Программы на языке Java состоят из классов, которые представляют объекты реального мира. Люди понимают по-­‐разному, что значит хороший стиль

программирования, однако большинство согласно, что нужно стараться программировать в так называемом объектно-­‐ориентированном стиле. Это значит, что хорошие программисты сначала решают, какие объекты включить в программу и какие Java классы будут их представлять, а только потом уже они начинают писать программу.

***Классы и объекты***

Давайте придумаем и обсудим класс, который будет называться VideoGame (видео игра). Этот класс может иметь несколько методов, из которых будет ясно, что могут

Делать объекты этого класса: начать игру, остановить её, сохранить (запомнить) счет и так далее.

А ещё этот класс может иметь какие-­‐нибудь атрибуты, например цена, цвет экрана, количество ручек управления, и все такое прочее.

Классы Java могут иметь и методы и атрибуты.

Методы определяют, что класс может сделать.

Атрибуты – это характеристики класса.

Во многих случаях программа может пользоваться классом только после создания его экземпляра.

Не смотря на то, что эти копии представляют тот же самый класс, их атрибуты могут иметь разные значения -­‐ какие-­‐то из них голубые, какие-­‐то перламутровые, и так далее.

Иными словами, программа может создавать множество экземпляров объектов GameBoyAdvance.

***Создаём Домашнее Животное***

Давайте придумаем и создадим класс Домашнее Животное (по- английски просто Pet).

Сначала нужно решить, какие действия наш Pet сможет выполнять. Как насчет есть, спать и говорить (eat, sleep, say) ?

Мы запрограммируем эти действия в методах класса Pet.

А ещё мы дадим Нашему домашнему животному такие аттрибуты: возраст (age), рост (height), вес (weight) и цвет(color).

Начнем с создания нового класса по имени Pet в проекте *My First Project*, но при этом не ставьте птичку, требующую создания метода main().



Теперь мы готовы объявить аттрибуты и методы в классе Pet.

Код в классах и методах должен быть окружен фигурными скобками. Каждая отрывающая скобка должна иметь свою закрывающую:

**class** Pet{

}

Давайте выберем типы данных для аттрибутов нашего класса. Я предлагаю int для возраста, float для веса и роста, String для цвета.

**class** Pet{

**int** age;

**float** weight;

**float** height;

String color;

}

Добавим несколько методов в наш класс. Здесь нужно решить будут ли эти методы иметь параметры и возвращать какие-­‐нибудь данные:

Метод sleep() будет просто печатать фразу *Спокойной ночи, до завтра* – ему не нужны никакие параметры и он не будет возвращать никаких значений.

То же самое относится и к методу eat()– он будет печатать сообщение *Я очень голоден…давайте перекусим чипсами!*

Хотя метод say() тоже будет печатать сообщение, но наше домашнее животное будет ещё и “произносить” слово или фразу, которую мы ему дадим, как *параметр*. Этот метод будет строить фразу, включающую значение этого параметра, после чего этого фраза будет возвращаться вызывающей программе. Новая версия класса Pet будет выглядет так:

**public class** Pet {

**int** age;

**float** weight;

**float** height;

String color;

**public void** sleep(){

System.out.println("Спокойной ночи! До завтра");

}

**public void** eat(){ System.out.println(

"Я очень голоден, давайте перекусим чипсами!");

}

**public** String say(String aWord){

String petResponse = "Ну ладно!! " +aWord;

**return** petResponse;

}

}

Давайте поговорим о заголовке (сигнатуре) метода sleep():

**public void** sleep()

Этот заголовок говорит нам, что метод sleep() можно вызывать из любого другого класса (public) и что метод не возвращает никаких данных (void). Пустые скобочки значат, что этот метод не имеет параметров – ему ведь не нужны никакие данные из окружающей среды, потому что он всегда печатает один и тот же текст.

Заголовок метода say() выглядит так: **public** String say(String aWord)

Он ещё и возвращает какой-­‐то текст – это и есть роль ключевого слова String, стоящего перед именем метода. А кроме того, этот метод ожидает какие-­‐то текстовые данные извне и для этого в заголовок метода включен параметр String aWord.

