Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов специальности 1 – 40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Минск 2015

Лабораторная работа №1

Построение кроссплатформенного графического интерфейса

Краткие теоретические сведения:

Введение

1. Swing
2. SWT
3. JavaFX

Введение

Современным программам необходим графический интерфейс пользователя (GUI). Пользователи совсем отвыкли использовать консоль: управление программой и ввод входных данных осуществляется посредством визуальных компонентов, к которым относятся текстовые поля, кнопки, выпадающие списки и т.д. Также ввиду интенсивного развития и роста популярности многих операционных систем возникла необходимость реализовывать программы под каждую из них. Через некоторое время стало очевидным, что неэффективно создавать программы отдельно под каждую операционную систему, то есть программы должны были приобрести свойство кроссплатформенности. Но одновременно с желанием сократить время разработки приложения возникает проблема различия состава визуальных компонентов в различных операционных системах.

Для решения этой проблемы появился по праву первый GUI framework: Abstract Window Toolkit. Идея была верная – AWT использует родные элементы интерфейса каждой операционной системы, то есть, они выглядят и физически являются стандартными, в независимости от того, где запускается приложение. К сожалению, оказалось, что общих для различных окружений элементов интерфейса мало, и составлять кроссплатформенные и одновременно родные интерфейсы так, чтобы всё отображалось корректно (не съехало во всех степенях свободы), очень сложно.

Для решения этой проблемы корпорация Sun начала разработку нового framework-а. В результате появилась очень мощная библиотека Swing, включающая большое количество компонент. Компоненты Swing в отличие от AWT не зависят от widget'ов операционной системы. За прорисовку и поведение GUI элементов полностью отвечает библиотека Swing, что позволило создать GUI компоненты, которые выглядят одинаково и функционируют на различных операционных системах. Для каждой операционной системы были разработаны настройки внешнего вида и поведения (Look&Feel), которые максимально эмулировали компоненты конкретной операционной системы. Использование Look&Feel позволило создать привлекательные и красочные интерфейсы Java приложений.

Именно популярность языка Java привлекла разработчиков корпорации IBM. Его использование решило проблему создания множества различных версий одинаковых продуктов для разных операционных систем. Раньше, в процессе разработки визуального интерфейса для VisualAge/SmallTalk, в сотрудничестве компании Object Technology International, Inc. (OTI) и корпорации IBM была разработана библиотека для построения визуального интерфейса Common Widgets.

Тем временем IBM реализовала свой framework используемый в Eclipse: Standard Widget Toolkit. Как и в AWT, используются родные компоненты. SWT не входит в JDK и использует JNI, поэтому не очень соответствует идеологии Java «написано однажды, работает везде». Технически можно упаковать в пакет реализацию SWT для всех платформ, и тогда приложение станет полностью кроссплатформенным, но только до тех пор, пока не появится какая-нибудь новая операционная система.

Наиболее новым GUI framework-ом, развитием которого занимается Oracle, является JavaFX. Идеологически JavaFX похож на Swing, то есть, элементы интерфейса не

являются родными элементами ОС. Среди интересных особенностей JavaFX следует отметить ускорение на уровне оборудования, создание GUI при помощи CSS и XML (FXML), возможность использовать элементы интерфейса JavaFX-а в Swing-е, а также большое количество новых элементов управления, в том числе для рисования диаграмм и 3D.

Каждый современный язык программирования предоставляет множество библиотек для работы со стандартным набором элементов управления. Библиотека в программировании - это набор готовых классов и интерфейсов, которые предназначены для решения определенного круга задач.

Каждая библиотека предоставляет набор классов для работы со списками, кнопками, меню, окнами и так далее, но все эти классы проектируются по-разному: они имеют различный набор методов с разными параметрами, а значит, «перевести» программу с одной библиотеки на другую (к примеру, с целью увеличения быстродействия) не так-то просто. Это как перейти с одного языка программирования на другой: все языки делают одно и то же, но каждый из них обладает своим синтаксисом, своей программной структурой и своими многочисленными хитростями.

По этой причине каждому следует разобраться с одной из предложенных библиотек.

1. Swing

Каждая GUI-программа запускается в окне и по ходу работы может открывать несколько дополнительных окон. В библиотеке Swing описан класс `JFrame`, представляющий собой окно с рамкой и строкой заголовка (с кнопками «Свернуть», «Во весь экран» и «Закрыть»). Оно может изменять размеры и перемещаться по экрану.

Базовые конструкции для запуска примитивного приложения описаны ниже:

- Конструктор `JFrame()` без параметров создает пустое окно.
- Конструктор `JFrame(String title)` создает пустое окно с заголовком title.
- Метод `setSize(int width, int height)` устанавливает размеры окна. Если не задать размеры, окно будет иметь нулевую высоту независимо от того, что в нем находится и пользователю после запуска придется растягивать окно вручную. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.
- Метод `setDefaultCloseOperation(int operation)` позволяет указать действие, которое необходимо выполнить, когда пользователь закрывает окно нажатием на крестик. Обычно в программе есть одно или несколько окон, при закрытии которых программа прекращает работу. Для того чтобы запрограммировать это поведение, следует в качестве параметра operation передать константу EXIT_ON_CLOSE, описанную в классе `JFrame`.
- Метод `setVisible(boolean visible)` необходим по причине того, что когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается данный метод с параметром true. Если вызвать его с параметром false, окно снова станет невидимым.

Теперь мы можем написать программу, которая создает окно, выводит его на экран и завершает работу после того, как пользователь закрывает окно.

```
import javax.swing.*;
public class MyClass {
      public static void main (String [] args) {
            JFrame trialWindow = new JFrame("Пробное окно");
            trialWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            trialWindow.setSize(400, 300);
            trialWindow.setVisible(true);
      }
```

```
}
```

Следует обратить внимание, что для работы с большинством классов библиотеки `Swing` понадобится импортировать пакет javax.swing.*

Как правило, перед отображением окна, необходимо совершить гораздо больше действий, чем в этой простой программке. Необходимо создать множество элементов управления, настроить их внешний вид, разместить в нужных местах окна. Кроме того, в программе может быть много окон и настраивать их все в методе `main()` неудобно и неправильно, поскольку нарушает принцип инкапсуляции: держать вместе данные и команды, которые их обрабатывают. Логичнее было бы, чтобы каждое окно занималось своими размерами и содержимым самостоятельно. Поэтому классическая структура программы с окнами выглядит следующим образом:

Файл SimpleWindow.java:
```
public class SimpleWindow extends JFrame {
	SimpleWindow(){
		super("Пробное окно");
		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setSize(250, 100);
	}
}
```

Файл Program.java:
```
public class Program {
	public static void main (String [] args) {
		JFrame trialWindow = new SimpleWindow();
		trialWindow.setVisible(true);
	}
}
```

Из примера видно, что окно описывается в отдельном классе, являющемся наследником `JFrame` и настраивающее свой внешний вид и поведение в конструкторе. Метод `main()` содержится в другом классе, ответственном за управление ходом программы. Каждый из этих классов очень прост, каждый занимается своим делом, поэтому в них легко разбираться и легко сопровождать (т.е. совершенствовать при необходимости).

Обратите внимание, что метод `setVisible()` не вызывается в классе `SimpleWindow`, что вполне логично: за тем, где какая кнопка расположена и какие размеры оно должно иметь, следит само окно, а вот принимать решение о том, какое окно в какой момент выводится на экран – прерогатива управляющего класса программы.

Напрямую в окне элементы управления не размещаются. Для этого служит панель содержимого, занимающая все пространство окна. Обратиться к этой панели можно методом `getContentPane()` класса `JFrame`. С помощью метода `add(Component component)` можно добавить на нее любой элемент управления.

Например, кнопка описывается классом `JButton` и создается конструктором с параметром типа `String` – надписью. Добавить кнопку в панель содержимого нашего окна можно посредством следующих команд:

```
JButton trialButton = new JButton();
getContentPane().add(trialButton);
```

В результате получим окно с кнопкой. Кнопка занимает всю доступную площадь окна. Такой эффект полезен не во всех программах, поэтому необходимо изучить различные способы расположения элементов на панели. Элементы, которые содержат

другие элементы, называются контейнерами. Все они являются потомками класса `Container` и наследуют от него ряд полезных методов:

- Метод `add(Component component)` – добавляет в контейнер элемент component.
- Метод `remove(Component component)` – удаляет из контейнера элемент.
- Метод `removeAll()` – удаляет все элементы контейнера.
- Метод `getComponentCount()` – возвращает число элементов контейнера.

Кроме перечисленных в классе `Container` определено около двух десятков методов для управления набором компонентов, содержащихся в контейнере. Как видно, они похожи на методы класса-коллекции. Это неудивительно, так как контейнер является коллекцией, но коллекцией особого рода – визуальной. Кроме хранения элементов контейнер занимается их пространственным расположением и прорисовкой. В частности, он имеет метод `getComponentAt(int x, int y)`, возвращающий компонент, в который попадает точка с заданными координатами (координаты отсчитываются от левого верхнего угла компонента) и ряд других.

Наиболее часто используемый контейнер – класс `JPanel`. Панель `JPanel` – это элемент управления, представляющий собой прямоугольное пространство, на котором можно размещать другие элементы. Элементы добавляются и удаляются методами, унаследованными от класса `Container`. У каждой панели есть так называемый *менеджер размещения*, который определяет стратегию взаимного расположения элементов, добавляемых на панель. Его можно изменить методом `setLayout(LayoutManager manager)`. Но чтобы передать в этот метод нужный параметр, необходимо разобраться с существующими менеджерами:

- менеджер последовательного размещения `FlowLayout`;
- менеджер граничного размещения `BorderLayout`;
- менеджер табличного размещения `GridLayout`;
- менеджер блочного размещения `BoxLayout` и класс `Box`.

Также следует учитывать особенности выравнивания элементов. Выравнивание по левому краю по горизонтали принято по умолчанию. Для того чтобы установить выравнивание любого визуального компонента, используются методы `setAlignmentX(float alignment)` – выравнивание по горизонтали и `setAlignmentY(float alignment)` – выравнивание по вертикали. В качестве параметра проще всего использовать константы, определенные в классе `JComponent`. Для выравнивания по горизонтали служат константы LEFT_ALIGNMENT (по левому краю), RIGHT_ALIGNMENT (по правому краю) и CENTER_ALIGNMENT (по центру). Для выравнивания по вертикали –BOTTOM_ALIGNMENT (по нижнему краю), TOP_ALIGNMENT (по верхнему краю) и CENTER_ALIGNMENT (по центру).

