Multithreading and concurrent programming in Java

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

Introduction to multithreading

- Multithreading allows concurrent execution of multiple commands in the program
  - Often called concurrent programming
- Allows maximum CPU utilization
- Each execution is called a “thread”
  - Threads are lightweight subprocesses within an instruction stream process
  - Smallest unit of processing
  - Executes a single task
  - 1 thread != 1 CPU
    - A single CPU will usually switch between executing different threads
- Helps optimizing performance of the program
  - Concurrent vs. sequential execution
- Improves application responsiveness
- Exceptions in one thread do not affect other threads
  - Independent
- Real life analogy: mutitasking

Multithreading in Java by inheriting from *Thread* class

- In Java we can create threads by extending *java.lang.Thread* class
  - Remember to import *java.lang.**
- Execute the thread with *start()* method
- Execution instructions are in *run()* method

Example: simple multithreaded program, print thread ID

Extend *Thread* class

Returns ID of the current thread

```
class MyClass extends Thread{
    public void run(){
        try{
            System.out.println ("Current thread: " + Thread.currentThread().getId());
        } catch (Exception e) {
            System.out.println (e);
        }
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<10; i++)
        {
            MyClass myClassObj = new MyClass();
            myClassObj.start();
        }
    }
}
```

Instructions are in the *run()* method

Returns reference to the current thread

Start a thread with *start()* method

Output:

```
Current thread: 9
Current thread: 10
Current thread: 11
Current thread: 12
Current thread: 13
Current thread: 14
Current thread: 15
Current thread: 16
Current thread: 17
Current thread: 18
```

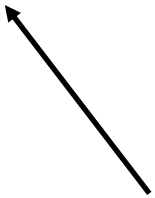
Thread ID is a positive long number generated when a thread is created; remains unchanged during the lifetime of a thread; once the tread is terminated its ID may be reused

Example: printing class attribute value in threads

```
class MyClass extends Thread{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    public void run(){
        try{
            System.out.println ("My attribute value: "+this.myAttribute);
        } catch (Exception e) {
            System.out.println (e);
        }
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<10; i++)
        {
            MyClass myClassObj = new MyClass("attribute"+i);
            myClassObj.start();
        }
    }
}
```

Output:

```
My attribute value: attribute0
My attribute value: attribute3
My attribute value: attribute4
My attribute value: attribute2
My attribute value: attribute1
My attribute value: attribute5
My attribute value: attribute6
My attribute value: attribute7
My attribute value: attribute8
My attribute value: attribute9
```



Print the value referenced by myAttribute field

Example: overriding *start()* thread method

```
import java.lang.*;

class MyClass extends Thread{
    public void run(){
        try{
            System.out.println ("Current thread: " + Thread.currentThread().getId());
        } catch (Exception e) {
            System.out.println (e);
        }
    }

    @Override
    public void start(){
        System.out.println ("Starting a thread");
        Thread t = new Thread (this);
        t.start();
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<10; i++)
        {
            MyClass myClassObj = new MyClass();
            myClassObj.start();
        }
    }
}
```

Output:

```
Starting a thread
Starting a thread
Starting a thread
Current thread: 10
Current thread: 12
Starting a thread
Current thread: 14
Starting a thread
Current thread: 16
Starting a thread
Current thread: 18
Starting a thread
Current thread: 20
Starting a thread
Current thread: 22
Starting a thread
Current thread: 24
Starting a thread
Current thread: 26
Current thread: 28
```

Runnable interface

- Remember that Java does not allow multiple inheritance?
- If our class extends *Thread* class it cannot inherit from other classes
- Sometimes it is necessary to create a class that has multithreading capabilities but also inherits from other classes
  - Runnable interface allows doing just that
- *java.lang.Runnable* interface
  - Class should implement *Runnable* interface
  - Implement *run()* method
  - Create new instance of *Thread* class and call *start()* method on it to start a new thread