А как же определить – должен ли метод возращать данные?

Если метод выполняет какие-­‐то действия над данными и должен передать результат этих действий вызывающему классу, то он должен возвращать данные. Вы можете возразить, что у класса Pet нет никакого вызывающего класса!

Верно, поэтому мы сейчас и создадим класс PetMaster (Хозяин).

У этого класса будет метод main(), содержащий код для работы с классом Pet. Создайте новый класс PetMaster, но на этот раз поставьте птичку в Eclipse возле вопроса *создавать-­‐ли метод main()*. Не забывайте, что без этого метода вы не сможете стартовать программу. Добавьте несколько строчек кода к классу, который сделал для вас Eclipse, чтобы он выглядел так:

**public class** PetMaster {

**public static void** main(String[] args) {

String petReaction;

Pet myPet = **new** Pet();

myPet.eat();

petReaction = myPet.say("Чик!! Чирик!!");

System.out.println(petReaction);

myPet.sleep();

}

}

PetMaster – это вызывающий класс и он сначала создает экземпляр объекта Pet. Он объявляет переменную myPet and и использует оператор new:

Pet myPet = **new** Pet();

Эта строчка объявляет переменную myPet типа Pet (да это так, вы можете использовать любые классы, созданные вами, как новые типы данных).

Теперь переменная myPet знает место в памяти, где был создан экземпляр объекта Pet, и можно пользоваться этой переменной, чтобы вызывать любые методы класса Pet, например: myPet.eat();

Если метод возвращает какое-­‐нибудь значение, его можно вызывать по-другому. Объявите переменную того-­‐же типа, что и возвращаемое значение и вызывайте метод так, чтобы присвоить это значение переменной, например так:

String petReaction;

petReaction = myPet.say("Чик!! Чирик!!");

Теперь возвращенное значение находится в переменной petReaction и его очень просто можно распечатать: System.out.println(petReaction);

***Наследование – Рыбка Тоже Домашнее Животное***

Наш класс Pet познакомит вас с ещё одной важной особенностью языка Java, которая называется *наследование (inheritance)*.

В реальном мире каждый человек наследует что-­‐то от своих родителей. В мире Java вы тоже можете создать новый класс по типу уже существующего.

Класс Pet, и ведет себя, и имеет атрибуты типичные для многих домашних животных – они едят, спят, некоторые из них издают звуки, их кожа имеет цвет и так далее.

С другой стороны, домашние животные отличаются друг от друга – собаки лают, рыбки беззвучно плавают, попугайчики разговаривают лучше, чем собаки. И все-­‐ е, все они спят, едят, и имеют рост и вес.

Поэтому гораздо легче создать класс Fish (рыба) так, чтобы он унаследовал общие черты и поведение у класса Pet, чем каждый раз создавать с начала классы для собак, попугаев и рыб.

Для этого и существует специальное ключевое слово extends:

Теперь вы имеете полное право сказать что Fish -­‐ это *подкласс* класса Pet, а класс Pet – это *супер-­‐класс* класса Fish.

Мы использовали класс Pet, как свообразный шаблон для создания класса Fish.

**class** Fish **extends** Pet{

}

Даже если вы оставите класс Fish таким, как он есть сейчас, всё равно уже можно использовать каждый метод и атрибут, унаследованный из класса Pet.

Вот посмотрите: Fish myLittleFish = **new** Fish();

myLittleFish.sleep();

Хоть мы ещё и не объявляли никаких методов в классе Fish, уже можно вызывать метод sleep(), находящийся в его супер-­‐классе!

Нет ничего легче создания подклассов в приложении Eclipse! Выберите меню *File*, *New*, *Class* и напечатайте слово Fish, как имя класса.

Замените в поле супер-­‐класс java.lang.Object на слово Pet.

Не забывайте, что мы создаем подкласс класса Pet, чтобы добавить то, что присуще только рыбам, а общий для всех животных код, объявленный в супер-­‐классе, мы просто используем.

Не все домашние животные могут нырять, но рыбки, конечно же, могут. Давайте добавим к классу Fish метод dive() -­‐ нырни.

