Multithreading

Shristi Technology Labs

Contents

- Introduction
- Multitasking Vs Multithreading
- Main Thread
- Thread class
- Lifecycle of a thread
- Creating child threads
- Using Runnable
- Synchronization

Introduction

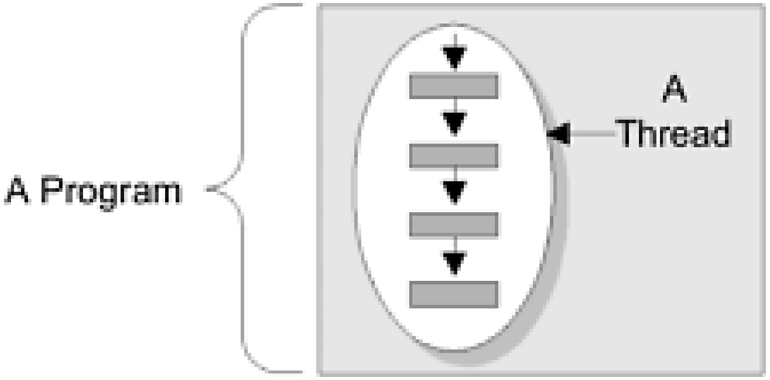
- Java is a multithreaded language.
- A Multithreaded program contains two or more parts that run concurrently.
- Each part of such program is called ***a thread***
- Each thread defines a separate path of execution.
- To utilize the idle time of CPU

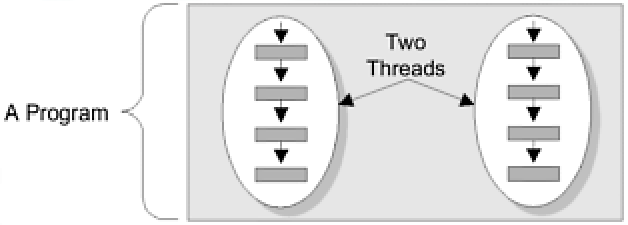
Multitasking Vs Multithreading

A ***process*** is a self-contained running program with its own address space.

A ***thread*** is a single sequential flow of control within a process.

A ***single process*** can have ***multiple concurrently executing threads***.





Main Thread

- When the Java Virtual Machine starts up, there is one thread that starts up for the main() function. This is a User thread.
- In a single-threaded application, this is the only thread.
- All the child threads emerge only from here.

Which thread occupies the CPU and how ?

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread has to run.
- Main thread has a priority of 5 which is the default priority.
- Priorities range between 1 and 10.
- 1 is the minimum and 10 is the maximum.

Thread.MIN_PRIORITY

Thread.NORM_PRIORITY

Thread.MAX_PRIORITY

Thread class

- Used to create child threads
- From lang package
- Has many methods
  - **static Thread currentThread()**
  - **void setName(String name)**
  - **String getName()**
  - **void setPriority(int number)**
  - **int getPriority()**
  - **void run()**
  - **void start()**
  - **static void sleep(long ms) throws InterruptedException**
  - **void join() throws InterruptedException**

static Thread currentThread()