Example: print thread id by implementing Runnable interface

Implements Runnable interface, is still free to extend a parent class

```
class MyClass implements Runnable{
    public void run(){
        try{
            System.out.println ("Current thread: " + Thread.currentThread().getId());
        } catch (Exception e) {
            System.out.println (e);
        }
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<10; i++)
        {
            Thread threadObj = new Thread(new MyClass());
            threadObj.start();
        }
    }
}
```

Create new instance of *Thread* class and pass in an instance of MyClass; *run()* method will automatically be called by JVM when we call *start()* method in this new *Thread* instance

Output:

```
Current thread: 10
Current thread: 12
Current thread: 11
Current thread: 9
Current thread: 13
Current thread: 14
Current thread: 15
Current thread: 16
Current thread: 17
Current thread: 18
```

Example: implement *start()* method inside the class that implements *Runnable* interface

```
class MyClass implements Runnable{
    private String objName;

    MyClass(String name){this.objName = name;}

    public void run(){
        try{
            System.out.println ("Class " + this.objName +
                ", current thread: " +
                Thread.currentThread().getId());
        } catch (Exception e) {
            System.out.println (e);
        }
    }

    public void start(){
        Thread threadObj = new Thread(this);
        threadObj.start();
    }
}

public class Test {
    public static void main(String[] args){
        List<MyClass> myClassLst = new ArrayList<MyClass>();
        myClassLst.add(new MyClass("instance1"));
        myClassLst.add(new MyClass("instance2"));
        myClassLst.add(new MyClass("instance3"));
        myClassLst.add(new MyClass("instance4"));
        myClassLst.add(new MyClass("instance5"));

        for(MyClass c : myClassLst){
            c.start();
        }
    }
}
```

implements Runnable interface

implement *run()* method to run thread tasks

implement *start()* method to call *run()* method

List of MyClass objects with different names

iterate through objects in the list and call *start()* method

Output:

```
Class instance1, current thread: 9
Class instance3, current thread: 11
Class instance2, current thread: 10
Class instance4, current thread: 12
Class instance5, current thread: 13
```

Useful thread methods

- *start()* – starts execution of a thread, calls *run()* method
- *sleep()* – tells the thread to sleep for the specified number of milliseconds
- *yield()* – tells the current thread to stop and allow other threads to execute
- *getName()* – returns the name of the thread
- setName() – sets/changes name of the thread
- *setPriority()* – sets/changes the thread priority
- *getPriority()* – returns the thread priority

Useful thread methods (cont'd)

- *interrupt()* – interrupts the current thread
- *isAlive()* – tests if the current thread is alive (between the point the tread was started and before it finishes the task)
- *join()* – waits for this thread to terminate until the thread from which it was called is dead
- isinterrupted() – tests if the thread has been interrupted
- *activeCount()* – returns the number of active threads
- *getState()* – returns the state of the thread

Example: using *getName()* thread method

Same example as before but calling *getName()* instead of *getId()*

```
class MyClass extends Thread{
    public void run(){
        try{
            System.out.println ("Current thread: " + Thread.currentThread().getName());
        } catch (Exception e) {
            System.out.println (e);
        }
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<10; i++)
        {
            MyClass myClassObj = new MyClass();
            myClassObj.start();
        }
    }
}
```

Output:

```
Current thread: Thread-0
Current thread: Thread-1
Current thread: Thread-2
Current thread: Thread-4
Current thread: Thread-5
Current thread: Thread-3
Current thread: Thread-6
Current thread: Thread-7
Current thread: Thread-8
Current thread: Thread-9
```