Предполагается, что программист явно задаёт размер окна методом `setSize()`. Но когда используется какой-либо менеджер расположения, расставляющий элементы и изменяющий их размеры по собственным правилам, трудно сказать заранее, какие размеры окна будут самыми подходящими. Безусловно, наиболее подходящим будет вариант, при котором все элементы окна имеют предпочтительные размеры или близкие к ним. Если вместо явного указания размеров окна, вызвать метод `pack()`, они будут подобраны оптимальным образом с учетом предпочтений всех элементов, размещенных в этом окне.

2. SWT

Standard Widget Toolkit является библиотекой компонент для построения графического интерфейса пользователя. Первым Java framework-ом для построения графического интерфейса пользователя была созданная корпорацией Sun Microsystems библиотека AWT (Abstract Window Toolkit).

Данный framework использовал визуальные компоненты операционной системы. А так как требовалось обеспечить кроссплатформенность, то возникла проблема различия состава визуальных компонент в различных операционных системах (ОС). По этой причине часть полезных компонент была исключена из состава AWT. В терминологии большинства ОС такие компоненты называют Widget.

Имея большой штат разработчиков VisualAge/SmallTalk, корпорацией IBM был инициирован проект по разработке универсальной платформы для создания Java приложений Eclipse. В процессе разработки Eclipse при сотрудничестве IBM и OTI был разработан новый графический framework, который позволил использовать ранее накопленный опыт без переучивания персонала и упрощения портирования ранее созданных продуктов на новую платформу. Этот framework получил название Standard Widget Toolkit (SWT). В SWT, как и AWT максимально используются компоненты операционной системы. Но в отличие от AWT, отсутствующие в конкретной операционной системе компоненты не исключены, а эмулируются. В результате, была создана быстрая высокоэффективная кроссплатформенная библиотека компонент, которые выглядят и ведут себя как родные компоненты операционной системы. Что в свою очередь упрощает процесс обучения пользователей Java приложений, так как SWT программы не отличаются от обычных приложений.

Итак, создадим новый Java проект. Для этого выберем меню «File->New». Выберем «JavaProject» в дереве мастеров. В следующей закладке мастера проекта введем имя проекта «by.bsuir.swt.hello». В закладке «Libraries» страницы «Java settings» мастера создания проекта добавим требуемую библиотеку (эту операцию можно сделать потом, редактируя свойства проекта). Нажмем кнопку «Add Library» и выберем «Standart Widget Toolkit (SWT)». Если с данным пунктом возникнут проблемы, то, случается, что необходимо попробовать следующий вариант: вместо «Add Library», необходимо выбрать «Add External JARs», после чего найти в папке с plugins для eclipse файл org.eclipse.swt.windowing_system.jar. >. Пути к библиотеке swt.jar для различных платформ сведены в таблицу:

ОС	Путь к библиотеке SWT
win32	INSTALLDIR/eclipse/plugins/org.eclipse.swt.win32_3.0.0/ws/win32/swt.jar
gtk	INSTALLDIR/eclipse/plugins/org.eclipse.swt.gtk_3.0.0/ws/gtk/swt.jar
motif	INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_3.0.0/ws/motif/swt.jar
photon	INSTALLDIR/eclipse/plugins/org.eclipse.swt.photon_3.0.0/ws/photon/swt.jar
macosx	INSTALLDIR/eclipse/plugins/org.eclipse.swt.cocoa.macosx.x86_64_....jar

Для некоторых платформ требуется дополнительные библиотеки. Например, для GTK требуются swt.jar, swt-pi.jar и swt-mozilla.jar. Соответственно, все эти файлы должны быть добавлены в путь поиска библиотек. Также, для отладки или запуска stand-alone SWT Java приложений, нужно в редакторе «VM arguments» указать путь к родной библиотеке SWT (закладка «Arguments» панели параметров запуска приложения). Варианты настройки путей для различных платформ сведены в таблицу:

ОС	Строка параметров
win32	-Djava.library.path=INSTALLDIRpluginsorg.eclipse.swt.win32_3.0.0oswin32x86
linux gtk	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.gtk_3.0.0/os/linux/x86
linux motif	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_3.0.0/os/linux/x86
solaris motif	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_3.0.0/os/solaris/sparc
aix motif	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_3.0.0/os/aix/ppc
hpux motif	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_3.0.0/os/hpux/PA_RISC
photon qnx	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.photon_3.0.0/os/qnx/x86
macosx	-Djava.library.path=INSTALLDIR/eclipse/plugins/org.eclipse.swt.carbon_3.0.0/os/macosx/ppc

Создадим профиль отладки приложения. Для этого вызовем меню «Run->Debug». Добавим новую конфигурацию отладки «Java Application».

Создадим пакет «by.bsuir.swt» и добавим в него новый класс «SwtHello», который будет содержать следующий код:

```
package by.bsuir.swt;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class SwtHello {
	public static void main(String[] args) {
		Display display = new Display();
		Shell shell = new Shell(display);
		shell.setText("SWT Hello");
		shell.setSize(200, 100);
		shell.open();
		while (!shell.isDisposed()) {
			if (!display.readAndDispatch()) {
				display.sleep();
			}
		}
		display.dispose();
	}
}
```

В главной функции создаём объект класса Display для связи SWT с дисплеем операционной системы. Затем создаём окно программы посредством вызова конструктора класса Shell. Display и Shell классы являются ключевыми компонентами приложения, GUI интерфейс которого основан на SWT framework-е. Класс org.eclipse.swt.widgets.Shell является главным окном приложения. Класс org.eclipse.swt.widgets.Display отвечает за управление событиями приложения, шрифтами, цветами, межпоточные взаимодействия между интерфейсом и другими частями приложения. Каждое SWT-приложение должно включать в себя по крайней мере один Display и не менее одного экземпляра Shell.

Следующим этапом по приведённому примеру является обработка корректного закрытия окна с освобождением ресурсов.

Рекомендуемые ресурсы:

- http://www.eclipse.org/
- http://eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html
- http://www.eclipse.org/articles/swt-design-2/swt-design-2.html
- http://www.vogella.com/tutorials/SWT/article.html

3. JavaFX

Прежде, чем приступить к созданию первого приложения с помощью JavaFX, необходимо настроить среду разработки. Инструкция, содержащая необходимую последовательность операций для Eclipse, может быть найдена по ссылке, представленной ниже:

- http://www.eclipse.org/efxclipse/install.html

Теперь можем создать приложение, основанное на framework-е JavaFX.

```
package helloworld;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Рассмотрим основные компоненты структуры JavaFX приложения:

- Главный класс `JavaFX` приложения унаследован от javafx.application.Application. Метод `start()` – главный метод приложения, в отличие от обычного Java приложения, где главный – `main()`. Метод `main()` присутствует в приложении, но в рамках его чаще всего происходит инициализация FS стека и загрузка в него приложения.
- В JavaFX приложении присутствуют два главных компонента stage и scene. В примере мы создаем сцену с определенными размерами и делаем ее видимой.
- Программа JavaFX имеет иерархическую структуру в виде дерева, где узлы – элементы программы (кнопка, текст и т.д.). В нашем случае корневой узел – объект StackPane – слой с изменяемым размером, это означает, что размер окна программы можно будет изменять.
- Наш корневой узел содержит одного потомка – кнопку с обработчиком нажатия (для вывода сообщения в консоль).

Рекомендуемый ресурс по настройке среды разработки:

- http://code.makery.ch/library/javafx-8-tutorial/ru/part1/.

Задание к лабораторной работе:

Необходимо разработать графический интерфейс полученного по варианту приложения (или предложить свою идею к реализации):

1. Игра «Классический тетрис»
2. Игра «Жизнь»
3. Игра «Змейка»
4. Игра «Быки и коровы»
5. Игра «Крестики-нолики»
6. Игра «2048»
7. Игра «Морской бой»

8. Игра «Танки»
9. Игра «Pacman»
10. Игра «Теннис»
11. Игра «Махджонг»
12. Игра «Пинг Понг»

Графический интерфейс должен быть реализован посредством одной из следующих технологий: Swing, SWT, JavaFX.

Требования к защите:

- Среда разработки сконфигурирована, приложение собирается.
- Графический интерфейс приложения спроектирован и реализован.
- Внешний вид должен одинаково хорошо отображаться на всех ОС.
- Swing: должна предоставляться возможность выбора стиля приложения.
- Должны быть определены окна приложения, которые будут включены в отчёт.
- Знание принципов создания графического интерфейса выбранной библиотеки.

Лабораторная работа №2

Программирование алгоритмов с использованием механизмов объектно-ориентированного программирования

Краткие теоретические сведения:

Введение
1. Классы
2. Наследование
3. Интерфейсы
4. Исключения

Введение

Класс – центральный компонент Java. Поскольку класс определяет форму и сущность объекта, он является той логической конструкцией, на основе которой построен весь язык. Как таковой, класс образует основу объектно-ориентированного программирования в среде Java. Любая концепция, которую нужно реализовать в программе Java, должна быть помещена внутрь класса.

Одним из фундаментальных понятий объектно-ориентированного программирования является наследование. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем этот класс может наследоваться другими, более специализированными, классами, каждый из которых будет добавлять свои уникальные характеристики.

Также в рамках данной лабораторной работы будут рассмотрены понятия интерфейс и исключение применительно к Java 8.

1. Классы

При определении класса объявляют его конкретную форму и сущность. Для этого указывают данные, который он содержит, и кода, воздействующего на эти данные. Для объявления класса служит ключевое слово class. Упрощённая общая форма определения класса имеет следующий вид:

```
class имя_класса {
    тип переменная_экземпляра1;
    // …
    тип переменная_экземпляраN:
    тип имя_метода1(список_параметров) {
        // тело метода
    }
    // …
    тип имя_методаN(список_параметров) {
        // тело метода
    }
}
```

Данные, или переменные, определённые внутри класса, называются переменными экземпляра. Код содержится внутри методов. Определённые внутри класса методы и переменные вместе называются членами класса. В большинстве классов действия с переменными экземпляров и доступ к ним осуществляют методы, определённые в этом классе.

При создании класса вы создаёте новый тип данных. Изначально требуется объявить переменную типа класса. Эта переменная не определяет объект. Она представляет собой переменную, которая может ссылаться на объект. Действительная физическая копия объекта появляется посредством вызова оператора new. Этот оператор

динамически резервирует память под объект и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором. Затем ссылка сохраняется в объявленной переменной.

Поскольку резервирование памяти выполняется динамически, может возникнуть вопрос, когда и как необходимо освобождать выделенную память, если проводить аналогию с языком C++. Однако в Java освобождение памяти выполняется автоматически и называется эта технология сбором «мусора». Механизм следующий: при отсутствии каких-либо ссылок на объект программа заключает, что этот объект больше не нужен, следовательно, и занимаемую им память можно очистить.

Иногда при уничтожении объект должен выполнять некое действие. Использую механизм финализации можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком «мусора». Чтобы добавить в класс средство финализации, достаточно определить метод `finalize()`.

Иногда необходимо, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово `this`. Оно может использоваться внутри любого метода для ссылки на текущий объект, для которого был вызван этот метод.

Язык Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют перегруженными, а процесс – перегрузкой методов. Перегрузка методов – один из способов поддержки полиморфизма в Java.

Язык Java 8 позволяет передавать ссылки на методы или конструкторы. Для этого можно воспользоваться ключевым термом «::».

В Java при передаче аргументов происходит передача по значению, конкретный эффект будет зависеть от того, передаётся ли базовый тип или ссылочный.

Чтобы иметь возможность использовать член класса, который может использоваться самостоятельно, без ссылки на конкретный экземпляр, необходимо в начало его объявления нужно поместить слово `static`.

Поле может быть объявлено как `final`. Это позволяет предотвратить изменение содержимого переменной. Значение также можно присвоить в пределах конструктора.

Язык Java позволяет определять класс внутри другого класса – вложенные классы. Область видимости вложенного класса ограничена областью видимости внешнего класса. Вложенный класс имеет доступ к членам (в том числе закрытым) класса, в который он вложен. Однако внешний класс не имеет доступа к членам вложенного класса.

2. Наследование

Чтобы наследовать класс, достаточно просто вставить определение одного класса в другой с использованием ключевого слова `extends`. В терминологии Java наследуемый класс называется суперклассом, а наследующий класс носит название подкласса. Общая форма наследования имеет вид:

```
class имя_суперкласса {
    // …
}

class имя_подкласса extends имя_суперкласса {
    // …
}
```

Для каждого создаваемого подкласса можно указывать только один суперкласс. Язык Java не поддерживает наследование нескольких суперклассов в одном подклассе. Если в этом есть необходимость, то можно создать иерархию наследования, в которой

подкласс становится суперклассом другого подкласса, но нельзя достичь ситуации, когда класс является своим собственным подклассом.

Для того, чтобы подкласс не получил доступа к определённым полям суперкласса их следует объявить как `private`. Чтобы дать доступ подклассу к его непосредственному суперклассу можно использовать ключевое слово super. Данное ключевое слово чаще всего используется для вызова конструктора суперкласса или для обращения к члену суперкласса.

Конструкторы классов вызываются в порядке наследования: от суперкласса к подклассу.

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, то метод подкласса переопределяет метод суперкласса. Если необходимо вызвать метод суперкласса, то используется ключевое слово super.

В ряде случаев нужно будет определять суперкласс, который объявляет структуру определённой абстракции без предоставления полной реализации каждого метода. Потребовать, чтобы определённые методы переопределялись подклассом, можно с использованием указания модификатора типа `abstract`.

```
abstract тип имя(список_параметров);
```

Следует указать в начале объявления метода ключевое слово final, чтобы запретить его переопределение. Таким же образом можно предотвратить наследование класса – путём указания данного ключевого слова перед идентификатором class.

В Java определён один специальный класс, который является суперклассом для всех остальных классов – Object. Это также означает, что все объекты имеют доступ к методам, определённым в этом классе:

- `Object clone()` – создание аналогичного объекта
- `boolean equals(Object object)` – сверка объектов на равенство
- `void finalize()` – метод, вызывающийся перед удалением неиспользуемого объекта
- `Class<?> getClass()` – получает класс объекта во время выполнения
- `int hashCode()` – формирует хеш-код объекта
- `void notify()` – возобновляет выполнение потока, который ожидает вызывающего объекта
- `void notifyAll()` – то же, что и предыдущий метод, только для всех объектов
- `String toString()` – получить строку описания объекта
- `void wait(…)` – ожидает другого потока выполнения

3. Интерфейсы

Применение ключевого слова interface позволяет полностью абстрагировать интерфейс класса от его реализации. Чтобы реализовать интерфейс, класс должен создать полный набор методов, определённых интерфейсом.

```
доступ interface имя {
    возвращаемый_тип имя_метода1 (список_параметров);
    тип имя_конечной_переменной1 = значение;
    // …
    возвращаемый_тип имя_методаN(список_параметров);
    тип имя_конечной_переменнойN = значение;
}
```

Однако Java 8 позволяет вам добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`.

Интерфейсы предназначены для поддержки динамического разрешения методов во время выполнения. Дело в том, что оба класса должны присутствовать во время компиляции в обычной ситуации, когда вы хотите, чтобы вызов метода мог выполняться из одного класса в другом. Данное требование создаёт статическую и нерасширяемую среду обработки классов. В такой системе функциональные возможности неизбежно передаются по иерархии классов всё выше и выше, в результате чего механизмы будут становиться доступными все большему количеству подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. Поскольку иерархия интерфейсов не совпадает с иерархией классов, классы могут реализовать один и тот же интерфейс.

```
доступ class имя_класса [extends суперкласс]
    [implements интерфейс [, интерфейс…]] {
    // тело класса
}
```

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов за счёт простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями.

Ключевое слово `extends` позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов.

Также вы можете поместить реальный класс внутрь интерфейса, однако следует понимать, что он автоматически становится `static public`.

4. Исключения

Исключение – это нештатная ситуация, возникающая во время выполнения последовательности кода. Исключения в Java представляют собой объект, который описывает ошибочную ситуацию, возникающую в части программного кода. Когда возникает такая ситуация, в вызвавшем ошибку методе создаётся и передаётся объект, который представляет исключение. В некоторой точке программы данное исключение перехватывается и обрабатывается.

Обработка исключений Java управляется пятью ключевыми словами: `try`, `catch`, `throw`, `throws` и `finally`.

```
try {
    // место потенциального возникновения ошибки
} catch (тип_исключения_1 exOb) {
    // обработчик первого исключения
} catch (тип_исключения_2 exOb) {
    // обработчик второго исключения
} finally {
    // блок кода, который должен будет быть выполнен несмотря на возникшие
    // исключительные ситуации
}
```

Все типы исключений являются подклассами встроенного класса `Trowable` (является суперклассом для всех исключений). Данный класс делится на два подкласса:

- `Exception` – этот класс используется для исключительных условий, которые должна перехватывать пользовательская программа;

- `Error` – этот класс определяет исключения, которые не должны были бы возникнуть при нормальном выполнении программы.

Существует возможность передавать исключения из ваших программ явным образом, используя оператор `throw` (это весьма полезная возможность при наличии как собственных, так и стандартных исключений).

```
throw экземпляр_Throwable;
```

Используя вышеописанный способ порождения исключений, стоит не забыть его обработать. Для этого, если вы не работаете с ним в том же методе, в котором генерируете, следует к объявлению данного метода добавить конструкцию throws. Данная конструкция перечисляет типы исключений, которые метод может передавать наверх по стеку вызовов.

```
тип имя_метода(список_параметров) throws список_исключений {
    // тело метода
}
```

Также рекомендуется самостоятельно ознакомиться со следующими понятиями и их применением:

- динамическая диспетчеризация методов
- сцепленные исключения

Задание к лабораторной работе:
Необходимо реализовать алгоритмы для выбранных в первой лабораторной работе заданий.

Требования к защите:

- Полностью работоспособное приложение.
- Приложение имеет следующие возможности:
  - выбор уровней сложности;
  - «автоматический» режим: компьютер выступает в качестве игрока.
- Диаграмма классов должна быть спроектирована и включена в отчёт.
- Обоснование выбранной компоновки классов, включённое в отчёт.
- Блок-схема основного алгоритма приложения, включённая в отчёт.
- Наличие комментариев в исходном коде.
- Наличие самогенерируемой документации (на основе комментариев).
- Сгенерированная документация должна быть включена в отчёт.
- Разработана и использована в приложении система исключений.
- Знание основных принципов ООП, их достоинств и недостатков.

Лабораторная работа №3

Разработка программ с использованием модульного подхода

Краткие теоретические сведения:

Введение

1. Модификаторы

2. Пакеты

Введение

По мере разрастания проектов возникает проблема уникальности имён. Дело в том, что при написании качественного кода обычно используются удобные описательные имена, а в результате изменения размеров проекта происходит их истощение.

К счастью, язык Java предоставляет механизм разделения пространства имён на более удобные для управления фрагменты. Этим механизмом является пакет, который одновременно используется и как механизм присвоения имён, и механизм управления видимостью.

Внутри пакета можно определить классы, не доступные коду вне этого пакета. Можно также определить члены класса, которые видны только другим членам этого же пакета. Такой механизм позволяет классам располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

1. Модификаторы

Модификаторы – это, по сути, служебные слова, которые придают классу, полю класса или методу определённые свойства.

Рассмотрим наиболее популярные модификаторы. Начнем с используемых при объявлении классов. Первый — это `public`. Этим модификатором мы указываем, что мы можем объявлять ссылки на этот класс в любом коде. Если же модификатор `public` не задан, то использовать класс мы можем исключительно внутри пакета. Помимо того, что мы можем создавать экземпляры класса, помеченного как `public`, нам также дозволено обращаться к полям и методам этого класса, помеченным этим же модификатором. В контексте отдельного пакета подразумевается, что класс с модификатором public должен соответствовать имени этого файла.

Следующий модификатор, предназначенный для рассмотрения, – это `abstract`. Он позволяет объявить класс неполноценным и не предназначенным для создания его экземпляров. Классы с таким модификатором можно только наследовать. Это обуславливается наличием у класса абстрактных методов. Если ваш класс не имеет абстрактных методов, то нет никакого смысла указывать модификатор `abstract` при описании вашего класса, если только вам по каким-либо причинам необходимо запретить создание объектов этого класса.

Следующий модификатор является некоторой противоположностью абстрактным классам и предназначен для того, чтобы запретить создание производных классов. Это модификатор `final`. Его можно использовать, когда вам необходимо не допустить наследования вашего класса.

Помимо вышеперечисленных модификаторов для объявления классов, существует также `strictfp`. Его цель – определить точное и единообразное выполнение операции над числами с плавающей запятой на всех JVM (виртуальная машина Java).

При объявлении класса вы можете использовать сразу несколько модификаторов. Из их определений можно заметить, что единовременно нельзя использовать модификаторы `final` и `abstract` по причине их противоположности.

Это что касается классов. Немного по-другому дело обстоит с полями классов. Здесь, помимо обычных модификаторов, присущих полям, имеют место быть также

модификаторы доступа. Модификаторы доступа – это зарезервированные слова, которые определяют область видимости/доступа полей или же методов класса. Заметьте, что классу не свойственно определение модификаторов доступа, понятно почему. В Java всего четыре модификатора доступа, расположенные в следующем списке в зависимости от убывания закрытости модификатора:

- `private` члены класса доступны только внутри класса;
- `package-private` или `default` (по умолчанию) члены класса внутри пакета;
- `protected` члены класса доступны внутри пакета и в классах-наследниках;
- `public` члены класса доступны всем.

Модификатор доступа у конструкторов, методов и полей может быть любой, а вот с классами и их блоками не так всё просто. Класс может быть только либо `public`, либо `default`, причём в одном файле может находиться только один public класс. У блока может быть только один модификатор – `default`.

После этого можно перейти к списку модификаторов у полей. И первое из них – это `static`. Использованию статических полей и методов можно посвятить отдельную статью. Статические поля – это поля, которые едины для всех объектов данного класса. Т.е. ссылка этого поля у любого экземпляра класса будет ссылаться на одно и то же значение. Например:

```
class staticBox {
      public static int iNum;
}

public MyBox {
      public static void main(String argv[]) {
            staticBox a = new staticBox();
            staticBox b = new staticBox();
            a.iNum = 123;
            System.out.println("staticBox b.iNum = " + b.iNum);
      }
}
```

Результатом выполнения данной программы будет: «`staticBox b.iNum = 123`». Следующий модификатор: `final` – это модификатор, позволяющий объявлять константные поля в классе. Если у вас есть некоторое свойство проектируемого вами объекта, значение которого не будет меняться, то вы может воспользоваться этим модификатором. Любая попытка переопределить значение поля с модификатором `final` приводит к выбросу исключения.

```
public final String boxCompanyName = "BlackBox :)";
```

Затем идут два специфических и очень важных модификатора:

- `transient` – решает проблему представления объекта в виде последовательности байтов данных (сериализации);
- `volatile` – требуется в процессе синхронизации потоков вычислений и управления памятью;

И последнее – это использование модификаторов при объявлении методов. Когда вы объявляете метод класса, то вправе также использовать и модификаторы доступа. Это, как и в случае с полями, позволяет ограничить доступ к вызову методов класса.

Опять же модификатор `static` позволяет определять статические методы. Это такие методы, которые являются общими для класса, а не для отдельного объекта этого класса. Также они могут работать лишь со статическими полями класса. Так как если бы

они могли обращаться к нестатическим полям, то объявление метода статическим теряло бы всякий смысл.

Метод, помеченный как `synchronized`, опять же, относится к проблеме управления вычислительными потоками, одновременно выполняющимися в контексте программного приложения.

Существует также замечательный модификатор `native`, позволяющий реализовывать любые системные операции на низком уровне. Это лишает Java-код переносимости между платформами, но зато позволяет использовать любые системные API вызовы операционной системы. Языком для написания `native`-методов, как правило, является C++. Библиотеки API предлагаются разработчиками виртуальных Java-машин для тех или иных платформ и языков. Например, JNI (Java Native Interface) – стандартная библиотека для C.

```
public native getMemoryCount();
```

Модификатор `strictfp` у методов, аналогично, как и одноименный модификатор для классов, позволяет организовать работу метода с числами с плавающей запятой единообразной для всех виртуальных машин Java.

Модификаторы в Java – это мощный инструмент для описания классов, их полей и методов, используя который вы сможете более четко и правильно описывать поведение своих объектов.

2. Пакеты

Для того, чтобы создать пакет необходимо включить команду `package` в качестве первого оператора исходного файла Java. Любые классы, объявленные внутри этого файла, будут принадлежать указанному пакету. Оператор `package` определяет пространство имён, в котором хранятся классы, и имеет следующую форму:

```
package пакет;
```

Здесь «пакет» задаёт имя пакета. Один и тот же оператор `package` может присутствовать в более чем одном файле. Этот оператор просто указывает пакет, к которому принадлежат классы, определённые в данном файле.

Язык Java позволяет создавать иерархию пакетов. Применение данного механизма имеет следующую форму:

```
package пакет1[.пакет2[.пакет3]];
```

Пакеты предлагают эффективный механизм изоляции различных классов друг от друга, посему все встроенные классы Java хранятся в пакетах. Поскольку внутри пакетов классы должны быть полностью определены именами их пакетов, длинное, разделённое точками имя пути пакета каждого используемого класса может оказаться слишком громоздким. Поэтому, чтобы определённые классы или весь пакет можно было сделать видимыми, в Java включён оператор import. После того, как класс импортирован, на него можно ссылаться непосредственно, используя только его имя.

Оператор import имеет следующую общую форму:

```
import пакет1[.пакет2].(имя_класса| *);
```

В этой форме `пакет1` – имя пакета верхнего уровня, `пакет2` – имя подчинённого пакета внутри внешнего пакета, отделённое символом «точка» (.). Глубина вложенности пакетов практически не ограничена ничем, кроме файловой системы. Конструкция

`имя_класса` может быть задана либо явно, либо с помощью символа «звёздочка» (*), который указывает компилятору Java о необходимости импорта всего пакета.

Классы и пакеты одновременно служат средствами инкапсуляции и хранилищем имен и области видимости переменных и методов. Пакеты играют роль контейнеров классом и других подчинённых пакетов. Классы служат контейнерами данных и кода. Класс – наименьшая единица абстракции Java. Вследствие взаимодействия между классами и пакетами Java определяет четыре категории видимости членов класса:

- подклассы в одном пакете;
- классы в одном пакете, не являющиеся подклассом;
- подклассы в различных пакетах;
- классы, которые не находятся в одном пакете и не являются подклассами.

Три модификатора доступа: `private`, `public`, `protected` – предоставляют разнообразные способы создания множества уровней доступа, необходимых для этих категорий.

Иногда возникает ситуация, когда импортировано несколько классов с одним и тем же именем из разных пакетов. Как результат мы получаем ситуацию, именуемую «конфликт имён». С целью того, чтобы специфицировать необходимый вам класс, следует указать квалифицированное имя, включая имя пакета. Например, следующий код не будет испытывать никаких проблем, несмотря на то, что класс `List` присутствует как в пакете `java.awt`, так и в пакете `java.util`:

```
import java.awt.*;
import java.util.*;
import java.util.List;

public class EmptyList {
      public static void main(String args) {
            List list = Collections.emptyList();
            System.out.println(list);
      }
}
```

Задание к лабораторной работе:

Необходимо разработать отдельный модуль, который будет содержать нотацию для игры. Данная нотация должна обладать следующими возможностями:

- воспроизведение сохранённой игры;
- отображаться в поле игры;
- сохраняться в файл.

*Нотация – система условных обозначений, принятых в какой-либо области знаний или деятельности. Включает множество символов, используемых для представления понятий и их взаимоотношений, составляющее алфавит нотации, а также правила их применения. В рамках данной лабораторной работы должны быть разработаны правила, по которым будут записаны действия, произведённые в приложении. По данной сохранённой записи данные действия должны иметь возможность воспроизводиться.

Требования к защите:

- Разработанная нотация, описанная в отчёте.
- Обоснование выбранного вида нотации.
- Должны поддерживаться возможности, описанные в списке задания.
- Разработанный формат хранения информации в файле.
- Иметь знания по понятию, достоинствам и недостаткам модульного подхода.

Лабораторная работа №4

**Разработка многопоточных программ,
использование стандартных примитивов синхронизации**

Краткие теоретические сведения:

Введение
1. Процессы
2. Потоки
3. Синхронизация

Введение

В данной лабораторной работе рассматриваются вопросы многопоточности и стандартных примитивов синхронизации в Java, модели их реализации и использования. Наиболее частые случаи использования многопоточности:

- программирование интерфейсов;
- клиент-серверные приложения;
- высокопроизводительные приложения.

Многопоточность является требуемым атрибутом приложения тогда, когда необходимо, чтобы графический интерфейс реагировал на действия пользователя независимо от действий, происходящий в фоне. В рамках лабораторных работ, заданиями к которым является создание игры, различные потоки могут отвечать за работу с сетью, анимацию, расчёт физики и тому подобные действия.

1. Процессы

Процессом является совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Большая часть приложений построена на базе одного процесса, однако часто существует необходимость использовать несколько процессов: с целью повысить общую отказоустойчивость приложения, его безопасность, или просто с целью более лаконичного разделения функциональности.

Процессы изолированы друг от друга, поэтому не существует возможности прямого доступа к памяти иного процесса. Операционная система создаёт для каждого процесса «виртуальное адресное пространство», которое является собственностью процесса и оный имеет к ней прямой доступ. Это пространство содержит данные только данного процесса. В обязанности же операционной системы входит отображение данного виртуального пространства на физическую память.

2. Потоки

Многопоточностью часто называют специализированную форму многозадачности. В среде поточной многозадачности наименьшим элементом управляемого кода является поток. Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. В отличие от процессов, каждый поток не требует собственного адресного пространства. Также тяжеловесным является процесс переключения контекстов от одного процесса к другому. Многопоточность позволяет писать программы с высокой эффективностью, которые максимально используют доступную мощь процессора системы.

Также немаловажным преимуществом является сведение к минимальному значению времени ожидания. Это важно для интерактивных сетевых сред, в которых работает Java.

Во многих отношениях система времени выполнения зависит от потоков, и все библиотеки классов спроектированы с учётом многопоточности. Однопоточные системы используют подход, который называется «циклом событий с опросом». В данной модели

единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, для принятия решения о том, что делать дальше. До тех пор, пока тот не вернёт управление, в программе ничего не может привести к тому, что одна часть будет доминировать над другими и не позволять обработку любых других событий.

Выгода многопоточности заключается в следующем: основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки остальных частей программы. Например, при чтении данных из сети время ожидания либо ожидание пользовательского ввода может быть утилизировано где угодно. Многопоточность позволяет циклам анимации «засыпать» на секунду между показом соседних кадров, не приостанавливая при этом работы всей системы. При блокировке потока в программе останавливается только один-единственный заблокированный поток, а все остальные потоки продолжают выполнение.

Java присваивает каждому потоку приоритет, определяющий поведение данного потока по отношению к другим. Приоритеты задаются целыми числами. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты:

- поток может добровольно уступить управление: остановка потока по причине ожидания ввода/вывода, ответа от сетевого устройства и так далее;
- поток может быть прерван другим, более приоритетным потоком: в данном случае используются выданные/назначенные приоритеты.

Многопоточная система Java встроена в класс `Thread`, его методы и дополняющий его интерфейс `Runnable`. Вы не можете обратиться напрямую к нематериальному состоянию работающего потока, и обязаны иметь дело с его заместителем – экземпляром класса Thread, который породил его. Для создания нового потока ваша программа должна либо расширить класс `Thread`, либо реализовать интерфейс `Runnable`.

Когда программа Java стартует, немедленно начинает исполняться главный поток. Главный поток создаётся автоматически при запуске программы, но им можно управлять через объект класса `Thread`. Для этого следует получить ссылку на него вызовом метода `currentThread()`, который является открытым и статическим методом класса Thread. Его общая форма выглядит следующим образом:

```
static Thread currentThread();
```

Данный метод возвращает ссылку на поток, из которого он был вызван. Получив ссылку на главный поток, можно управлять им точно так же, как и любым другим. В Java существует два способа создать поток:

- реализация интерфейса `Runnable`;
- расширение класса `Thread`.

Для реализации многопоточной модели используя первый способ достаточно объявить один единственный метод `run()`. Внутри данного метода вам следует определить код, который определяет поведение нового потока. Ниже представлен краткий пример.

```
// поток с использованием первого метода
class CustomThread implements Runnable {
      Thread thread;

      CustomThread() {
            thread = new Thread(this, "Новый поток");
            System.out.println("Поток Custom создан: " + thread);
            thread.start();
      }
```

```
	public void run() {
		System.out.println("Поток Custom завершается");
	}
}

class ThreadMain {
	public static void main(String args[]) {
		new CustromThread();
		System.out.println("Главный поток завершается");
	}
}
```

3. Синхронизация

Многопоточность придаёт вашим программам модель асинхронного поведения, посему зачастую существует необходимость обеспечить определённость поведения вашего приложения. Для этой цели в Java реализован монитор. Монитор – это управляющий механизм, впервые реализованный Чарльзом Энтони Ричардом Хоаром. Вы можете осознавать монитор как чёрный ящик, принимающий только один единственный поток в единицу времени. Как только поток вошёл в монитор, все другие потоки ожидают, пока тот не покинет его. Этот механизм синхронизации (монитор) имеет весьма схожее поведение с одним из системных средств межпроцессного взаимодействия операционной системы Windows на языке Си – критическая секция.

После того, как вы разделите программу на отдельные потоки, вам необходимо будет определить, как они станут общаться друг с другом. При программировании с использованием Java вы не зависите от операционной системы, что зачастую происходит при использовании других языков программирования. Java предоставляет экономичный и ясный способ общения двух или более потоков между собой: вызов предопределённых методов, которыми обладают объекты. Система сообщений Java позволяет потоку войти в синхронизированный метод объекта и ждать, пока какой-либо иной поток явно не уведомит его о прибытии.

Предположим, вам необходимо синхронизировать доступ к объектам классов, которые не были предназначены для многопоточного доступа. Самым простым способом решения данной проблемы является заключение в блок синхронизации вызовы методов этого класса. Вот как выглядит общая форма оператора `synchronized`:

```
synchronized(объект) {
	// то, что необходимо синхронизировать
}
```

Объектом является ссылка на синхронизируемый объект. Также в Java существует более элегантный механизм межпроцессных коммуникаций:

- метод `wait()` – вызывающий поток должен отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдёт в тот же монитор и не вызовет метод `notify()`;
- метод `notify()` – возобновляет работу потока, который вызвал метод `wait()`;
- метод `notifyAll()` – возобновляет работу всех потоков, которые вызвали метод `wait()` в том же объекте.

Иногда, для выполнения потока нужно дождаться завершения другого потока. В таких случаях вам поможет метод `join()`. Если поток вызывает метод `join()` для другого потока, то вызывающий поток приостанавливается до тех пор, пока вызываемый поток не завершит свою работу. Данный метод имеет две перегруженные реализации:

- `join()` – ожидает пока вызываемый поток не завершит свою реализацию;
- `join(long millisecond)` – ожидает завершения вызываемого потока указанное время, после чего передает управление вызывающему потоку.

Вызов метода `join()` может быть прерван вызовом метода `interrupt()` для вызывающего потока, поэтому метод `join()` размещают в блоке `try/catch`.

```
class Sleeper extends Thread {
      private int duration;
      public Sleeper(String name, int sleepTime) {
            super(name);
            duration = sleepTime;
            start();
      }

      public void run() {
            try {
                  sleep(duration);
            } catch (InterruptedException e) {
                  System.out.println(getName() + " прерван");
                  return;
            }
            System.out.println(getName() + " активизировался.");
      }
}

class Joiner implements Runnable {
      private Sleeper sleeper;
      private Thread thread;
      public Joiner(String name, Sleeper sleeper) {
            this.sleeper = sleeper;
            thread = new Thread(this);
            thread.setName(name);
            thread.start();
      }

      public void run() {
            try {
                  sleeper.join();
                  System.out.println(thread.getName() + " завершен.");
            } catch (InterruptedException e) {
                  System.out.println(thread.getName() + " прерван.");
            }
      }

      public Thread getThread() {
            return thread;
      }
}

public class App {
      public static void main(String[] args) {
            Sleeper sleepy1 = new Sleeper("Sleepy 1", 1500),
                  sleepy2 = new Sleeper("Sleepy 2", 2000);
            Joiner joiner1 = new Joiner("Joiner 1", sleepy1),
                  joiner2 = new Joiner("Joiner 2", sleepy2);
            sleepy1.interrupt();
            joiner2.getThread().interrupt();
      }
}

/* В результате вывод будет содержать следующие строки:
Joiner2 прерван.
Sleepy1 прерван
Joiner1 завершен.
Sleepy2 активизировался.
*/
```

Как видно из данного примера при прерывании метода `join()` для потока joiner2 было выброшено исключение `InterruptedException`. Данное исключение привело к тому, что поток joiner2 прервался, не дожидаясь завершения потока sleepy2. В случае прерывания потока sleepy1, управление было передано потоку joiner1, который ожидал завершения потока sleepy1.

Рассмотренные примитивы синхронизации являются лишь частью всей области многопоточного программирования с использованием языка Java.

Задание к лабораторной работе:

Необходимо модернизировать разработанное приложение следующим образом:

- выделить отдельно поток-клиент и поток-сервер:
  - требования к потоку-клиенту:
    - представляет собой графический интерфейс приложения;
    - следит за изменениями/воздействиями;
    - аккумулирует внешнее воздействие на приложение и отсылает запросы потоку-серверу;
    - получает ответ от потока-сервера и применяет решения к графическому интерфейсу;
  - требования к потоку-серверу:
    - должен быть способен получать сообщения от потока-клиента;
    - должен быть способен просчитать следующее состояние приложения;
    - по результатам просчёта выполнить отсылку соответствующего результата потоку-клиенту;
- *должна иметься возможность запуска нескольких графических интерфейсов;
- *на сервере должен присутствовать «предсказатель» (реализованный в отдельных потоках), который рассчитывает наименьшее количество операций для достижения некоторой определённой студентом ситуации: победа/поражение/следующий уровень и т.д.
- *также «предсказатель» должен иметь возможность угадывания следующего хода игрока.

Требования к защите:

- Должны поддерживаться возможности, описанные в списке задания.
- В отчёте должно быть описано клиент-серверное взаимодействие.
- Разработать систему расчёта производительности «предсказателя».
- Отразить производительности «предсказателя» в отчёте.

*под производительностью понимается скорость просчёта шагов в единицу времени (или иная разработанная и обоснованная реализация термина).

Лабораторная работа №5

Функциональное программирование с использованием языка Scala. Выражения и функции. Обобщённые типы и методы. Работа с файлами. Хвостовая рекурсия

Краткие теоретические сведения:

Введение

1. Настройка среды разработки
2. Выражения и функции
3. Обобщённые типы и методы
4. Работа с файлами
5. Хвостовая рекурсия

Введение

Язык Scala был создан в 2001-2004 гг в лаборатории методов программирования EPFL. Он стал результатом исследований, направленных на разработку более хорошей языковой поддержки компонентного программного обеспечения. Язык Scala соответствует двум нижеописанным гипотезам. Во-первых, Scala легко масштабируем в том смысле, что существует возможность с помощью одних и тех же концепций описать как маленькие, так и большие части. Во-вторых, масштабируемая поддержка компонентов может быть предоставлена языком программирования, унифицирующим и обобщающим объектно-ориентированное и функциональное программирование. Некоторые из основных технических новшеств Scala – это концепции, представляющие собой сплав этих парадигм программирования. В статически типизированных языках, к которым относится Scala, эти парадигмы до сих пор были почти полностью разделены.

Scala разрабатывалась в расчете на совместную работу с C# и Java. Она позаимствовала у этих языков значительную часть синтаксиса и системы типов. Но в то же время, иногда прогресс требует отбросить некоторые существующие соглашения. Поэтому Scala – это не расширенный вариант любого из этих языков. Некоторые возможности отсутствуют, другие же переосмыслены для улучшения согласованности концепций.

Некоторые ключевые аспекты языка:

- Scala-программы во многом похожи на Java-программы, и могут беспроблемно взаимодействовать с Java-кодом.
- Scala включает единообразную объектную модель – в том смысле, что любое значение является объектом, а любая операция – вызовом метода.
- Scala – это также функциональный язык в том смысле, что функции – это полноправные значения.
- В Scala включены мощные и единообразные концепции абстракций, как для типов, так и для значений.
- Она содержит гибкие симметричные конструкции mixin-композиции для композиции классов и trait-ов.
- Она позволяет производить декомпозицию объектов путем сравнения с образцом.
- Образцы и выражения были обобщены для поддержки естественной обработки XML-документов.
- В целом, эти конструкции позволяют легко выражать самостоятельные компоненты, использующие библиотеки Scala, не пользуясь специальными языковыми конструкциями.

1. Настройка среды разработки

Подключение требуемых библиотек в Scala IDE не отличается от подключения оных в Eclipse IDE for Java. Базовое конфигурирование проекта на Scala и Java выглядит следующим образом для примера с использованием библиотеки SWT:

- Создать новый Scala проект: File -> New -> Project… -> Scala Wizards -> Scala Project -> Next.
- Выбрать вкладку Projects -> Add…, выбрать org.eclipse.swt, следом нажать Finish.

Пример приложения, вычисляющего факториал, приведён ниже: UI – Java + SWT, ядро приложения – Scala.