У метода dive() есть параметр howDeep, который “говорит” рыбке, как глубоко она должна нырнуть.

А ещё мы объявили переменную currentDepth, куда будем помещать текущее значение глубины при каждом вызове метода dive().

Этот метод возвращает значение переменной currentDepth вызывающему классу. Сделайте, пожалуйста, вот такой класс

FishMaster:

Пора рассказать маленький секрет – все классы в языке Java унаследованы из супер-­‐дупер класса Object,

**public class** Fish **extends** Pet {

**int** currentDepth=0;

**public int** dive(**int** howDeep){

currentDepth=currentDepth + howDeep;

System.out.println("Ныряю на глубину "

+ howDeep + " футов");

System.out.println("Я на глубине "

+ currentDepth + " футов ниже уровня моря");

**return** currentDepth;

}

}

Метод main() в классе FishMaster создает экземпляр объекта Fish и дважды вызывает его метод dive() с разными параметрами.

После этого он вызывает метод sleep().

**public class** FishMaster {

**public static void** main(String[] args) {

Fish myFish = **new** Fish();

myFish.dive(2);

myFish.dive(3);

myFish.sleep();

}

}

Во время выполнения FishMaster напечатает следующее:

Ныряю на глубину 2 футов

Я на глубине 2 футов ниже уровня моря

Ныряю на глубину 3 футов

Я на глубине 5 футов ниже уровня моря

Спокойной ночи, до завтра

Вы заметили, что FishMaster вызывает не только методы объявленные в классе Fish, но также и метод sleep() его супер-­‐класса Pet?

То-­‐то! В этом и есть вся прелесть наследования – вам не нужно копировать код из класса Pet. Просто напишите слово extends и класс Fish сможет пользоваться методами класса Pet! 􀀁

Да, вот ещё что, хотя метод dive() и возвращает значение переменной currentDepth, наш FishMaster им не пользуется. Это не беда, просто нашему классу FishMaster оно не нужно.

Но каким-­‐нибудь другим классам, которые тоже могут работать с классом Fish, это значение может быть очень даже полезно. Представьте, например, класс FishTrafficDispatcher (регулировщик движения рыб), который должен знать положения других рыб в море, прежде чем разрешить ныряние во избежание дорожно-­‐транспортных происшествий.

***Переопределение методов***

Вы, конечно, знаете, что рыбы не говорят (по крайней мере он не делают это громко).

Но наш класс Fish был унаследован из класса Pet, у которого есть метод say().

Это значит, что вы беспрепятственно можете написать что-­‐то в этом роде: myFish.say();

Ну и ну, наши рыбки заговорили…

Чтобы избежать этого, в классе Fish нужно переопределить (override) метод say(), обьявленный в классе Pet.

Это работает так: если вы объявляете в под-­‐классе метод имеющий точно такой-­‐ е заголовок, как в его-­‐же супер-­‐классе, Java выполнит метод под-­‐класса, вместо метода супер-­‐класса.

Давайте добавим к классу Fish метод say().

**public** String say(String something){

**return** "Ты чё не знаешь, что рыбы не разговаривают?";

}

А теперь вызовем метод say() из метода main() класса FishMaster: myFish.say("Привет"); Выполните эту программу и она напечатает следующее:

Ты чё не знаешь, что рыбы не разговаривают?

Это подтверждает, что метод say() класса Pet был переопределен.

Давайте передохнём.

 Если заголовок метода включает ключевое слово final, такой метод переопределить нельзя, например:

**final public void** sleep(){…}

***Практические упражнения***

1.

Создайте новый класс Car (автомобиль) и включите в него следующие

методы:

**public void** start()

**public void** stop()

**public int** drive(**int** howlong)

Метод drive() (едь) должен возвращать общее расстояние пройденное автомобилем за заданное время.

Используйте следующую формулу дла рассчёта расстояния: distance = howlong\*60;

2. Создайте ещё один класс CarOwner (хозяин автомобиля), который будет создавать экземпляр объекта Car и вызывать его методы. Результат каждого такого вызова должен быть напечатан с помощью

System.out.println().

