Introduction to Software Design & Concurrency

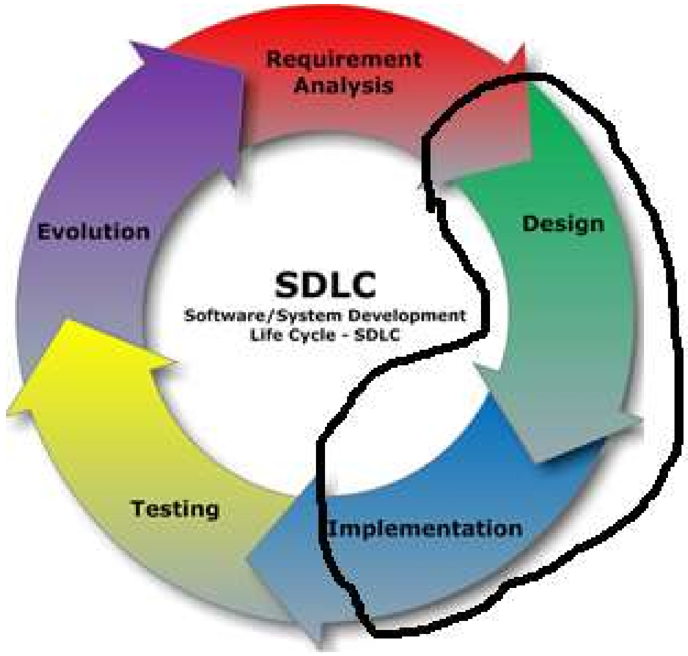
Lecture Notes

Software Design (SE324)

(Prepared by Dr. Khaldoon Al-Zoubi)

Software Design

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
  - SDLC → to produce software product
- Software Design → How to build software Systems
  - The **process** for <u>turning</u> a specification into operational software with acceptable quality
  - The **process** of <u>implementing</u> software solutions to one or more sets of problems with acceptable quality
  - The **process** of <u>How to build software</u> to meet its intended functional and non-functional requirements
    - Software Modelling → representations of the real thing (software system)
    - Programming → to build the real thing

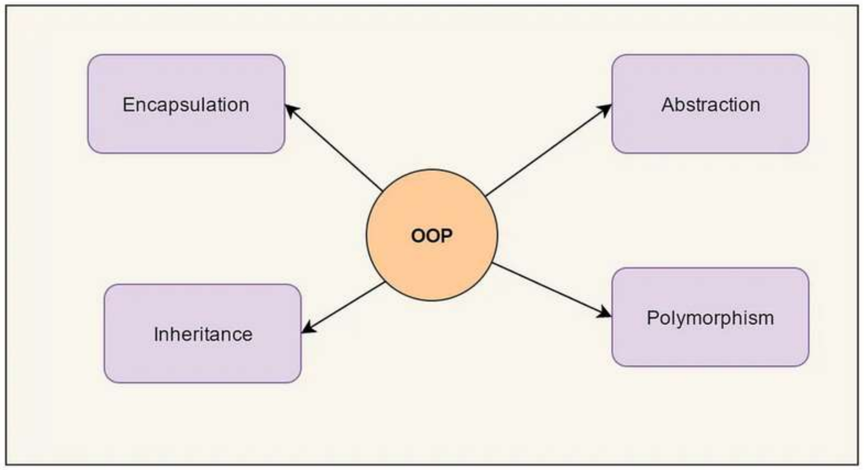


Why Software Design matters?

- Because it always takes more time to write something from scratch than changing something that is already there.
- With good design the time you spend to change (and test) something is also minimized
- Maintainable software
  - Can implement new requirements with less cost
  - Can change existing implementation with less cost
    - Because requirement keep changing
    - Unit Test Programs also need to be changed
      - They also need to have good design
- What is a bad code?

Why Software Design matters?

- Object-oriented organizes software design around data, or objects, rather than functions and logic (procedural-programming).
- Object-oriented principles
  - Encapsulation — hide information + public accessor
    - WHY????
  - Abstraction means a concept or an Idea.
    - Using abstract class/Interface we express the intent of the class rather than the actual implementation
    - WHY????
  - Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class.
  - Polymorphism describes situations in which something occurs in several different forms.



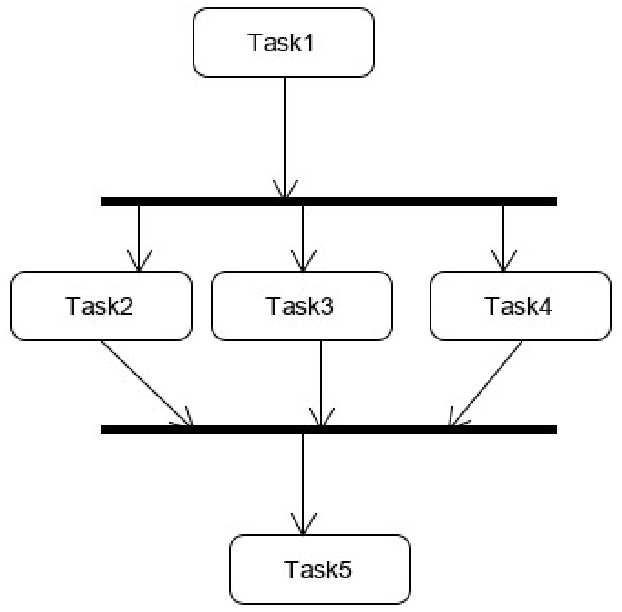
Why Software Design matters?