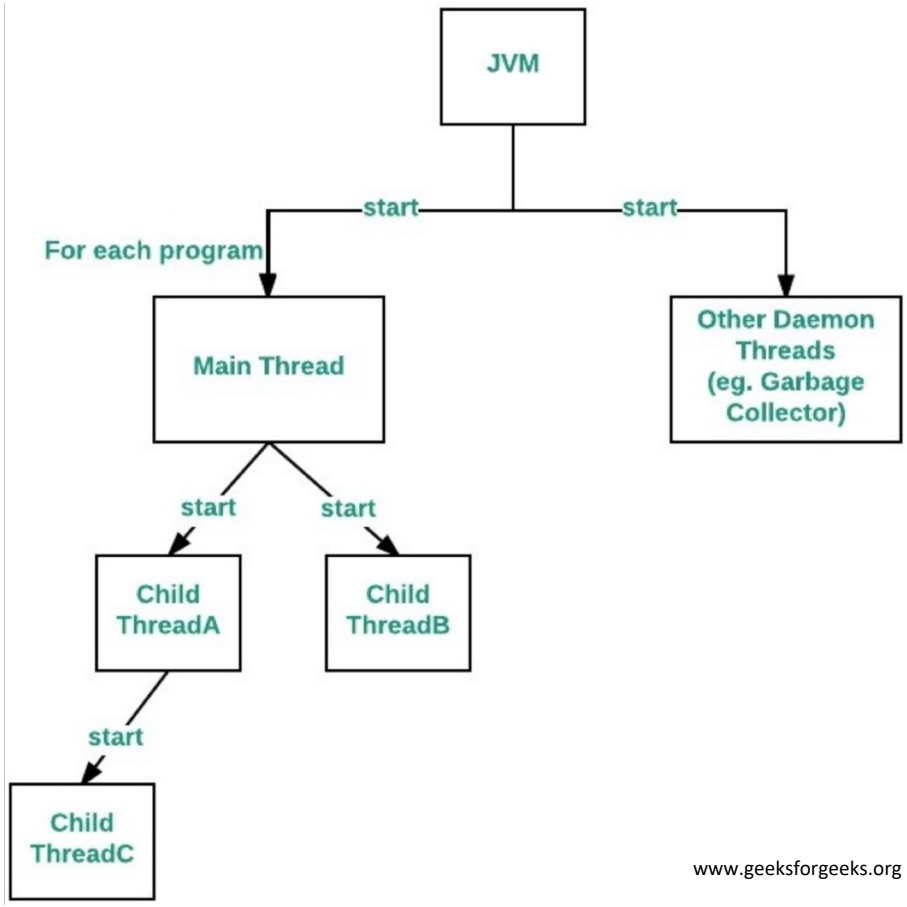
Main thread in Java

- When any Java program starts running a new thread is always created by JVM
  - Main thread of the program
  - Called “main” thread
- Any new thread created within this program is a child thread that spans from the main thread
- Last thread to terminate when the program finishes
- The first thing the main thread does is checks the existence of the *main()* method
- Reference to main thread can be obtained by *currentThread( )*

Daemon threads in Java

- When a thread is marked as daemon Java doesn't wait for it to finish executing before it terminates main thread
- Can set a thread to be daemon using *setDaemon()* method
  - *Thread.setDaemon(true)*
- Can check if a thread is daemon using *isDaemon()* method
- Can have more than one daemon threads running
- JVM designates the lowest priority to daemon threads
  - If JVM terminates it will check for any daemon threads and terminate them before shutting down itself
- These threads are usually used for functionality that is not vital to the program execution and can complete independently from program completion

