**Лабораторная работа №6**

## **РАЗРАБОТКА ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ МНОГОПОТОЧНОСТИ**

**Цель:** разработка многопоточного приложения.

### Потоки

Когда запускается любое приложение, то начинает выполняться поток, называемый главным потоком (main). От него порождаются дочерние потоки. Главный поток, как правило, является последним потоком, завершающим выполнение программы.

Несмотря на то, что главный поток создаётся автоматически, им можно управлять через объект класса Thread. Для этого нужно вызвать метод currentThread(), после чего можно управлять потоком.

Класс Thread содержит несколько методов для управления потоками.

* getName() - получить имя потока
* getPriority() - получить приоритет потока
* isAlive() - определить, выполняется ли поток
* join() - ожидать завершение потока
* run() - запуск потока
* sleep() - приостановить поток на заданное время
* start() - запустить поток вызовом метода start()

### Многозадачность

Многозадачность, основанная на потоках как минимальной единице диспетчеризации называется многопоточностью, т.е. многопоточность – это реализация процесса на основе параллельно выполняющихся потоков.

В отличие от многозадачности, где задачи используют собственное адресное пространство, потоки используют одно и то же адресное пространство и разделяют процессорное время. В многопоточном режиме блокировка одного потока не приводит к остановке всей программы, например, в ожидании занятого ресурса, а вызывает передачу управления потоку, который может выполняться. Java обеспечивает встроенную поддержку для многопоточного программирования. В Java назначается приоритет, определяющий порядок обработки одного потока относительно другого, т.е. порядок переключения процессора от выполняющегося потока к следующему (переключение контекста). Абсолютное значение приоритета потока само по себе не имеет значения.

### Правила переключения:

* добровольный отказ от управления (кооперация) – явный переход в режим ожидания или блокированием на ожидании ввода/вывода и передача управления самому высокоприоритетному потоку, готовому к выполнению;
* упреждающая многозадачность – поток может быть приостановлен более приоритетным потоком. В этом случае выполнение низкоприоритетного потока приостанавливается, независимо от того, какие действия он выполняет, более высокоприоритетным потоком

Распределение процессора для потоков с одинаковым приоритетом может быть реализовано, например, квантованием времени использования процессора, добровольной передачи управления, либо другими более сложными стратегиями.

Т.к. многопоточность предполагает асинхронное выполнение потоков, то в каких-то точках может понадобиться синхронизация потоков. Для этого используется «монитор», называемый так же семафором – механизм управления связью между процессами – это блок, который может использоваться только одним потоком, а следовательно, монитор можно использовать для разделения ресурса между несколькими потоками. В Java нет отдельного класса объектов «монитор». Объект потока располагает собственным неявным монитором, который вводится автоматически, как только поток управления начинает выполнять синхронизированный метод, т.е. никакой другой поток не может вызвать данный метод до его освобождения.

Взаимодействия потоков в Java реализуются через вызовы предопределенных методов, которыми обладают все объекты.

Один из потоков – **«главный**» начинает выполняться первым при запуске Java-программы. От него порождаются дочерние потоки. Главный поток является последним выполняющимся потоком. Программа завершается, когда главный поток останавливается.

### Создание нового потока

Поток может быть создан на основе базового класса Thread или реализации интерфейса Runnable. Класс Thread инкапсулирует ряд методов, которые помогают управлять потоками.

java.lang.Object

java.lang.Thread

Главный поток создается автоматически с именем main и приоритетом 5 по умолчанию. Им можно управлять через Thread объект. Ссылку на него можно получить через статический метод static currentThread ( ), т.е. выполнить, например:

Thread tp = Thread.CurrentThread ( ) – *возвращает ссылку на поток, в котором он вызывается. В дальнейшем эту объектную ссылку можно использовать для вызова методов в управлении потоками.*

*Например, установить новое имя потока при помощи (string).*

Thread tp = Thread.CurrentThread ( )

System.ont.println («новое имя потока:» + tp.setName («New name»);.