- To get the details about the current thread occupying the CPU

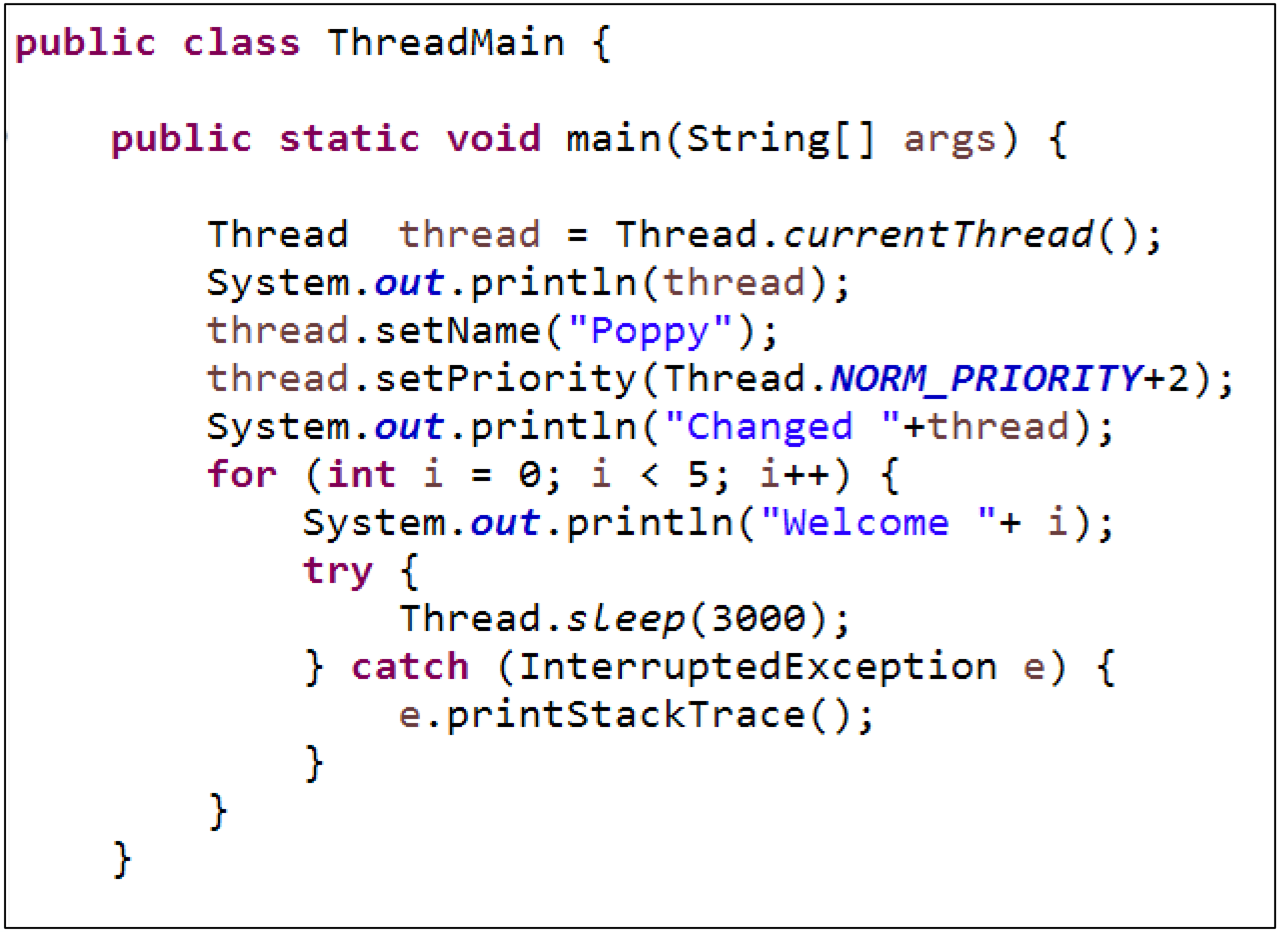
```
Thread  t = Thread.currentThread();
```

static void sleep(long ms)

- To make the thread sleep for a period of time so that other threads can occupy the cpu and do the work.