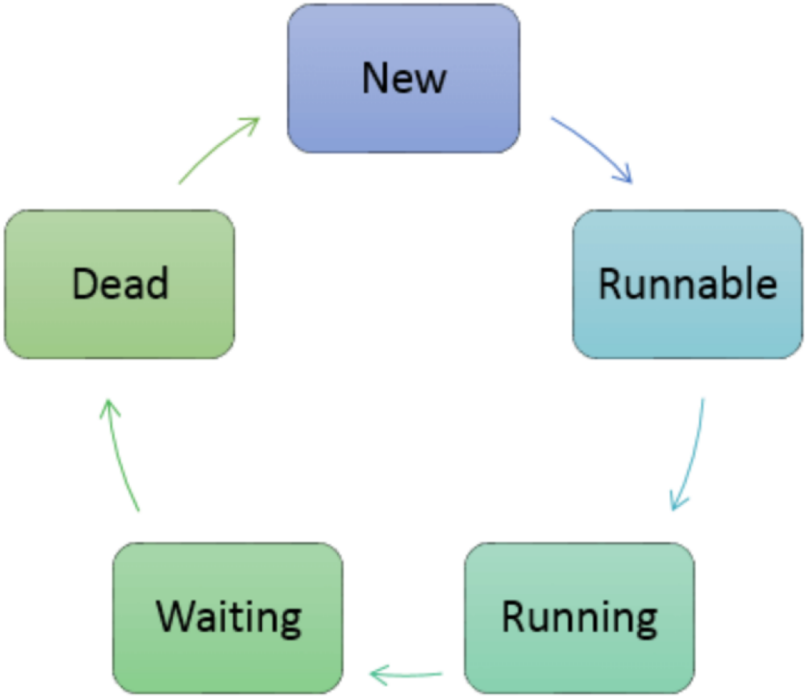
Types of threads in JVM



Stages in lifecycle of a thread

- New – a thread remains in this state until between the time it is created and it is started
- Runnable – after the thread is started it becomes runnable
- Running – when the thread begins to execute its task
- Waiting – a thread is waiting while it waits for another thread to finish executing tasks and until it is signaled to continue executing
- Timed waiting – similar to waiting state but for a fixed amount of time, once that time lapses the thread transitions to runnable state
- Terminated – also called “dead”, thread enters this state when it completes executing its task or is explicitly terminated

Stages in lifecycle of a thread (cont'd)



https://www.guru99.com/

Example: stages of a thread lifecycle

```
class MyClass implements Runnable{
    private String objName;
    private Thread threadObj;

    MyClass(String objName, String threadName){
        this.objName = objName;
        this.threadObj = new Thread(this);
        this.threadObj.setName(threadName);
        System.out.println("Creating " + this.objName +
            ": " +  this.threadObj.getName());
    }

    public void run(){
        System.out.println("Running " +  this.threadObj.getName() +
            " (" + this.objName + ")");
        try{
            System.out.println ("Instance " + this.objName +
                ", current thread: " +
                this.threadObj.getName() +
                "(" + this.threadObj.getId() + ")");

            if(this.threadObj.getName() == "thread3-1"){
                Thread.currentThread().interrupt();
                }

            for(int i = 5; i > 0; i--) {
                System.out.println("Thread " +
                    this.threadObj.getName() + " is running");

                if(i == 3){
                    System.out.println("Thread " +
                        this.threadObj.getName() + " is sleeping");
                    this.threadObj.sleep(20);  // sleep for 20 milliseconds
                    System.out.println("Is thread " +
                        this.threadObj.getName() + " alive: "+this.threadObj.isAlive());
                    }
                }
            } catch (Exception e) {
                System.out.println (e);
                System.out.println("Thread " +
                    this.threadObj.getName() + " was interrupted");
            }
            System.out.println("Exiting " +  this.threadObj.getName() +
                " (" + this.objName + ")");
        }

        public void start(){
            System.out.println("Starting " +  this.threadObj.getName() +
                " (" + this.objName + ")");
            this.run();
        }
}
```

```
public class Test {
    public static void main(String[] args){
        List<MyClass> myClassLst = new ArrayList<MyClass>();
        myClassLst.add(new MyClass("instance1","thread1-1"));
        myClassLst.add(new MyClass("instance2","thread2-1"));
        myClassLst.add(new MyClass("instance3","thread3-1"));

        for(MyClass c : myClassLst){
            c.start();
        }
    }
}
```

