Concurrent Object-Oriented Programming (Part One)

Concurrent and Parallel Programming

14 marzo 2022

Shared memory and variables

- We know that shared and mutable memory regions represent dangers in multi-threaded programs. Synchronization tools have to be used for correct access.
- But, how this consideration impact on the source code of an object oriented program?
- At the code level, which are the types of variables that can be shared between threads and what kind of risk are associated?

SUPSI

Dipartimento tecnologie innovative

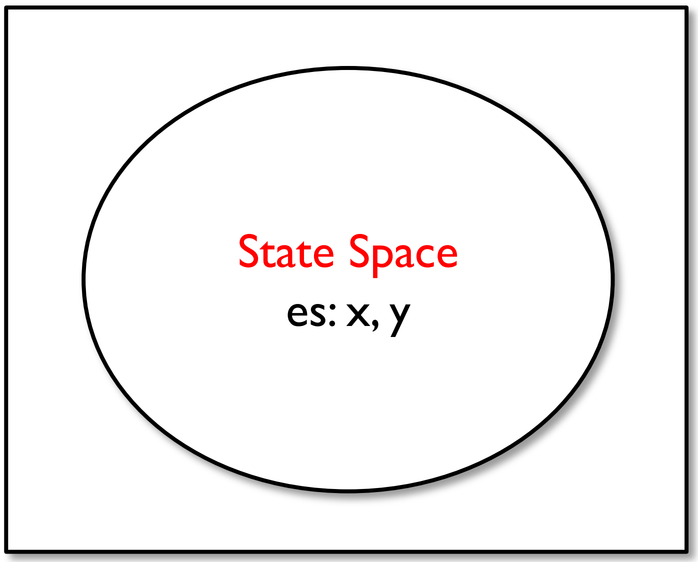
Variables and shared memory

- Local variables and parameters: no problem. Are on the stack.
- Final static variables: no problem. Are constants.
- Static (non final) variables: are global variables at the class level, always at risk of being shared. Are considered the nightmare of concurrent programming. Should be avoided.
- Primitive type fields: are shared when inside shared objects. Might be problematic.
- Reference type fields: behave like primitive type fields. When shared, the referenced array/object is also shared. Warning: an array/object can also be shared by copying the reference.

State of an object

- The state of an object is composed of the values of its fields:
  - If they are all primitive type fields, their values represent the entire state of the object.
  - If some fields are references to other objects, the state may include the values of the fields of the referenced object too.

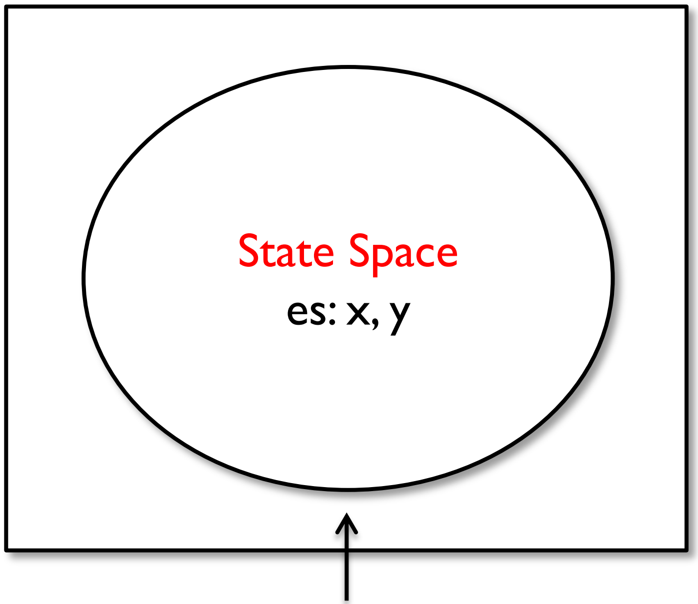
State space - review



Every object has its own state-space. It's the space of all the consistent states in which the object can be.

The state-space is delimited by the invariants: rules that have to be respected by the object to avoid falling in an inconsistent state.

Invariants - review



Invariant

example:

0 <= x < 1000 &&

0 <= y < 10 &&

x * y != 5000

Invariants represent the limits of the values that can be assumed by the variables (both static and instance).

Invariants have to be established by the constructors and have to be preserved by the methods.

Example

```java
public class NumberRange {
    // INVARIANT: lower <= upper
    private volatile int lower = 0;
    private volatile int upper = 0;

    public void setLower(int i) {
        if (i > upper)
            throw new IllegalArgumentException("can't set lower to " + i + " > upper");
        lower = i;
    }

    public void setUpper(int i) {
        if (i < lower)
            throw new IllegalArgumentException("can't set upper to " + i + " < lower");
        upper = i;
    }

    public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
```

State-space and invariants

- There are invariants that constrain more than one state variable at the same time. For example, it is the case of dependent variables in compound actions.

- The smaller the state-space, the easier it is to manage the state of the object.
- As we will see later, it is possible to use the “final” keyword to reduce the state-space of an object.

Example

```java
public class NumberRange {
    // INVARIANT: lower <= upper
    private volatile int lower = 0;
    private volatile int upper = 0;

    public void setLower(int i) {
        // Warning -- unsafe check-then-act
        if (i > upper)
            throw new IllegalArgumentException("can't set lower to " + i + " > upper");
        lower = i;
    }

    public void setUpper(int i) {
        // Warning -- unsafe check-then-act
        if (i < lower)
            throw new IllegalArgumentException("can't set upper to " + i + " < lower");
        upper = i;
    }

    public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
```

SUPSI

Dipartimento tecnologie innovative

Example

```java
public class NumberRange {
    // INVARIANT: lower <= upper
    private volatile int lower = 0;
    private volatile int upper = 0;

    public void setLower(int i) {
        // Warning -- unsafe check-then-act
        if (i > upper)
            throw new IllegalArgumentException("can't set lower to " + i + " > upper");
        lower = i;
    }

    public void setUpper(int i) {
        // Warning -- unsafe check-then-act
        if (i < lower)
            throw new IllegalArgumentException("can't set upper to " + i + " < lower");
        upper = i;
    }

    public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
```

There's not enough protection!

Concurrency and objects

- In your opinion, when and how should the state of an object be protected in a multi-threaded application?

- Which approach do you propose to use?

Thread-safe classes and objects

- One of the most frequently used approaches is to implement and use thread-safe classes and objects.
- An object is said to be thread-safe if it can be accessed from multiple threads at the same time:
  - independently from the execution order (interleaving) of the operations executed by the threads.
  - without using synchronization tools (e.g. locks) in the client code that uses the object.

Stateless objects

- Stateless objects are always thread-safe.
- Possible state exists only in parameters and local variables of the methods, which live in the thread stack.

Example (1/2)

```java
class StatelessObject {
    public double doTheWork(final double r) {
        final double volume = (4.0 / 3.0) * Math.PI * r * r * r;
        System.out.println("Volume for r: " + r + " is = " + volume);
        return volume;
    }
}

class Runner implements Runnable {
    private final StatelessObject statelessObject;
    private final int r;

    public Runner(final StatelessObject statelessObject, final int r) {
        this.statelessObject = statelessObject;
        this.r = r;
    }

    @Override
    public void run() {
        statelessObject.doTheWork(r);
    }
}
```

Example (2/2)

```
public class TestStatelessObject {
    static final ArrayList<Thread> threads = new ArrayList<Thread>();
    static final StatelessObject statelessObject = new StatelessObject();

    public static void main(final String[] args) {
        for (int i = 0; i < 10; i++)
            threads.add(new Thread(new Runner(statelessObject, i)));

        try {
            for (final Thread t : threads)
                t.start();
            for (final Thread t : threads)
                t.join();
        } catch (final InterruptedException e) {
            return;
        }
    }
}
```

Statefull objects

- For statefull objects, thread-safety can be obtained in the following 3 ways:
  - By transforming them into immutable objects. Immutable objects are always thread-safe.
  - By avoiding to share the object state, i.e. by applying confinement techniques.
  - By using synchronization tools (volatile variables, atomic variables, locks, ...) inside the object.

Non-thread-safe objects

- A thread-safe program does NOT mandatorily require to be composed of just thread-safe objects.
- A thread-safe program can also contain non-thread-safe objects.
- For example, it might be the case of objects always accessed from just one of the threads; or objects used with synchronization (e.g. locks) in the calling client code.

Development of thread-safe classes

- Encapsulation and data hiding techniques are very helpful, they allow to isolate and hide the state variables of the object.
- Encapsulation helps assessing whether a class is thread-safe or not, without analyzing the whole program.

Development of thread-safe classes

- The process of designing a thread-safe class includes:
  - Identification of the variables that represent the state of the object.
  - Identification of the invariants that limit the state-space.
  - Definition of rules called synchronization policies for correctly managing concurrent accesses to the state variables.

Encapsulation and synchronization

- The constraints imposed by the invariants generate synchronization requirements. Operations that may cause invalid transitions or other types of errors must be made atomic.

- On the other hand, if there are no constraints, encapsulation and synchronization requirements can be relaxed to obtain greater flexibility and better performances.

Synchronization policy

- The synchronization policy defines how an object coordinates the multi-threaded access to its state, by a combination of:
  - Immutability
  - Confinement
  - Locking and other synchronization tools

- Developing a thread-safe class means ensuring that its invariants are not violated under concurrent accesses.

Read-only shared state

- In case of shared state that is read-only, there's no need for synchronization.
- Values are never changed, therefore there's no risk for visibility problems (registers/caches with inconsistent content) or race-conditions.

- In multi-threaded object-oriented programs, it is possible to take advantage of this feature!
- The Java language has specific support for read-only state variables by means of the "final" keyword.

Final fields - review

- A final field can only be initialized once, either by default initialization or by an assignment in the constructor (or static initializer for static fields).
- Each final field must be initialized at the latest before the end of the constructor or static initializer of the class in which is declared.
- ATTENTION: for reference types, it's the reference itself that is final. Not the referenced object!

Initialization safety

- The Java memory model provides specific support for the initialization safety of final fields.
- Without initialization safety, there could be the risk of accessing objects still under construction.
- Thanks to initialization safety, it is possible to access shared (read-only) fields even without synchronization.

- ATTENTION: if a final field is a reference to a mutable object, synchronization is then required (when accessing the mutable object).

Immutable objects

- If the mutability of an object is fully avoided, the result is an immutable object.
- An object is immutable if it is not possible to modify its state after the object has been constructed.
- Final fields are used to implement immutable objects. Thanks to initialization safety, it is possible to access shared immutable objects without synchronization.
- However, to develop immutable objects it is not sufficient to just set all fields to final because of the potential presence of final references to mutable objects.

Immutable objects

To obtain an immutable object:

- Specify all fields as private and final.
- Don't let subclasses override methods. The easiest way is to declare the class or its methods as final. A more sophisticated approach is to declare the constructor as private and manage the creation of objects with factory methods.
- If the fields contain references to mutable objects, prevent these objects from being modified:
  - No code in the methods that modify the contained mutable objects.
  - Never share references to mutable objects. Always make protection copies. Use the copy-on-write approach.

Immutable objects: example

```
public final class Planet {

    /** Final primitive data is always immutable. */
    private final double fMass;

    /** An immutable object field. */
    private final String fName;

    /** A mutable object field. */
    private final Date fDateOfDiscovery;

    public Planet(final double aMass, final String aName, final long discoveryTime) {
        fMass = aMass;
        fName = aName;
        fDateOfDiscovery = new Date(discoveryTime);
    }

     ...
```

Immutable objects: example

```
    ...

    public double getMass() {
        return fMass;
    }

    // String is immutable and cannot be changed.
    public String getName() {
        return fName;
    }

    // Returns a defensive copy of the field.
    public Date getDateOfDiscovery() {
        return new Date(fDateOfDiscovery.getTime());
    }

    public Planet rediscoverPlanet(final long newDiscoveryTime) {
        return new Planet(fMass, fName, newDiscoveryTime);
    }
}
```

copy-on-write

Confinement techniques

- In addition to the use of synchronization tools, several specific techniques can be exploited for the development of multi-threaded applications.
- These techniques are very effective and should always be taken into consideration when developing a concurrent program.
- Many of these techniques have a strong impact on the architectural design of the application. It is advised to apply them already in early design stages to avoid the need of significant refactoring.

Confinement techniques

- As discussed, in a multi-threaded application it is required to correctly manage the state that is shared and mutable.
- But if there is state that is not shared and mutable, no protection is needed!
- Consequently, if it is possible to avoid sharing variables and fields, things are easier.
- For this reason the following techniques can be used:
  - Thread Confinement
  - Stack Confinement
  - Instance Confinement

Thread Confinement

- Thread confinement is one of the easiest ways to achieve thread-safety.
- If a field or referenced object is confined to the use of a single thread, the execution from the point of view of that field/object can be compared to the execution of a mono-threaded application, ... therefore, no risk of concurrency problems has to be considered.

Thread Confinement

- During the confinement time, it's important to ensure that the objects have no chance to escape (undesired sharing of objects).
- For the confined objects, it is important to ensure that they are also created by the same thread.
- The confinement might also be temporary. Later, it could be interrupted by sharing the values or the objects with a synchronization tool (e.g. a lock).
- Thread confinement is a design element of the program. To obtain it, there's NO language-specific solution. It must implemented with discipline.

SUPSI

Dipartimento tecnologie innovative

Example

```
class Counter implements Runnable {
    private int value = 0;

    @Override
    public void run() {
        while (true) {
            value++;
            if (value % 10000 == 0) System.out.println(value);
            if (value == 100000) return;
        }
    }
}

public class CounterExample {
    static final ArrayList<Thread> threads = new ArrayList<>();

    public static void main(final String[] args) {
        for (int i = 0; i < 10; i++)
            threads.add(new Thread(new Counter()));
        for (final Thread t : threads)
             t.start();
        try {
            for (final Thread t : threads)
                t.join();
        } catch (final InterruptedException e) {  return;  }
    }
}
```

Stack Confinement

- Stack confinement is a special version of thread confinement, where a value or referenced object can only be accessed through method parameters or local variables.
- Parameters and local variables exist on the thread stack, which is not accessible from other threads, therefore are automatically confined to the running thread.
- Stack confinement is easier and more intuitive to implement than thread confinement and results in less fragile code.
- Parameters and local variables of primitive type are always stack confined. For reference types, it is required to ensure that the reference never escapes the thread.

Example

```java
public class StackConfinementExample {
    public int computeOperation(final int value1, final int value2) {
        int y = value1;
        for (int i = 0; i < 10; i++) {
            y *= value2;
        }

        if (y < 0)
            return 0;
        return y;
    }
}
```

Instance confinement

- One of the most effective techniques is to confine the values and referenced objects that compose the state of an object directly in its instance.
- Thread-safety can then be obtained by ensuring that:
  - the instance (container object) is accessed only by a single thread (thread confinement), or
  - all accesses to the instance are protected by synchronization means (i.e. it is a thread-safe object).
- Encapsulation is of strong help to achieve instance confinement. All code paths that access an encapsulated value or object are fully under control of the class.

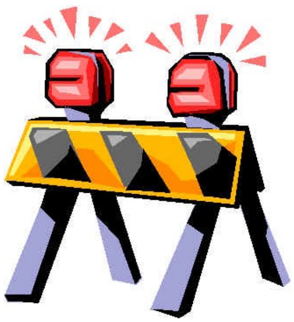
Example

```
public class PersonSet {
    private final Set<Person> mySet = new HashSet<>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }

    interface Person {
        // ...
    }
}
```



Pattern for thread-safe objects

- The concept of instance confinement introduces a possible design pattern:

encapsulate all mutable states in an object and protect them from concurrent accesses by using synchronized methods:

  - all the mutable state of the object is encapsulated
  - all the mutable state of the object is protected by the intrinsic lock of the object
- This design pattern is simple but often not very effective in terms of performance.
- For better performances other types of locks (such as RW-locks), or for example atomic variables, might be used.

Example

```java
public class SynchronizationPatternExample {
    private String name = null;
    private boolean nameModified = false;

    public synchronized String getName() {
        return name;
    }

    public synchronized boolean isNameModified() {
        return nameModified;
    }

    public synchronized void unsetNameModified() {
        this.nameModified = false;
    }

    public synchronized void setName(final String name) {
        this.name = name;
        nameModified = true;
    }
}
```

One writer - many readers

- Another design pattern that profits from the confinement techniques, but allows data to be shared between threads is called one writer - many readers.
- In this solution, just a single thread is allowed to modify (write) the shared state, while other threads have only rights to read it. The consequence is that no read-modify-write race-condition might ever happen.
- This pattern, can be implemented with just one volatile variables for sharing the state. Keep in mind that the visibility effect of volatile variables also extends to the other shared variables (if things are done correctly).

One writer - many readers

- To increase the effectiveness of this approach, it is recommended that the only thread that modifies the state uses thread/stack confinement until the instant of sharing.
- For the threads that read the state, to avoid check-then-act race-conditions, it is also recommended to switch to thread/stack confinement immediately after the initial read of the shared state.

Example

It has to be ensure that only one thread is allowed to invoke this method!

```
public class OneWriterExample {
    private volatile int x = 1;

    public void modifyValue(final int value) {
        int y = x;
        y *= value;
        if (y < 0)
            return;
        x = y;
    }

    public int readValue() {
        int tempX = x;
        if (tempX > 1000)
            return 1000;
        else
            return tempX;
    }
}
```

Summary of topics

- Shared memory and types of variables
- Objects: state-space and invariants
- Thread-safe objects and classes
- The added value of encapsulation
- Final fields and immutable objects
- Confinement techniques
- Design pattern for thread-safe objects
- The one-writer many-readers pattern