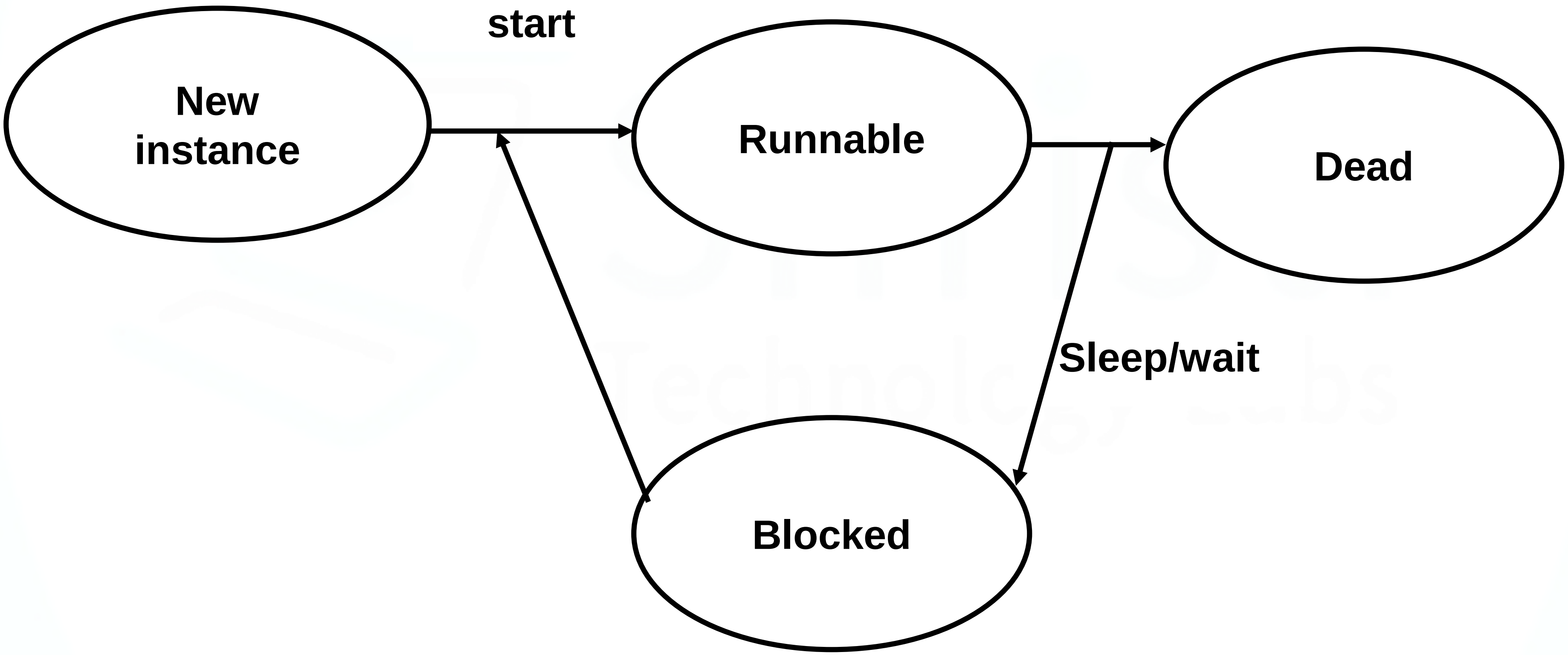
```
Thread.sleep(1000);
```

Example - Main Thread



```
public class ThreadMain {

    public static void main(String[] args) {

        Thread  thread = Thread.currentThread();
        System.out.println(thread);
        thread.setName("Poppy");
        thread.setPriority(Thread.NORM_PRIORITY+2);
        System.out.println("Changed "+thread);
        for (int i = 0; i < 5; i++) {
            System.out.println("Welcome "+ i);
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

Lifecycle of thread



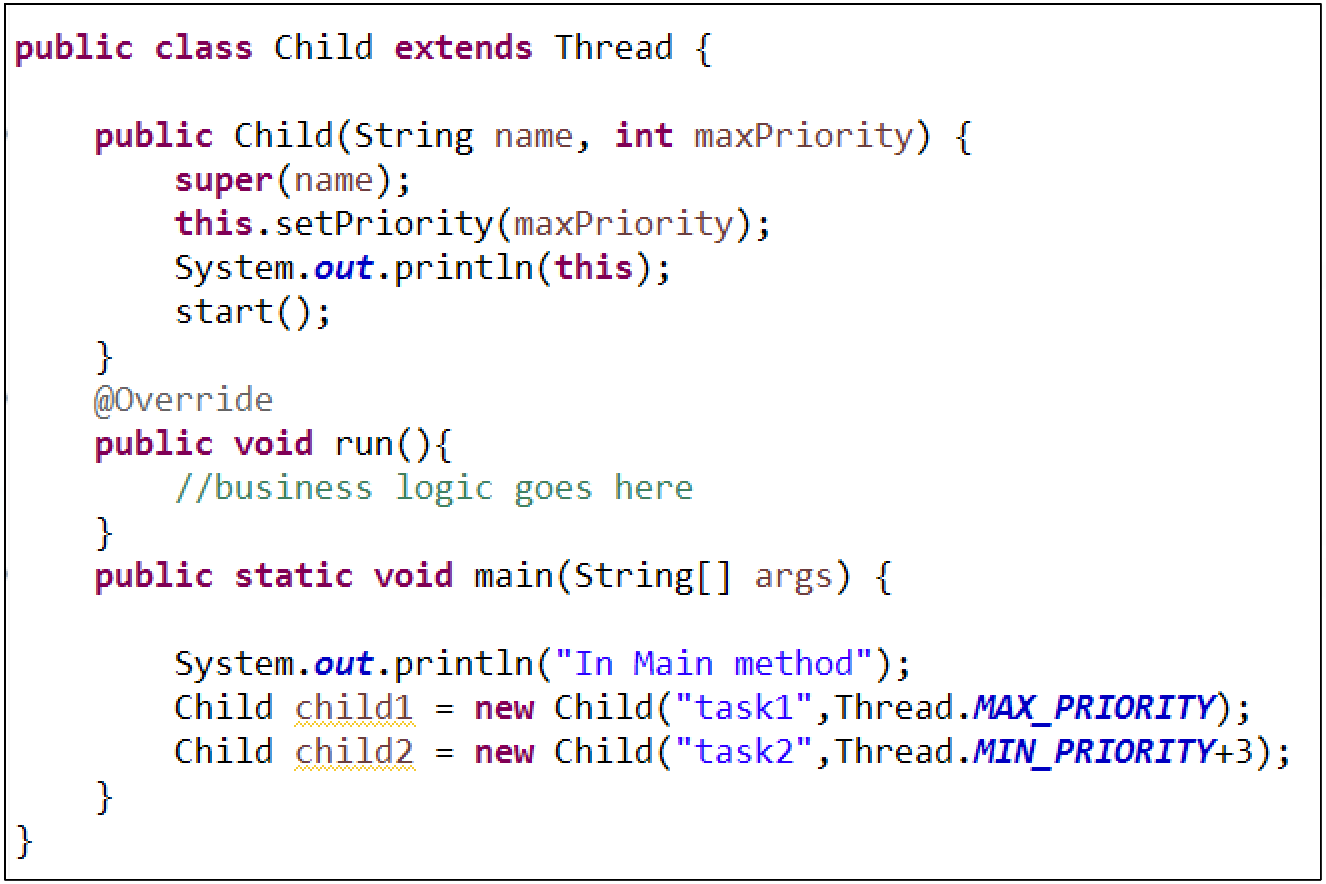
Thread States

- When a thread is first created, it is in the **NEW** State ***Thread t = new Thread();***
- When you invoke ***start*** method, it gets ready to get the CPU. ***t.start()***
