Java Concurrency
на практике

Chapter-4:
Composing Objects.
Компоновка объектов.

Upcode Software
Engineer Team

-2023-

CONTENT

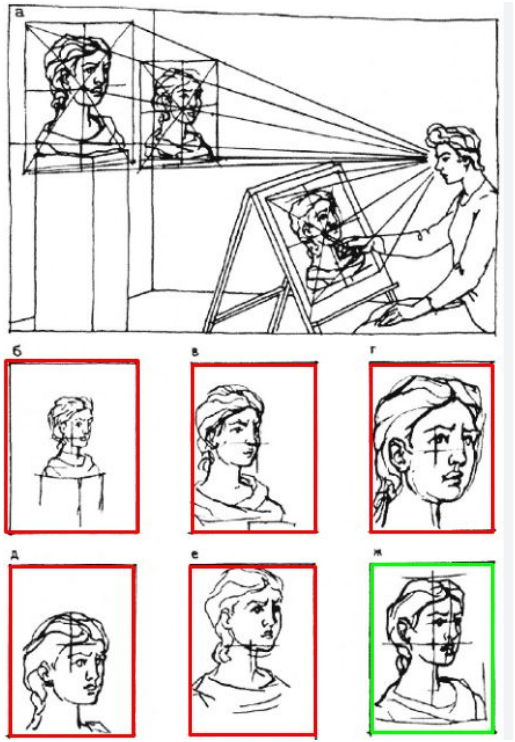
1. Что такое **Компоновка объектов** ?
2. Проектирование **потокобезопасного класса.**
3. Thread process.
4. **Документирование** политик синхронизации.

Что такое **Компоновка объектов ?** (1/1)

- Компоновка объектов в **Java** относится к способу организации объектов (например, **классов**) в вашем *коде*.

  Это позволяет создавать более сложные структуры данных и программы, объединяя **объекты** вместе для более удобного управления и использования.

- В **Java** компоновка объектов может быть реализована через различные механизмы, включая:



Что такое **Компоновка объектов ?** (1/2)

Классы:

Вы создаете классы для определения типов объектов. Классы могут содержать поля (переменные) и методы (функции), которые описывают состояние и поведение объектов.

```
// Определение класса
class МойКласс {
    int число;

    // Конструктор класса
    public МойКласс(int число) {
        this.число = число;
    }
}
```

Что такое **Компоновка объектов ?** (1/2)

Объекты:

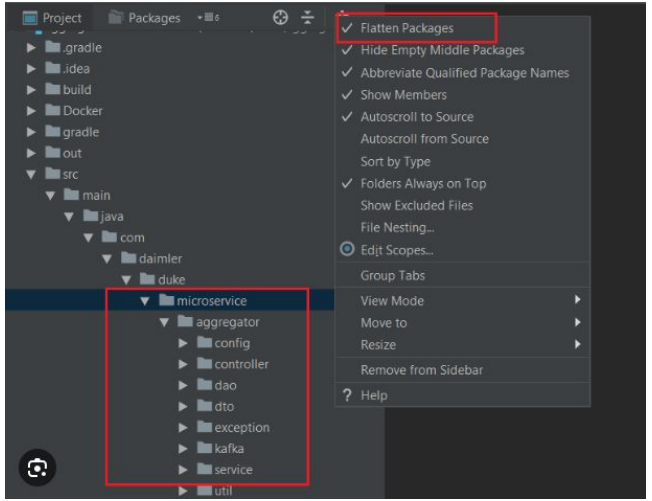
Вы создаете экземпляры классов, называемые объектами. Эти объекты хранят реальные данные и могут вызывать методы, определенные в классах.

```
public class Пример {
    public static void main(String[] args) {
        // Создание экземпляра класса МойКласс
        МойКласс экземпляр = new МойКласс(10);

        // Теперь у вас есть экземпляр класса
МойКласс с установленным значением числа
    }
}
```

Что такое **Компоновка объектов ?** (1/2)

Пакеты:

Вы можете организовать классы в пакеты, чтобы логически группировать связанные классы вместе. Пакеты помогают управлять исходным кодом и делают его более **структурированным**.