***Практические упражнения для умников и умниц***

Сделайте подкласс класса Car, назовите его JamesBondCar (автомобиль Джемса Бонда) и переопределите в нем метод drive().

Используйте следующую формулу дла рассчёта расстояния: distance = howlong\*180;

Будьте изобретательны! Печатайте смешные сообщения.

***Специальные методы: конструкторы***

В Java для создания экземпляров классов и выделения под них памяти используется оператор new, например:

Fish myFish = **new** Fish();

Круглые скобки после слова Fish говорят о том, что у этого класса определен метод Fish().

Так и есть, существуют специальные методы, которые называются *конструкторами (constructors)*, и у этих методов есть следующие особенности:

Конструкторы вызываются только один раз при создании объекта в памяти.

Ни должны называться так же, как называется класс.

Они ничего не возвращают, не нужно даже писать слово void в сигнатуре этого метода.

У класса может быть несколько конструкторов. Если вы не написали ни одного конструктора, во время компиляции Java автоматически создаст за вас так называемый *пустой конструктор по умолчанию (default no-­‐ argument constructor)*.

Вот почему компилятор никогда не будет “ругаться” на выражение new Fish(), даже если в классе Fish вы не объявили ни одного конструктора.

В основном, конструкторы используются для присваивания начальных значений атрибутам класса, к примеру, следующая версия класса Fish включает конструктор с одним аргументом, который задаёт начальное значение атрибута currentDepth равным значению аргумента конструктора.

**public class** Fish **extends** Pet {

**int** currentDepth;

Fish(**int** startingPosition){

currentDepth=startingPosition;

}

}

Теперь класс FishMaster может создать экземпляр класса Fish и задать начальное положение рыбки. Ниже создаётся экземпляр класса Fish, который изначально погружает рыбку в море на глубину 20 метров:

Fish myFish = **new** Fish(20);

Для класса, в котором был определён конструктор с аргументами, конструктор по умолчанию создаваться автоматически не будет. Если вам необходим конструктор без аргументов -­‐ напишите его.

**Освоюємо Java/Графічні компоненти Swing**

<https://uk.wikibooks.org/wiki/%D0%9E%D1%81%D0%B2%D0%BE%D1%8E%D1%94%D0%BC%D0%BE_Java/%D0%9C%D0%B5%D0%BD%D0%B5%D0%B4%D0%B6%D0%B5%D1%80%D0%B8_%D1%80%D0%BE%D0%B7%D1%82%D0%B0%D1%88%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F>

# Справочник по Java Swing GUI

<http://darkraha.com/rus/java/swing/swing09.php>

### **WindowBuilder - is a powerful and easy to use bi-directional Java GUI designer**

<https://www.eclipse.org/windowbuilder/>

Java содержит широкий набор классов, которые позволяют создавать графические приложения. Существует две основные группы классов для создания окон в Java. ***AWT и Swing***

В первой версии языка Java для работы с графикой имелась только библиотека -­‐ AWT.

Эта библиотека – простой набор классов, таких, как Button (кнопка), TextField (текстовое поле), Label (текстовая метка или иконка) и другие.

Вскоре была создана более совершенная библиотека, которую назвали Swing. Она так же включает в себя кнопки, текстовые поля и другие элементы управления графическими приложениями. Названия компонентов этой библиотеки начинается с буквы J. Например, JButton, JTextField и так далее.

Всё в Swing чуточку лучше, быстрее и удобнее.

***Пакеты и ключевое слово import***

Java поставляется с большим количеством полезных классов, которые организованы в пакеты (*packages*). Некоторые пакеты содержат классы для рисования графики, другие – классы для работы с интернетом и так далее. Например, класс String находится в пакете с названием java.lang и полное имя этого класса

java.lang.String.

Компилятор Java знает, где найти классы, находящиеся в java.lang, поэтому я не указывал явно полное имя String в предыдущих примерах кода, но существует много других пакетов с полезными классами и ваша задача сообщить компилятору, в каком пакете содержатся классы, используемые в программе.