- The thread changes to **RUNNABLE** state (eligible for execution) depending on its priority
- When ***sleep/wait*** method is called on a **RUNNABLE** thread, it may enter the NOT RUNNABLE state. ***Thread.sleep();***
- When a thread is **BLOCKED,** it is still alive, but it is not eligible for execution.
- A **BLOCKED** thread becomes ready to **run** again when the sleeping thread wakes up.
- This thread occupies the CPU depending on its **PRIORITY**
- When a thread terminates, it is said to be **DEAD**

Creating child threads

- Can be created by extending Thread class or by implementing Runnable.
- Override or implement the run method.

By Extending Thread class

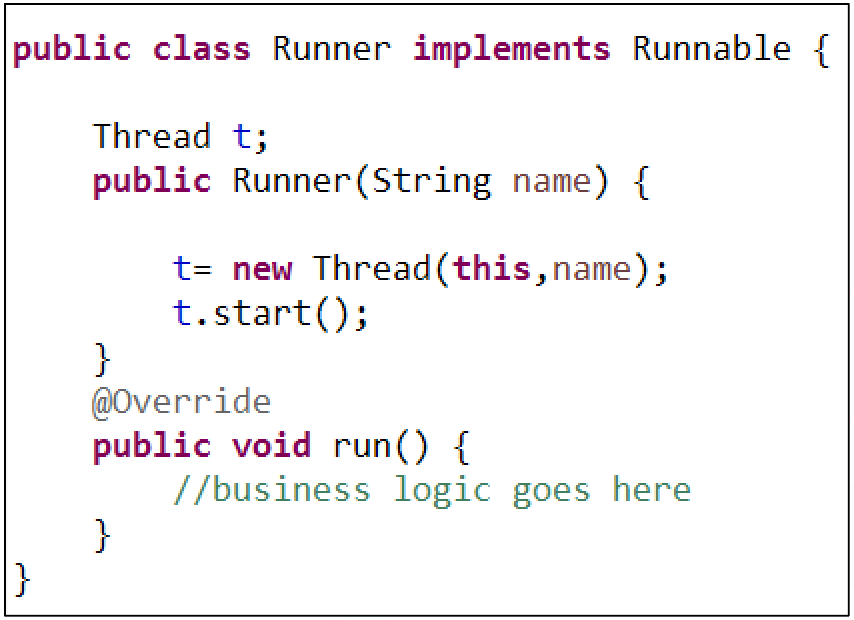