Новое имя потока: Thread [New name, 5,main], где NewName – новое имя потока; 5- приоритет, заданный по умолчанию.

main –имя группы потоков, которой принадлежит данный поток.

Группа потоков – структура данных, контролирующая состояние совокупности потоков в целом.

Самый простой способ создания – реализация интерфейса Runnable, который требует определения в классе реализации метода: public void run ( ); , устанавливающего в программе точку входа для запуска нового конкурирующего потока выполнения.

### Примеры создания потоков

* анонимный класс

// Создание потока

Thread t = new Thread(new Runnable() {

public void run() {

System.out.println("Hello");

}

});

// Запуск потока

t.start();

* класс, реализующий Runnable и метод run()

public class myTask  
implements Runnable {

public void run() {

// действия

}

}

// Создание потока

Thread t = new Thread(new myTask());

// Запуск потока

t.start();

* Класс, наследующийся от Thread и реализующий метод run()

public class myTask  
extends Thread {

public void run() {

// действия

}

}

// Создание потока

Thread t = new myTask();

// Запуск потока

t.start();

В данном случае также можно вызвать *start()* в конструкторе

### Синхронизация

Возможность синхронизации как бы встроена в каждый объект, создаваемый приложением Java. Для этого объекты снабжаются защелками, которые могут быть использованы для блокировки потоков, обращающихся к этим объектам.

Чтобы воспользоваться защелками, вы можете объявить соответствующий метод как synchronized, сделав его синхронизированным:

public synchronized void decrement()

{

}

При вызове синхронизированного метода соответствующий ему объект (в котором он определен) блокируется для использования другими синхронизированными методами. В результате предотвращается одновременная запись двумя методами значений в область памяти, принадлежащую данному объекту.

Использование синхронизированных методов - достаточно простой способ синхронизации потоков, обращающихся к общим критическим ресурсам, наподобие описанного выше банковского счета.

Заметим, что не обязательно синхронизовать весь метод - можно выполнить синхронизацию только критичного фрагмента кода.

synchronized(Account)

{

if(Account.check(3000000))

Account.decrement(3000000);

}

Здесь синхронизация выполняется для объекта Account.

### Блокировка потока

Синхронизированный поток, определенный как метод типа *synchronized*, может переходить в заблокированное состояние автоматически при попытке обращения к ресурсу, занятому другим синхронизированным методом, либо при выполнении некоторых операций ввода или вывода. Однако в ряде случаев полезно иметь более тонкие средства синхронизации, допускающие явное использование по запросу приложения.

#### **Блокировка на заданный период времени**

С помощью метода *sleep* можно заблокировать поток на заданный период времени:

try

{

Thread.sleep(500);

}

catch (InterruptedException ee)

{

}

В данном примере работа потока  *Thread* приостанавливается на 500 миллисекунд. Заметим, что во время ожидания приостановленный поток не отнимает ресурсы процессора.

Так как метод *sleep* может создавать исключение *InterruptedException*, необходимо предусмотреть его обработку. Для этого мы использовали операторы *try* и *catch*.

#### **Временная приостановка и возобновление работы**

Методы *suspend*и *resume* позволяют, соответственно, временно приостанавливать и возобновлять работу потока.

В следующем фрагменте кода поток *m\_Rectangles* приостанавливает свою работу, когда курсор мыши оказывается над окном аплета:

public boolean mouseEnter(Event evt,

int x, int y)

{

if (m\_Rectangles != null)

{

m\_Rectangles.suspend();

}

return true;

}

Работа потока возобновляется, когда курсор мыши покидает окно аплета:

public boolean mouseExit(Event evt,

int x, int y)

{

if (m\_Rectangles != null)

{

m\_Rectangles.resume();

}

return true;

}

#### **Ожидание извещения**

Если вам нужно организовать взаимодействие потоков таким образом, чтобы один поток управлял работой другого или других потоков, вы можете воспользоваться методами *wait*, *notify* и *notifyAll*, определенными в классе *Object*.

Метод *wait* может использоваться либо с параметром, либо без параметра. Этот метод переводит поток в состояние ожидания, в котором он будет находиться до тех пор, пока для потока не будет вызван извещающий метод *notify, notifyAll*, или пока не истечет период времени, указанный в параметре метода *wait.*

Как пользоваться методами *wait, notify* и *notifyAll*?

Метод, который будет переводиться в состояние ожидания, должен быть синхронизированным, то есть его следует описать как *synchronized*:

public synchronized void run()

{

while (true)

{

try

{

this.wait();

}

catch (InterruptedException e)

{

}

}

}

В этом примере внутри метода run определен цикл, вызывающий метод wait без параметров. Каждый раз при очередном проходе цикла метод run переводится в состояние ожидания до тех пор, пока другой поток не выполнит извещение с помощью метода notify.

Ниже мы привели пример потока, вызывающией метод notify:

public void run()

{

while (true)

{

try

{

Thread.sleep(30);

}

catch (InterruptedException e)

{

}

synchronized(STask)

{

STask.notify();

}

}

}

Этот поток реализован в рамках отдельного класса, конструктору которого передается ссылка на поток, вызывающую метод wait. Эта ссылка хранится в поле STask.

Обратите внимание, что хотя сам метод run не синхронизированный, вызов метода notify выполняется в синхронизированном режиме. В качестве объекта синхронизации выступает поток, для которого вызывается метод notify.

#### **Ожидание завершения потока**

С помощью метода join вы можете выполнять ожидание завершения работы потока, для которой этот метод вызван.

Существует три определения метода join:

public final void join();

public final void join(long millis);

public final void join(long millis,

int nanos);

Первый из них выполняет ожидание без ограничения во времени, для второго ожидание будет прервано принудительно через millis миллисекунд, а для третьего - через millis миллисекунд и nanos наносекунд. Учтите, что реально вы не сможете указывать время с точностью до наносекунд, так как дискретность системного таймера компьютера намного больше.

### Примеры разработки GUI приложений с использованием многопоточности

#### Пример 1.

Разработать многопоточное приложение, отображающее хаотичное движение двух бабочек. Бабочки со временем уменьшаются в размере до полного их исчезновения.

package mypackage;

import java.awt.\*;

import java.awt.image.BufferedImage;

import java.io.\*;

import java.util.Random;

import javax.imageio.ImageIO;

import javax.swing.\*;

public class DemoThread extends JFrame{

public DemoThread() {

setTitle("Demo app");

setSize(new Dimension(480, 440));

setLocationRelativeTo(null);

setVisible(false);

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JPanel content = new JPanel(new BorderLayout());

setContentPane(content);

content.add(new Img1(), BorderLayout.WEST);

content.add(new Img2(), BorderLayout.EAST);

}

private class Img1 extends JPanel{

public Img1() {

setPreferredSize(new Dimension(240, 220));

try {

buffImg1 = ImageIO.read(new File("D:\\Butterfly1.jpg"));

}

catch (IOException exc) {};

new Thread(new Runnable() {

@Override

public void run() {

while (buffImg1Width != 0) {

buffImg1Width--;

buffImg1Height--;

repaint();

try {

Thread.sleep(150);

}

catch (Exception exc) {};

}

}

}).start();

}

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D graphics2D = (Graphics2D)g;

randomBuffImg1 = new Random();

buffImg1X = randomBuffImg1.nextInt(getWidth() - buffImg1Width);

buffImg1Y = randomBuffImg1.nextInt(getHeight() - buffImg1Height);

graphics2D.drawImage(buffImg1, buffImg1X, buffImg1Y, buffImg1Width, buffImg1Height, this);