Например, большинство классов библиотеки Swing находятся в следующих двух пакетах:

javax.swing

javax.swing.event

Было бы очень утомительно каждый раз, когда используется класс, писать его полное имя. Чтобы избежать этого, вы можете написать ключевое слово import всего один раз перед объявлением класса, как показано в примере:

**import** javax.swing.JFrame;

**import** javax.swing.JButton;

**class** Calculator{

JButton myButton = **new** JButton();

JFrame myFrame = **new** JFrame();

}

Ключевое слово import позволяет использовать короткие имена классов, такие как JFrame или JButton и сообщает компилятору, где искать эти классы.

Если нужно использовать несколько классов из одного пакета, нет необходимости перечислять каждый из них в строке с import, можно просто использовать символ \*.

**import** javax.swing.\*;

***Основные элементы Swing***

Вот некоторые основные объекты, из которых состоят Swing-­‐ приложения:

Окно или фрейм (*frame),* который может быть создан с помощью класса JFrame.

Невидимая панель (*panel)* или, как ещё её называют, *pane (оконное стекло)* содержит все кнопки, текстовые поля, метки и другие компоненты. Панели создаются с помощью класса JPanel.

Оконные элементы управления, такие как кнопки JButton, текстовые поля JTextfield, списки JList, и так далее.

*Схемы размещения* (layout managers) компонент, которые помогают организовать все эти кнопки и поля на панели. Например, можно создать экземпляр класса JPanel и назначить для него схему размещения. Затем создайте различные графические компоненты и добавьте их на панель. После этого добавьте панель на фрейм, задайте его размер и сделайте его видимым.

Но отображение фрейма это только половина работы. Нужно ещё добавить обработку различных событий, например нажатий на кнопки. В этой главе я расскажу, как создавать окна с компонентами, а в следующей – обрабатывать события (*events*), которые могут произойти с компонентами окна.

Наша основная цель в этой главе – написать калькулятор, который позволяет сложить два числа и увидеть результат.

Создайте новый проект в Eclipse, назовите его *My Calculator* и добавьте в него новый класс SimpleCalculator со следующим кодом:



Это, может быть, не самый красивый в мире калькулятор, но в данном примере видно, как добавлять компоненты и отображать окно. В следующей секции мы попробуем сделать более красивый интерфейс с помощью схем размещения*.*

***Схемы Размещения***

В некоторых старомодных языках программирования необходимо было указывать координаты и размеры каждого компонента окна. Это работало хорошо, если было известно разрешающая способность экрана каждого пользователя.

В Java есть схемы размещения (Layout Managers), которые позволяют разместить компоненты на экране, не зная точных позиций компонентов. Схемы гарантируют, что та часть интерфейса, за которую они отвечают, будет выглядеть правильно вне зависимости от размеров окна и разрешения экрана.

Swing предоставляет следующие схемы:

FlowLayout

GridLayout

BoxLayout

BorderLayout

CardLayout

GridBagLayout

Чтобы использовать любой из этих менеджеров, необходимо создать его экземпляр и затем назначить этот объект какому-­‐нибудь *контейнеру (container)*, например панели, как это было в примере с SimpleCalculator.

***FlowLayout - построчное расположение***

По этой схемe компоненты размещаются в окне (или другом контейнере) строка за строкой.

Например, текстовые метки, иконки, текстовые поля и кнопки будут добавляться в первую условную строку, пока в ней есть место. Когда первая строка заполнится, оставшиеся компоненты будут добавляться в следующую строку и так далее.

Если пользователь изменит размер окна, картина может измениться. Просто потяните за угол калькулятора, чтобы поменять его размер. Посмотрите как java.awt.FlowLayout переупорядочивает элементы окна во время изменения его размеров.

Согласен, FlowLayout не лучшим образом подходит для нашего калькулятора. Давайте теперь попробуем что-­‐нибудь другое.

***GridLayout - табличное расположение***

Класс java.awt.GridLayout позволяет организовать компоненты, как *строки* и *столбцы* в таблице. Компоненты будут добавляться в ячейки условной таблицы. Если размер окна будет увеличен, ячейки станут больше, но положение компонентов относительно друг друга останется прежним.