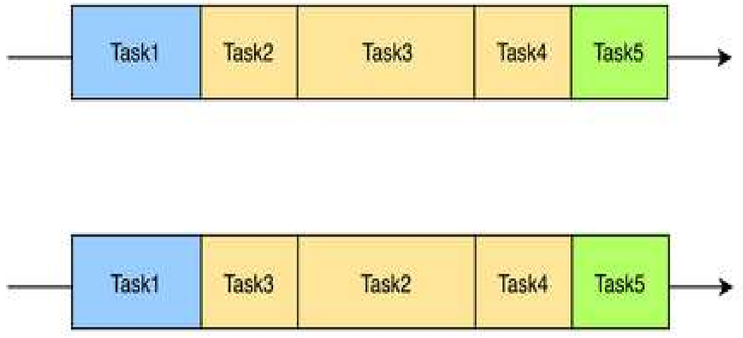
- ***How Software designers solve problems (Remember: object-oriented)?***
  - Dividing the problem
    - Modularization
    - Component-based software
  - Isolating parts (each part has one goal/responsibility)
    - High Cohesion
      - High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
      - **Cohesion** refers to the degree to which the elements of an entity belong together
  - Reducing dependencies between different related elements
    - Lower Coupling
      - The goal of a loose/low coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements.
      - **Coupling** refers to degree of interdependence between software entities.

Concurrency

Concurrency

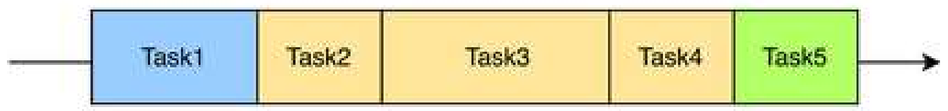
- In theory: concurrency refers to the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.
  - If you have tasks having inputs and outputs, and you want to **schedule** them so that they produce correct results, you are solving a concurrency problem.
  - Operating system manage all resources like
    - Threads scheduling
    - Hardware access (processor & memory)
- The figure shows a data flow with input and output dependencies.
  - Tasks (threads) 2, 3, 4 need to run after 1, and before 5.
    - There is no specific order between them
    - so we have multiple alternatives for running it sequentially. Showing only two of them below

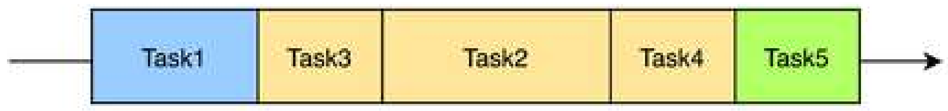


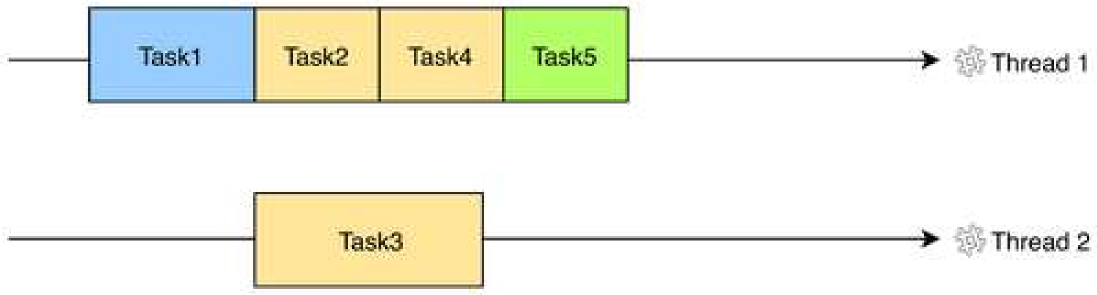


Concurrency

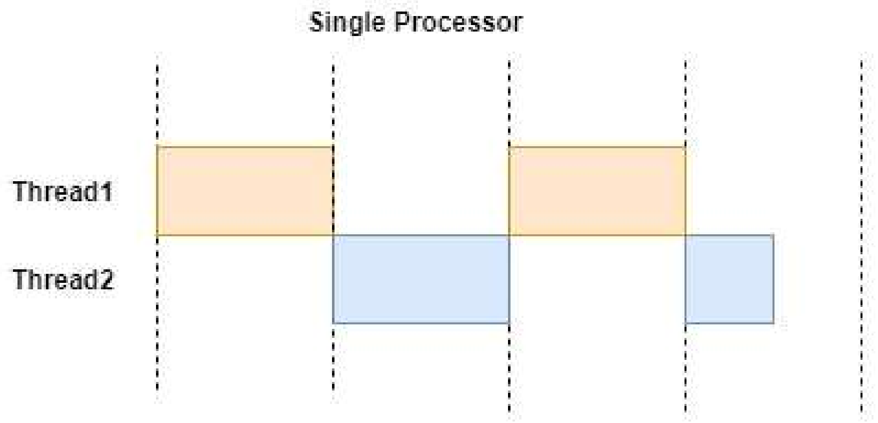
- Alternatively, these tasks can run in parallel,
  - e.g. on another processor core, another processor, or an entirely separate computer.
  - Parallel = tasks run at the same time on multi-core processors
    - Special case of concurrency
- Running concurrent tasks in parallel can reduce the overall computation time $\rightarrow$ speed it up
- How about concurrency on single processor core?
  - Reduce latency responsiveness