Что такое **Компоновка объектов ?** (1/2)

Наследование:

Вы можете создавать новые классы, основанные на существующих, с помощью наследования. Это позволяет вам повторно использовать код и расширять функциональность существующих классов.

```java
// Родительский класс
class Shape {
  protected String color;

  public Shape(String color) {
    this.color = color;
  }

  public void draw() {
    System.out.println("Drawing a shape");
  }
}
```

```java
// Дочерний класс, наследующий от Shape
class Circle extends Shape {
  private double radius;

  public Circle(String color, double radius) {
    super(color);
    this.radius = radius;
  }

  @Override
  public void draw() {
    System.out.println("Drawing a circle with radius " + radius);
  }

  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    Circle circle = new Circle("Red", 5.0);
    circle.draw();
    System.out.println("Area: " + circle.getArea());
  }
}
```

Что такое **Компоновка объектов ?** (1/2)

Интерфейсы:

Интерфейсы определяют контракты, которые классы должны реализовать. Это помогает в создании общего API для различных классов.

```
interface Printable {
  void print();
}

class Document implements Printable {
  @Override
  public void print() {
    System.out.println("Printing a document");
  }
}

class Photo implements Printable {
  @Override
  public void print() {
    System.out.println("Printing a photo");
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Printable printable1 = new Document();
    Printable printable2 = new Photo();

    printable1.print();
    printable2.print();
  }
}
```

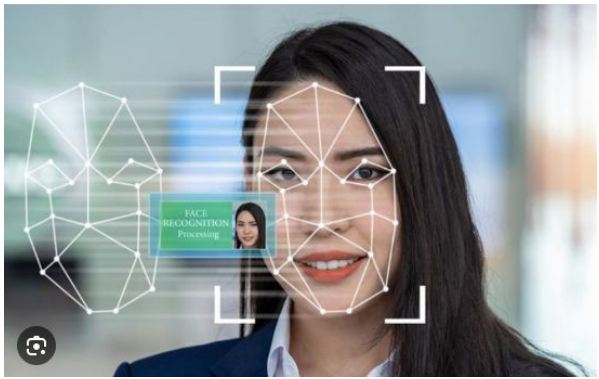
Потокобезопасный класс. (2/1)

- Потокобезопасный класс (или "**thread-safe class**") - это класс в программировании, который спроектирован и реализован таким образом, чтобы его **методы** могли безопасно использоваться из нескольких **параллельных потоков** выполнения (threads) в многозадачной программе. Потокобезопасность означает, что даже при одновременном доступе из разных потоков, состояние объекта класса остается согласованным, и операции выполняются корректно.

Потокобезопасный класс. (2/2)

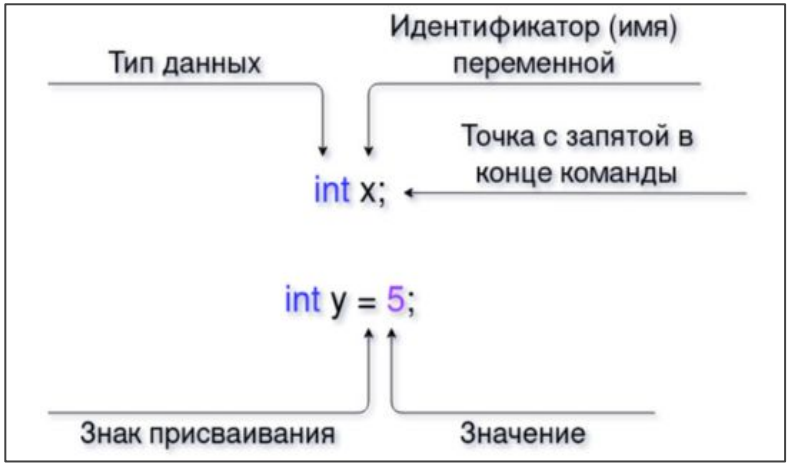
Проектирование потокобезопасного класса включает три этапа:

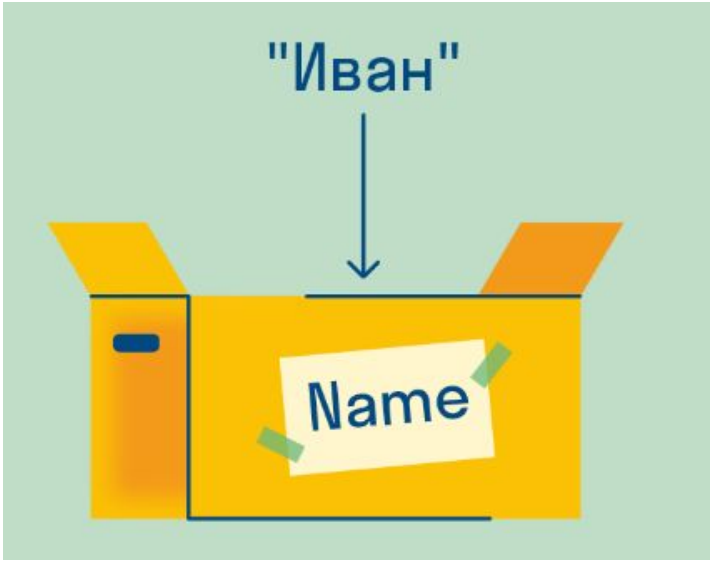
- **идентификация переменных, формирующих состояние объекта;**
- **идентификация инвариантов, ограничивающих переменные состояния;**
- **создание политики для управления конкурентным доступом к состоянию объекта.**



Идентификация переменных. (2/3)

- Идентификация переменных, формирующих состояние объекта: Это означает определение тех переменных, которые содержат данные, определяющие состояние объекта.





Идентификация инвариантов. (2/4)

- **Инвариант в программировании** - это условие или свойство, которое остается неизменным в течение выполнения программы. Он обеспечивает определенные гарантии о состоянии данных или условиях в программе на протяжении её выполнения. Инварианты используются для обеспечения правильности и надежности программного кода.

Политики для конкурентного доступа ? (3/n)

- **Создание политики** для управления конкурентным доступом к состоянию объекта - это процесс определения правил использования общих ресурсов объекта, чтобы избежать конфликтов при одновременном доступе нескольких потоков к этому объекту.