В нашем калькуляторе семь компонентов – три текстовые метки, три текстовых поля и кнопка. Мы можем разместить их в таблице с четырьмя строками и двумя колонками (одна ячейка останется пустой):

GridLayout gr = **new** GridLayout(4,2);

Также можно задать расстояние между ячейками по вертикали и горизонтали, например в пять пикселей:

GridLayout gr = **new** GridLayout(4,2,5,5);

После небольших изменений в нашем калькуляторе (они подсвечены ниже), он станет выглядеть гораздо симпатичнее. А теперь создайте и скомпилируйте новый класс SimpleCalculatorGrid в проекте *My Calculator*.

После запуска программы SimpleCalculatorGrid, вы увидите такое окно:



Попробуйте поменять размеры этого окна – размеры элементов управления будут меняться вместе с ним, но их положение относительно друг друга не изменится:



Ещё одна важная вещь, которую стоит запомнить про табличный компоновщик – в нем все ячейки имеют одинаковую длину и ширину.

***BorderLayout - размещение по областям***

Класс java.awt.BorderLayout разделяет окно на Южную, Западную, Северную, Восточную и Центральную области. Северная область находится наверху окна, Южная – снизу, Западная – слева, а Восточная – справа.

Например, в калькуляторе, который будет продемонстрирован на следующей странице, текстовые поля, которые отображают числа, находятся в Северной области.

Создать BorderLayout и поместить в него текстовые поля можно следующим образом:

BorderLayout bl = **new** BorderLayout();

**this**.setLayoutManager(bl);

JTextField txtDisplay = **new** JTextField(20);

**this**.add("North", txtDisplay);

Совсем не обязательно помещать элементы управления во все пять областей. Если необходимы только Северная, Центральная и Южная области, то Центральная станет шире, т.к. Западная и Восточная пустуют.

Я буду использовать BorderLayout чуть позже, в следующей версии нашего калькулятора, который будет называться Calculator.java.

***Комбинирование схем размещения***

Как вы думаете, можно ли с помощью GridLayout создать калькулятор, который будет выглядеть так же, как стандартный калькулятор в Microsoft Windows?

К сожалению, нет, так как ячейки этого калькулятора имеют разные размеры – текстовое поле больше кнопок. Но содержимое окна можно представить с помощью нескольких панелей, у которых схемы разные.



Попробуем использовать комбинацию нескольких схем в новом калькуляторе. Для этого необходимо выполнить следующие шаги:

Назначить BorderLayout панели фрейма, которая будет основной, и в которой будут содержаться остальные панели.

Добавить JTextField в северную часть, для того чтобы отображать введённые числа.

Создать панель p1 с GridLayout, добавить на неё 20 кнопок и затем поместить эту панель в центральную область основной панели.

Создать панель p2 с GridLayout, добавить на неё четыре кнопки и затем поместить панель p2 в западную область основной панели.

Давайте начнём с более простой версии калькулятора, которая будет выглядеть вот так:



Создайте новый класс Calculator и запустите программу. Чтобы понять, как она работает, прочитайте комментарии в примере кода, продемонстрированном ниже.

***GridBag Layout - более гибкое табличное расположение***

А сейчас я покажу ещё один способ создания окна калькулятора. Здесь будет использоваться java.awt.GridBagLayout вместо комбинации схем и панелей.

В нашем калькуляторе есть строки и столбцы, но в GridLayout они обязаны иметь одинаковые размеры. Это не подходит, так как у нас есть поле для ввода, ширина которого равна ширине трёх кнопок с числами.

GridBagLayout -­‐ более продвинутая схема размещения. Она позволяет задавать размер ячейки, равным нескольким клеткам таблицы. GridBagLayout имеет вспомогательный класс, который называется GridBagConstraints (ограничения на клетки таблицы). Эти ограничения не что иное, как атрибуты ячеек, которые необходимо задавать для каждой ячейки таблицы отдельно.