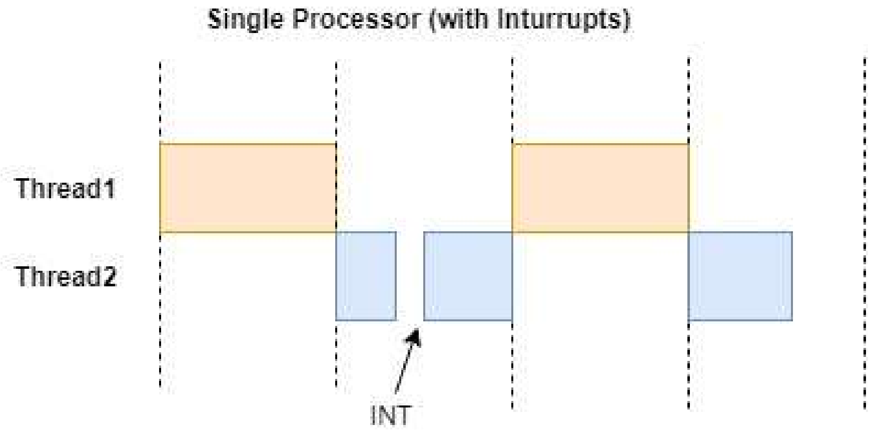


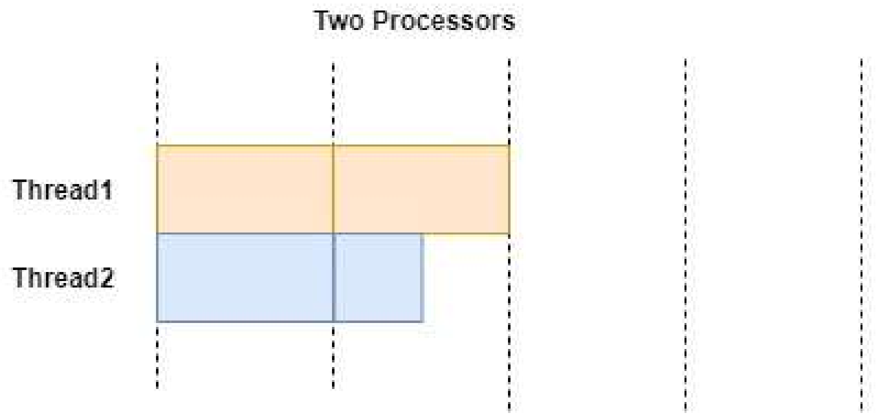




Possible Executions by OS

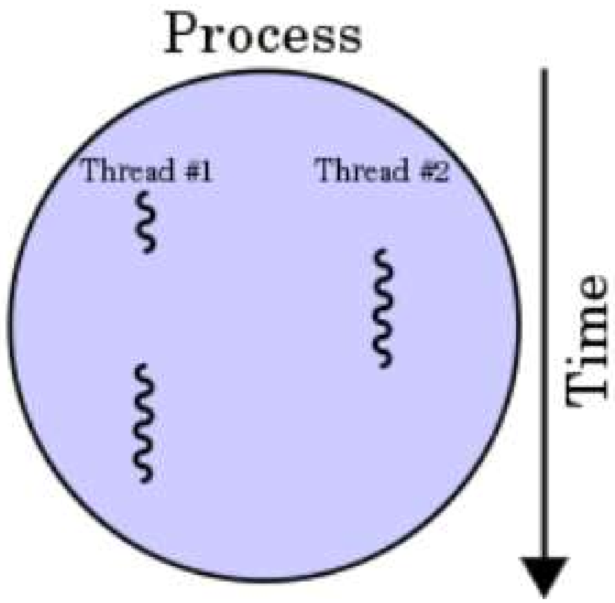






Process & Threads

- A process has a self-contained execution environment.
  - A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
  - Communicate with each other via operating system
    - *Inter Process Communication* (IPC)
- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time

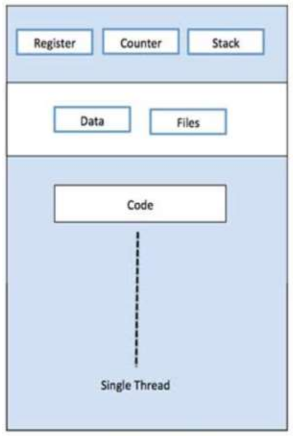


Thread Context switching

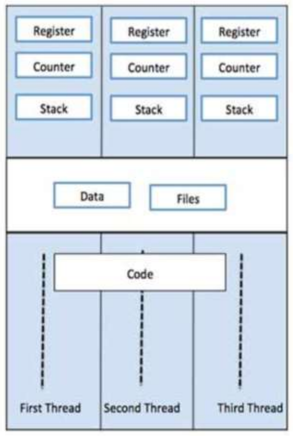
- Context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point
- *Thread*'s *context block contains things like*
  - *Processors registers, Thread's Stack, PC, etc.*
  - *Allowing threads to resume execution correctly*
- OS calls routine *Switch(oldThread, nextThread)* , which performs:
  1. Save all registers in *old Thread*'s *context block*
  2. Save Program Counter (PC) in *old Thread*'s *context block*
  3. load next thread values into the registers from the context block of the next thread
  4. Restore next thread saved PC. At this point, next thread is executing (Context routine is done)
- Typical Reasons for Threads context switching:
  – Time slice expired
    – ***OS Time slice*** *is the period of time for which a thread is allowed to run in a multitasking system*
  – Timer or I/O interrupt
  – Higher priority thread needs to run

Threads Within same Program (Process)

- Not Shared between Threads
  - Thread stacks: special region of your computer's memory that stores temporary variables created by each function
    - "LIFO" (last in, first out) data structure
    - **Local variables & method parameters**
    - Each thread gets a stack (not shared with others)
- Shared between Threads (within same program)
  - Heap structure
  - Shared resources between threads
  - Threads from different programs managed by OS
    - Programmers don’t need to worry about it