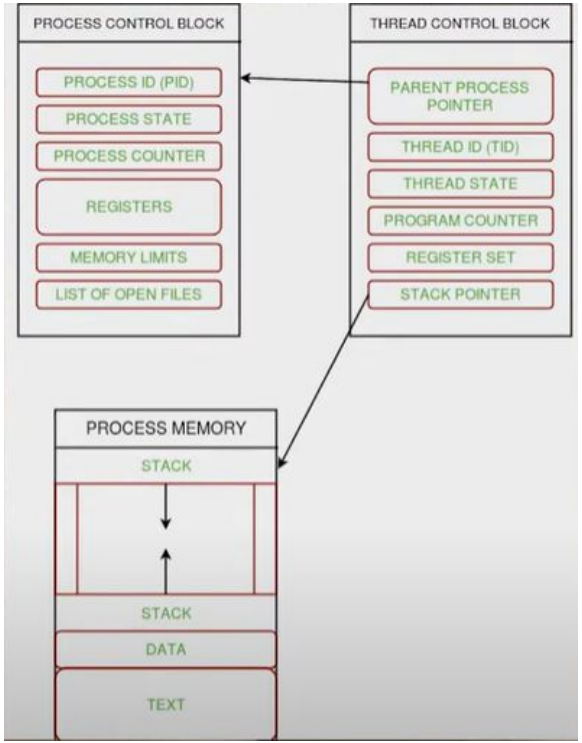
```
public class MyObject {
    private int value;
    private final Lock lock = new ReentrantLock();
    public void setValue(int newValue) {
        lock.lock();
        try {
            value = newValue;
        } finally {
            lock.unlock();
        }
    }
```

```
    public int getValue() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

Threaded process. (4/n)

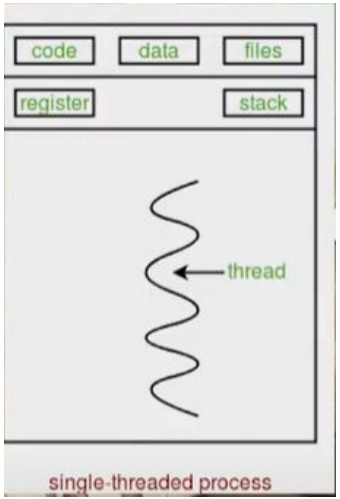
Thread process (процесс потока) в Java - это процесс, который выполняется в отдельном потоке. В отличие от многопоточного процесса, в котором операции выполняются параллельно в нескольких потоках, в процессе потока операции выполняются последовательно в одном потоке.

Threaded process. (4/n)



Threaded process.(5/n)

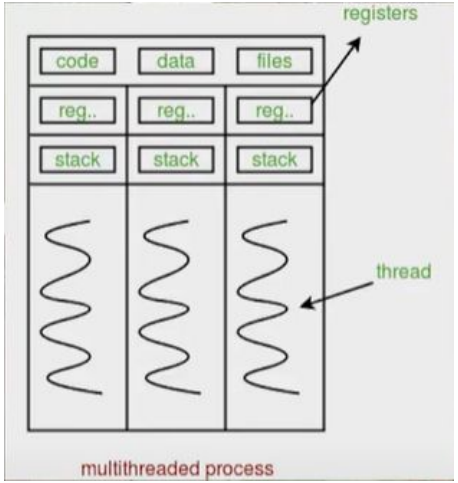
- **Single thread process** (однопоточный процесс) в Java - это процесс, который выполняется в единственном потоке. В таком процессе все операции выполняются последовательно, одна за другой, и не может быть одновременного выполнения нескольких операций.




```java
public class SingleThreadProcess {
    public static void main(String[] args) {
        System.out.println("Starting the process");
        System.out.println("Performing operation 1");
        System.out.println("Performing operation 2");
        System.out.println("Performing operation 3");
        System.out.println("Finishing the process");
    }
}
```

Threaded process.(5/n)

- **Multithread process** (многопоточный процесс) в Java - это процесс, который выполняется в нескольких потоках. В таком процессе операции могут выполняться одновременно в разных потоках, что позволяет увеличить производительность и эффективность работы приложения.



Threaded process.(5/n)

- **Single thread process** (однопоточный процесс) в Java - это процесс, который выполняется в единственном потоке. В таком процессе все операции выполняются последовательно, одна за другой, и не может быть одновременного выполнения нескольких операций.

```
public class MultiThreadProcess {
    public static void main(String[] args) {
        MyRunnable runnable1 = new MyRunnable("Runnable 1");
        MyRunnable runnable2 = new MyRunnable("Runnable 2");
        Thread thread1 = new Thread(runnable1);
        Thread thread2 = new Thread(runnable2);
        thread1.start();
        thread2.start();
    }
}
```

```
class MyRunnable implements Runnable {
    private String name;
    public MyRunnable(String name) {
        this.name = name;
    }
    public void run() {
        System.out.println("Starting " + name);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Finishing " + name);
    }
}
```

Data race. (6/n)

- **Data Race / Состояние гонки** — это ошибка проектирования многопоточной системы, при которой работа программы зависит от порядка выполнения частей кода, которые не синхронизированы должным образом.

Data race возникает при условии:

— два или более потока обращаются к одной и той же общей переменной;
— как минимум один из потоков пытается менять значение этой переменной;
— потоки не используют блокировки для обращения к этой переменной

Data race. (6/1)