}

}

private class Img2 extends JPanel{

public Img2() {

setPreferredSize(new Dimension(240, 220));

try {

buffImg2 = ImageIO.read(new File("D:\\Butterfly2.jpg"));

}

catch (IOException exc) {};

new Thread(new Runnable() {

@Override

public void run() {

while (buffImg2Width != 0) {

buffImg2Width -= 2;

buffImg2Height -= 2;

repaint();

try {

Thread.sleep(250);

}

catch (Exception exc) {};

}

}

}).start();

}

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D graphics2D = (Graphics2D)g;

randomBuffImg2 = new Random();

buffImg2X = randomBuffImg2.nextInt(getWidth() - buffImg2Width);

buffImg2Y = randomBuffImg2.nextInt(getHeight() - buffImg2Height);

graphics2D.drawImage(buffImg2, buffImg2X, buffImg2Y, buffImg2Width, buffImg2Height, this);

}

}

private BufferedImage buffImg1, buffImg2;

private int buffImg1Width = 60, buffImg1Height = 60, buffImg2Width = 60, buffImg2Height = 60;

private int buffImg1X, buffImg1Y, buffImg2X, buffImg2Y;

private Random randomBuffImg1, randomBuffImg2;

public static void main(String[] args) {

new DemoThread().setVisible(true);

}

}

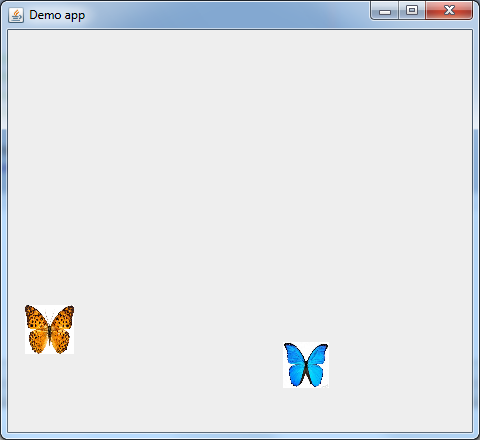


Рис. 1 – Результат выполнения примера 1

#### Пример 2. Туча в верхней части экрана, а в нижней части экрана – сугроб. При нажатии на кнопку «Старт» начинает идти снег. При этом размер тучи уменьшается, a сугроб растет.

### import java.awt.\*;

### import java.awt.event.ActionEvent;

### import java.awt.event.ActionListener;

### import java.awt.image.BufferedImage;

### import java.io.\*;

### import java.util.concurrent.ThreadLocalRandom;

### import javax.imageio.ImageIO;

### import javax.swing.\*;

### public class DemoThread extends JFrame{

### private BufferedImage buffImg1, buffImg2;

### private int cloudWidth = 1000, cloudHeight = 150, snowBankWidth = 1000, snowBankHeight = 200, snowBankY = 400, snowflakeX = 200, snowflakeY = 50,

### snowflakeX1 = 500, snowflakeY1 = 505, snowflakeX2 = 550, snowflakeY2 = 505,snowflakeX3 = 640, snowflakeY3 = 415, snowflakeX4 = 752, snowflakeY4 = 420,

