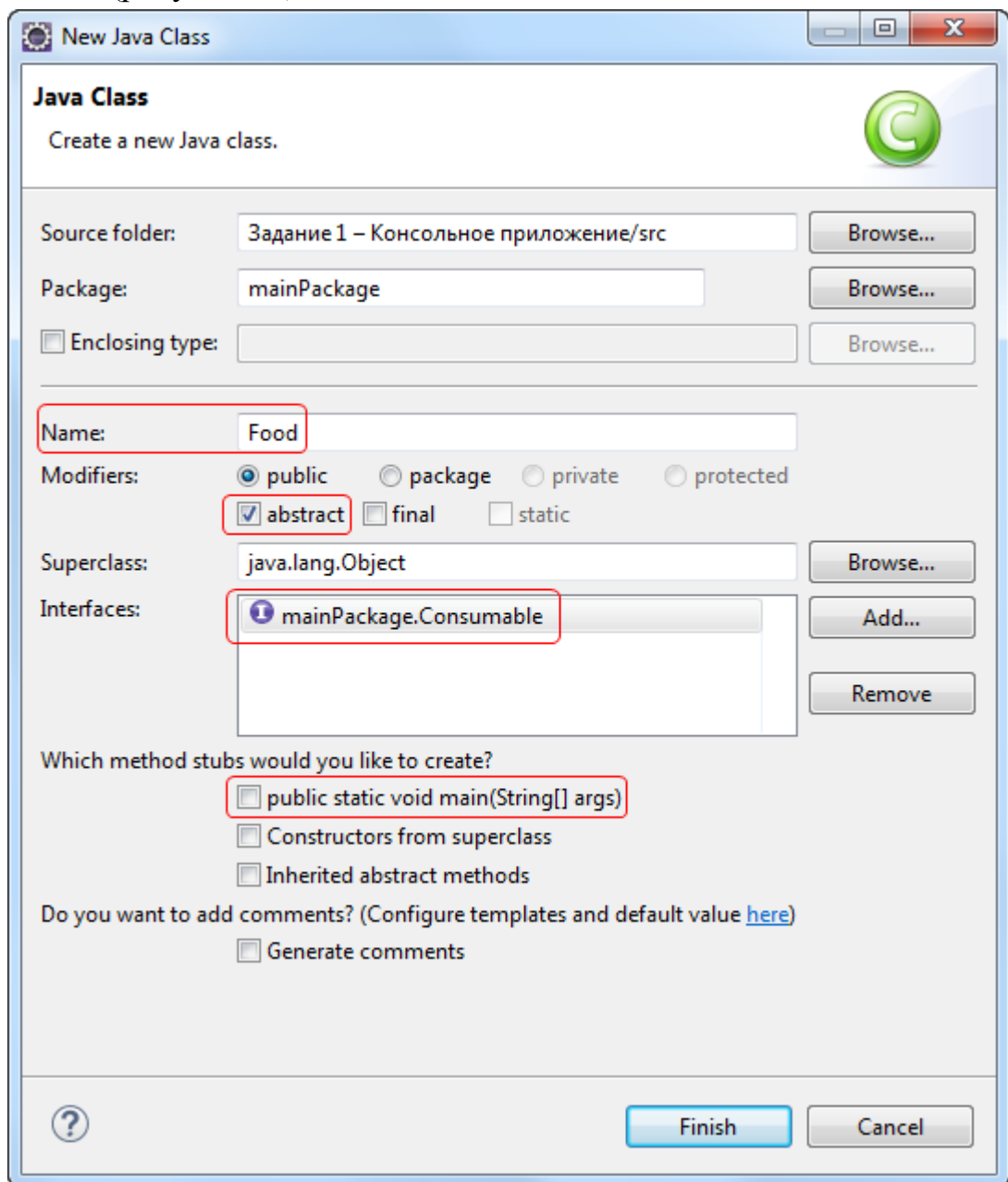


Рис 11. Установки в окне создания нового класса, необходимые для создания каркаса класса *Food*

В появившемся диалоговом окне необходимо указать:

- имя нового класса (поле «Name»);
- модификаторы, определяющие тип доступа к классу (*public*, *default*, *private*, *protected*);
- дополнительные модификаторы класса (*abstract*, *final*, *static*);
- класс-предок (поле «Superclass») (по умолчанию *java.lang.Object*);
- реализуемые классом интерфейсы (в поле «Interfaces»);

- необходимость создания метода *main* (поставить соответствующий флажок в группе «Which method stubs would you like to create?»);
- необходимость автоматической генерации прототипов унаследованных от суперкласса конструкторов (флажок «Constructors from superclass»);
- необходимость автоматической генерации прототипов унаследованных абстрактных методов (флажок «Inherited abstract methods»);
- необходимость генерации комментариев (флажок «Generate comments»).

Установки, которые необходимо выполнить в окне создания нового класса для класса *Food*, представлены на рисунке 11.

Сразу после нажатия кнопки «Finish» в диалоговом окне на рисунке 11, в папке «Src» будет создан новый файл с именем, соответствующим имени создаваемого класса, и расширением .java (в примере, представленном на рисунке 11, *Food.java*). Содержимое вновь созданного класса, а также обновленная информация в видах «Project explorer», «Navigator» и «Outline», представлены на рисунке 12.