```java
public class DataRaceExample {
    private static int counter = 0;

    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter++;
            }
            System.out.println("thread1: " + counter);
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter++;
            }
            System.out.println("thread2: " + counter);
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Counter value: " + counter);
    }
}
```

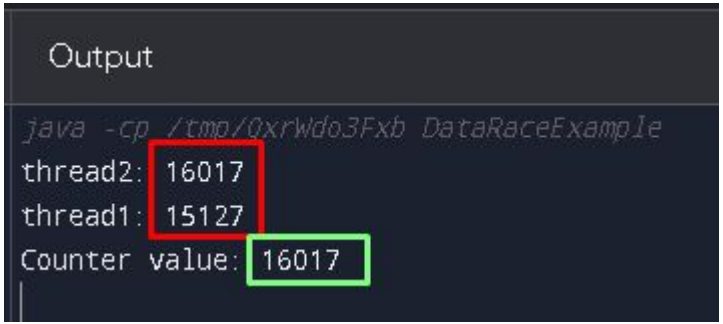
Data race. (6/2)


Output
java -cp /tmp/QxrWdo3Fxb DataRaceExample
thread1: 2000
thread2: 2000
Counter value: 2000

Data race. (6/3)

```java
public class DataRaceExample {
    private static int counter = 0;

    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter++;
            }
            System.out.println("thread1: " + counter);
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter++;
            }
            System.out.println("thread2: " + counter);
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Counter value: " + counter);
    }
}
```

Data race. (6/4)


Output
java -cp /tmp/QxrWdo3Fxb DataRaceExample
thread2: 16017
thread1: 15127
Counter value: 16017

Race condition.(7/n)

- **Состояние гонки в Java** — это ситуация, когда два или более потока одновременно манипулируют общим ресурсом, и при этом возникают непредсказуемые результаты.

Для решения проблемы race condition в Java используются механизмы синхронизации, такие как synchronized блоки и методы, volatile переменные и атомарные операции.

Deadlock. (8/n)

- Это ситуация, когда два или более потока блокируют друг друга и ожидают ресурсы, которые другой поток удерживает. В результате программы может зависнуть и не продолжать свою работу.

Deadlock. (8/2)

Deadlock. (8/2)

Blocking. (9/n)

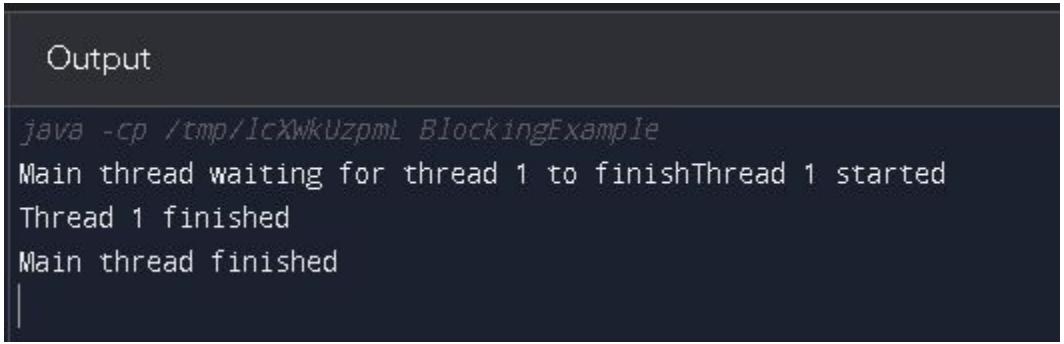
- **Блокировка (Blocking):**

  Иногда поток может заблокироваться и ждать некоторого события или ресурса. Если это происходит в главном потоке, то пользовательский интерфейс может "зависнуть" и не реагировать на действия пользователя.

Blocking. (9/1)