### snowflakeX5 = 670, snowflakeY5 = 385, snowflakeX6 = 480, snowflakeY6 = 5;

### private static Image background;

### private static Image snowflake;

### private static Image snowbank;

### private static Image cloud;

### JButton bt;

### public DemoThread() {

### setTitle("Demo app");

### setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);

### setSize(1000, 600);

### setLocationRelativeTo(null);

### setResizable(false);

### // Background() - панель, для которой переопределен paintComponent с фоном

### setContentPane(new Background()); // панель устанавливается как contentPane в наследнике JFrame

### Container content = getContentPane(); //теперь все элементы интерфейса будут на этой панели.

### bt = new JButton("Старт");

### bt.setPreferredSize(new Dimension(1000,50));

### bt.setBackground(Color.white);

### bt.setForeground(Color.BLACK);

### bt.addActionListener(new ActionListener() {

### @Override

### public void actionPerformed(ActionEvent e) {

### bt.setVisible(false);

### Thread snowMove = new Thread(new SnowThread());

### snowMove.start();

### Thread snowMove1 = new Thread(new SnowFlakeThread());

### snowMove1.start();

### }

### });

### content.add(bt);

### content.add(new CloudSnowbank());

### }

### private static class Background extends JPanel{ // отрисовка нового фона

### @Override

### protected void paintComponent(Graphics g){

### super.paintComponent(g);

### try {

### background = ImageIO.read(new File("background.png"));

### } catch (IOException e) {

### e.printStackTrace();

### }

### g.drawImage(background,0,0,null);

### }

### }

### private class CloudSnowbank extends JPanel{

### public CloudSnowbank() {

### setOpaque(false);

### setPreferredSize(new Dimension(1000, 600));

### try {

### cloud = ImageIO.read(new File("cloud.png"));

### snowbank= ImageIO.read(new File("snowball.png"));

### snowflake = ImageIO.read(new File("snowflake1.png"));

### }

### catch (IOException exc) {};

### }

### @Override

### protected void paintComponent(Graphics g) {

### super.paintComponent(g);

### Graphics2D graphics2D = (Graphics2D)g;

### graphics2D.drawImage(cloud, 0, 0, cloudWidth, cloudHeight, this);

### graphics2D.drawImage(snowbank, 0,snowBankY, snowBankWidth, snowBankHeight, this);

### graphics2D.drawImage(snowflake,snowflakeX ,snowflakeY, 25, 25, this);

### graphics2D.drawImage(snowflake, snowflakeX1,snowflakeY1, 55, 55, this);

### graphics2D.drawImage(snowflake, snowflakeX2,snowflakeY2, 35, 35, this);

### graphics2D.drawImage(snowflake, snowflakeX3,snowflakeY3, 45, 45, this);

### graphics2D.drawImage(snowflake, snowflakeX4,snowflakeY4, 35, 35, this);

### graphics2D.drawImage(snowflake, snowflakeX5,snowflakeY5, 45, 45, this);

### graphics2D.drawImage(snowflake, snowflakeX6,snowflakeY6, 25, 25, this);

### }

### }

### public class SnowThread implements Runnable{

### @Override

### public void run() {

### while (cloudHeight > 0) {

### cloudHeight -= 2;

### snowBankHeight+=4;

### snowBankY-=4;

### repaint();

### try {

### Thread.sleep(130);

### }

### catch (Exception exc) {};

### }

### }

### }

### public class SnowFlakeThread implements Runnable{

### @Override

### public void run() {

### while (cloudHeight > 0) {

### snowflakeX = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX1 = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY1 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX2 = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY2 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX3 = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY3 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX4 = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY4 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX5 = ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY5 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### snowflakeX6= ThreadLocalRandom.current().nextInt(0, cloudWidth + 1);

### snowflakeY6 = ThreadLocalRandom.current().nextInt(cloudHeight, snowBankY + 1);

### try {

### Thread.sleep(150);

### }

### catch (Exception exc) {};

### }

### }

### }

### public static void main(String[] args) throws IOException {

### new DemoThread().setVisible(true);

### }

### }

### 

Рис. 2 – Результат выполнения примера 2

### Задания для самостоятельного выполнения

Разработать GUI-приложение c 3-4-мя потоками, выводящее соответствующие изображения. Предусмотреть кнопки «Стоп» и «Старт», позволяющие запускать и останавливать движение в приложении.