Output:

```
Creating instance1: thread1-1
Creating instance2: thread2-1
Creating instance3: thread3-1
Starting thread1-1 (instance1)
Running thread1-1 (instance1)
Instance instance1, current thread: thread1-1(9)
Thread thread1-1 is running
Thread thread1-1 is running
Thread thread1-1 is running
Thread thread1-1 is sleeping
Is thread thread1-1 alive: false
Thread thread1-1 is running
Thread thread1-1 is running
Exiting thread1-1 (instance1)
Starting thread2-1 (instance2)
Running thread2-1 (instance2)
Instance instance2, current thread: thread2-1(10)
Thread thread2-1 is running
Thread thread2-1 is running
Thread thread2-1 is running
Thread thread2-1 is sleeping
Is thread thread2-1 alive: false
Thread thread2-1 is running
Thread thread2-1 is running
Exiting thread2-1 (instance2)
Starting thread3-1 (instance3)
Running thread3-1 (instance3)
Instance instance3, current thread: thread3-1(11)
inside interrupt
Thread thread3-1 is running
Thread thread3-1 is running
Thread thread3-1 is running
Thread thread3-1 is sleeping
java.lang.InterruptedException: sleep interrupted
Thread thread3-1 was interrupted
Exiting thread3-1 (instance3)
```

Java thread priority

- You can set thread priority with *setPriority()* method
  - Positive numeric value
    - Cannot set priority to zero
- You can also obtain current thread's priority with *getPriority()* method
- Default priority of main thread is 5 and every child thread inherits main thread's priority
- Higher priority threads are scheduled first
  - Precedence over lower priority threads

Example: setting and retrieving thread priority

```
class MyClass extends Thread{
    public void run(){
        try{
            System.out.println ("Current thread id: " + Thread.currentThread().getId());
        } catch (Exception e) {
            System.out.println (e);
        }
    }

    @Override
    public void start(){
        Thread t = new Thread (this);
        t.start();
        System.out.println ("Starting a thread with priority "+t.getPriority());
    }
}

public class Test {
    public static void main(String[] args){
        for (int i=0; i<5; i++)
        {
            MyClass myClassObj = new MyClass();
            myClassObj.start();
            System.out.println ("After the start thread priority is: "+myClassObj.getPriority());
            myClassObj.setPriority(i+1);
            System.out.println ("After the change thread priority is: "+myClassObj.getPriority());
        }
    }
}
```

Output:

```
Starting a thread with priority 5
Current thread id: 10
After the start thread priority is: 5
After the change thread priority is: 1
Starting a thread with priority 5
After the start thread priority is: 5
After the change thread priority is: 2
Current thread id: 12
Starting a thread with priority 5
Current thread id: 14
After the start thread priority is: 5
After the change thread priority is: 3
Starting a thread with priority 5
After the start thread priority is: 5
Current thread id: 16
After the change thread priority is: 4
Starting a thread with priority 5
After the start thread priority is: 5
Current thread id: 18
After the change thread priority is: 5
```

Example: demonstrating priority inheritance

```
public class Test extends Thread{
    public void run() {
        System.out.println("Executing run() method");
    }

    public static void main(String[]args) {
        Thread.currentThread().setPriority(8);
        System.out.println("main thread priority is " + Thread.currentThread().getPriority());

        Test myTest = new Test();
        System.out.println("myTest thread priority is " + myTest.getPriority());
    }
}
```

Output:

```
main thread priority is 8
myTest thread priority is 8
```

Java thread synchronization

- When threads share object resources, collisions and data corruptions can happen
  - Two threads change the same object
  - Concurrency issues
- Synchronization of multiple threads
  - Making sure only one thread can access a resource at a given point
- Monitors – only one thread can hold a lock on a monitor at a time
  - Also called “locks”
  - *holdLock()* method in *Thread* class returns true if current thread holds a lock on the specified object’s monitor

How to achieve synchronization in Java

- Synchronized method
- Synchronized block
  - synchronized(objReference){…}
  - Once the code reaches synchronized block no other thread can call the same method on the object referenced by synchronized block
- Static synchronization

