Un *thread* è un flusso di esecuzione, e può essere pensato come un processo (sebbene tecnicamente non lo sia).

Normalmente esiste un solo thread (il *Main Thread*), la cui esecuzione inizia quando si lancia la JVM su una classe eseguibile: il thread esegue in sequenza le istruzioni del metodo main; durante il thread verranno creati oggetti, e invocati metodi che potranno a loro volta creare oggetti e invocare metodi. Il thread termina quando il metodo main termina.

Per creare un nuovo thread, occorre in primo luogo scrivere una classe che estenda `Thread`; tale classe deve fornire tipicamente:

- un costruttore che inizializzi le variabili che servono al thread;
- un metodo

  `public static void run()`

  che viene eseguito quando il thread viene lanciato.

Una volta costruito un oggetto di questa classe, il thread viene lanciato invocando il metodo `start()`. Quando il metodo `run()` del thread termina, il thread termina.

```java
/** Uno Stampatoreè  un thread che stampa un dato messaggio un dato
 *  numero di volte. Fra un messaggio e l'altro fa una pausa di un
 *  dato numero di millisecondi. */

public class Stampatore extends Thread {

    private String msg;
    private int times, millis;

    public Stampatore( String msg, int times, int millis ) {
        super();
        this.msg = msg;
        this.times = times;
        this.millis = millis;
    }

    public void run() {
        for ( int i = 0; i < times; i++ ) {
            System.out.println( msg );
            try {
                Thread.currentThread().sleep( millis );
            } catch ( InterruptedException e ) {
                return;
            }
        }
    }
}
```

Nell'esempio abbiamo visto due metodi della classe `Thread`:

- il metodo statico `Thread.currentThread()` che restituisce il thread corrente (quello che ha eseguito la chiamata);
- il metodo `sleep( x )` che interrompe il thread per circa `x` millisecondi; questo metodo, come altri della classe `Thread`, può sollevare una `InterruptedException` nel caso in cui il `Thread` venga interrotto (per esempio, perché il `Thread` che lo ha generato è stato interrotto — il Main Thread si può interrompere premendo Ctrl-C).

```
public class StampatoreTest1 {

    public static void main( String arg[] ) {
        Stampatore stampatore =
            new Stampatore( "Nel mezzo", 5, 500 );
        stampatore.start();
    }

}
```

```
Nel mezzo
Nel mezzo
Nel mezzo
Nel mezzo
Nel mezzo
```

```
public class StampatoreTest2 {

    public static void main( String arg[] ) {
        Stampatore stampatore0 =
            new Stampatore( "Nel mezzo", 5, 300);
        Stampatore stampatore1 =
            new Stampatore( "del cammin", 5, 500 );
        Stampatore stampatore2 =
            new Stampatore( "di nostra vita", 5, 200 );
        stampatore0.start();
        stampatore1.start();
        stampatore2.start();
    }

}
```

```
Nel mezzo
del cammin
di nostra vita
di nostra vita
Nel mezzo
di nostra vita
del cammin
Nel mezzo
di nostra vita
di nostra vita
...
```

```
public class Contatore {

    private int conta;

    public Contatore() {
        conta = 0;
    }

    public void inc() {
        int incrementato;
        incrementato = conta + 1;
        conta = incrementato;
    }

    public void dec() {
        int decrementato;
        decrementato = conta -1;
        conta = decrementato;
    }

    public int valore() {
        return conta;
    }
}
```

```
public class Incrementatore extends Thread {

    private Contatore c;
    private int volte;

    public Incrementatore( Contatore c, int volte ) {
        this.c = c;
        this.volte = volte;
    }

    public void run() {
        for ( int i = 0; i < volte; i++ )
            c.inc();
    }
}
```

```
public class Decrementatore extends Thread {

    private Contatore c;
    private int volte;

    public Decrementatore( Contatore c, int volte ) {
        this.c = c;
        this.volte = volte;
    }

    public void run() {
        for ( int i = 0; i < volte; i++ )
            c.dec();
    }
}
```

```
public class IncDecTest {

    public static void main( String arg[] ) {
        Contatore c = new Contatore();
        Incrementatore inc = new Incrementatore( c, 10000000 );
        Decrementatore dec = new Decrementatore( c, 10000000 );
        inc.start();
        dec.start();
        try {
            inc.join();
            dec.join();
            System.out.println( c.valore() );
        } catch ( InterruptedException e ) {}

    }

}
```

```
lithium[14:54:37]>java IncDecTest
313379
lithium[14:54:45]>java IncDecTest
626327
lithium[14:54:46]>java IncDecTest
623861
lithium[14:54:49]>java IncDecTest
-888
lithium[14:54:52]>java IncDecTest
-312578
lithium[14:54:55>java IncDecTest
0
```

Ad ogni oggetto è associato un *lock*: il lock di un oggetto può in ciascun istante essere libero oppure assegnato a un thread.

Per richiedere il lock sull'oggetto `t`:

```
synchronized( t ) {
    .... codice ....
    }
```

Se il lock non è disponibile, ci si blocca sul synchronized in attesa che si liberi il lock e che venga assegnato a noi. Il lock verrà rilasciato alla fine del blocco synchronized.

```
public class Incrementatore extends Thread {
    ...
    public void run() {
        for ( int i = 0; i < volte; i++ )
            synchronized( c ) {
                c.inc();
            }
    ...
}
```

Meglio ancora: lasciamo la sincronizzazione nel codice del contatore.

```
public void dec() {
    int decrementato;
    synchronized( this ) {
      decrementato = conta -1;
      conta = decrementato;
    }
}
```

o, equivalentemente:

```
public synchronized void dec() {
    int decrementato;
    decrementato = conta -1;
    conta = decrementato;
}
```

Gli stati di un thread:

- RUNNING($S$): in esecuzione; possiede l'insieme $S$ di lock;
- READY($S, T$): pronto per l'esecuzione: possiede l'insieme $S$ di lock, ed è in attesa dell'insieme $T$ di lock ($S \cap T = \emptyset$);
- WAITING($S, t$): in attesa sull'oggetto $t$; prima possedeva l'insieme $S$ di lock ($t \in S$);
- NOTIFIED($S, t$): era nello stato WAITING($S, t$) e ha ricevuto la notifica.

$$\text{RUNNING}(S) \to \text{READY}(S, \emptyset)$$

era running e ha perso la CPU.

$$\text{READY}(S, \emptyset) \to \text{RUNNING}(S)$$

era pronto, e non aspettava nessun lock; ha guadagnato la CPU.

$$\text{RUNNING}(S) \to \text{READY}(S, \{t\}) \text{ con } t \notin S$$

era in esecuzione ed è entrato in un blocco `synchronized(t)`.

$$\text{RUNNING}(S) \to \text{RUNNING}(S \setminus \{t\}) \text{ con } t \in S$$

era in esecuzione ed è uscito da un blocco `synchronized(t)`.

$$\text{READY}(S, T) \to \text{READY}(S \cup \{t\}, T \setminus \{t\}) \text{ con } t \in T$$

era in attesa di un lock su $t$: si è liberato, e nella competizione il lock è stato assegnato a lui.

$$\text{RUNNING}(S) \to \text{WAITING}(S, t) \text{ con } t \in S$$

era running e si è messo in attesa su $t$ (cioè `t.wait()`). Libera il lock su $t$, ma continua a possedere i lock su $S \setminus t$.

$$\text{WAITING}(S, t) \to \text{NOTIFIED}(S, t)$$

era in attesa su $t$ e qualcuno ha invocato una `t.notify()`: fra tutti quelli in attesa su $t$ è stato scelto lui.

$$\text{NOTIFIED}(S, t) \to \text{READY}(S, \emptyset)$$

il lock su $t$ si è liberato ed è stato acquisito da lui.