```
public class Child extends Thread {

    public Child(String name, int maxPriority) {
        super(name);
        this.setPriority(maxPriority);
        System.out.println(this);
        start();
    }
    @Override
    public void run(){
        //business logic goes here
    }
    public static void main(String[] args) {

        System.out.println("In Main method");
        Child child1 = new Child("task1",Thread.MAX_PRIORITY);
        Child child2 = new Child("task2",Thread.MIN_PRIORITY+3);
    }
}
```

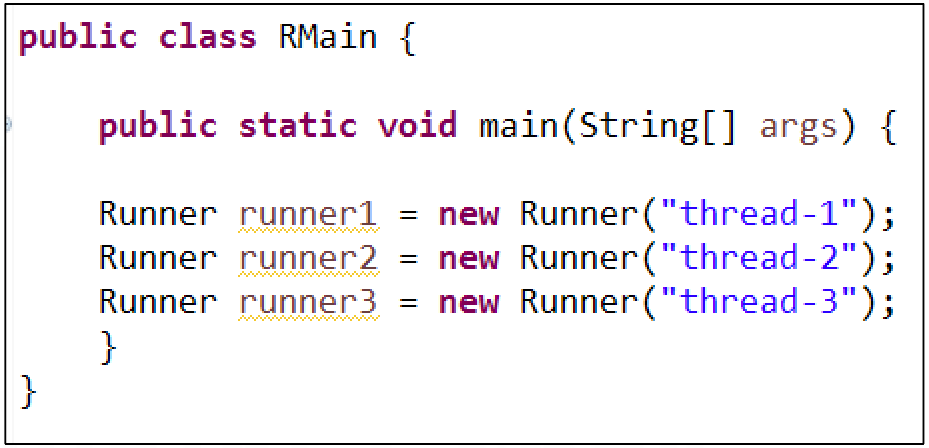
Using Runnable

- Runnable is a functional interface.
- Has one method run()
- The run method has the work to be done
- It is called by the Thread object
- The Runnable is the task to perform.
- The Thread is the worker doing this task.

By Implementing Runnable