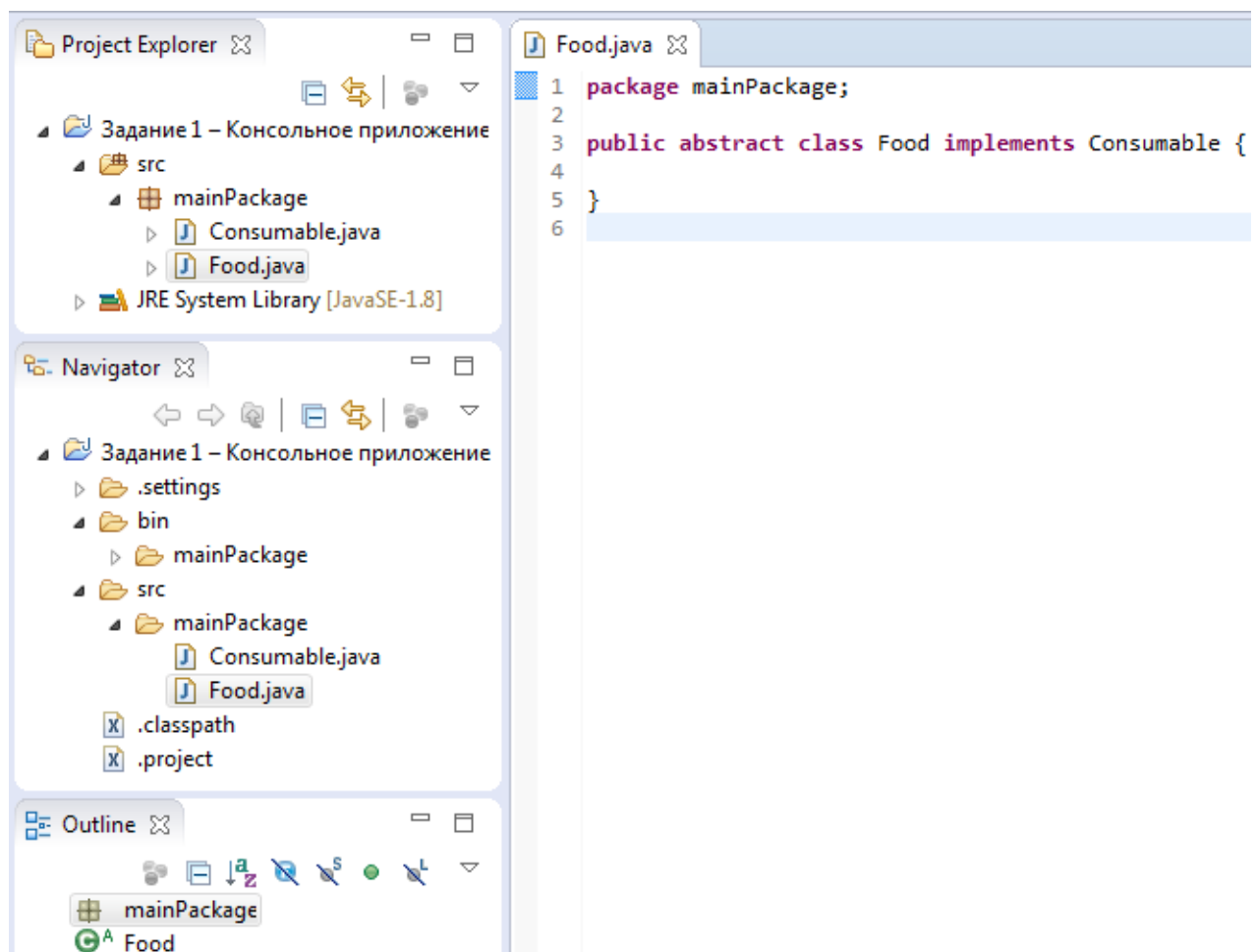


Рис 12. Представление созданного класса *Food* в среде Eclipse

Теперь перейдем к реализации базового класса *Food*. Для представления названия продукта, включаемого в завтрак, добавим в класс внутреннее поле

*name*, имеющее тип *String*. Для соблюдения принципа инкапсуляции объектно-ориентированной концепции программирования для поля *name* используем модификатор *private*, который делает его недоступным напрямую из других классов приложения. Код класса примет вид:

```
public abstract class Food implements Consumable {  
    private String name;  
}
```

Для работы с полем *name* из других классов приложения добавим два специальных метода:

- метод для получения значения поля, называемый «геттером»;
- метод для задания значения поля, называемый «сеттером».

В среде Eclipse пару таких методов для поля, уже определенного в классе, можно сгенерировать автоматически. Для этого перейдем в редактор исходного кода для файла «*Food.java*», щелкнем левой кнопкой мыши в любом месте в рамках блока кода, относящегося к определению класса *Food* (между соответствующими ему фигурными скобками) и активируем пункт меню «*Source* → *Generate Getters and Setters*», который откроет окно генерации методов (рисунок 13).