```
//MainWindow.java
package by.bsuir.wmsis.kpp.testapp;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;

public class MainWindow {
  public static void main(String[] args) {
    Display display = Display.getDefault();
    Shell shell = new Shell();
    shell.setSize(250, 250);
    Text txtInput = new Text(shell, SWT.BORDER);
    txtInput.setBounds(10, 12, 76, 21);
    Label lblFact = new Label(shell, SWT.NONE);
    lblFact.setBounds(31, 55, 55, 15);
    Button btnFact = new Button(shell, SWT.NONE);
    btnFact.addSelectionListener(new SelectionAdapter() {
      @Override
      public void widgetSelected(SelectionEvent e) {
        int value = Integer.parseInt(txtInput.getText());
        lblFact.setText(Integer.toString(FactorialFunc.factorial(value)));
      }
    });
    btnFact.setBounds(120, 10, 27, 25);
    btnFact.setText("!");
    shell.open();
    shell.layout();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
    }
  }
}

//FactorialFunc.scala
package by.bsuir.wmsis.kpp.testapp

object FactorialFunc {
  def factorial(n : Int) : Int = {
    def factorIter(res : Int, n : Int) : Int = {
      if (n == 0) res;
      else factorIter(res * n, n - 1);
    }
    factorIter(1, n);
  }
}
```

2. Выражения и функции

После установки и настройки среды разработки существует два варианта работы с языком программирования Scala: консольный интерпретатор и полноценная среда разработки.

Рассмотрим первый вариант. Самым популярным способом запуска консольного варианта считается запуск из-под системы сборки проектов языка Scala. Производится это следующим образом: набор команды «sbt» в командной строке, а затем, после того, как вы успешно попадёте в систему сборки, следующим шагом является набор команды «console». Запустив консольный интерпретатор, можно протестировать его следующим образом:

```
scala> 23 + 123
res0: Int = 146
```

Существует возможность именования и использования впоследствии подвыражений:

```
scala> def  mark = 10
mark: Int
scala> 10 * mark
res1: Int = 100
scala> 10.0 * mark
res2: Double = 100.0
```

Данные выражения не вычисляются во время определения – подстановка и расчёт выполняется лишь во время обращения к этому выражению за получением его непосредственного значения. Определение функции выглядит следующим образом:

```
scala> def cube(x: Double) = x * x * x
cube: (Double)Double
scala> cube(2)
res3: Double = 8.0
```

Следует отличать различный порядок вычисления результата функции: вызов по имени (call-by-name) – порядок, когда интерпретатор изначально рассчитывает переданные аргументы, а затем их использует; вызов по значению (call-by-value) – альтернативный вариант, в котором функция применяется к неприведённым аргументам.

Ниже представлен пример использования условной конструкции языка, также одновременно она является заменой тернарных операторов в иных языках программирования:

```
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (Double)Double
```

Используемые в Scala выражения для определения истинности и обратного явления похожи на такие же конструкции в языке Java. Этими конструкциями являются true/false, операторы сравнения, булевское отрицание !, булевские операторы && и ||.

Язык Scala также обладает следующей функциональностью: вложенные функции – механизм, очень похожий на механизм использования вложенных методов в Java.

3. Обобщённые типы и методы

Язык Scala имеет механизм, очень напоминающий шаблонные классы в C++ как с точки зрения синтаксиса, так и с точки зрения функционала. Данное действие имеет свойство называться «параметризация определения». Параметризация по типу выглядит это следующим образом:

```
abstract class Substance[A] {
  def create(x: A): Substance[A] = new Instance[A](x, this)
  …
}
```

В приведённом определении `A` является типовым параметром класса. Данные параметры являются произвольными именами, которые заключены в квадратные скобки. Методы, использующие параметризованные типы называются полиморфными.

Иногда требуется вернуть из функции более одного значения. Можно создать класс, который будет состоять из необходимого количества полей, однако существует более лаконичное решение данной проблемы, так как весьма неэффективно создавать класс для каждого возможного возвращаемого значения из функции. Решением является использование кортежей.

Рассмотрим популярный пример с двумя возвращаемыми значениями - mod && div. В Scala для решения этого вопроса можно использовать класс Tuple2. С Tuple2 функция будет выглядеть следующим образом:

```
scala> def moddiv(x: Int, y: Int) = new Tuple2[Int, Int](x % y, x / y)
```

Следует также обратить внимание, что типовые параметры, рассмотренные чуть ранее, могут быть опущены по причине вывода их из аргументов - это позволяет сделать код более чистым.

Существует два пути обращения к элементам кортежей:

– имя параметров конструктора

```
val tuple = moddiv(x, y)
println("rest: " + tuple._1 + "quotient: " + tuple._2)
```

– сопоставление с образцом

```
moddiv(x, y) match {
  case Tuple2(rest, quotient) =>
    println("rest: " + rest + ", quotient: " + quotient)
}
```

Следует запомнить, что непозволительно использовать в образцах типовые параметры, например: `caseTuple2[Int, Int](rest, quotient).`

4. Работа с файлами

Рассмотрим простейшие примеры работы с файлами, которые своим способом применения могут напомнить один из существующих путей реализации похожего функционала в Java.

```
import scala.io.Source

val filename = “fileopen.scala”
for (line <- Source.fromFile(filename).getLines) {
    println(line)
}
```

В предложенном выше примере рассмотрен способ чтения файла построчно. Если в этом есть необходимость или желание, то можно сразу из файла сформировать требуемую коллекцию. Например, следующим образом:

```
val lines = Source.fromFile("fileopen.scala").getLines.toList
val lines = Source.fromFile("fileopen.scala").getLines.toArray
val content = Source.fromFile("fileopen.scala").getLines.mkString
```

В последнем примере рассмотрен вариант, как можно получить содержимое файла в качестве одной строки.

На текущий момент Scala не предоставляет собственной возможности произвести запись в файл, поэтому программисты вынуждены использовать один из предоставленных Java способов реализации данного действия, например PrintWriter и FileWriter:

```
//PrintWriter
import java.io._
val pw = new PrintWriter(new File("filewrite.scala" ))
pw.write("Hello, world")
pw.close

//FileWriter
val file = new File(canonicalFilename)
val bw = new BufferedWriter(new FileWriter(file))
bw.write(text)
bw.close()
```

5. Хвостовая рекурсия

Приведём распространённый пример с функцией для нахождения наибольшего общего делителя двух чисел.

```
scala> def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
```

Используя модель подстановки пошаговое вычисление будет выглядеть следующим образом:

```
gcd(14, 21)
      if (21 == 0) 14 else gcd (21, 14 % 21)
      if (false) 14 else gcd(21, 14 % 21)
      gcd(21, 14 % 21)
      gcd(21, 14)
      if (14 == 0) 21 else gcd(14, 21 % 14)
            gcd(14, 21 % 14)
      gcd(14, 7)
      if (7 == 0) 14 else gcd(7, 14 % 7)
            gcd(7, 14 % 7)
      gcd(7, 0)
      if(0 == 0) 7 else gcd(0, 7 % 0)
            7
```

Теперь вместо рассмотренного алгоритма Евклида обратимся к стандартному алгоритму расчёта факториала посредством рекурсии.

```
scala> def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

Используем эту функцию для вычисления значения факториала для числа 5:

```
factorial(5)
      if (5 == 0) 1 else 5 * factorial(5 – 1)
      5 * factorial(5 – 1)
      5 * factorial(4)
…           5 * (4 * factorial(3))
…           5 * (4 * (3 * factorial(2)))
…           5 * (4 * (3 * (2 * factorial(1))))
```

```
…           5 * (4 * (3 * (2 * (1 * factorial(0)))))
…           5 * (4 * (3 * (2 * (1 * 1))))
…           120
```

Вероятно, несложно заметить разницу между процессом выполнения обоих этих функций: в первом случае длина конструкций вызова остаётся прежней на протяжении всего времени вычисления (постоянна), во втором же случае конструкция планомерно разрастается с каждым вызовом.

Обратите внимание, что вызов gcd в показанной реализации является последним действием в теле вычисления. Данный рассмотренный пример и является примером хвостовой рекурсии. Результирующий вызов функции может быть реализован скачком к началу данной рекурсивной функции. Параметры вызова в таком случае должны быть вписаны вместо текущих параметров выполнения алгоритма, в результате чего нет необходимости выделять новую память на стеке. В результате получается: функции, применяющие способ решения задачи - хвостовая рекурсия – выполняются по факту итеративно, что позволяет им работать в памяти фиксированного размера.

