# CORE JAVA

# Multithreading

## Introduction

### Multitasking (mt)

Executing several tasks simultaneously

checking mobile

Taking notes

Listening classroom

Looking around environment

Process based mt

Simultaneously do the tasks

Where each tasks **independent program** (process)

e.g. listen music, type code, downloading movie

at OS level

#### thread based mt

Simultaneously do the tasks .where each task is an **independent part of a same program** each independent part is called **thread.**

### Advantage of multitasking

Increase performance by reducing response time

Multimedia graphics

Animation movies (many objects need to move at a time)

Video games

Request 1

Web server/

App server

Request 2

Request n

Develop web servers and application servers

\* Each thread handles one request, tomcat able to handle 60 request simultaneously

If Independent tasks there we need to use threads

Java provide rich-API to support multithread

Rich-api classes

[Thread, Runnable, ThreadGrop..]

Thread :

Separate follow of execution main thread



To do separate independent job

If only one thread (main thread): single thread application t4 t5 t6



## Defining threads

### By extending thread class

public class MainThread {

    public static void main(String[] args) {



        MyThread myThread = new MyThread();

        myThread.start();



        for (int i = 0; i < 10; i++) {



            System.out.println("main thread");

        }

    }

}

Main-thread is responsible to call main method

Create child-thread obj

Start child-thread

Do remaining jobs

class MyThread extends Thread{

    public void run(){ // override

        for(int i=0; i<20; i++){



            System.out.println("child thread");



        }



    }



}



#### Thread scheduler

It is part of jvm

Responsible to schedule thread:

While many threads are waiting, giving chance to execution

But we can’t expect thread execution order

We can’t provide exact output

#### t.start() V.s. t.run()

start() is in the parent class (Thread)

When this called, separate execution order runs



1, Register thread with thread-scheduler

2, Do other mandatory activities

3, after that invoke into run()

run() called directly then this method will called by main-thread

so there is only one thread available



#### overload run()

class MyThread extends Thread{

    public void run(){



        for(int i=0; i<20; i++){



            System.out.println("child thread");



        }

    }

    public void run(int j){

        for(int i=0; i<j; i++){

            System.out.println("child thread" + j);

        }

    }

}

Overloading of run() method is possible

But t.start() can invoke no-arg run() method

Like this main() overload is not called by jvm calls

Main(string arg) only

#### Overriding run()

1. Thread class has empty run() method
2. We need to override run method, with in that

we need to done our job

1. if you don’t override then parent class’s empty run() will be executed

Should override run()

#### Overriding start()

Now child class’s start() will be executed so new thread won’t be created

Should not override start()

class MyThread extends Thread{

    public void start(){

        super.start();

        System.out.println("hello start");

    }



    public void run(){

        for(int i=0; i<20; i++){

            System.out.println("child thread");



        } “hello start



    }



} run() remain code



public class MainThread {

    public static void main(String[] args) {

        MyThread myThread = new MyThread();

        myThread.start();

        for (int i = 0; i < 10; i++) {

            System.out.println("main thread");

        }

    }

}

#### Thread lifecycle



1,t.start() 2,thread-scheduler allocate processor

3,run() is complete

IllegalThreadStateException

public class MainThread {

    public static void main(String[] args) {

        MyThread myThread = new MyThread();

        myThread.start();

        myThread.start();

after start we try to start again

### By implementing Runnable interface

Runnable(Interface)

Implements

implements

Thread

Extends

MyRunnable

MyThread

public class MainThread {

    public static void main(String[] args) {

        MyRunnable myRunnable = new MyRunnable();

        Thread t = new Thread(myRunnable);

        t.start();

Thread is only has start() => to start new thread

Start() invoke into Thread’s run() method

But that’s empty implementation

So we pass myRunnable obj as arg

Now this runnable’s run method will be executed

public class MyRunnable implements Runnable {

    @Override

    public void run() {

        for (int i = 0; i < 20; i++) {

            System.out.println("my runnable");

        }

    }

}

### Runnable method is recommend to use

MyThread extends Thread

So we cant extend another class

MyRunnable extends MyClass implements Runnable

So we can get inheritance benefits

### Thread class constructors

Thread t1 = new Thread();

Thread t2 = new Thread(Runnable target);

Thread t3 = new Thread(String name);

Thread t4 = new Thread(Runnable target, String name);

Thread t5 = new Thread(ThreadGroup group, String name);

Thread t6 = new Thread(ThreadGroup group, Runnable target);

Thread t7 = new Thread(ThreadGroup group, Runnable target, String name);

Thread t8 = new Thread(ThreadGroup group, Runnable target, String name, long stacksize);

Thread t9 = new Thread(myThread);

t9.start() this will work because myThread implements runnable

## Getting and setting name of threads

Every thread has default name is generated by jvm

(main, Thread-0, Thread-1, Thread-2, ..)