```
public class Runner implements Runnable {

    Thread t;
    public Runner(String name) {

        t= new Thread(this,name);
        t.start();
    }
    @Override
    public void run() {
        //business logic goes here
    }
}
```

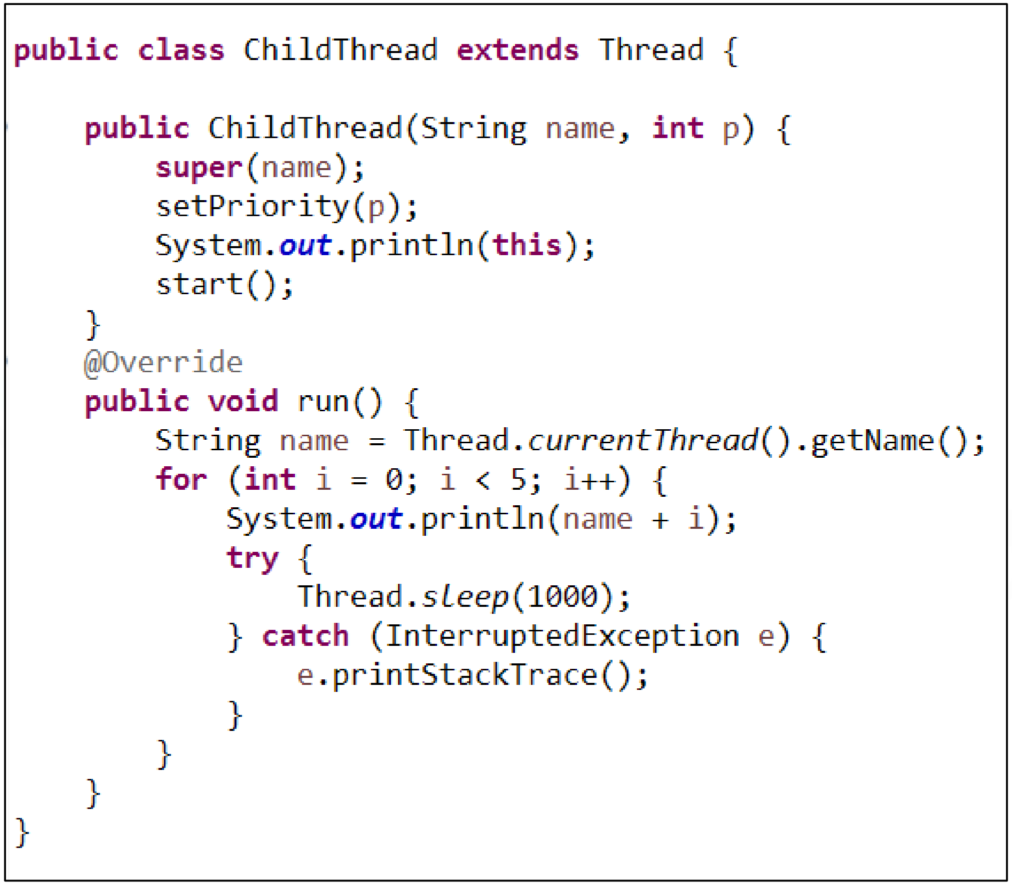


```
public class RMain {

    public static void main(String[] args) {

    Runner runner1 = new Runner("thread-1");
    Runner runner2 = new Runner("thread-2");
    Runner runner3 = new Runner("thread-3");
    }
}
```

Thread class or Runnable

- Extending the Thread class means that the subclass cannot extend any other class
- Implementing Runnable means the class can extend any other class

Creating multiple threads



```
public class ChildThread extends Thread {

    public ChildThread(String name, int p) {
        super(name);
        setPriority(p);
        System.out.println(this);
        start();
    }
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        for (int i = 0; i < 5; i++) {
            System.out.println(name + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Use of join method

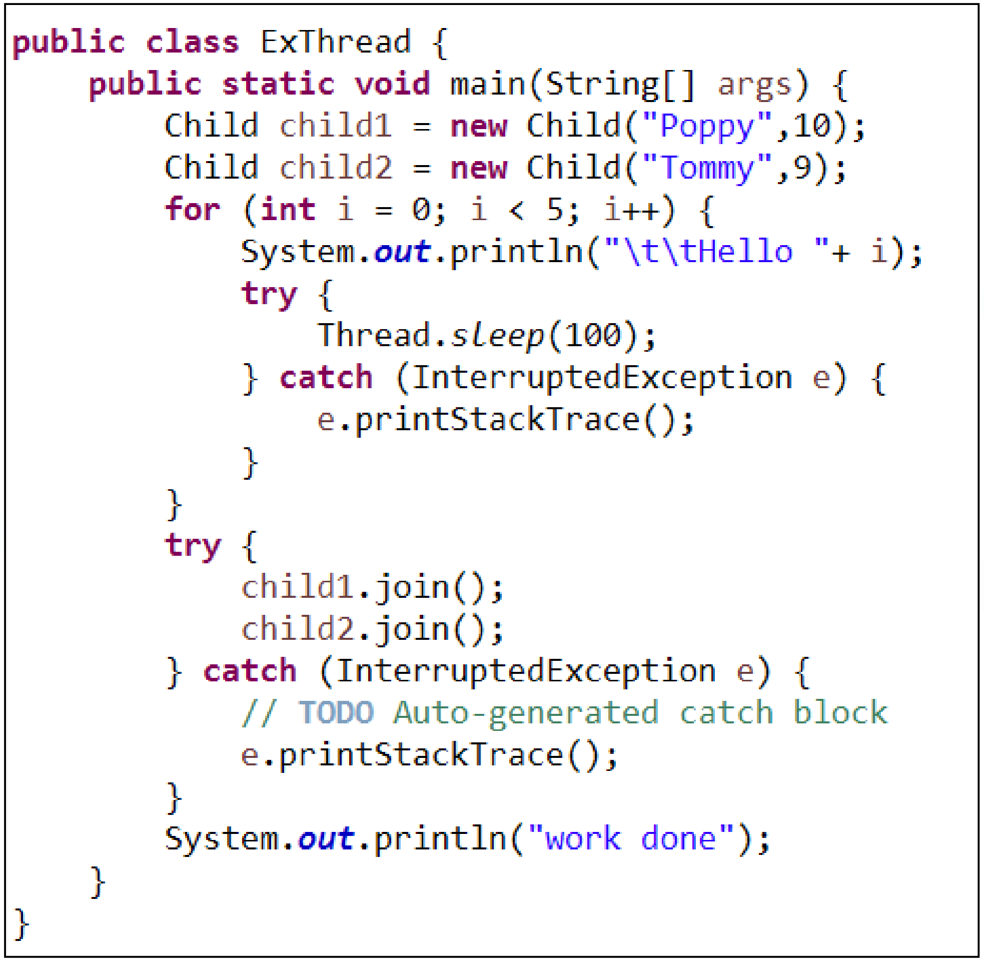
- This allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing,