Example: non-synchronized thread operations

```
class Stopwatch {
    private int count;

    Stopwatch(int c){this.count = c;}

    public void countDown() {
        for(int i = 1; i <= this.count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

class StopwatchExecutor extends Thread {
    private Thread executionThread;
    private String threadName;
    private Stopwatch sw;

    StopwatchExecutor(String name, Stopwatch sw){
        this.threadName = name;
        this.sw = sw;
    }

    public void run() {
        this.sw.countDown();
        System.out.println("Thread " +  this.threadName + " is exiting");
    }

    public void start() {
        System.out.println("Thread " +  this.threadName + " is starting");
        try{
            executionThread = new Thread(this, this.threadName);
            executionThread.start();
        }catch(NullPointerException e){
            System.out.println(e);
        }
    }
}
```

```
public class Test {
    public static void main(String[]args) {
        Stopwatch sw = new Stopwatch(5);

        StopwatchExecutor se1 = new StopwatchExecutor("thread1", sw);
        StopwatchExecutor se2 = new StopwatchExecutor("thread2", sw);

        se1.start();
        se2.start();

        try {
            se1.join();
            se2.join();
        } catch ( Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Thread thread1 is starting
Thread thread2 is starting
Count =  1
Count =  2
Count =  3
Count =  1
Count =  4
Count =  5
Thread thread1 is exiting
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread2 is exiting
```

This example was inspired by www.tutorialspoint.com

Example: synchronized thread operations with synchronized block

```
class Stopwatch {
    private int count;

    Stopwatch(int c){this.count = c;}

    public void countDown() {
        for(int i = 1; i <= this.count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

class StopwatchExecutor extends Thread {
    private Thread executionThread;
    private String threadName;
    private Stopwatch sw;

    StopwatchExecutor(String name, Stopwatch sw){
        this.threadName = name;
        this.sw = sw;
    }

    public void run() {
        synchronized(this.sw) {
            this.sw.countDown();
        }
        System.out.println("Thread " +  this.threadName + " is exiting");
    }

    public void start() {
        System.out.println("Thread " +  this.threadName + " is starting");
        try{
            executionThread = new Thread(this, this.threadName);
            executionThread.start();
        }catch(NullPointerException e){
            System.out.println(e);
        }
    }
}
```

adding synchronized block puts a lock on the this.sw monitor

```
public class Test {
    public static void main(String[]args) {
        Stopwatch sw = new Stopwatch(5);

        StopwatchExecutor se1 = new StopwatchExecutor("thread1", sw);
        StopwatchExecutor se2 = new StopwatchExecutor("thread2", sw);

        se1.start();
        se2.start();

        try {
            se1.join();
            se2.join();
        } catch ( Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Thread thread1 is starting
Thread thread2 is starting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread1 is exiting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread2 is exiting
```

This example was inspired by www.tutorialspoint.com

Example: synchronized thread operations with synchronized method

```
class Stopwatch {
    private int count;

    Stopwatch(int c){this.count = c;}

    synchronized public void countDown() {
        for(int i = 1; i <= this.count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

class StopwatchExecutor extends Thread {
    private Thread executionThread;
    private String threadName;
    private Stopwatch sw;

    StopwatchExecutor(String name, Stopwatch sw){
        this.threadName = name;
        this.sw = sw;
    }

    public void run() {
        this.sw.countDown();
        System.out.println("Thread " +  this.threadName + " is exiting");
    }

    public void start() {
        System.out.println("Thread " +  this.threadName + " is starting");
        try{
            executionThread = new Thread(this, this.threadName);
            executionThread.start();
        }catch(NullPointerException e){
            System.out.println(e);
        }
    }
}
```

declaring the method as synchronized achieves synchronization when that method is called

```
public class Test {
    public static void main(String[]args) {
        Stopwatch sw = new Stopwatch(5);

        StopwatchExecutor se1 = new StopwatchExecutor("thread1", sw);
        StopwatchExecutor se2 = new StopwatchExecutor("thread2", sw);

        se1.start();
        se2.start();

        try {
            se1.join();
            se2.join();
        } catch ( Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Thread thread1 is starting
Thread thread2 is starting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread1 is exiting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread2 is exiting
```