Single Process P with single thread



Single Process P with three threads

Method Call Stack Example

```
 1: #include <stdio.h>
 2:
 3: int mogrify(int a, int b){
 4:   int tmp = a*4 - b / 3;
 5:   return tmp;
 6: }
 7: double truly_half(int x){
 8:   double tmp = x / 2.0;
 9:   return tmp;
10: }
11: int main(){
12:   int a = 7, y = 17;
13:   int mog = mogrify(a,y);
14:   printf("Done with mogrify\n");
15:
16:   double x = truly_half(y);
17:   printf("Done with truly_half\n");
18:
19:   a = mogrify(x,mog);
20:
21:   printf("Results: %d %lf\n",mog,x);
22:   return 0;
23: }
```

main() called

Method	Line	Var	Value	Addr
main()	12	a	?	1024
		y	?	1028
		mog	?	1032
		x	?	1036

One line of main() executed

Method	Line	Var	Value	Addr
main()	13	a	7	1024
		y	17	1028
		mog	?	1032
		x	?	1036

mogrify() called

mogrify()	4	a	7	1044
		b	17	1048
		tmp	?	1052
main()	13	a	7	1024
		y	17	1028
		mog	?	1032
		x	?	1036

mogrify() first line executed

mogrify()	5	a	7	1044
		b	17	1048
		tmp	23	1052
main()	13	a	7	1024
		y	17	1028
		mog	?	1032
		x	?	1036

mogrify() second line (return) executed

Method	Line	Var	Value	Addr
main()	13	a	7	1024
		y	17	1028
		mog	23	1032
		x	?	1036

What could go wrong in concurrency?

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Multi-word operations are not atomic
- To solve any of these, you need **<u>synchronization</u>**.
  - You want program behavior to be a specific function of input
    - not of the sequence of who went first.
  - You want the behavior to be deterministic
    - not to vary from run to run

Synchronization

Synchronization refers to the coordination of simultaneous threads or processes to complete a task with correct runtime order and no unexpected race conditions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock (mutex = mutual exclusion): prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)
- only one thread

Semaphores

- Use a semaphore when you (thread) want to sleep till some other thread tells you to wake up
- Allow one or more threads to enter

Synchronization

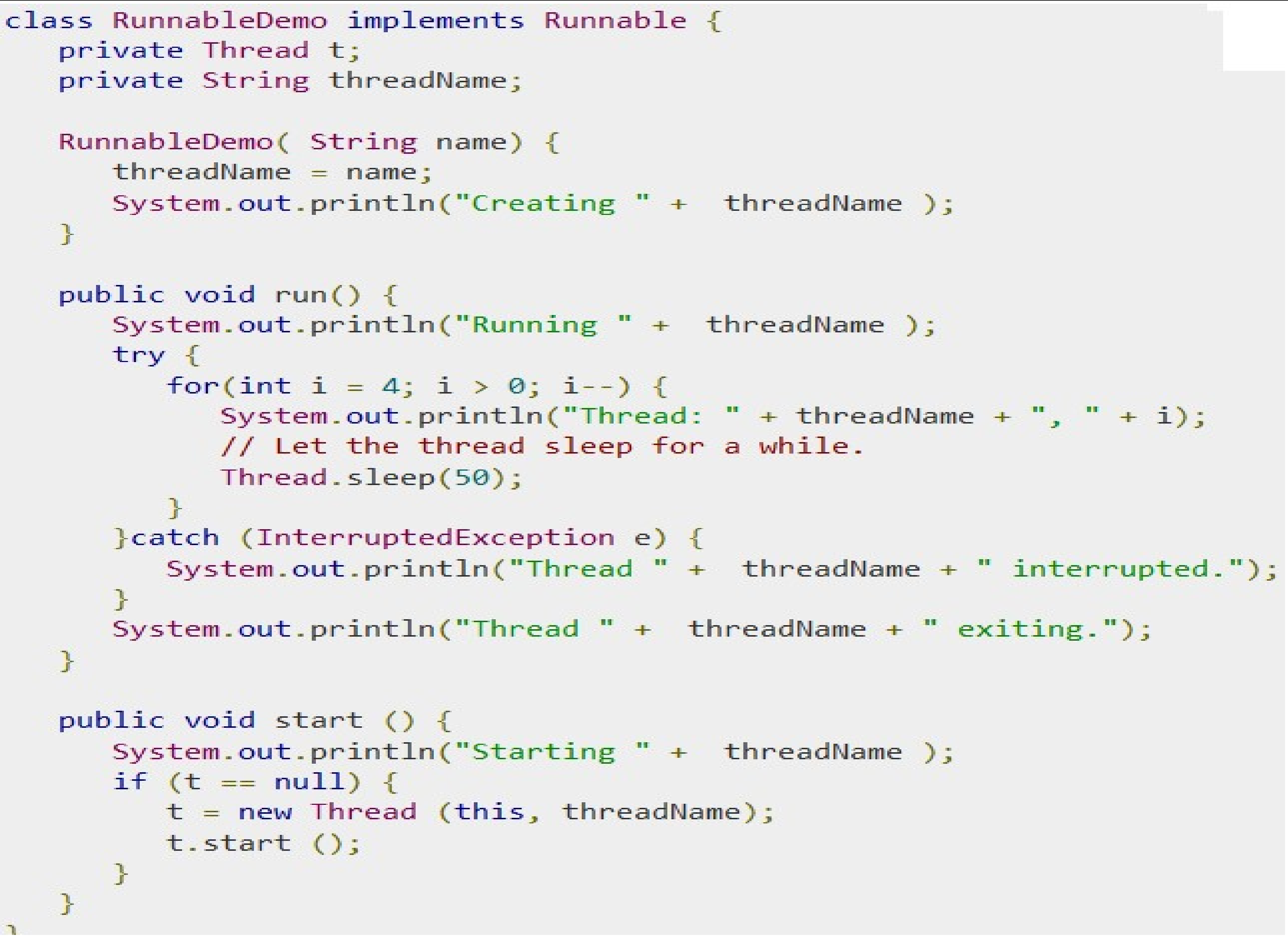
- A program with lots of concurrent threads can still have poor performance on a multiprocessor
  - Threads creation overhead & Context switching
  - Unnecessary synchronization (no need for protection)
    - Big critical sections
- Starvation: thread waits indefinitely to enter the critical section
- Deadlock: circular waiting for resources