Второй рассмотренный пример заканчивается операцией умножения. Из архитектуры конструирования и разработки программ известно, что для каждого рекурсивного вызова выделяется новый блок стека, в котором хранится вспомогательная информация для вызывающейся функции. Данный сегмент очищается после завершения следования инструкциям подпрограммы (функции). Из этого можно сделать вывод, что функция неитеративна, то есть в данном случае не обладает свойствами алгоритмов, использующих хвостовую рекурсию – это означает, что данная реализация требует количество памяти, пропорциональное количеству вызовов.

Пример реализации алгоритма нахождения факториала посредством хвостовой рекурсии мог бы выглядеть следующим образом:

```
def factorial(accumulator: Int, number: Int) : Int = {
    if(number == 1)
        return accumulator
    factorial(number * accumulator, number - 1)
}
```

И в таком случае стек вызовов будет выглядеть аналогично стеку вызовов для рассмотренного примера к алгоритму Евклида для нахождения наименьшего общего делителя.

Подведём итог: если последней операцией функции является вызов функции, то обе данные функции будут использовать один блок стека - пример хвостового вызова.

Задание к лабораторной работе:

Необходимо настроить среду разработки и ваш проект для совместной работы с функциональным языком программирования Scala.

Заданием является выполнение сортировки сохранённой нотации игр. Результатом лабораторной работы должен быть готовый список игр, отсортированный от лучших игр к худшим. Решение принимается на основе избранного критерия. Для реализации этого задания на функциональном языке программирования следует использовать хвостовую рекурсию.

Провести подобную сортировку на Java. Произвести измерения производительности и обосновать полученные результаты.

Требования к защите:

- Обладание навыком настройки среды разработки под язык Scala
- Следующие пункты должны быть отражены в отчёте:
  - Выбранный критерий сравнения игр и его обоснование

  - Результаты сравнения производительности и их обоснование
- Приложение должно предоставлять возможность выбора способа сортировки
- Должна поддерживаться возможность воспроизведения лучшей/худшей игры

Лабораторная работа №6

Коллекции и абстракции

Краткие теоретические сведения:

Введение
1. Последовательности
2. Списки
3. Множества
4. Абстракции

Введение

В рамках данной лабораторной работы вам предстоит познакомиться с коллекциями, которыми пестрит язык Scala. К имеющимся типам коллекций доступны абстракции. Этот механизм позволяет вам использовать код, который изначально проектировался быть использованным коллекцией, например, SomeCollection, но в результате может быть использован коллекциями List, Set, или чем-то ещё без необходимости приведения/изменения/адаптирования.

На данном этапе стоит ввести понятие «трейт». «Трейтами» являются абстрактные классы, которые спроектированы, чтобы существовала возможность добавления их к другим классам. Первый сценарий использования «трейтов»: добавление методов и полей к имеющимся классам, второй – вбор в себя сигнатур некоторой функциональности, обеспечиваемой классами (схожая функциональность с Java интерфейсами).

Основные «трейты» коллекций представлены ниже:

```
                  → «Трейт» Seq     → «Трейт» IndexedSeq
«Трейт» Iterable  → «Трейт» Set     → «Трейт» SortedSet
                  → «Трейт» Map     → «Трейт» SortedMap
```

У коллекций, имеющих в своём составе Iterable, существует возможность возвращать итератор Iterator, обеспечивающий доступ ко всем элементам коллекции:

```
val collect = … // некоторая коллекция типа Iterable
val iter = coll.iterator
while (iter.hasNext)
    // здесь можно выполнить некое действие над iter.next()
```

При поверхностном осмотре данной иерархии стоит отметить, что Seq – это упорядоченная коллекция значений, Set – это неупорядоченная коллекция значений, Map – множество пар ключ/значение.

Scala поддерживает изменяемые и неизменяемые коллекции. Неизменяемые коллекции никогда не изменяются, по этой причине их безопасно можно передавать по ссылке и использовать в многопоточных приложениях.

1. Последовательности

Наиболее популярными неизменяемыми последовательностями являются List, Stream, Stack, Queue, Vector и Range.

Создание стандартного связанного списка выглядит следующим образом:

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)
```

Также вы можете создать его используя функциональный стиль:

```
scala> 1::2::3::Nil
res1: List[Int] = List(1, 2, 3)
```

Создание последовательности, имеющей определённый порядок, выглядит следующим образом:

```
scala> Seq(1, 1, 2)
res3: Seq[Int] = List(1, 1, 2)
```

Обратите внимание на возвращаемый результат, в котором Seq является «трейтом», не стоит путать его с фабричным объектом Seq, который создаёт списки.

Тип Vector – индексируемая последовательность с быстрым произвольным доступом к элементам. Векторы реализованы как деревья, где каждый узел может иметь до 32 потомков.

Класс Range представляет последовательность целых чисел. Объект данного типа хранит лишь начальное и конечное значения, а также шаг. В результате для конструирования используются методы to и until.

К изменяемым последовательностям относятся: ArrayBuffer, Stack, Queue, Priority Queue, LinkedList, Double LinkedList.

Тип ArrayBuffer является изменяемым аналогом Vector. Стеки, очереди и очереди с приоритетами ничем особенным от стандартных структур данных, имеющих аналогичные наименования, не выделяются.

2. Списки

В Scala список может быть либо значением Nil, либо объектом с элементом head и элементом tail. Например, в списке List(4, 7) элемент head имеет значение 4, а tail соответственно 7.

Правоассоциативный оператор «::» создаёт новый список из заданных значений.

Списки в Scala обладают огромным количеством встроенных функций, которые позволяют обрабатывать элементы структуры данных без введения дополнительных действий, например:

```
scala> List(5, 5, 5).sum
res0: Int = 15
```

Изменяемый тип LinkedList аналогичен List, за исключением того, что позволяет изменять голову присваиванием по ссылке elem и хвост присваиванием по ссылке next. Например, следующий цикл удалит каждый второй элемент из списка:

```
val lst = scala.collection.mutable.LinkedList(1, 2, 3, 4)
var cur = lst
while (cur != Nil && cur.next != Nil) {
    cur.next = cur.next.next
    cur = cur.next
}
```

Есть тип DoubleLinkedList, который отличается от вышеописанного лишь тем, что обладает изменяемой ссылкой prev.

3. Множества

Множество – это коллекция, в которой каждый элемент может присутствовать в единственном экземпляре. Вы можете выполнять добавление нового такого же элемента в данную коллекцию – это не приведёт к иным последствиям, кроме тех, что коллекция останется в прежнем состоянии.

Множества не сохраняют порядок следования элементов, также чаще всего они реализованы как хеш-множества, что позволяет использовать метод `hashCode`.

Однако если есть необходимость сохранения порядка добавления элементов, то можно использовать связанное хеш-множество (следует принять во внимание существенную потерю производительности).

Если необходимо обойти множество в порядке сортировки, то можно использовать сортированное множество `scala.collection.immutable.SortedSet`, которое организовано как красно-черные деревья.

Метод contains проверяет наличие в множестве указанного значения.

Метод subset проверяет, входят ли все элементы множества в другое множество.

```
scala> val digits = Set(1, 2, 3, 4)
scala> digits contains 0
res1: Boolean = false
scala> Set(1, 2) subsetOf digits
Res2: Boolean = true
```

Методы union, intersect и diff реализуют наиболее типичные операции над множествами. Существуют также операции объединения «++» и вычитания «--» над множествами.

4. Абстракции

При построении компонентных систем наиболее важным и серьёзным вопросом является процесс абстрагирования от иных модулей. В языках программирования существует два основных способа решения данной дилеммы: абстрактные члены и параметризация. Первый способ применяется для объектов, а второй обычно для функций.

Приведённый ниже класс представляет собой trait, являющийся функциональной абстракцией ячейки, доступной на чтение и запись:

```
class Cell[T](init: T)
{
    private var value: T = init;
    def get: T = value;
    def set(x: T) : unit = { value = x }
}
```

Класс абстрагирует значение типа, определяемого параметром T. Как известно, методы тоже обладают возможностью иметь параметры типов.

Описанная выше функциональная абстракция может быть заменена объектно-ориентированной, которая может использоваться как альтернативная:

```
abstract class Cell
{
    type T;
    val init: T;
    private var value: T = init;
    def get: T = value;
    def set(x: T) : unit = { value = x }
}
```

Данный класс содержит абстрактный член типа T с абстрактной переменной init. Экземпляры данного класса могут быть созданы с помощью реализации этих абстрактных членов с конкретными определениями:

```
val cell = new Cell
{
```

```
    type T = int;
    val init = 1
}
cell.set(cell.get * 5)
```

В данном примере тип класса Cell определён: { type T = int }. Это позволяет исходному коду работать со значением cell, как с известным типом. Исходя из этого, допустимы операции над cell, приведённые выше.

Задание к лабораторной работе:

На данном этапе каждая сохранённая игра представляет собой последовательность действий, хранимых в списке и прочитанных из файла. Используя модель абстракции списков (list comprehension) необходимо построить карту игры.

Данная карта должна содержать статистическую информацию об играх: например, наиболее часто посещённое поле, наиболее часто используемое движение и т.д.

Требования к защите:

– Окно со статистикой по сыгранным играм, содержащее информацию, описанную выше в задании к лабораторной работе

– Иметь представление в области существующих в Scala коллекций

– В отчёте должны быть отражены выбранные статистические показатели и полученные результаты

Лабораторная работа №7

Изменяемое состояние. Сопоставление с образцом и case-классы. Параметризованные типы

Краткие теоретические сведения:

Введение

1. Изменяемое состояние
2. Сопоставление с образцом и case-классы
3. Параметризованные типы

Введение

Scala обладает качественным механизмом сопоставления с образцами. Основные области применения: инструкции-переключатели, извлечение частей сложных выражений и определение типа. Для реализации данного механизма чаще всего используются case-классы.

В Scala возможно использовать параметры типов при реализации необходимой функциональности. Эта функциональность многим напоминает шаблонные конструкции в C-подобных языках программирования, однако в них нет возможности накладывания ограничений на созданный тип: в Scala же данная функциональность присутствует.

1. Изменяемое состояние

Данный раздел содержит рассуждения на тему того, по каким причинам апологеты функционального программирования поощряют неизменяемые состояния. Доказательство основывается в основном на концентрации внимания на опасностях изменяемого состояния:

– неявные изменения – отсутствие целостности внутреннего состояния программы, выполнение нескольких инвариантов (например, совпадение полей наследуемых объектов);

– кэширование и разделение – осложнение корректного кэширования;

– многопоточность – несколько корректных последовательностей изменений, переплетаясь в условиях многопоточности, в результате образуют некорректную;

– сложный код – с появлением изменяемых данных в коде появляется сущность: время – как на уровне взаимодействия компонентов, так и на уровне последовательности строк кода;

– наследование – необходимо следить, что класс-наследник всегда удовлетворяет принципу подстановки Лисков, что затруднительно, так как часто класс-наследник накладывает дополнительные ограничения на изменяемые состояния, чего делать не должен.

Следует обратить внимание, что рассмотренные пункты относятся к объектам, имеющим в своём определении/процессе существования различные состояния или возможность различных состояний.

Исходя из практик использования неизменяемых состояний и stateless сервисов можно сделать вывод, что там данные проблемы отсутствуют, что приводит к экономии рабочего времени, затраченного на проект, однако следует считаться с рисками уникальности исходного кода и экзотики техник программирования.

2. Сопоставление с образцом и case-классы

Сопоставление с образцом является обобщением выражения switch в C/Java для иерархий классов, только вместо switch используется стандартный метод match, доступный для всех объектов.

Массово сопоставление разделяют на сопоставление с типами и сопоставление со структурами данных. Например, сопоставление с типом может выглядеть следующим образом:

```
obj match {
    case s: String      => Integer.parseInt(s)
    case z: Int         => z
    case _: BigInt      => Int.MaxValue
    case _         => 0
}
```

Данная форма записи является более предпочтительной, чем использование оператора isInstanceOf. В приведённом примере следует обратить внимание на несколько моментов. В первом случае сопоставление будет выполнено с переменной s как с типом String. Во втором случае аналогично с целочисленным типом. В третьем случае сопоставление соответствует любому объекту BigInt. В четвёртом случае представлен эквивалент С-подобной конструкции default – универсальный образец. Считается хорошим тоном включать его в сопоставление с целью предотвращения исключения MatchError.

Для того, чтобы выполнить сопоставление с заполненной структурой данных, необходимо её же использовать в качестве образца. Например, для Array это будет выглядеть следующим образом:

```
arr match {
    case Array(0) => "0"
    case Array(x, y)    => x + " " + y
    case _         => "default"
}
```

В данном примере объект-компаньон Array является экстрактором – объектом с методом unapply или unapplySeq, который позволяет извлекать значение из объекта.

Case-классы – это оптимизированные для использования в операциях сопоставления классы. Процедура объявления case-класса сопровождается следующими действиями:

- если параметры не объявлены как var, они становятся значением val;
- создаётся объект-компаньон с методом apply и unapply;
- в отсутствии методов toString, equals, hashCode, copy они генерируются.

3. Параметризованные типы

Для определения параметра типа в Scala используются квадратные скобки:

```
class DictionaryItem[K, V](val key: K, val value: V)
```

Класс, в котором присутствуют параметры типов, называется обобщённым (generic). Методы и функции также обладают данной особенностью (параметр типа помещается после имени). Для того, чтобы ограничить диапазон изменения, например, верхней границей, можно использовать:

```
class DictionaryItem[K <: Comparable[K]](val key: K, val value: V) {
    def smaller = if (first.compareTo(value) < 0) key else value
}
```

Следует обратить внимание, что в таком случае тип K должен быть подтипом Comparable[K]. Благодаря данному ограничению вы сможете создать экземпляр

DictionaryItem[java.io.String], но не сможете DictionaryItem[java.io.File], потому что File не реализует интерфейс Comparable[File].

Был рассмотрен пример границы изменения типов, также существуют границы представления и границы контекста, множественные границы.

Задание к лабораторной работе:

Используя технику сопоставления с образцом выполнить одно из следующих заданий (на выбор):

- поиск и вывод наиболее длинной последовательности произведённых операций, повторившейся более одного раза
- поиск наиболее часто встречающейся последовательности и расчёт её содержания в общем количестве последовательностей
- подмена разработанной нотации псевдокодом для повышения её читаемости неискушённым знаниями нотации пользователем

Требования к защите:

- Выбрано не менее одного варианта из задания к реализации
- Отчёт должен содержать используемый алгоритм сопоставления
- Обладать понятиями изменяемых состояний и обстоятельств их использования

Лабораторная работа №8

Отложенные вычисления. Абстракции многопоточности. Модель акторов. Парсинг. Комбинаторы синтаксического анализа

Краткие теоретические сведения:

Введение

1. Ленивые вычисления
2. Модель акторов
3. Комбинаторы синтаксического анализа

Введение

Одним из важных преимуществ Scala является её реализация для многопоточных приложений, так как сам подход функционального программирования позволяет создавать сложные отказоустойчивые системы, ориентированные на корректную работу в любых условиях. В рамках данной лабораторной работы вам предстоит познакомиться со стандартными паттернами параллельного программирования.

1. Ленивые вычисления

К данной стратегии вычисления относят методику, согласно которой вычисления следует откладывать до тех пор, пока не потребуется их результат. Отложенные вычисления относятся к нестрогим вычислениям.

Рассмотрим следующий пример: найти второе простое число между 1000 и 10000.

```
scala> def isPrime(n: Int):Boolean = (2 until n) forall (d => n % d != 0)
scala> ((1000 to 10000) filter isPrime)(1)
```

Этот код верен, однако он вычисляет все простые числа в заданном промежутке, хотя нам в этом нет необходимости – необходимо найти лишь второй простое число.

Можно попытаться снизить предел с 10000 до 5000, однако есть более качественное решение: избежать вычисления хвоста до тех пор, пока явно не возникнет необходимость это сделать.

Структура данных Stream построена по данному принципу и напоминает собой список, за исключением описанной выше особенности, которой мы желаем достичь:

```
scala> Stream.cons(1, Stream.cons(2, Stream.empty))
scala> Stream(1, 2, 3)
```

Однако данная реализация не является произведением искусства, если элемент списка затребован несколько раз, ибо в таком случае каждый раз будет вычисляться список до этого элемента. Обходным путём для такой ситуации является сохранение вычисленного значения.

Этот приём имеет название «ленивая инициализация» и для его реализации используется ключевое слово lazy. Таким образом, модификация будет выглядеть так:

```
def cons[T](_head: T, _tail: => Stream) = new Stream {
    def isEmpty = false
    def head = _head
    lazy val tail = _tail
}
```

2. Модель акторов

Акторы являются конкурентным механизмом реализации параллельных вычислений, созданным в альтернативу традиционным механизмам на основе блокировок.

Актором является класс, наследующий трейт Actor, который обладает методом act – реализуя данный метод, вы приводите поведение актора. Частой практикой является реализация актора согласно модели почтового ящика: приём и отправка сообщений.

Ниже приведён пример актора и способа его создания/вызова:

```
import scala.actors.Actor

class SantaActor extends Actor {
    def act() {
        while (true) {
            receive {
                case "HowHowHow" => println("I'm afraid…")
            }
        }
    }
}

val claus = new SantaActor
claus.start()
```

После того, как вы создали экземпляр, метод act будет исполняться параллельно, и вы можете посылать ему сообщения. Актор является объектом, обрабатывающим асинхронные сообщения. Сообщение отправляется посредством оператора !:

```
claus ! "HowHowHow"
```

Как вы могли обратить внимание, на роль сообщений хорошо подходят case-классы, благодаря чему актор для обработки может использовать сопоставление с образцом. Для приёма сообщений используется метод receive, который при вызове извлекает из «почтового ящика» очередное сообщение.

Если необходимо превратить асинхронное взаимодействие в синхронное, то можно воспользоваться оператором ?!, в случае чего актор может отправить сообщение и ждать ответа. Чтобы этот приём работал, получатель должен вернуть сообщение отправителю. Отправитель же блокируется, пока не получит ответное сообщение.

Метод act запускается после вызова метода start и обычно входит в цикл по приёму сообщений. Работа актора завершается в следующих случаях:

- метод act возвращает управление;
- работа метода act прерывается исключением;
- актор вызывает метод exit.

3. Комбинаторы синтаксического анализа

Простейшие анализаторы легко справляются с терминальными символами, однако когда дело доходит до полноценного анализа входных цепочек лексем, то также надо учитывать и нетерминальные символы. Для более общего случая и существуют отдельные функции высших порядков, или, так называемые, комбинаторы синтаксического анализа.

Такие комбинаторы получают на вход два «парсера» и возвращают новый, реализующий определённую функциональность на основе функциональности двух заданных «парсеров».

В Scala включен пакет scala.util.parsing, который включает в себя инструменты для описания грамматики External DSL на Scala – это «внутренний DSL» для создания «внешних DSL».

С целью осознания принципов работы анализаторов, необходимо погрузиться в методики реализации комбинаторов. Каждый «парсер» представляет собой функцию (или case-класс), которая получает некоторые входные данные и возвращает объект-парсер. Комбинаторы самого нижнего уровня базируются на простейших «парсерах», которые

принимают на вход объекты, считывающие элементы из входного потока и возвращающие объекты, обладающие некоторый высокоуровневой семантикой.

Разумеется, библиотека комбинаторов «парсеров» значительно более обширна, чем может показаться. Фактически комбинаторы «комбинируют» «парсеры» в целях повышения уровня абстракции и реализации нужной схемы синтаксического анализа.

Для желающих более подробно изучить построение своего собственного синтаксического анализатора рекомендуется к прочтению http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.pdf

Задание к лабораторной работе:

Задание предоставляется на выбор студента:

- изменить серверную часть приложения с целью использования модели акторов и концепцию отложенных вычислений, произвести измерение изменения производительности ввиду добавленной функциональности
- с целью модификации «предсказателя» из 4-ой лабораторной работы (теперь предсказания должны быть основаны на статистике сыгранных игр), произвести построение синтаксического дерева, используя один из следующих путей:
  - парсеры на основе регулярных выражений
  - парсеры на основе лексем
  - комбинаторы синтаксического анализа

Требования к защите:

- Реализация выбранного варианта по заданию к лабораторной работе
- Оценка изменения производительности серверной части/«предсказателя»
- Финальная подготовка и защита отчёта