# ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA: ООП

## МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

# ПОТОК:

- **Java** поддерживает такую важную функциональность как **многопоточность**.
- При помощи **многопоточности** можно выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно.

# КЛАСС THREAD:

- В **Java** функциональность отдельного потока заключается в классе **Thread**.
- Чтобы создать новый поток, нам надо создать объект этого класса.
- Когда запускается программа, начинает работать главный поток этой программы **main**.
- От этого главного потока порождаются все остальные дочерние потоки.

# МЕТОДЫ КЛАССА THREAD:

- **getName()**: возвращает имя потока
- **setName(String name)**: устанавливает имя потока
- **getPriority()**: возвращает приоритет потока
- **setPriority(int proirity)**: устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
- **isAlive()**: возвращает true, если поток активен
- **isInterrupted()**: возвращает true, если поток был прерван
- **join()**: ожидает завершение потока
- **run()**: определяет точку входа в поток
- **sleep()**: приостанавливает поток на заданное количество миллисекунд
- **start()**: запускает поток, вызывая его метод `run()`

# СОЗДАНИЕ ПОТОКА:

- Для создания нового потока мы можем создать новый класс, либо наследуя его от класса **Thread**, либо реализуя в классе интерфейс **Runnable**.

# НАСЛЕДОВАНИЕ ОТ КЛАССА THREAD:

```
class JThread extends Thread {

    JThread(String name){
        super(name);
    }

    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s fiished... \n", Thread.currentThread().getName());
    }
}
```

```
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        new JThread("JThread").start();
        System.out.println("Main thread finished...");
    }
}
```

# НАСЛЕДОВАНИЕ ОТ КЛАССА THREAD:

- Здесь в методе **main** в конструктор **JThread** передается произвольное название потока, и затем вызывается метод **start()**. По сути этот метод как раз и вызывает переопределенный метод **run()** класса **JThread**.

```
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        new JThread("JThread").start();
        System.out.println("Main thread finished...");
    }
}
```

# РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА RUNNABLE:

- Этот интерфейс имеет один метод **run.**
- В методе **run()** собственно определяется весь тот код, который выполняется при запуске потока:

```
interface Runnable{

    void run();
}
```

# РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА RUNNABLE:

```
class MyThread implements Runnable {


    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        Thread myThread = new Thread(new MyThread(),"MyThread");
        myThread.start();
        System.out.println("Main thread finished...");
    }
}
```

# ЗАВЕРШЕНИЕ ПОТОКА:

- Переменная **isActive** указывает на активность потока.
- С помощью метода **disable()** мы можем сбросить состояние этой переменной.

# СИНХРОНИЗАЦИЯ ПОТОКОВ:

- При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми.
- Чтобы избежать подобной ситуации, надо **синхронизировать** потоки. Одним из способов синхронизации является использование ключевого слова **synchronized**. Этот оператор предваряет блок кода или метод, который подлежит синхронизации.

# СИНХРОНИЗАЦИЯ ПОТОКОВ:

- Каждый объект в **Java** имеет ассоциированный с ним **монитор**. **Монитор** представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора **synchronized**, монитор объекта блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта освобождается и становится доступным для других потоков.
- После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

# МЕТОДЫ:

- **wait():** освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод **notify()**
- **notify():** продолжает работу потока, у которого ранее был вызван метод **wait()**
- **notifyAll():** возобновляет работу всех потоков, у которых ранее был вызван метод **wait().**