Java Concurrency

Java Threads Creation

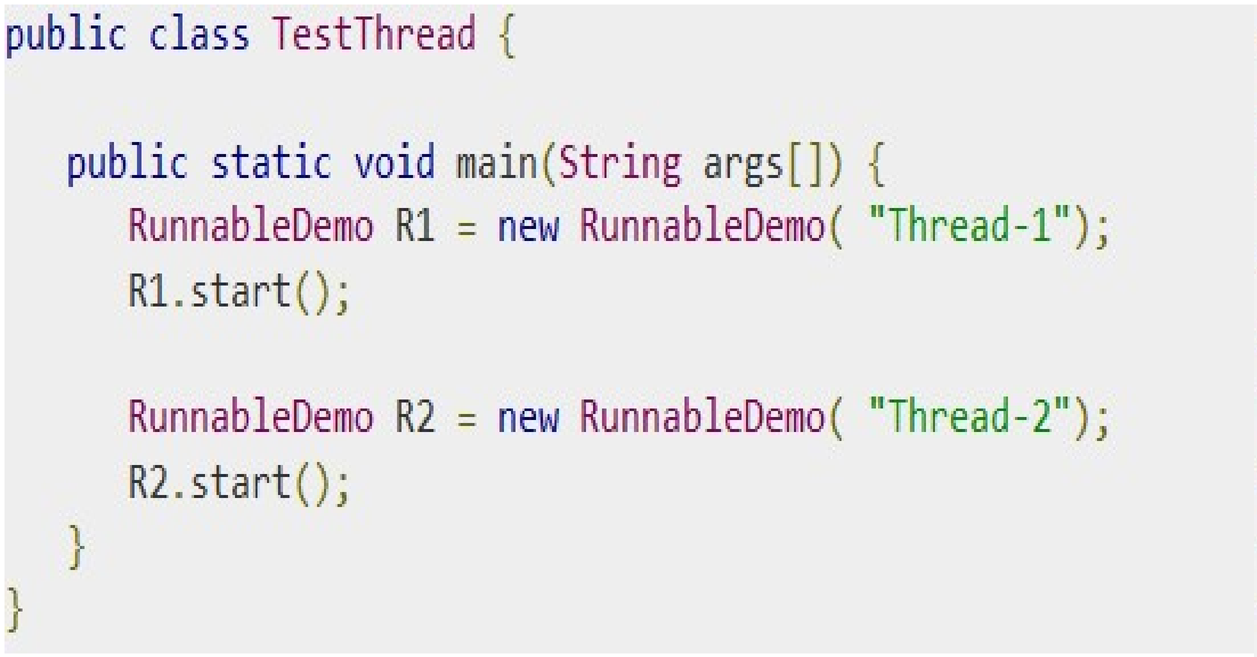
- **Option 1:**
  - If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface
  - you need to implement a **run()** method provided by a Runnable interface.
    - run() provides an entry point for the thread and you will put your complete business logic inside this method
  - you will instantiate a **Thread** object using the following constructor : *Thread(Runnable threadObj, String threadName);*
  - Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method

Java Threads (runnable interface Example)

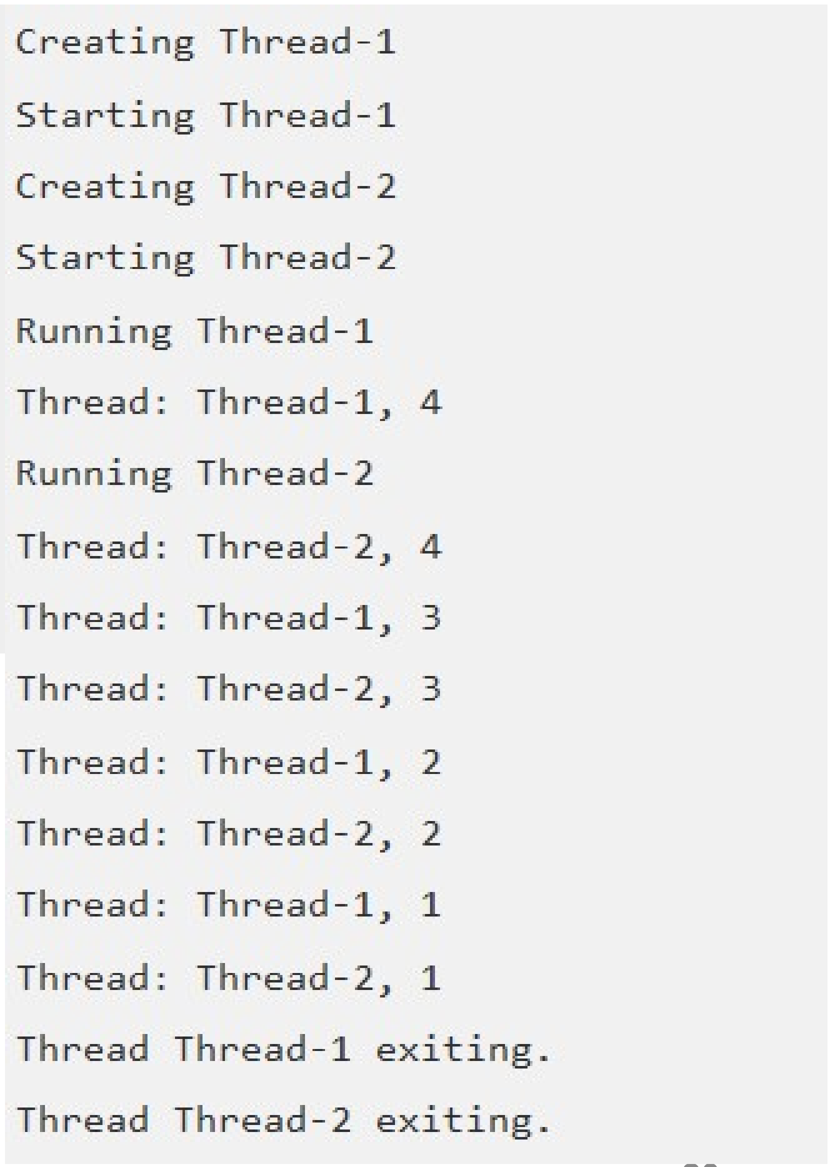


```java
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      }catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

Java Threads (runnable interface Example Cont.)



