

# Example04

## Table of contents

- [Atomic references](#)
  1. [`AtomicReference.java`](#)
- [Example04](#)
  1. [`Example04.java`](#)

## Atomic references

Alcune volte la mutua esclusione ci basta per risolvere tutti i problemi di sincronizzazione che dobbiamo affrontare. Con le **reference atomiche** verifichiamo il valore di una reference in mutua esclusione, oppure facciamo letture/scritture sul reference garantendo la mutua esclusione.

Operazioni fornite sono:

- **deference** del reference;
- **assign** del reference;
- operazione **atomic deference & reference**, permettere a chi chiama un metodo di leggere un valore e cambiarlo a un altro
  - leggo e poi scrivo
  - scrivo e poi leggo;
- interfaccia funzionale  $f(R)$  che ritorna  $R$ ;
- interfaccia funzionale  $f(R)$  che ritorna  $f(R)$ .

### AtomicReference.java

Costruiamo una classe con argomento generico `<T>`, un oggetto su cui faremo le sezioni critiche e poi un valore reference che prendiamo dall'esterno.

```
package it.unipr.informatica.concurrent.atomic;
import java.util.function.UnaryOperator;

public class AtomicReference<T> {
    private T value;
    private Object lock;

    // costruttore che fa riferimento ad un altro
    public AtomicReference() {
        this(null);
    }

    public AtomicReference(T value) {
        this.value = value;
        this.lock = new Object();
    }

    // ...
}
```

La `get()` costruisce la sezione critica e ritorna il valore.  
Va a fare la deference.

```
// ...
// equivalente all'operatore di deference
public T get() {
```

```

        synchronized(lock) {
            return value;
        }
    }
    // ...

```

La `set()` prende un riferimento a un valore di tipo `T`, blocca la sezione critica entra e scrive.

```

    // ...
    public void set(T value) {
        synchronized(lock) {
            this.value = value;
        }
    }
    // ...

```

Prende la sezione critica su `lock`, sovrascrive il valore di `value` e ritorna il valore vecchio prima dell'aggiornamento.

```

    // ...
    public T getAndSet(T value) {
        synchronized(lock) {
            result = this.value;
            this.value = value;
            return result;
        }
    }
    // ...

```

`UnaryOperator` ha interfaccia con un metodo con argomento tipo `T` ritornante tipo `T`. Interfaccia funzionale standard con un unico metodo: `apply()`.

Prima leggiamo il valore di `value` e lo ritorniamo quando è il momento.

```

    // ...
    public T getAndUpdate(UnaryOperator<T> update) {
        synchronized(lock) {
            T result = value;
            this.value = update.apply(value);
            return result;
        }
    }
    // ...

```

Prima applica la funzione  $f()$  e poi ritorna il risultato  $R$ .

```

    // ...
    public T updateAndGet(UnaryOperator<T> update) {
        synchronized(lock) {
            T result = update.apply(value);
            this.value = result;
            return result;
        }
    }
}

```

## Example04

Example04.java

```

package it.unipr.informatica.examples;

public class Example04 {
    private void go() {
        AtomicReference<Integer> counter = new AtomicReference<>(1);
        int i = counter.get();
        Incrementer incrementer = new Incrementer();

        while ((i = counter.get()) ≤ 10) {
            System.out.println(i);
            i = counter.updateAndGet(incrementer);
        }

        counter.set(1);
        i = counter.get();

        UnaryOperator<Integer> operator
            = new UnaryOperator<Integer>() {
            @Override
            public Integer apply(Integer value) {
                return value+1;
            }
        };

        while(i ≤ 20) {
            System.out.println(i);
            i = counter.updateAndGet(operator);
        }

        while(i ≤ 30) {
            System.out.println(i);
            i = counter.updateAndGet((x) → {
                return x+1;
            });
        }

        while(i ≤ 40) {
            System.out.println(i);
            i = counter.updateAndGet((x) → x+1);
        }
    }

    public static void main(String[] args) {
        new Example04().go();
    }

    private static class Incrementer
        implements UnaryOperator<Integer> {
        @Override
        public Integer apply(Integer value) {
            return value+1;
        }
    }
}

```