When thread invoked synchronized method it automatically acquires a lock for the object

This example was inspired by www.tutorialspoint.com

Static synchronization

- Static methods can be synchronized as well
- The lock is put on the class, not the object

Example: synchronized thread operations with synchronized method

```
class Stopwatch {
    synchronized public static void countDown(int count) {
        for(int i = 1; i <= count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

class StopwatchExecutor extends Thread {
    private Thread executionThread;
    private String threadName;

    StopwatchExecutor(String name){
        this.threadName = name;
    }

    public void run() {
        Stopwatch.countDown(5);
        System.out.println("Thread " +  this.threadName + " is exiting");
    }

    public void start() {
        System.out.println("Thread " +  this.threadName + " is starting");
        try{
            executionThread = new Thread(this, this.threadName);
            executionThread.start();
        }catch(NullPointerException e){
            System.out.println(e);
        }
    }
}
```

declaring the method as synchronized will put a lock on Stopwatch class whenever it is called

```
public class Test {
    public static void main(String[]args) {
        StopwatchExecutor se1 = new StopwatchExecutor("thread1");
        StopwatchExecutor se2 = new StopwatchExecutor("thread2");

        se1.start();
        se2.start();

        try {
            se1.join();
            se2.join();
        } catch ( Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Thread thread1 is starting
Thread thread2 is starting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread1 is exiting
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread2 is exiting
```

Output if we remove keyword synchronized:

```
Thread thread1 is starting
Thread thread2 is starting
Count =  1
Count =  2
Count =  1
Count =  2
Count =  3
Count =  4
Count =  5
Thread thread1 is exiting
Count =  3
Count =  4
Count =  5
Thread thread2 is exiting
```

Other ways of implementing thread safety in Java

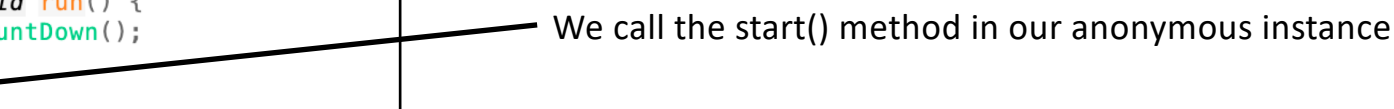
- Atomic wrappers in *java.util.concurrent.atomic* package
  - Toolkit of classes that support programming of lock-free thread-safe variables
  - E.g. *AtomicInteger*
- Locks in *java.util.concurrent.locks* package
  - Toolkit of interfaces and classes for locking objects and waiting for conditions
- Thread-safe collections and classes
- Use *volatile* keyword with variables
  - Indicates that the variable should always be read straight from memory and not from the thread's cache
  - Used when the variable might be modified by multiple threads

Example: using anonymous class to instantiate a thread

```
class Stopwatch {
    private int count;

    Stopwatch(int c){this.count = c;}

    synchronized public void countDown() {
        for(int i = 1; i <= this.count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

public class Test {
    public static void main(String[]args) {
        Stopwatch sw = new Stopwatch(5);

        Thread t1 = new Thread() {
            public void run() {
                sw.countDown();
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                sw.countDown();
            }
        };

        t1.start();
        t2.start();
    }
}
```

Example: using anonymous instance of a Thread class

```
class Stopwatch {
    private int count;

    Stopwatch(int c){this.count = c;}

    synchronized public void countDown() {
        for(int i = 1; i <= this.count; i++){
            System.out.println("Count =  " + i);
        }
    }
}

public class Test {
    public static void main(String[]args) {
        Stopwatch sw = new Stopwatch(5);

        new Thread() {
            public void run() {
                sw.countDown();
            }
        }.start();

        new Thread() {
            public void run() {
                sw.countDown();
            }
        }.start();
    }
}
```



