Reference atomiche, abbiamo chiarito che il problema di base della sincronizzazione è la mutua esclusione, alle volte questa è sufficiente, l'idea è di mettere a disposizione in modo semplice di estrarre il valore di una reference in mutua esclusione , oppure fare letture e scritture in mutua esclusione.

L'idea di base è quello che un reference in java sono cose che vengono lette e scritte.
poi vogliamo che vengano fatte della operazione atomiche:

-Leggere una reference e sostituisce il valore con uno nuovo ma ritorna quello vecchio
getAndSet()

Costruiamo una classe AtomicReference con argomento generico T, avremo un oggetto qualsiasi chiamato Lock per la gestione sincronizzata delle sezioni critiche.

- *get()* costruisce la sezione critica sull'oggetto lock e poi ritorna il valore.
- *set()* prende come argomento una reference a un tipo T, poi costruisce la sezione critica entra e scrive all'interno della reference.
- *getAndSet()* utilissima quando dobbiamo verificare un valore booleano e se questo è falso va "alzato" a true se no bisogna fare altre "cose", quindi quello che fa è fare un operazione di set() e prende la sezione critica su lock, poi sovrascrive il valore di value(cambia lo stato dell'oggetto) e il risultato che viene ritornato è il valore vecchio di value.
- Per le *Update* andiamo a considerare l'interfaccia UnaryOperator che è un'interfaccia con un unico metodo, e vediamo che è un metodo che ha come argomento un valore di tipo T e ritorna un valore di tipo T.
- *getAndUpdate()* prima leggiamo il valore corrente di value, che lo ritorneremo quando sarà il momento, poi usiamo la funzione update, il valore risultato di questa operazione lo mettiamo in value.
- *updateAndGet()* prima usiamo la funzione update poi ritorneremo il valore aggiornato.

```
package it.unipr.informatica.concurrent.atomic;

import java.util.function.UnaryOperator;

public class AtomicReference<T> {
	private T value;

	private Object lock;

	private AtomicReference() {
		this(null);
	}
```

```
        public AtomicReference(T value) {
                this.value =value;
                this.lock = new Object();
        }
        public T get() {
                synchronized(lock) {
                        return value;
                }
        }
        public void set(T value) {
                synchronized (lock) {
                        this.value = value;
                }
        }
        public T getAndSet(T value) {
                synchronized (lock) {
                        T result = this.value;
                        this.value = value;
                        return result;
                }
        }

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

                }
        }
}
```

EXAMPLE04.JAVA

```
package it.unipr.informatica.examples;

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

```java
public class Example04 {
	private void go() {
		AtomicReference<Integer> counter = new AtomicReference<Integer>(1);

		int i = counter.get();

		Incrementer incrementer = new Incrementer();

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

		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;
		}
	}
```

```
}
```

Un'altra astrazione molto utile è il Pools of Resources, risorse condivise nel flusso di esecuzione, una possibilità è garantire l'accesso condiviso alle singole risorse ma .....
abbiamo 10 risorse e 20 flussi ogni flusso richiede una risorsa e l'acquisisce e poi la rilascia a questo punto nessuno si blocca se ci sono risorse disponibili, se invece non sono disponibili si blocca.
I thread si esecuzione sono degli oggetti che vengono utilizzati dal flusso di esecuzione per fare i loro calcoli, ma non sempre si hanno i thread che si vuole per questo andiamo a fare un Thread Pools.

Quello che vediamo è il cosiddetto Esecutore a cui andremo ad affidare dei Task, e lui deciderà se esguire o meno i task nel caso crea una coda.
Esecutore1 diretto
Esecutore2 crea un nuovo thread ogni volta che gli arriva una richiesta di un nuovo Task
Esecutore3 crea un pool di k thread è il più importante.