Все ограничения должны быть заданы *до того,* как в ячейку помещаются компоненты. Например, один из атрибутов GridBagConstraints называется gridWidth. Он позволяет задать ширину какой-­‐то одной ячейки, равной ширине нескольких других.

Во время работы с GridBagLayout необходимо сначала создать экземпляр класса GridBagConstraints, и затем задать значения для его свойств. После того как это сделано, можно добавлять объект в ячейку контейнера.



Следующий пример кода, усыпан комментариями, которые помогут понять, как использовать GridBagLayout.

// Задаём GridBagLayout для панели окна

GridBagLayout gb = **new** GridBagLayout();

**this**.setLayout(gb);

// Создаём экземпляр класса GridBagConstraints

// Эти строки кода нужно повторить для каждой компоненты

// которая добавляется в ячейку

GridBagConstraints constr = **new** GridBagConstraints();

//задаём ограничения для строки ввода калькулятора

// координата x в таблице

constr.x=0;

// координата y в таблице

constr.y=0;

// эта ячейка имеет такую же высоту, как стандартные ячейки

constr.gridheight =1;

// эта ячейка имеет ширину равную ширине 6 стандартных ячеек

constr.gridwidth= 6;

// заполняем всё пространство ячейки

constr.fill= constr.BOTH;

// пропорция по горизонтали, которую будет заниматькомпонент

constr.weightx = 1.0;

// пропорция по вертикали, которую будет занимать компонент

constr.weighty = 1.0;

// позиция компонента внутри ячейки

constr.anchor=constr.CENTER;

displayField = **new** JTextField();

// устанавливаем ограничения для поля ввода

gb.setConstraints(displayField,constr);

// добавляем поле ввода в окно

windowContent.add(displayField);

**Можно ли создавать окна, не используя схемы?**

Конечно, можно! Ничто не мешает явно задавать координаты каждого компонентa при добавлении на окно. Для этого класс должен чётко указать, что он не использует схемы размещения.

В Java есть специальное слово null, которое означает “значение не задано”. Мы будем использовать это ключевое слово довольно часто в будущем, и в следующем примере оно означает, что никакая схема не используется:

windowContent.setLayout(null);

Но, если отказаться от схем, то необходимо назначить координаты левого верхнего угла, ширину и высоту каждого оконного компонента.

В следующем примере показано, как можно установить ширину кнопки в 40 пикселей, высоту в 20, и разместить её на 100 пикселей вправо и 200 пикселей вниз от верхнего левого угла окна:

JButton myButton = **new** Button("New Game");

myButton.setBounds(100,200,40,20);

***Компоненты окна***

Я не буду описывать все компоненты Swing но вы можете найти ссылку на онлайн учебник по Swing,

Наш калькулятор использует только JButton, JLabel и JTextField.

В Swing есть много различных компонентов, чтобы сделать ваши окна симпатичными.

В этой главе мы создавали Swing компоненты, просто вводя код, без использования специальных инструментов. Но есть специальные утилиты, которые позволяют выбрать компонент на панели инструментов и перетащить его в создаваемое окно. Эти приложения автоматически генерируют соответствующий Java код для компонентов Swing. Один из таких графических дизайнеров, который позволяет легко создавать Swing приложения, называется Matisse. Другая – Gigloo GUI Builder.

В следующей главе будет рассказано, как окно может реагировать на действия пользователя.

***Практические упражнения***

1.Модифицируйте класс Calculator.java добавив в него кнопки +, -­‐, /, и \*. Поместите эти кнопки на панель p2, и положите эту панель на Восточную область основной панели.

2. Прочитайте про класс JFormattedTextField в интернете и измените исходный код калькулятора так, чтобы этот класс использовался вместо JTextField.

Целью является создание поля ввода с выравниванием по правому краю, как в настоящих калькуляторах.

***Практические упражнения для умников и умниц***

Модифицируйте класс Calculator.java так, чтобы все кнопки с цифрами хранились в массиве с десятью элементами, который должен быть объявлен вот так:

Buttons[] numButtons= **new** Buttons[10];

Замените 10 строк кода, которые начинаются с button0=**new** JButton("0"); циклом, который создаёт кнопки и добавляет их в массив.