Thread cooperation and inter-thread communication

- Sometimes two or more threads in your program need to exchange some information
- Ways threads can communicate:
  - *wait()* – causes current thread to wait until it is notified by another thread
  - *notify()* – gives a notification to a particular thread that is waiting for the object
  - *notifyAll()* – gives a notification to all threads waiting for the object

Example: thread cooperation

```
class BankAccount{
    private double balance;

    BankAccount(double initialDeposit){
        this.balance = initialDeposit;
    }

    public double getBalance(){return this.balance;}

    synchronized public void withdraw(double amount){
        System.out.println("Withdrawl for amount $"+amount+" is initiated");

        if(this.balance < amount){
            try{
                wait();
            }catch(Exception e){System.out.println(e);}
        }

        this.balance -= amount;
        System.out.println("Withdrawl for amount $"+amount+" is completed");
    }

    synchronized public void deposit(double amount){
        System.out.println("Deposit for amount $"+amount+" is initiated");
        this.balance += amount;
        System.out.println("Deposit for amount $"+amount+" is completed");
        notify();
    }
}
```

```
public class Test {
    public static void main(String[]args) {
        BankAccount b = new BankAccount(1000.00);
        System.out.println("Initial bank balance is "+b.getBalance());

        Thread t1 = new Thread(){
            public void run(){b.withdraw(1500.00);}
        };
        t1.start();

        Thread t2 = new Thread(){
            public void run(){b.deposit(2000.00);}
        };
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch(Exception e){System.out.println(e);}

        System.out.println("Final bank balance is "+b.getBalance());
    }
}
```

Output:

```
Initial bank balance is 1000.0
Withdrawl for amount $1500.0 is initiated
Deposit for amount $2000.0 is initiated
Deposit for amount $2000.0 is completed
Withdrawl for amount $1500.0 is completed
Final bank balance is 1500.0
```

Thread deadlock

- Two or more threads are blocked waiting for each other
- Usually this situation occurs when two threads need to obtain the same locks but obtain them in different order
- Deadlocks do not resolve themselves
- Changing the order of the locks can solve the deadlock problem
- Lock timeouts is another way to solve a deadlock problem
  - Timeout on lock attempts
- Finally, lock detection in your program
  - Advanced topic in Java programming

Exceptions you might encounter when dealing with Java threads

- ExceptionInMain – comes up when there is a mismatch in compile vs. execution JDK versions
- NoClassDefFoundError – JRE tries to load a class but the class definition is no longer available
- NoSuchMethodError – method is called that is not present in the class definition
  - Compiled against a different version of the class?
- NullPointerException – provided reference is null
- InterruptedException – thrown when a thread is interrupted before or during activity
- IllegalMonitorStateException – thread tried to notify other threads about or wait on a monitor that it has no control over

Recap of what we learned

- What is multithreading and how to achieve it in Java
- Create new instances of a thread
  - Extend *Thread* class
  - Implement *Runnable* interface
  - Overriding *start()* and *run()* methods
  - Anonymous *Thread* class and *Thread* instance
- Special threads
  - main thread
  - Daemon thread
- Lifecycle of a thread
- Thread priorities
- Thread safety
  - Thread synchronization
- Thread cooperation
- Thread control
- Deadlocks

Concluding remarks

- Multithreading is a great way to
  - Maximize CPU utilization
  - Improve performance
  - Improve application responsiveness
- However, multithreading is challenging
  - Multiple threads are accessing and writing to the same memory
    - Thread-safety might be difficult to achieve in practice
  - Some of the issues may not be detected on a single CPU machine
  - Concurrent coding is more difficult to write and is harder to read
  - Use caution!
  - Test your application well!