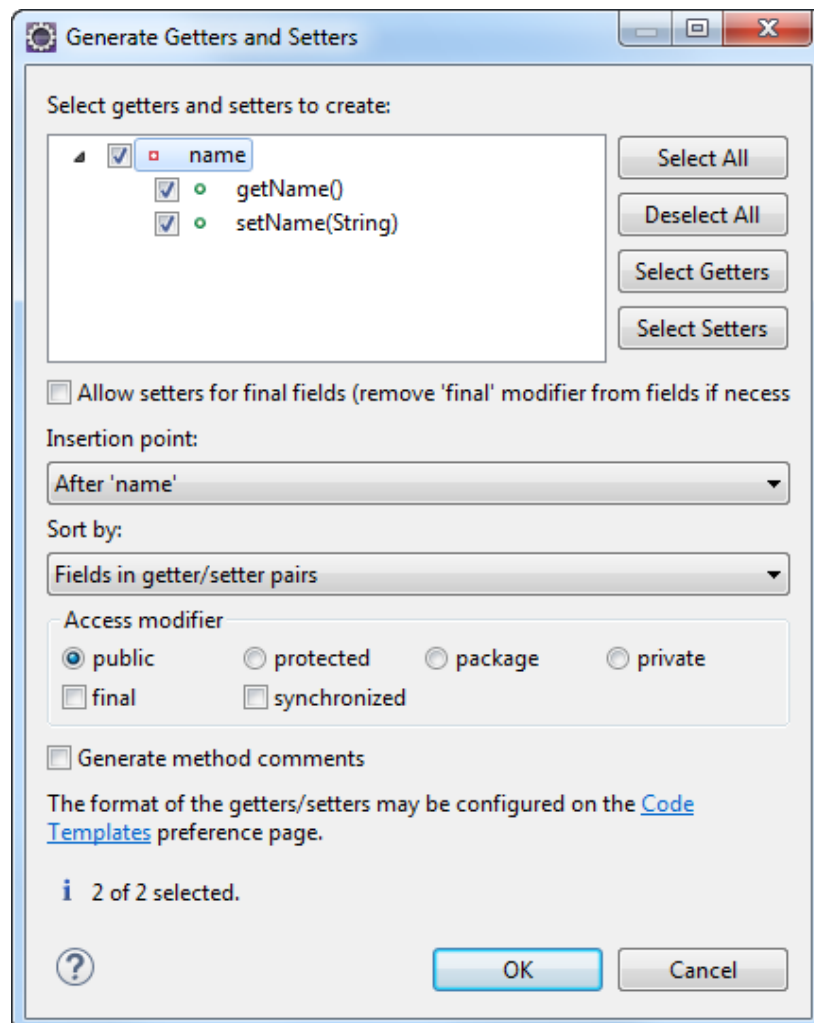


Рис 13. Диалоговое окно добавления геттеров и сеттеров

В появившемся диалоговом окне пометим флажком поля данных, для которых необходимо сгенерировать геттеры и сеттеры (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 модификаторы для создаваемых методов (*public*). Далее нажмем кнопку «*Ok*». С учетом сгенерированных методов исходный код класса примет вид:

```
public abstract class Food implements Consumable {
    private String name;

    public String getName() {
        return name;
    }

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

Для того, чтобы отличать обращение к полю *name*, определенному в классе *Food*, и одноименному аргументу метода *setName*, при обращении к полю в данном методе используется ссылка *this*.

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

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

Переопределим в классе *Food* методы *toString* и *equals*, унаследованные от базового класса *Object*. Для этого активируем пункт меню «*Source → Override/Implement Methods*» и в появившемся диалоговом окне (рисунок 14) отметим флажками соответствующие методы, а также зададим место вставки нового кода. После нажатия кнопки «*Ok*» и создания каркасов соответствующих методов, переопределим их содержимое, чтобы оно имело вид:

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

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

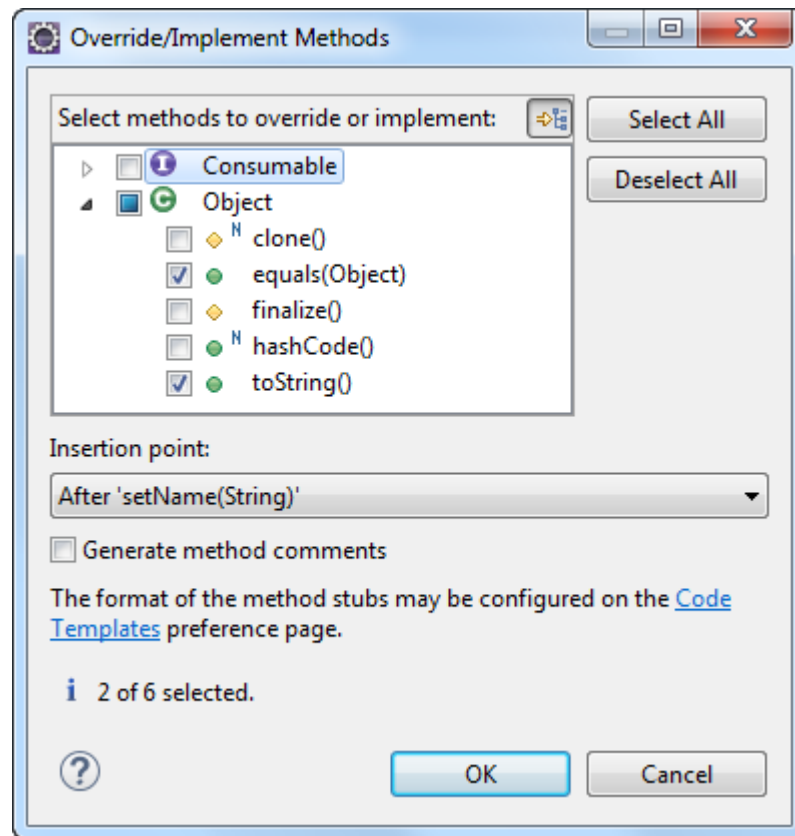


Рис 14. Диалоговое окно перегрузки/реализации методов

Рассмотрим более детально содержимое метода *equals*, предназначенного для сравнения двух продуктов. Проверка равенства двух объектов продуктов реализована как последовательность из трех 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** проверка равенства внутреннего поля *name* (являющегося ссылкой) константе *null* позволяет определить, полностью ли сконструированы вызывающий объект (равенство которого проверяется) и объект *arg0* (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** проверка равенства поля *name* у обоих объектов определяет результат всей операции (равенство или неравенство объектов).

Окончательный код реализованного класса *Food*:

```

package mainPackage;

public abstract class Food implements Consumable {
    private String name;

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

    public String getName() {
        return name;
    }

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

    public String toString() {
        return 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
    }
}

```

### 3.6 Реализация классов *Apple* и *Cheese*

Для представления продукта «яблоко» создадим класс *Apple*. Данный класс будет унаследован от класса *Food* и будет отличаться от него следующим:

1. не будет являться абстрактным и будет позволять создавать его объекты;
2. будет содержать дополнительное поле данных *size*, характеризующее размер яблока (например, малое, среднее, большое);
3. будет содержать реализацию метода *consume*, отражающую то, как яблоко употребляется;
4. объекты для яблока будут считаться равными только в том случае, если их размеры равны (необходимо переопределить метод *equals*);
5. строковое представление яблока должно учитывать информацию о его размере (необходимо переопределить метод *toString*).

Создание каркаса класса *Apple* выполняется в последовательности, аналогичной созданию каркаса класса *Food*: щелкнуть правой кнопкой мыши на имени пакета; активировать пункт меню «*New → Class*»; указать имя суперкласса *Food*; отметить флажок «*Inherit abstract methods*» (так как



необходимо реализовать нереализованный ранее метод *consume*). Исходный код заготовки созданного класса будет иметь следующий вид:

```
package mainPackage;

public class Apple extends Food {
    @Override
    public void consume() {
        // TODO Auto-generated method stub
    }
}
```

Выполним реализацию класса, которую начнем с добавления в класс нового поля данных *size*. Поскольку значения данного поля предполагается задавать в строковом виде (малое, среднее, большое), то в качестве типа для данного поля используем *String*:

```
private String size;
```

С помощью ранее описанной процедуры создадим для этого поля «геттер» и «сеттер»:

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

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

Добавим конструктор класса *Apple*, который будет принимать в качестве аргумента размер яблока, который далее будет записываться в поле *size*. Кроме того, для задания имени продукта, будет вызываться конструктор класса *Food*, куда будет передаваться значение “Яблоко”:

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

Определим метод *consume*, отвечающий за употребление яблока в процессе завтрака, что будет выражаться в отображении в консоли строкового представления объекта, после чего будет добавлено слово «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* (пока унаследованный от класса *Food*) для преобразования объекта в строку. Результат вывода:

Яблоко съедено

Для учета в строковом представлении объекта типа *Apple* значения поля *size*, переопределим метод *toString*:

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

При использовании приведенной выше реализации метода *toString*, метод *consume* будет выводить в консоль строку, содержащую информацию о размере яблока, например:

Яблоко размера 'БОЛЬШОЕ' съедено

При реализации метода *equals* необходимо в первую очередь выполнить проверки, уже реализованные в классе *Food*, которые гарантируют, что сравнивается два продукта, которые имеют в поле *name* значение “Яблоко”. Если это не так, то сразу можно считать, что вызывающий объект и объект переданный в качестве аргумента метода *equals* не равны и вернуть в качестве результата значение *false*:

```
if (!super.equals(arg0)) return false;
```

Если метод *equals* класса *Food* сообщил о равенстве объектов (было возвращено значение *true*), то имеет смысл продолжать сравнение объектов, анализируя значения их полей *size*. Для этого необходимо в первую очередь убедиться, что объект, переданный в качестве аргумента *arg0*, принадлежит классу *Apple* и, соответственно, имеет поле *size*. С этой целью необходимо добавить в метод *equals* проверку вида:

```
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);
```

Полностью код метода *equals* выглядит следующим образом:

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

Создание и реализация класса *Cheese*, представляющего продукт сыр выполняется аналогично, причем данный класс не будет иметь дополнительных полей, а, следовательно, будет включать лишь конструктор и реализацию метода *consume*.

### 3.7 Реализация главного класса приложения

Создание каркаса главного класса приложения *Main* выполняется аналогично ранее рассмотренным классам для продуктов. Основным отличием является то, что данный класс содержит статический общедоступный метод *main*, получающий в качестве аргумента массив строк (соответствующий параметрам командной строки). Данный метод является точкой входа в Java-приложение и будет вызван первым при его запуске на выполнение. Установки, которые необходимо выполнить для класса Main в окне создания нового класса, приведены на рисунке 15.

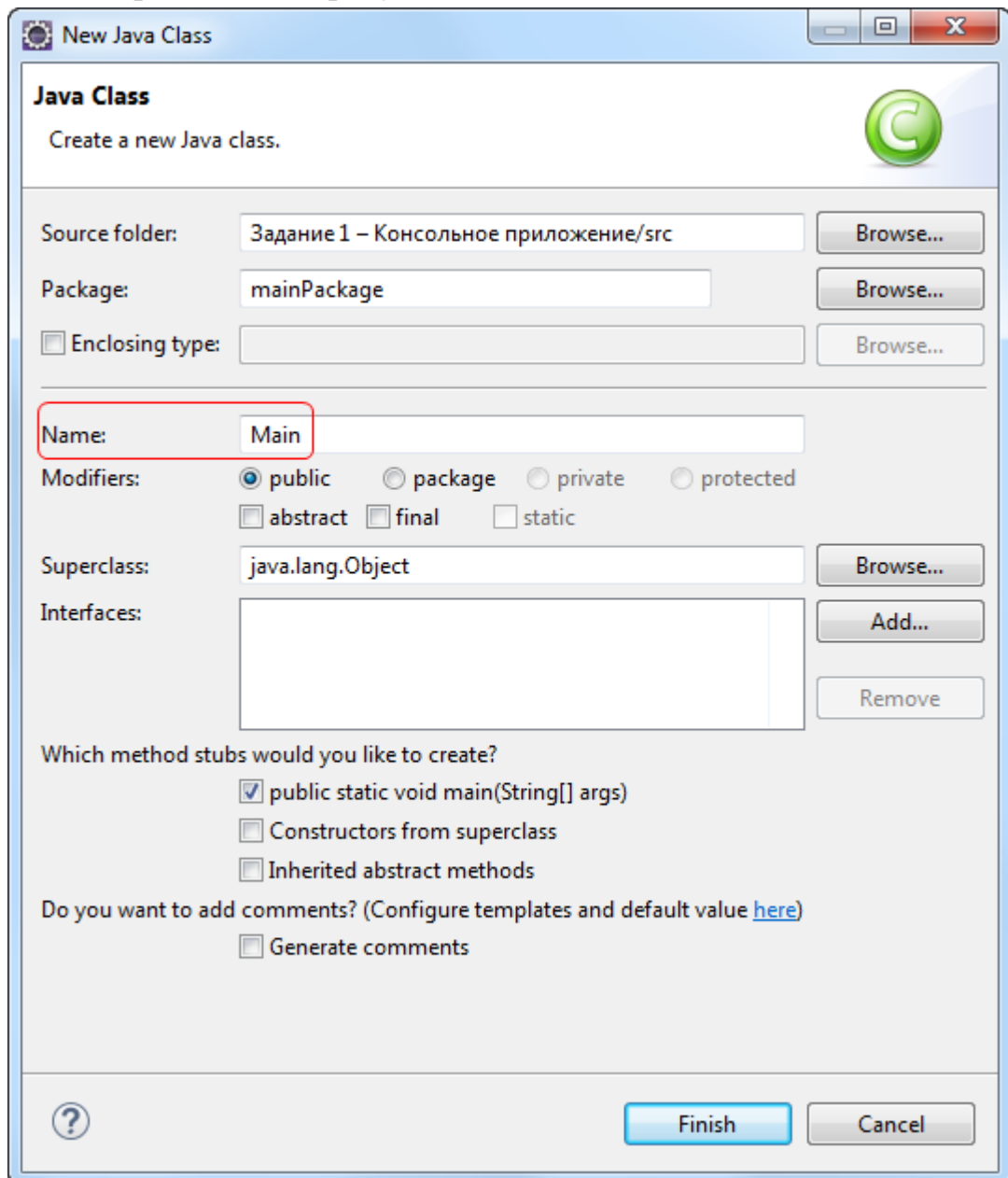


Рис 15. Установки в окне создания нового класса, необходимые для создания каркаса класса *Main*

После нажатия кнопки «*Finish*» в окне создания нового класса, в папке «*Src*» будет создан новый файл с именем *Main.java*, содержащий следующий код:

```
package mainPackage;

public class Main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Задачами класса *Main* являются:

1. анализ аргументов командной строки (в качестве них передаются продукты, включаемые в завтрак);
2. построение массива объектов для указанных продуктов;
3. проведение некоторых операций над этим массивом;
4. организация процедуры завтрака (употребления продуктов).

Программный код для решения этих задач будет добавлен в метод *main*.

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];
```

Стоит отметить, что при создании массива происходит выделение памяти под хранение адресов объектов типа *Food*, в то время как резервирование памяти для самих этих объектов при этом не происходит!

Проанализируем теперь аргументы командной строки (передаваемые в виде массива строк *String[] args*) и создадим для них объекты соответствующих классов продуктов. Для перебора массива введенных аргументов будем использовать цикл *for* вида:

```
for(int itemIndex=0; itemIndex<args.length; itemIndex++) {
    //Код для обработки отдельного аргумента,
    //ссылка на который расположена в переменной arg
    ...
}
```

В общем случае, отдельный элемент массива *args* 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»). По этой причине каждый отдельный элемент массива *args* необходимо сначала разбить на компоненты, используя в качестве разделителя символ «/». Для этого можно использовать метод *split* объекта класса *String*:

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

Дальнейшая обработка требует для каждого элемента массива *args* проанализировать имя класса (элемент массива *parts* с индексом 0) и определить количество его параметров (число элементов массива *parts* за исключением элемента с индексом 0), а затем создать объект указанного класса с занесением ссылки на него в массив *breakfast*. Эта операция может выполняться по-разному. Способ ее реализации для заданий уровней сложности А и В описан в разделе 3.7.1, а заданий уровня сложности С – в разделе 3.7.2. Сортировка массива (необходимая на уровнях сложности В, С) описана в разделе 3.7.3. Обработка объектов массива описана в разделе 3.7.4.

### 3.7.1 Создание экземпляров классов с помощью сравнения строк

Данный способ предполагает, что имена всех классов продуктов и соответствующее им число параметров известны. Выбор класса для создания объекта некоторого продукта осуществляется с помощью цепочки связанных блоков *if-else if*:

```
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 if (...) {  
    ...  
}
```

В этом случае, для каждого класса продукта должен быть реализован соответствующий отдельный блок *if*.

### 3.7.2 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("mainPackage." + 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 учета этого необходимо либо окружить рассматриваемый код блоком *try-catch* вида:

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

**Замечание:** в общем случае, попытка обнаружения конструктора с заданным числом и типом аргументов может закончиться неудачей – исключительной ситуацией типа *NoSuchMethodException*. Для учета 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 такого типа.

### 3.7.3 Сортировка массива

Для операций над массивами в библиотеке Java есть специальный класс *Arrays*, содержащий ряд статических методов обработки. Для сортировки



массива по заданному критерию будем использовать метод *sort(T[], Comparator c)*. Интерфейс *Comparator* определяет метод *compare(Object o1, Object o2)*, возвращающий отрицательное целое, если объект *o1* меньше *o2*; ноль, если они равны; положительное целое, если *o1* больше *o2*. Существует два способа определения класса-компаратора, представленные ниже.

**Определение класса-компаратора в виде отдельного класса** (для заданий варианта В). При таком способе определения класса-компаратора следует добавить в проект новый класс, реализующий интерфейс *Comparator*, и реализовать в нем метод *compare*. Ниже приведен пример такого класса, позволяющего сортировать массив объектов продуктов по содержимому их поля *name* в алфавитном порядке:

```
package mainPackage;

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. перемещается в начало массива
        if (arg1==null) return -1;
        // если оба объекта не null, то результат сравнения определяется
        // сравнением их полей name
        return arg0.getName().compareTo(arg1.getName());
    }
}
```

Сортировка массива *breakfast* с помощью определенного компаратора выполняется следующим образом:

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

**Определение класса-компаратора в виде анонимного класса** (для заданий варианта С). В данном случае класс-компаратор является анонимным классом, который определяется непосредственно в месте использования. Его определение и применение для создания объекта-компаратора 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());
    }
});
```



**Замечание:** в данном примере оператор *new* создает объект анонимного класса (имя которого не указано в явном виде). Следующее за оператором *new* имя интерфейса *Comparator* указывает на то, что объявленный анонимный класс реализует данный интерфейс. Содержимое объявленного анонимного класса указано в фигурных скобках после записи *new Comparator()* и представляет собой в данном случае единственный метод *compare*.

Результат применения представленного анонимного класса аналогичен результату использования класса *FoodComparator*, определенного в явном виде.

### 3.7.4 Выполнение операций над элементами массива

Простейшей операцией над элементами массива продуктов, включенных в завтрак, является реализация их «употребления». Для выполнения данной операции необходимо организовать цикл по элементам массива *breakfast* с использованием одного из вариантов оператора *for*, либо оператора *while*.

**Применение цикла *for*, использующего индекс:**

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

**Применение специальной формы цикла *for* для работы с массивами:**

```
for (Food item: breakfast) {  
    if (item==null) break;  
    item.consume();  
}
```

**Применение цикла *while*:**

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

Во всех циклах, приведенных выше, выполняется проверка элементов массива *breakfast* на равенство *null*, поскольку число продуктов, заданных аргументами командной строки может быть меньше длины массива *breakfast*. Если найден элемент равный *null*, то это означает что созданные объекты продуктов закончились и поэтому выполняется выход из цикла при помощи оператора *break*.

## 3.8 Запуск приложения

Полные версии исходных кодов классов базового приложения данной лабораторной работы (без реализации заданий, предложенных в пункте 4) размещены в Приложении 1.

Для задания аргументов командной строки необходимо определить конфигурацию запуска приложения. Для этого следует активировать пункт меню «Run → Run Configuration», в открывшемся диалоговом окне (рисунок 16) в списке слева выбрать пункт «Java Application» и нажать кнопку «New launch configuration» (первая слева кнопка на панели инструментов, расположенной в левой части окна над списком).