```
public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```



```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

- **Note: You could use *join* to allow a thread to wait**
- **Example : t.join();**
  - **causes the current thread to pause execution until t's thread terminates.**

Java Threads Creation

- **Option 2:**
  - The second way to create a thread is to create a new class that extends **Thread** class
  - you need to implement a **run()** method available in Thread class.
    - run() provides an entry point for the thread and you will put your complete business logic inside this method
  - Once a Thread object is created, you can start it by calling **start()** method, which executes a call to **run()** method

Java Threads (Thread Class)

```
class ThreadDemo
{
   public static void main (String [] args)
   {
      MyThread mt = new MyThread ();
      mt.start ();
      for (int i = 0; i < 50; i++)
           System.out.println ("i = " + i + ", i * i = " + i * i);
   }
}
class MyThread extends Thread
{
   public void run ()
   {
      for (int count = 1, row = 1; row < 20; row++, count++)
      {
           for (int i = 0; i < count; i++)
                System.out.print ('*');
           System.out.print ('\n');
      }
   }
}
```

Synchronization in Java

- Any object can serve as a lock
  - Separate object: `Object myLock = new Object();`
  - Current instance: “this” object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
     // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others “block” or wait until the first thread releases the lock
- Common situation: all the code in a method is a critical section
  - add synchronized keyword to method signature.
  - E.g. `public` <u>`synchronized`</u> `void update(…) {`

- `synchronized(this) {`

```
        // code here
}
```

Synchronization in Java

- Any object can serve as a lock
  - Separate object: **static** `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
    // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others "block" or wait until the first thread releases the lock
- Common situation: all the code in a method is a critical section
  - add synchronized keyword to method signature.
  - E.g. `public synchronized static void update(…) {`

– `synchronized (DemoClass.class) {`

```
        //other thread safe code
}
```

Example

```
public class BankAccount {

    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Is this a thread safe?

Example

```
public class BankAccount {

    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Is this a thread safe?

Example

```
public class BankAccount {

  private double balance;

  public synchronized void deposit(double amount) {
      balance += amount;
  }

  public synchronized void withdraw(double amount) {
      balance -= amount;
  }
}
```

Is this a thread safe?

Example

```
public void method1() {
    do something ...
    synchronized(this) {
        a ++;
    }
    ................
}


public void method2() {
    do something ...
    synchronized(this) {
        b ++;
    }
    ................
}
```

```
  class Test {
        private Object lockA = new Object();
        private Object lockB = new Object();

public void method1() {
    do something ...
    synchronized(lockA) {
        a ++;
    }
    ................
}


public void method2() {
    do something ...
    synchronized(lockB) {
        b ++;
    }
    ................
 }
```

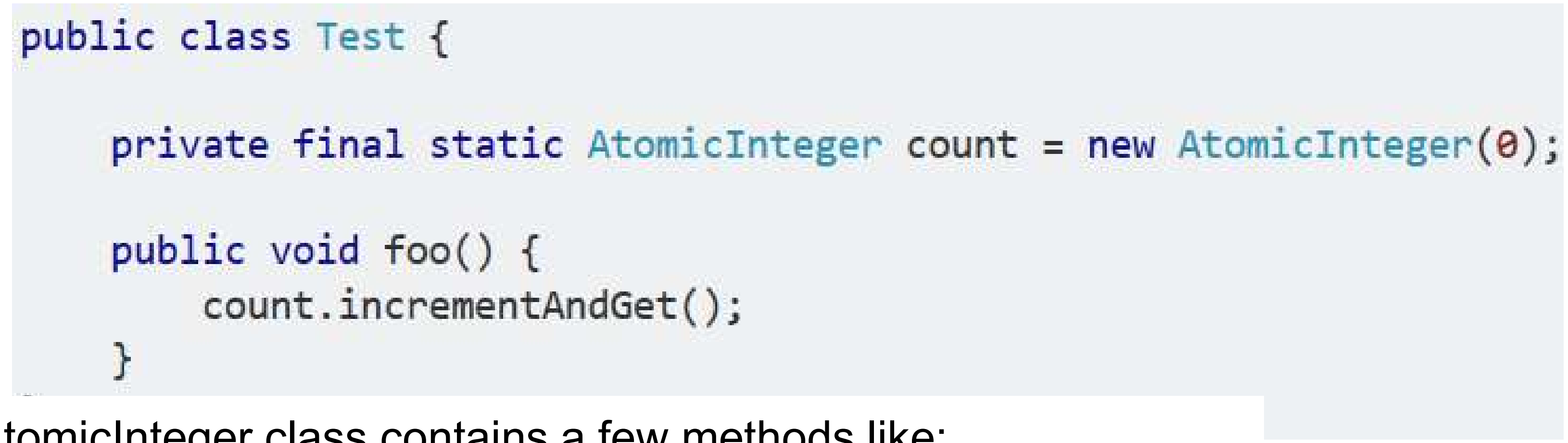
Example

```java
public class Test {
    private static int count = 0;

    public void incrementCount() {
        synchronized (Test.class) {
            count++;
        }
    }
}
```

```java
public class Test {
    private static int count = 0;

    public static synchronized void incrementCount() {
        count++;
    }
}
```

```java
public class Test {
    private static int count = 0;
    private static final Object countLock = new Object();

    public void incrementCount() {
        synchronized (countLock) {
            count++;
        }
    }
}
```

Atomic Access Example