1. Машина, проезжающая мимо идущего человека и растущего дерева. При нажатии на кнопку «Старт» машина начинает двигаться, человек идти.
2. Облако в верхней части экрана. При нажатии на кнопку «Старт» облако начинает двигаться, из него начинает идти дождь. Внизу появляется лужа. При этом размер облака постепенно уменьшается. Лужа постепенно растет.
3. Туча в верхней части экрана, а в нижней части экрана – емкость для воды. При нажатии на кнопку «Старт» из тучи начинает идти дождь, и размер тучи уменьшается. При нажатии на кнопку «Стоп» дождь прекращает идти, а в емкости появляется вода.
4. Туча в верхней части экрана, а в нижней части экрана – сугроб. При нажатии на кнопку «Старт» начинает идти снег. При этом размер тучи уменьшается, a сугроб растет.
5. Летящие самолеты с разной скоростью. При нажатии на кнопку «Старт» самолеты начинают лететь. При нажатии на кнопку «Стоп» самолеты исчезают, и отображается дальность полета каждого самолета.
6. Пушка. При нажатии на кнопку «Старт» в правой части экрана появляется и исчезает случайным образом мишень. Нажатием кнопки «Стрелять» производится выстрел из пушки. Момент попадания фиксируется в виде взрыва.
7. Движущегося слева направо парусника с постоянной скоростью. Движение парусника начинается нажатием на кнопку «Старт». Скорость всякий раз задается генератором случайных чисел. В нижней части экрана расположена пушка. При нажатии на кнопку происходит выстрел торпедой с постоянной скоростью. При попадании торпеды в пушку смоделировать взрыв парусника и его исчезновение. При промахе парусник достигает правой границы экрана и начинает движение сначала с новой постоянной скоростью.
8. Циферблат механических часов с секундной, минутной и часовой стрелками. При нажатии на кнопку «Старт» часы начинают идти. Организовать срабатывание будильника в заданное время.
9. Песочные часы. При нажатии на кнопку «Старт» моделируется процесс падения песчинок, уменьшение уровня песка в верхней части колбы и увеличение в нижней части колбы.
10. Треугольник. При нажатии клавиши «Курсор вправо» треугольник вращается по часовой стрелке, клавиши «Курсор влево» − против часовой стрелки.
11. Бабочка и сачок. При нажатии на кнопку «Старт» бабочка начинает хаотичное движение. При нажатии на кнопку «Ловить» сачок опускается и поднимается. При нажатии на кнопку «Стоп» отображается результат – количество раз попадания бабочки в сачок и промахов.
12. Три вложенные друг в друга окружности, представляющие собой беговые дорожки. На линию старта выходят три спортсмена (произвольные фигуры). При нажатии на кнопку «Старт» участники стартуют с одинаковой угловой скоростью. После старта угловые скорости участников забега изменяются по случайному закону. На финише указать место, занятое каждым участником забега.
13. Изображение домика с окном и трубой. При нажатии на кнопку «Старт» в окне загорается свет, а из трубы начинает идти дым.
14. Два автомобиля. После нажатия на кнопку «Старт» они начинают двигаться, при этом скорость их движения изменяется случайным образом. При достижении финишной черты автомобили останавливаются, и выводится результат (какой автомобиль первым добрался до финиша).
15. Летящие вверх воздушные шарики. Шариков должно быть не менее 5. При нажатии на кнопку «Старт» шарики начинают подниматься вверх, каждый со своей скоростью. Пользователь мышкой должен попасть по шарику. Смоделировать взрывы шариков. При нажатии на кнопку «Стоп» отображается результат – количество попаданий по шарикам и количество промахов.

### Контрольные вопросы

1. Что такое многопоточность?
2. Что такое поток?
3. Что такое класс Thread, какие методы он содержит?
4. Что такое интерфейс Runnablе, для чего он используется?
5. Что такое метод или блок кода с ключевым словом synchronized?
6. Что такое обоюдная блокировка?
7. Какие существуют правила переключения?
8. Что такое многозадачность?
9. Как можно приостанавливать выполнение потока?
10. Какие существуют основные методы для работы с потоками?