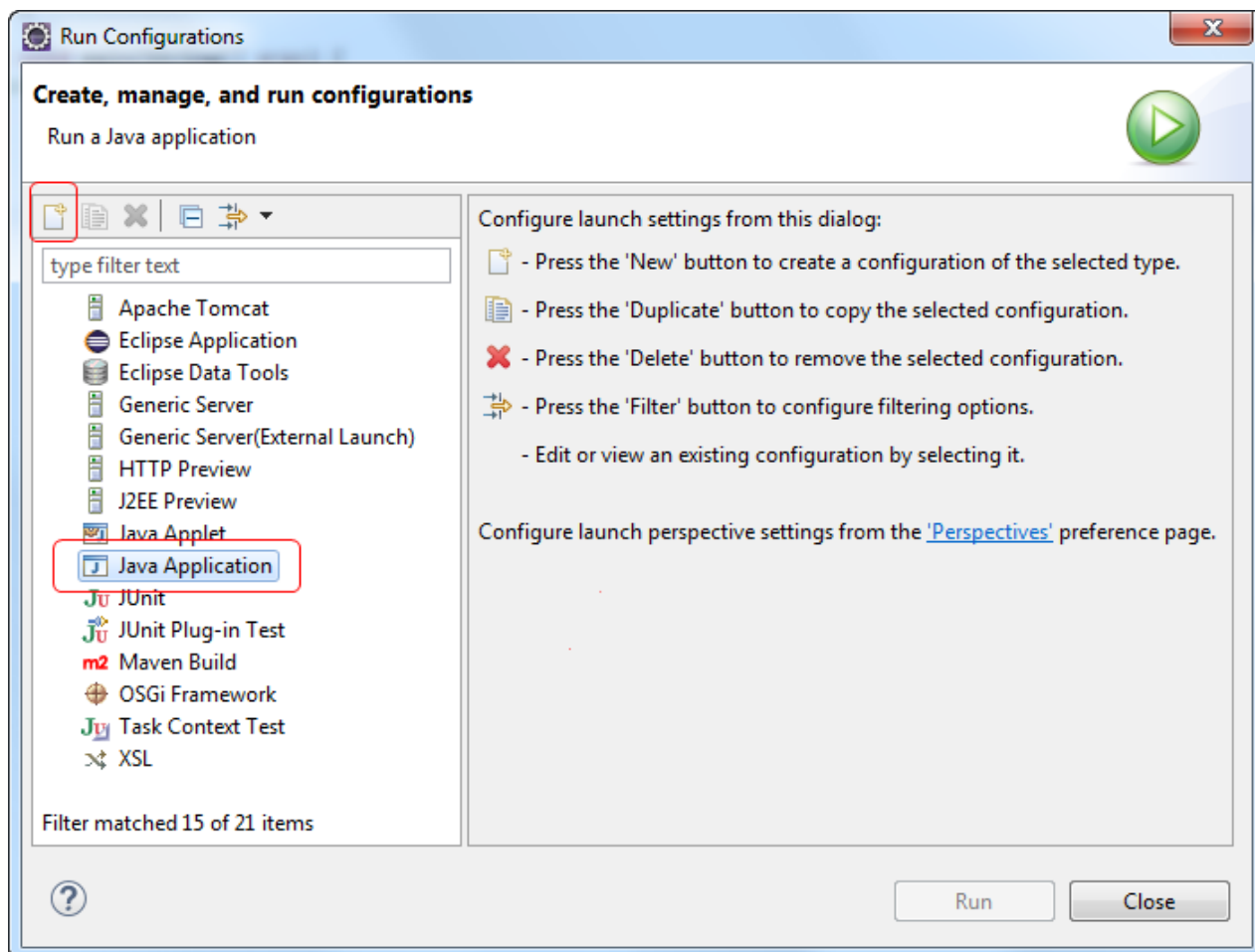


Рис 16. – Создание новой конфигурации запуска приложения

После создания новой конфигурации запуска приложения перейти на вкладку «Arguments», и в поле ввода «Program arguments» указать перечень аргументов (рисунок 17). После этого следует нажать кнопки «Apply» и «Run».

**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/со сливками и шоколадом* будет передан методу *main* в виде четырех отдельных строк {“IceCream/со”, “сливками”, “и”, “шоколадом”}, в то время как “IceCream/со сливками и шоколадом” будет передан одной строкой вида {“IceCream/со сливками и шоколадом”}.

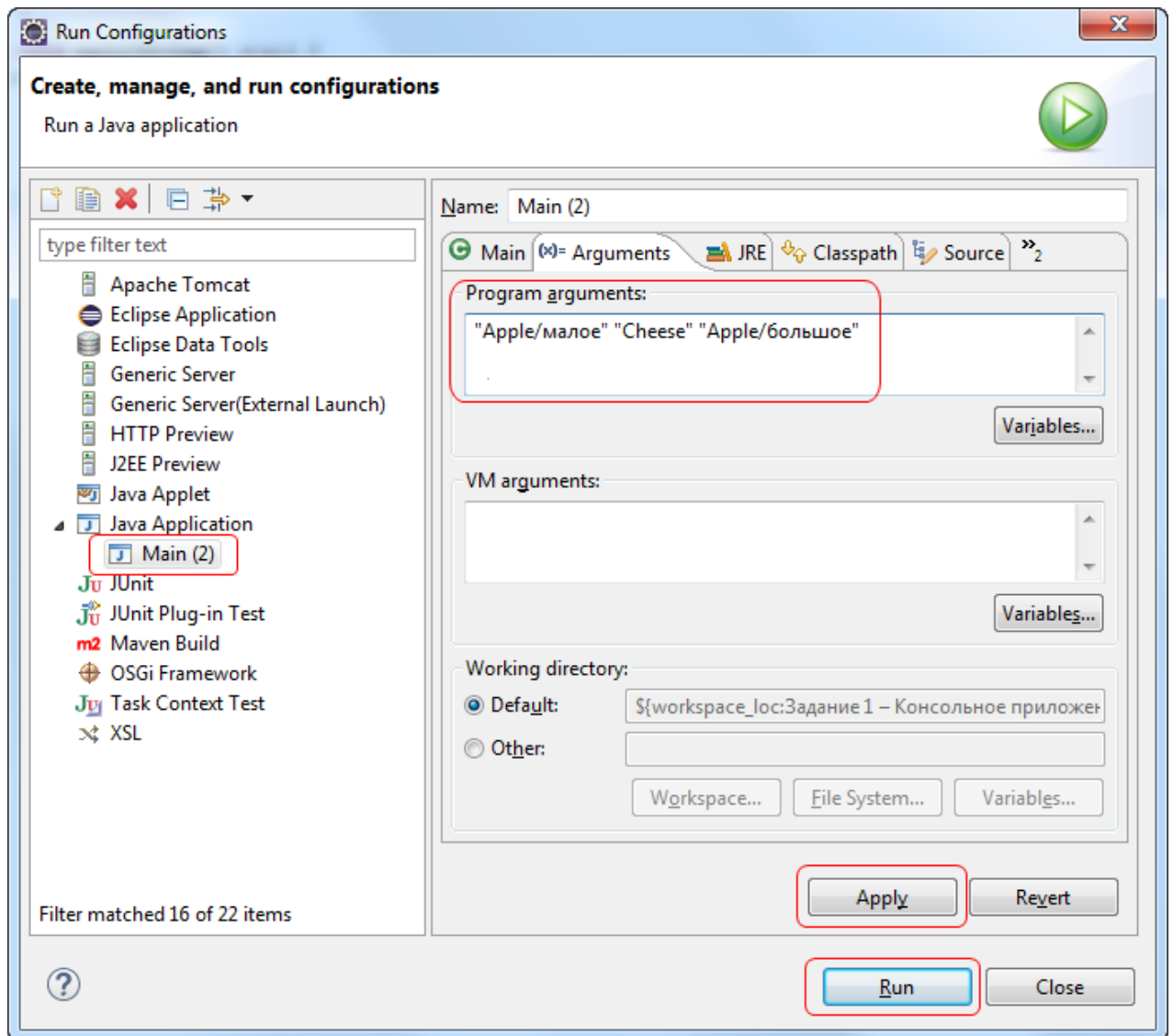


Рис 17. – Задание аргументов командной строки в диалоговом окне настройки конфигурации запуска

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

## 4 Задания

### 4.1 Вариант сложности А

- а) Реализовать (по вариантам) класс продукта (смотри таблицу 8) с одним параметром, подключить его в основную программу наряду с классами *Apple* и *Cheese*.
- б) Реализовать процедуру подсчета в завтраке продуктов заданного типа без сравнения по внутренним полям (использовать метод *equals*, определенный в классе *Food*, без его переопределения в классах-потомках для сравнения объектов массива *breakfast*).

Таблица 8 Классы продуктов для реализации по вариантам (для уровня сложности А)

№ п/п	Класс продукта	Параметр	Значение параметра
1	Tea (чай)	color (цвет)	черный, зеленый
2	Pie (пирог)	filling (начинка)	вишневая, 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 (прикус)	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 сложности В

- а) Реализовать (по вариантам) класс продукта (смотри таблицу 9) с одним параметром, подключить его в основную программу наряду с классами *Apple* и *Cheese*.
- б) Реализовать процедуру подсчета в завтраке продуктов заданного типа со сравнением по внутренним полям (использовать методы *equals*, переопределенные в классах конкретных продуктов).
- в) Определить отдельный интерфейс *Nutritious* (Питательный), содержащий метод *int calculateCalories()*, предназначенный для вычисления калорийности продукта. Для каждого из классов продуктов реализовать данный интерфейс и в зависимости от значений параметров вычислять калорийность.
- г) Реализовать обработку специальных параметров (начинающихся с дефиса), которые могут быть заданы наряду с продуктами как параметры

командной строки. 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 в порядке, указанным в таблице 9 (по вариантам).

Таблица 9 Классы продуктов для реализации по вариантам (для уровня сложности В)

№ п/п	Класс продукта	Параметр	Значение параметра	Сортировка
1	Tea (чай)	color (цвет)	черный, зеленый	По длине строкового представления объекта в прямом порядке
2	Pie (пирог)	filling (начинка)	вишневая, клубничная, яблочная	По калорийности в прямом порядке
3	Milk (молоко)	fat (жирность)	1.5%, 2.5%, 5%	По длине строкового представления объекта 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 строкового представления объекта 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 строкового представления объекта в прямом порядке
12	Beef (мясо)	preparedness (готовность)	с кровью, норма, прожаренное	По калорийности в обратном порядке

### 4.3 Вариант сложности С

- а) Реализовать (по вариантам) класс продукта (смотри таблицу 10 ниже) с двумя параметрами, подключить его в основную программу наряду с классами *Apple* и *Cheese* используя Reflection API.

- б) Реализовать процедуру подсчета в завтраке продуктов заданного типа со сравнением по внутренним полям (использовать методы *equals*, переопределенные в классах конкретных продуктов).
- в) Определить отдельный интерфейс *Nutritious* (Питательный), содержащий метод *int calculateCalories()*, предназначенный для вычисления калорийности продукта. Для каждого из классов продуктов реализовать данный интерфейс и в зависимости от значений параметров вычислять калорийность.
- г) Реализовать обработку специальных параметров (начинающихся с дефиса), которые могут быть заданы наряду с продуктами как параметры командной строки. 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 неопределенного 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 в порядке, указанным в таблице 10 (по вариантам).

Таблица 10 Классы продуктов для реализации по вариантам (для уровня сложности С)

№ 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 базового приложения

### Интерфейс *Consumable*

```
package mainPackage;

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

### Базовый класс для продуктов *Food*

```
package mainPackage;

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;
    }
}
```

### Класс *Cheese*

```
package mainPackage;

public class Cheese extends Food {

    public Cheese() {
        super("Сыр");
    }

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



## Класс *Apple*

```
package mainPackage;

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");
    }

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

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

    // Переопределенная 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)) return false; // Шаг 1
        if (!(arg0 instanceof Apple)) return false; // Шаг 2
        return size.equals(((Apple)arg0).size); // Шаг 3
    }

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

## Главный класс приложения *Main*

```
// Объявление класса частью пакета
package mainPackage;

public class Main {

    // Конструктор класса отсутствует!!!
    // Главный метод главного класса
    public static void main(String[] args) throws Exception {

        //Создание массива ссылок на продукты завтрака
        Food[] breakfast = new Food[20];
    }
}
```

```

// Анализ аргументов командной строки и создание для них
// экземпляров соответствующих классов для завтрака
for(int itemIndex=0; itemIndex <args.length; itemIndex++) {
    String[] parts = args[itemIndex].split("/");
    if (parts[0].equals("Cheese")) {
        // У сыра дополнительных параметров нет
        breakfast[itemIndex] = new Cheese();
    } else if (parts[0].equals("Apple")) {
        // У яблока - 1 параметр, который находится в parts[1]
        breakfast[itemIndex] = new Apple(parts[1]);
    }
    // ... Продолжается анализ других продуктов для завтрака
}

// Перебор всех элементов массива
for (Food item: breakfast) {
    // 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
    if (item==null) break;
    // Если элемент не null - употребить продукт
    item.consume();
}
System.out.println("Всего хорошего!");
}
}

```