```java
public class BlockingExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            System.out.println("Thread 1 started");
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread 1 finished");
        });

        thread1.start();

        System.out.println("Main thread waiting for thread 1 to finish");
        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Main thread finished");
    }
}
```

Blocking. (9/2)


Output
java -cp /tmp/lcXWkUzpmL BlockingExample
Main thread waiting for thread 1 to finishThread 1 started
Thread 1 finished
Main thread finished

Context Switching. (9/n)

- **Переключение контекста (Context Switching):**

  Когда происходит переключение между потоками, требуется сохранение состояния одного потока и восстановление состояния другого. Это может замедлить выполнение программы и потреблять больше ресурсов процессора.

Context Switching. (9/1)

```java
public class ContextSwitchingExample {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        for (int i = 0; i < 100000; i++) {
            Math.sin(i);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Time taken without threads: " + (endTime - startTime) +
            "ms");

        startTime = System.currentTimeMillis();

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 50000; i++) {
                Math.sin(i);
            }
        });
```

```java
        Thread thread2 = new Thread(() -> {
            for (int i = 50000; i < 100000; i++) {
                Math.sin(i);
            }
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        endTime = System.currentTimeMillis();
        System.out.println("Time taken with threads: " + (endTime - startTime) + "ms"
            );
```

Context Switching. (9/2)

Resource Starvation (10/n)

- **Недостаток ресурсов (Resource Starvation):** Если слишком много потоков пытаются использовать ограниченные ресурсы, такие как память или сетевые соединения, то может возникнуть недостаток ресурсов, что может привести к снижению производительности или даже к сбоям программы.

Resource Starvation (10/1)

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ResourceStarvationExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        for (int i = 0; i < 10; i++) {
            int taskNumber = i;
            executorService.submit(() -> {
                try {
                    System.out.println("Task " + taskNumber + " started by thread " + Thread.currentThread().getName());
                    Thread.sleep(1000);
                    System.out.println("Task " + taskNumber + " completed by thread " + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        executorService.shutdown();
    }
}
```

Resource Starvation (10/2)

```
Task 0 started by thread pool-1-thread-1
Task 1 started by thread pool-1-thread-2
Task 0 completed by thread pool-1-thread-1
Task 2 started by thread pool-1-thread-1
Task 1 completed by thread pool-1-thread-2
Task 3 started by thread pool-1-thread-2
Task 2 completed by thread pool-1-thread-1
Task 4 started by thread pool-1-thread-1
Task 3 completed by thread pool-1-thread-2
Task 5 started by thread pool-1-thread-2
Task 4 completed by thread pool-1-thread-1
Task 6 started by thread pool-1-thread-1
Task 5 completed by thread pool-1-thread-2
Task 7 started by thread pool-1-thread-2
Task 6 completed by thread pool-1-thread-1
Task 8 started by thread pool-1-thread-1
Task 7 completed by thread pool-1-thread-2
Task 9 started by thread pool-1-thread-2
```

Future() (11/n)

```java
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();

        Future<Integer> future = executorService.submit(() -> {
            // Выполняем длительную операцию
            Thread.sleep(2000);
            // Генерируем исключение
            throw new RuntimeException("Oops! Something went wrong.");
        });

        System.out.println("Задача отправлена в поток исполнения.");

        try {
            Integer result = future.get();
            System.out.println("Результат: " + result);
        } catch (InterruptedException | ExecutionException e) {
            Throwable cause = e.getCause();
            System.err.println("Произошло исключение: " + cause.getMessage());
        }

        executorService.shutdown();
    }
}
```

Output:

```
Задача отправлена в поток исполнения.

Произошло исключение: Oops! Something went wrong.
```

Future() (11/1)

Reference

1. Java Concurrency in Practice (page 96- 120) in russian

Thank you!

Presented by **Nodirkhuja Tursunov**

(ameriqano@gmail.com)