```
public class Test {

    private final static AtomicInteger count = new AtomicInteger(0);

    public void foo() {
        count.incrementAndGet();
    }
```

AtomicInteger class contains a few methods like:
•addAndGet()
•getAndAdd()
•getAndIncrement()
•incrementAndGet()

•Also AtomicDouble, etc.

Guarded Blocks

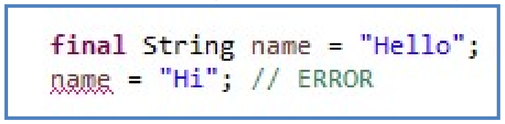
```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

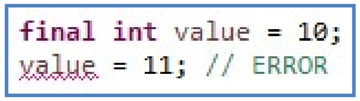
```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Immutable Objects

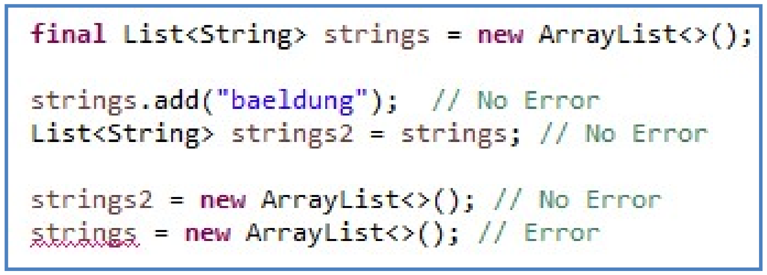
- An object is considered immutable if its state cannot change after it is constructed.
  - read-only
  - In Java, variables are **mutable** by default
    - meaning we can change the value they hold.
- **“final”** keyword
  - forbids us from changing the reference the variable holds, it doesn't protect us from changing the internal state of the object it refers to by using its public API



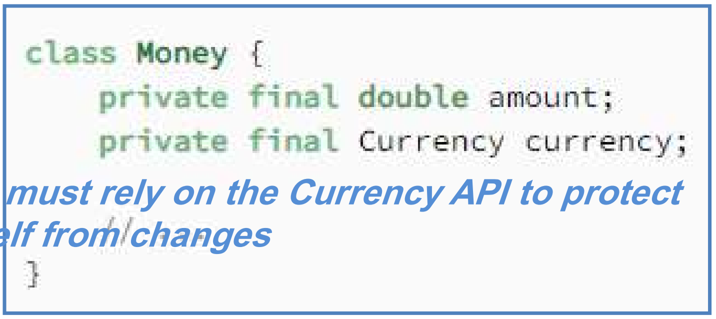
```java
final String name = "Hello";
name = "Hi"; // ERROR
```



```java
final int value = 10;
value = 11; // ERROR
```

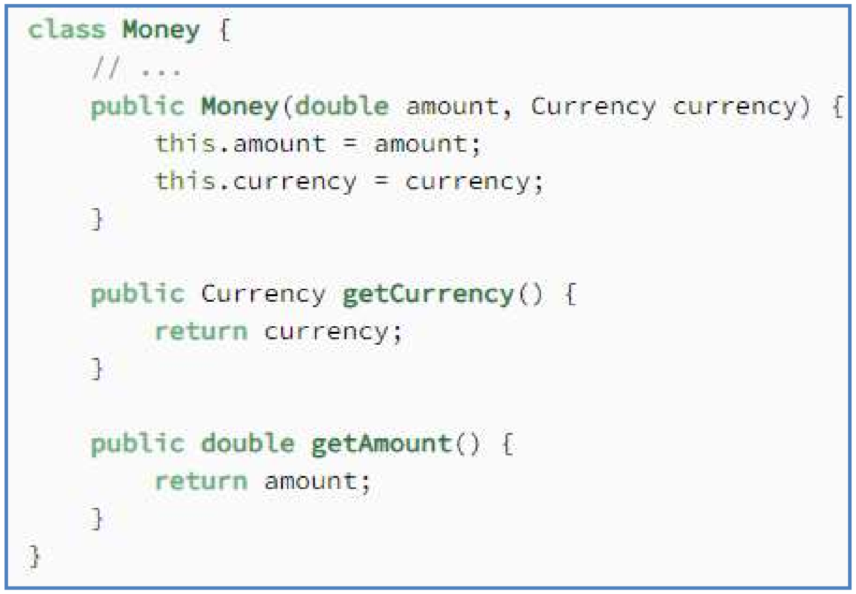


```java
final List<String> strings = new ArrayList<>();

strings.add("baeldung");   // No Error
List<String> strings2 = strings; // No Error

strings2 = new ArrayList<>(); // No Error
strings = new ArrayList<>(); // Error
```



```java
class Money {
    private final double amount;
    private final Currency currency;

    //
}
```

we must rely on the Currency API to protect itself from changes

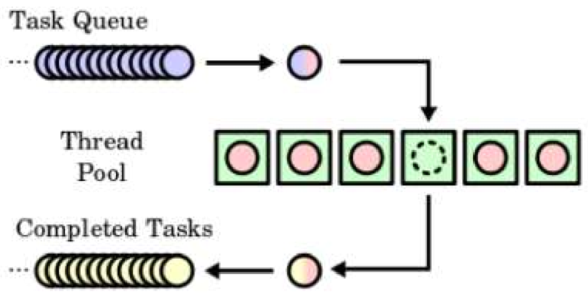


```java
class Money {
    // ...
    public Money(double amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public Currency getCurrency() {
        return currency;
    }

    public double getAmount() {
        return amount;
    }
}
```

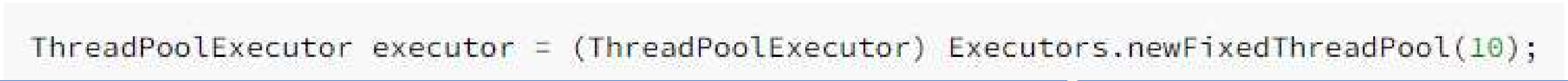
To be Immutable

1. **Don't provide "setter" methods *(methods that change internal data)***
2. **Make all fields final and private**
3. **Don't allow subclasses to override methods *(declare the class as final)***
4. **don't allow referenced mutable objects to be changed**

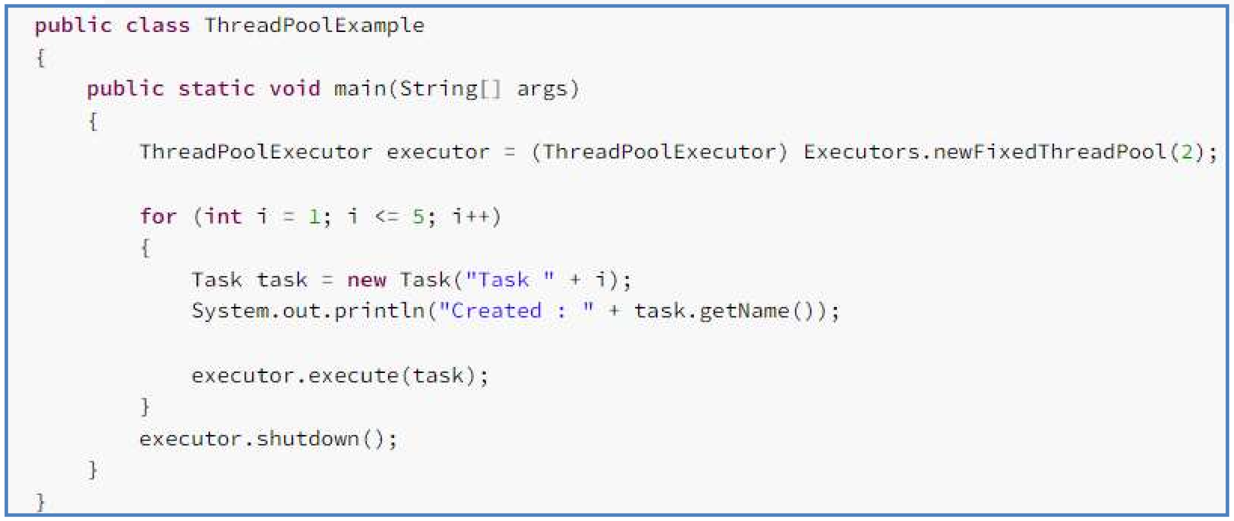
Thread Pool



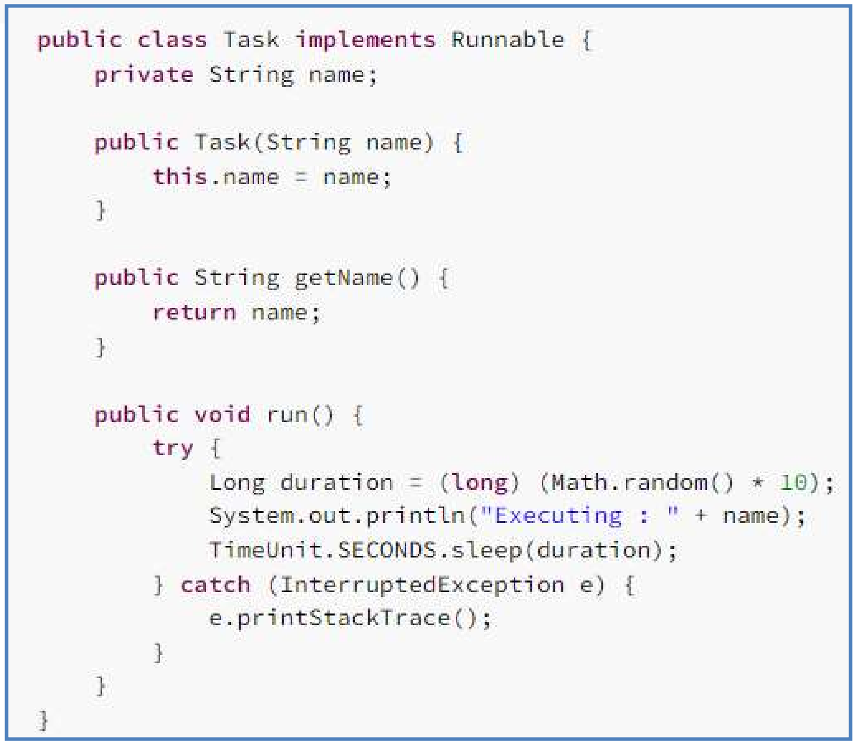
- One kind is the Fixed thread pool executor
  - *There are other kinds of pool executors*



```java
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
```



```java
public class ThreadPoolExample
{
    public static void main(String[] args)
    {
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(2);

        for (int i = 1; i <= 5; i++)
        {
            Task task = new Task("Task " + i);
            System.out.println("Created : " + task.getName());

            executor.execute(task);
        }
        executor.shutdown();
    }
}
```



```java
public class Task implements Runnable {
    private String name;

    public Task(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void run() {
        try {
            Long duration = (long) (Math.random() * 10);
            System.out.println("Executing : " + name);
            TimeUnit.SECONDS.sleep(duration);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Collections

Synchronized Collection Methods of Collections class
Collections.synchronizedCollection(Collection<T> c)
Collections.synchronizedList(List<T> list)
Collections.synchronizedMap(Map<K,V> m)
Collections.synchronizedSet(Set<T> s)
Collections.synchronizedSortedMap(SortedMap<K,V> m)
Collections.synchronizedSortedSet(SortedSet<T> s)

```
List<String> syncList = Collections.synchronizedList(new ArrayList<String>());

syncList.add("one");
syncList.add("two");
syncList.add("three");
```