Practical Session #06 - Threads And Concurrency / Safety

Tom Mahler
In This Recitation We’ll Cover

- General
- Threads
- Concurrency / Safety
- Callback Functions
- The Singleton
- Summary
General
All the computer programs you've seen so far were sequential
  Only one thing was performed at any given time

Sometimes it is desirable to do several things simultaneously
Modern operating systems (Windows / Linux / OSX) allow the execution of several programs (several **processes**) at the same time

- If the tasks to be performed are not related in any way, using a different process for each task is a good solution
- If the tasks are somewhat related, and need to share data between them, it is beneficial to put them inside the same process, so they can share resources (e.g., memory, code and files)

**This is achieved using** Threads
General

- Threads enable us to perform several tasks at a given time
- Several Threads can run simultaneously on a single process, sharing its resources
- Special care should be taken when using those shared resources
In this practical session we will learn how to use threads in Java, and discuss some of the problems that arise from threads usage.

In future practical sessions, we'll see how to solve these problems.
Threads
How To Use Threads

- In order to run several tasks in parallel, two steps should be taken:
  - Define the tasks
  - Create threads and assign each with a task
How To Use Threads Step 1: Define The Tasks

- **Defining a task** is done by implementing the Runnable interface

- This is a very simple interface with a single method: void run(), which defines the work unit (task) to be performed
Examples

See code...
(See Threads01e.java SimpleRunnable_e)
How To Use Threads Step 2: Assigning Tasks To A Thread

- We will now see two ways of assigning tasks to a thread:
  - Using Java's Thread class
  - Using Java's Executor framework
Assigning Tasks To A Thread: Using Java's Thread Class

- Thread is a class which its constructor takes a single Runnable object

- Upon calling the start() method, the thread will invoke the supplied Runnable run() method (without blocking the calling thread)

- Calling start():
  - First sets up the thread internals (allocates resources)
  - Then when it is ready, calls the Runnable run() method
Examples

- Implementing Runnable, and creating a Thread that will run it

- See code...
  (See Threads01.java and its output Threads01.txt)
  (See Threads01e.java and its output Threads01e.txt)
Assigning Tasks To A Thread: Using Java's Executor Framework

- An Executor is a service that internally manages one or more threads and allows you to assign tasks for them.

- There are many types of Executors (each of them with a different policy regarding how to manage the internal threads - how many tasks should be performed at the same time, etc), but all of them implement the ExecutorService interface.
Examples

- In the following example, we create several Runnable tasks, and then pass them to the ExecutorService for execution.

- See code...
  (See Threads01e.java and its output Threads01e.txt)
Threads Can Be Dangerous

- Having things run "at the same time, in the same place", can be dangerous

- The two tasks can interfere with each other, and unexpected results might occur

- Next we present two programs which will work fine in a "regular" single-threaded environment, but fail in the case of multi-threaded execution
Threads Can Be Dangerous: Synchronization Problems #1

- In this example, we wish to print complete lines to the screen using a dedicated Printer object, which is shared between two threads.

- Unsynchronized shared resource - the Printer is shared, so we get a mixture of lines.
Examples

- See code...
  (See [Thread02.java](#) and its output [Thread02.txt](#))
Threads Can Be Dangerous: Synchronization Problems #2

- In this example we use the object `Even`, which holds an even numeric counter.

- We'll define the pre and post conditions and see if they are being kept.

- The conditions are kept when using a single thread; yet in a multi-threaded environment this class is not safe.
Examples

- See code...
  (See Threads03.java and its output Threads03.txt)
Threads Can Be Dangerous: Synchronization Problems #2

- A simple solution: check if the pre / post conditions are broken, and throw an exception if so
Examples

- See code...
(See [Threads04.java](#) and its output [Threads04.txt](#))
Threads Can Be Dangerous: Synchronization Problems #2

- This doesn't really solve the problem - we may still get a bad output and no exception

- Notice that the last number of the output can be less than 1000, this means that the command n++ in the next() method hadn't been executed 1000 times as you would expect
Threads Can Be Dangerous: Synchronization Problems #2

To understand more about what commands are composed of, you can print the java byte code generated after its compilation using the console command `javap -c [name of the tested class file]`

Further details on `javap` can be found [here](#)
Shared Resources

- Safety problems arise when multiple threads access the same resource

- But what exactly is a shared resource?
  - A shared resource is any object that is visible to several threads
  - This includes global (static) objects, non-primitive method, parameters and class members, but not local function variables
Examples

- Consider the following example illustrating a problem with sharing a resource that is mutable (value can be changed) between different threads in the system.

- See code...
  (See unprotectedCounter)
Shared Resources

Why does this happen?

Let's consider the following lines:
public void increment(){
    count++;
}

The second line in this code involves three basic operations:

- Read the value of count
- Increment that value by 1
- Assign the new value to count
Shared Resources

- Let's say a thread is preempted after he performed operations #1 (read value X) and #2 (will save X+1 but not assign it to count yet)

- If another thread tries to perform those actions, the value of count he reads at #1 is not yet updated (value X)
Shared Resources

- This means that both of these threads will assign the same value \((X+1)\) to \(\text{count}\).