### Список литературы:

1. [www.studfiles.ru/preview/2854789/](http://text.ru/rd/aHR0cDovL3d3dy5zdHVkZmlsZXMucnUvcHJldmlldy8yODU0Nzg5Lw%3D%3D)
2. [ru.wiki2.org/wiki/Java-апплет](http://text.ru/rd/aHR0cHM6Ly9ydS53aWtpMi5vcmcvd2lraS9KYXZhLdCw0L%2FQv9C70LXRgg%3D%3D)
3. [habrahabr.ru/post/164487/](http://text.ru/rd/aHR0cDovL2hhYnJhaGFici5ydS9wb3N0LzE2NDQ4Ny8%3D)
4. [wm-help.net/books-online/book/12462/12462-72.html](http://text.ru/rd/aHR0cDovL3dtLWhlbHAubmV0L2Jvb2tzLW9ubGluZS9ib29rLzEyNDYyLzEyNDYyLTcyLmh0bWw%3D)
5. [javalang.website/c14.html](http://text.ru/rd/aHR0cDovL2phdmFsYW5nLndlYnNpdGUvYzE0Lmh0bWw%3D)
6. [libeldoc.bsuir.by/bitstream/123456789/549/1/Языки\_...VA\_Лаб\_практ\_Ч\_2.pdf](http://text.ru/rd/aHR0cDovL2xpYmVsZG9jLmJzdWlyLmJ5L2JpdHN0cmVhbS8xMjM0NTY3ODkvNTQ5LzEv0K%2FQt9GL0LrQuF%2FQv9GA0L7Qs9GA0LDQvNC80LjRgF%2FQtNC70Y9f0YDQsNC30YDQsNCxX9GB0LXRgtC10LLRi9GFX9C%2F0YDQuNC70L7Qtl%2FQr9C30YvQul%2FQv9GA0L7Qs9GA0LDQvF9KQVZBX9Cb0LDQsV%2FQv9GA0LDQutGCX9CnXzIucGRm)
7. [javac.at.ua/news/mnogopotochnost\_java\_ochen\_ponjat...statja/2011-10-24-14](http://text.ru/rd/aHR0cDovL2phdmFjLmF0LnVhL25ld3MvbW5vZ29wb3RvY2hub3N0X2phdmFfb2NoZW5fcG9uamF0bmFqYV9ydXNza2FqYV9zdGF0amEvMjAxMS0xMC0yNC0xNA%3D%3D)
8. [www.studfiles.ru/preview/834380/](http://text.ru/rd/aHR0cDovL3d3dy5zdHVkZmlsZXMucnUvcHJldmlldy84MzQzODAv)
9. [studopedia.org/8-213606.html](http://text.ru/rd/aHR0cDovL3N0dWRvcGVkaWEub3JnLzgtMjEzNjA2Lmh0bWw%3D)
10. [ru.wikipedia.org/wiki/Java-апплет](http://text.ru/rd/aHR0cHM6Ly9ydS53aWtpcGVkaWEub3JnL3dpa2kvSmF2YS3QsNC%2F0L%2FQu9C10YI%3D)
11. [studopedia.net/14\_60251\_paket-AWT.html](http://text.ru/rd/aHR0cDovL3N0dWRvcGVkaWEubmV0LzE0XzYwMjUxX3Bha2V0LUFXVC5odG1s)
12. [studopedia.net/14\_60254\_mnogopotochnoe-programmirovanie.html](http://text.ru/rd/aHR0cDovL3N0dWRvcGVkaWEubmV0LzE0XzYwMjU0X21ub2dvcG90b2Nobm9lLXByb2dyYW1taXJvdmFuaWUuaHRtbA%3D%3D)
13. [lomos.wen.ru/Shildt\_Java2/](http://text.ru/rd/aHR0cDovL2xvbW9zLndlbi5ydS9TaGlsZHRfSmF2YTIvMTkuaHRtbA%3D%3D)