  t.join()

  causes the current thread to pause execution and wait till t’s thread terminates.

- Join responds to an interrupt by exiting with an InterruptedException.

Example for join



```
public class ExThread {
    public static void main(String[] args) {
        Child child1 = new Child("Poppy",10);
        Child child2 = new Child("Tommy",9);
        for (int i = 0; i < 5; i++) {
            System.out.println("\t\tHello "+ i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        try {
            child1.join();
            child2.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("work done");
    }
}
```

Types of Threads

- User Threads
- Daemon Threads

User Thread

- Are threads that are created by the application.
- Are foreground threads
- JVM waits for these threads to finish their task.
- JVM will exit once all user threads finish their execution.

.

Daemon Thread

- Are background threads that are mostly created by the JVM.
- Are used to perform some background tasks like
  - garbage collection
  - poll remote systems for status changes
  - sending out email notifications
  - timer operations to perform scheduled maintenance
- They are less priority threads.
- JVM will not wait for daemon threads to finish their task.
- JVM will exit as soon as all user threads finish their execution.

Daemon Thread

- To make a thread as a daemon thread

```
Thread thread = new Thread();
 thread.setDaemon(true);
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- Achieved with the process of synchronization.
- Synchronization in Java can be achieved using the keyword – **synchronized**
- Synchronized can be associated only with methods and cannot be associated with member fields

Synchronization

- If a thread is in synchronized method, all the other threads trying to call this method or any other synchronized method using the same object have to wait.

- Synchronized statement has to be applied to **an Object** and cannot be applied to a primitive data type

- There are two types of synchronized usage:
  - **Syncronized** methods
  - **Syncronized** blocks

Synchronized Method

- A method can be declared synchronized.
- Behaves as if its body were contained in a synchronized statement.

Syntax

```
public synchronized returntype methodname(parameterlist){  }
```

Example

```
public synchronized double calcLoan(double s) {

       // method logic
}
```

Synchronized Block

- performs two actions:
  - After getting a reference to an object, it locks that object
  - The thread entering this block gets the lock of this object
  - The method is called on the object
  - After execution of the body has completed, either normally or abruptly, it unlocks that same lock.

Syntax

```
synchronized(object){  }
```

Gets the lock of this object

Example

```
synchronized (loan) {
      double d = loan.calcLoan(amount, n);
      System.out.println("interest is " + d);
}
```

Disadvantages of using thread class

- Creating a new thread causes some performance overhead.
- Too many threads can lead to reduced performance, as the CPU needs to switch between these threads.
- It is difficult to control the number of threads, so, may run into out of memory errors due to too many threads

Thank You