or we can edit it

#### Get current executing thread obj

Thread.currentThread()

currentThread() is static method present inside Thread class

#### Getname

t.getName(); // t is thread instance

Thread.currentThread().getName(); // main thread’s name

#### Setname

t.setName("arul");

Thread.currentThread().setName("tharsi");

## Thread priorities

Every thread in java has some priority

It may be generate by jvm

We can provide priority (1 to 10)

Thread.MIN\_PRIORITY => 1

Thread.NORM\_PRIORITY => 5

Thread.MAX\_PRIORITY => 10

T1

6

T2 7  5 t3

4

T4

Thread scheduler will use priority while allocating processor

In this case T2 will get the first chance to get the processor

If 2 threads have same priority, we can’t tell the execution order

#### SetPriority getPriority

In Thread class

public final void setPriority(int newPriority) {}

public final int getPriority() {}

1 to 10, else illegalArgumentException

t.setPriority(8);

Thread.currentThread().setPriority(3);

#### Default priority

Default priority only for main thread is 5

But all remaining threads default priority will be inherited from parent to child



Set priority to main thread



Thread.currentThread().setPriority(3);



MyRunnable myRunnable = new MyRunnable();



Thread t = new Thread(myRunnable);



System.out.println(t.getPriority());

Now this child priority will be 3



Main 3, t1 5, t2 8

This won’t give us t2….., t1….., main…. Out put

Because some platform not provide support to thread priority

(Problem with OS not our code)

## The methods to prevent thread execution

### Yield()



1, to pause current executing thread

To give the chance for waiting thread of same priority

2, same thread can continue its execution

If there is no waiting thread/ have low priority

3, many threads with same priority

We can’t tell which thread will get chance

4, the thread which leaving the processor (yielded), when it will get the chance again??

We can’t expect exactly it depends on thread scheduler

Because now T4, T7, Tx have same priority 7,

T4(p=7)



Tx(p=7) T6(p=7)



Thread.yild() T7(p=7)

Tx(p=7)

In Thread class..

public static native void yield();



2,thread-scheduler

1,t.start() allocate processor

Thread.yield() 3,run() is

complete

public class MainThread {

    public static void main(String[] args) {

MyThread myThread = new MyThread();

myThread.start();

for (int i = 0; i < 10; i++) {

      System.out.println("main thread");

}

public void run(){

for(int i=0; i<10; i++){

            System.out.println("child thread");

            Thread.yield();



      }

}

* If no yield()

We can’t expert which(main/child) thread complete first

* With yield()

Child always calls yield(), so main thread gets chance more number of times, chance of main completing is high

* Thread which required more processing time, in the middle it needs to call yield()
* Some platforms won’t provide proper support for yield() method

### Join()

T1 T2

T2.join()

finish

Waiting

* If a thread wants to wait some other thread’s result then we should go for join method
* If T1 executes T2.join() then T1 enters into waiting state until T2 completes, after that T1 can continue its execution

Wedding Date Wedding Card Wedding Card

Fixing printing distribution

T1 T2 T3

T2.join()

T1.join()

T2 needs to wait until T1 completes

T3 needs to wait until T2 completes

public class MainThread {

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



MyThread myThread = new MyThread();

        myThread.start();



        myThread.join();



wait until, myThread completes



        for (int i = 0; i < 10; i++) { public void run(){



            System.out.println("main thread"); for (int i = 0 …



        }



}

#### 3 types of join method

    public final void join() throws InterruptedException {

    public final void join(long ms) throws InterruptedException {

    public final void join(long ms, int ns) throws InterruptedException {

So we need to handle (throws, tryCutch)

otherwise we will get compile time interruption exception

T1 T2

T2.join(100ms)

Waiting max 100ms

finish

#### Thread life cycle

T2.join()

If T2 completes T2.join(1200)

If time completes T2.join(1200, 100)

If waiting thread got interrupted

2,thread-scheduler

1,t.start() allocate processor

Thread.yield() 3,run() is

complete

#### child wait until main complete

public class MainThread {

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

        MyThread myThread = new MyThread();

        myThread.start();

        // Thread.currentThread()  == main thread obj

        MyThread.mainThread = Thread.currentThread();

        for (int i = 0; i < 10; i++) {

            System.out.println("main thread");

        }

    }

class MyThread extends Thread {

    // we need a reference of main thread

    static Thread mainThread;

    public void run() {

        try {

            mainThread.join();

        } catch (InterruptedException e1) {

            e1.printStackTrace();

        }

        for (int i = 0; i < 10; i++) {

            System.out.println("child thread");

            try {

                Thread.sleep(2000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

### Sleep()

## Synchronization

## Inter thread communication

## Deadlock

## Daemon Threads

## Multithreading enhancements

# Concurrent collections

Traditional collections

1. Most traditional collections are not thread safe

So there may be a data inconsistency problem

E.g. array list, hash set, linked list, tree set ..

Thread 4

1. Thread safe (only one thread allow to operate at a time)

So increase waiting time, create performance problem

E.g. Vector, hash table, synchronizedList, synchronizedSet, synchronizedMap