- Because of this, a lot of increment operations can be lost.
This example demonstrates the problematic nature of sharing resources between threads.

In order to take care of this problem, we need to handle synchronization in a way that doesn't allow the threads to access the problematic parts of our code (the increment part) at the same time.
Shared Resources

MULTITHREADING

THREADS ARE NOT GOING TO SYNCHRONIZE THEMSELVES
Few Design Approaches Towards Avoiding Safety Problems: Thread Confinement

- If a variable/resource is accessed only by one thread, no synchronization is required

- You may design your program such that certain variables are confined to one thread
Examples

See code... (ThreadConfinedExample)
Few Design Approaches Towards Avoiding Safety Problems: Thread Confinement

- In the following example, everything in the `createCar()` method is confined, and so the method is safe to use in a multi-threaded situation.

- However, the `createCar(Engine e)` method gets an Engine as a parameter, and as a result might not be thread safe.
  - The `e` is not thread confined - you don't know anything about the engine, and who else can access it beside your thread.
Few Design Approaches Towards Avoiding Safety Problems: Immutability

- An immutable object's state cannot be changed after construction

- Thus, it can be safely used as a read-only object by multiple threads

- As an example, the String and Integer classes in Java are immutable
Few Design Approaches Towards Avoiding Safety Problems: Immutability

- For an object to be immutable, it is recommended that:
  - All of its fields will be final
  - The types of the object field will be either primitive types or references to immutable objects

- The object is immutable only after it has been safely published
  - Meaning that reference to this hasn't escaped during construction and no thread have accessed the object before the construction completed
Few Design Approaches Towards Avoiding Safety Problems: Immutability

- **Definition:** The `final` keyword (similar to `const` in C/C++):
  - `final` fields can't be modified, but objects they refer to (internal objects) can be modified if they are mutable

- It is good practice to make all fields `final` unless they need to be mutable
Few Design Approaches Towards Avoiding Safety Problems: Immutability

- It is not always possible to make all your fields immutable
  - For example, you want to hold a list of things, but your collection class isn't immutable
  - In such cases, you can still get the immutability effect if you follow the following rules (next slide)
Few Design Approaches Towards Avoiding Safety Problems: Immutability

A class will be immutable if all of the following are true:

1. All of its fields are final
2. The class is declared final
3. The this reference is not allowed to escape during construction
Few Design Approaches Towards Avoiding Safety Problems: Immutability

4. Any fields that contain references to mutable objects, such as arrays, collections, or mutable classes like Date:
   ▶ Are private
   ▶ Are never returned or otherwise exposed to callers
   ▶ Are the only reference to the objects that they reference
   ▶ Do not change the state of the referenced objects after construction
Examples

▶ See code... (ThreeStooges)
Callback Functions
Callback Functions

- A callback function is a function that is passed to another function as a parameter, and can be activated there.
  - For example, a function F2 can receive a function F1 as a parameter.
  - F2 can activate F1 from inside its scope.

- In Java, we can simulate this behavior by using interfaces.
Examples

- See code...(Simulator)
Anonymous classes (and lambdas) can capture variables with some restrictions:

1. An anonymous class has access to the members of its enclosing class.
2. An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.

For example: comments (1) and (3) in the code won't compile.
3. A declaration of a variable in an anonymous class shadows any other declarations in the enclosing scope that have the same name

- In lambdas, this is only true for the enclosing class
- For example: comment (2) in the code will shadow the class variable "id"
Callback Functions

- An effectively final variable is a variable whose value is never changed after it is initialized.

- The reason why anonymous classes and lambdas can only access final and effectively final variables in the enclosing scope.
  - When you create an instance of an anonymous class, any variables which are used within that class have their values copied in via the autogenerated constructor.
  - This way the compiler doesn’t have to create extra types to manage these local variables.
The Singleton
In some cases, you'll want to have exactly one instance of a class in your program

For example: window managers, user-made loggers, and filesystems

Typically, those types of objects (which are known as singletons) are accessed throughout a software system, and therefore require a global point of access
The Singleton

- The **Singleton design pattern** addresses these concerns

- With the Singleton design pattern you can ensure that only one instance of a class is created and that a global point of access to the object is provided
Examples

- See code... (ClassicSingleton)
The Singleton

- In a single threaded application, this Singleton implementation works

- Let's assume that we are running a multi-threaded application

- Let's have a look at the following lines:

  ```java
  if(instance == null) {
    instance = new ClassicSingleton();
  }
  ```
The Singleton

- If a thread is preempted at Line 2 before the assignment is made, the instance member variable will still be null, and another thread can subsequently enter the if block.

- In that case, two distinct singleton instances will be created.

- In the next practical session, we'll explain how to solve this problem.
The Most Useful Policies For Using & Sharing Objects In A Concurrent Program

- **Thread-confined**
  - A thread-confined object is owned exclusively by and confined to one thread, and can be modified only by the thread that owns it
The Most Useful Policies For Using & Sharing Objects In A Concurrent Program

- **Shared read-only**
  - A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread
  - Shared read-only objects are immutable
The Most Useful Policies For Using & Sharing Objects In A Concurrent Program

- **Shared thread-safe**
  - A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization