PROGETTAZIONE ORIENTATA AD OGGETTI

Progettare applicazioni orientate ad oggetti significa dover passare per diverse fasi che sono comuni anche a molti altri software. Per questo, ci soffermiamo un attimo ad analizzare il **ciclo di vita di un software**, cioè tutte le attività dall'analisi iniziale fino all'obsolescenza. Le diverse fasi sono:

- **analisi**. Si redige un documento che contiene un'analisi dei requisiti. Descrive cosa farà il software quando verrà terminato, allegando anche un manuale utente per l'utilizzo e dei criteri per la misurazione delle prestazioni (dati in ingresso, fabbisogno di spazio sul disco).
- **progettazione**. Si pianifica come implementare il sistema, decidendo metodi, classi e facendo dei diagrammi per le relazioni.
- **implementazione**. Si scrive e si compila il codice, usando le classi e i metodi scoperti durante la progettazione.
- **collaudo**. Si eseguono dei test per vedere se il programma funziona correttamente e si redige un documento riassuntivo.
- **installazione**. Gli utenti installano il programma e lo utilizzano.

In base a questi criteri, possiamo avere diversi tipi di organizzazione. Possiamo usare quella **a cascata**, che è un semplice modello sequenziale che non è molto efficiente. Oppure, quella **a spirale**, quando l'intero processo è diviso in multiple fasi che si concentrano sulla realizzazione di prototipi.
La prima cosa da fare per una buona progettazione orientata ad oggetti, è la **scelta delle classi**. Una classe deve poter rappresentare entità concrete o astratte. Il nome viene definito guardando i nomi usati nella descrizione e, invece, per i metodi, si utilizzano i verbi. Non tutte le classi possono essere scoperte in fase di analisi. Per ogni metodo è necessario stabilire la classe in cui collocarlo. Si fa compilando le schede **CRC**, ***Classe-Responsabilità-Collaboratori***, che descrive una classe, le sue responsabilità e i suoi collaboratori. Le classi possono essere in

relazione tra di loro. Abbiamo già parlato di **ereditarietà**, cioè la relazione tra una superclasse e una sottoclasse. Però, per esempio, possiamo anche avere relazioni di tipo **associazione-aggregazione**, cioè quando gli oggetti di una classe contengono riferimenti ad oggetti di un'altra classe. Menzioniamo anche una **dipendenza**. Si tratta di un caso particolare di quella precedentemente descritta, molto più rigida.
Per generare la **documentazione**, è possibile utilizzare `javadoc`, che genera documenti in formato HTML.

CORREZIONE DI ERRORI (DEBUGGING)

Come in un qualsiasi linguaggio di programmazione, anche Java mette a disposizione diversi metodi per l'individuazione e la correzione di errori. Uno di questi è il **Program Trace**. Sono dei messaggi che mostrano un cammino di esecuzione e stampano il suo contenuto tramite il metodo `printStackTrace()`. Questo metodo ha diversi svantaggi. Infatti, bisogna rimuovere il messaggio una volta che il testing è completo e non ci sono errori e reinserirlo dove ci sono. In alternativa, si può quindi usare la classe `Logger` che, anziché stampare a video, scrive in memoria. Per farlo, generalmente, si appoggia su un file, tramite la classe `FileHandler`, che poi vedremo. Quando il testing è completo, il logging dei messaggi può essere disattivato tramite la classe `Level` e il suo metodo `OFF`.
Per controllare condizioni di errore è possibile anche usare le **asserzioni**. Se una determinata asserzione, contrassegnata con la parola chiave `assert`, non si verifica, allora il programma si interrompe. Per i programmi molto grandi, queste due strade non sono percorribili. Per questo, si usa un **debugger**. È fondamentalmente un programma che esegue il programma sotto collaudo, analizzando il suo comportamento. Utilizza dei **breakpoint** per arrestare e far ripartire il programma ed esegue il programma un passo alla volta (**single-step**). I debuggers, di solito, fanno parte di un IDE. Quando l'esecuzione viene arrestata si ispezionano le variabili e si riparte. Una volta terminato il programma, termina anche il debugger. Attenzione, perchè alla base di un buon debugging, c'è un concetto fondamentale. Una volta rieseguiti i test e una volta visualizzato un risultato, il debugger non potrà dirci se il programma funziona correttamente. Dovremo basarci noi sugli input forniti dal nostro test campione.

PROGRAMMAZIONE GENERICA

Abbiamo già parlato di programmazione generica, quando abbiamo definito i tipi di polimorfismo ma, adesso, ci tocca scendere di più nel dettaglio. Programmare in maniera generica significa creare costrutti che possano essere utilizzati con tipi di dati diversi. In Java, si può raggiungere questo obiettivo:

- usando l'**ereditarietà**, per esempio usando `LinkedList`, che permette di realizzare una lista generica.
- usando **variabili di tipo**, come l'`ArrayList` di Java che abbiamo visto abbondantemente.

Di particolare importanza è la realizzazione di una **classe generica**, dichiarata usando le variabili di tipo tra parentesi angolari dopo il nome della classe. Le variabili di tipo rendono il codice più sicuro e di più facile comprensione in un contesto generico. Per rendere l'idea, è impossibile aggiungere un oggetto `String` ad un `ArrayList<BankAccount>`, ma è possibile aggiungere un oggetto `String` ad un `ArrayList` che sia stato creato con l'intenzione di inserire conti bancari. Le variabili di tipo, però, non sono tipi di Java. Infatti, se `T` è un'ipotetica variabile di tipo, non troveremo mai il suo riferimento all'interno del file system, che sia `.class` o `.java`. Infatti, esse passano sotto un processo di ***type erasure***, cioè di completa rimozione dei tipi generici. Esse vengono sostituite con tipi di Java opportuni e vengono aggiunti dei casting dove è necessario per non incorrere in problemi di compilazione. Ovviamente, esistono diversi tools che permettono di risalire al bytecode iniziale del codice sorgente.
Il concetto di programmazione generica è strettamente legato alle **collezioni**, di cui poi parleremo nel dettaglio. Un primo limite è che, per memorizzare tipi primitivi, è opportuno utilizzare classi *Wrapper*, che trasformano i tipi primitivi in oggetti mettendoli con la lettera grande. Inoltre, con le collezioni abbiamo un controllo sugli oggetti abbastanza limitato. Infatti, potremo memorizzare qualsiasi oggetto e, quando accediamo ad esso, non avremo il controllo sul tipo, potendo incorrere in errori al tempo di compilazione se viene inserito un oggetto di un tipo

diverso da quello atteso. E, quindi, torniamo poi all'esempio di prima fatto con le stringhe e gli ArrayList

COLLEZIONI

La libreria di Java fornisce una serie di classi che consentono di lavorare con gruppi di oggetti, dette collezioni. Queste classi passano sotto l'egida di ***Java Collections Framework***, che è una raccolta di interfacce e classi, tra loro collegate, all'interno del pacchetto java.util. Durante il nostro percorso, abbiamo già visto un tipo di collezione, che può essere l'array. Come detto, però, gli array hanno una dimensione fissa e sono di natura statica. Così, il JCF mette a disposizione array di natura dinamica, liste, insiemi, mappe e code, che ora vedremo nel dettaglio. In cima alla gerarchia ci sono le interfacce `Collection` e `Map`. La prima contiene molti metodi utili, come `size()`, `isEmpty()`, `add()`, `remove()` e `contains()`. La seconda, invece, è molto meno usata e prendere come parametri una coppia **chiave-valore**, dove la chiave ha lo scopo di identificare il valore associato. I metodi cambiano un po', infatti troveremo il metodo `put(key, value)`, che associa il valore alla chiave, oppure `keySet()`, che restituisce l'insieme di chiavi presenti nella mappa.
Ci soffermeremo, adesso, su alcune interfacce che estendono `Collection`. Esse si dividono innanzitutto in due macrocategorie. La prima è `Set`, che descrive le funzionalità di un insieme. Quindi, non ammette duplicati e aggiunge alcuni vincoli ai metodi di `Collection`. Infatti farà l'`add()` soltanto se l'elemento non esiste all'interno dell'insieme. A sua volta, `Set` mette a disposizione altre due classi:

- `HashSet`. È l'implementazione più comune e ha la struttura di una **tabella hash**. Una tabella hash non è altro che un array di liste. Usa una **funzione hash** che, dato un elemento, restituisce un numero. Quindi l'elemento si trova in posizione `hash(i)` dell'array.
- `TreeSet`. Seconda implementazione e viene usato soprattutto quando ci serve tener conto di una lista ordinata secondo l'ordine naturale. Al suo interno usa una rappresentazione ad **albero** e le operazioni sono eseguite in tempo logaritmico rispetto al numero di elementi dell'insieme.

La seconda interfaccia è `List`. Non ci soffermeremo su una delle sue classi, cioè `ArrayList`, dato che l'abbiamo già ampiamente descritta. Le altre due classi sono:

- `Vector`. È molto simile all'`ArrayList`, dato è un **array dinamico**. Quindi tutti gli oggetti vengono memorizzati in un array e le operazioni sono tutte semplificate.
- `LinkedList`. Quando dobbiamo frequentemente inserire o rimuovere elementi da una lista, è opportuno usare questa **lista doppiamente puntata**. Ogni elemento ha un riferimento all'elemento successivo e precedente.

Una volta definiti i tipi di collezione del JCF, ci soffermiamo brevemente sull'**iteratore**. L'iteratore è un oggetto di supporto che serve per accedere agli elementi di una collezione. Esso implementa l'interfaccia `Iterator` e, poi, `Collection` contiene il metodo `iterator()` per restituirlo. L'interfaccia mette a disposizione diversi metodi utili, come `next()`, che restituisce il prossimo elemento della collezione, `hasNext()`, che verifica se c'è un elemento successivo da fornire, e `remove()`, che rimuove l'elemento restituito da `next()`. È buona norma non apportare modifiche mentre si sta usando l'iteratore.
Ovviamente, `Collections` permette anche l'implementazione dell'interfaccia `Queue`, usata per la gestione delle code.

GESTIONE DELLE ECCEZIONI

Una condizione di errore può avere molte cause. Può essere un semplice errore di programmazione, oppure qualche errore di sistema, dovuto ad un malfunzionamento delle componenti fisiche del nostro computer. Oppure errori di utilizzo, come degli input incorretti, o la scrittura su file inesistenti. Quando all'interno di un programma si presenta una situazione imprevista, bisogna saperla gestire. Java, per fare ciò, mette a disposizione le eccezioni. Si tratta di oggetti che possono essere creati e lanciati (**throw**) in determinate condizioni e che possono essere catturati (**catch**) dal codice appositamente scritto per la loro gestione. Generalmente, quando parliamo di eccezione, ci riferiamo ad un'anomalia di cui non è possibile effettuare un recupero durante l'esecuzione del programma. Creando un'eccezione, creiamo un evento che interrompe la normale esecuzione del programma. Una volta verificatasi, il metodo che la lancia, trasferisce il controllo al **gestore delle eccezioni**. Java, per rappresentare tutti i tipi di errore, mette a disposizione la classe `Throwable`, che ha due sottoclassi, che sono `Error` e `Exception`. La prima è poco utilizzata, perché si occupa di errori fatali che non possono essere gestiti dal programma. Poi, ovviamente, `Exception` ha molte altre sottoclassi che rappresentano le tipologie di errore. In generale si dividono in due macrocategorie, cioè le eccezioni **non controllate** e **controllate**. Le eccezioni non controllate si occupano di circostanze che possono essere evitate dal programmatore correggendo il programma, mentre quelle controllate prevedono circostanze che non possono essere evitate e che, quindi, si deve pronosticare il loro comportamento anomalo. Possiamo distinguere:

- `IOException`. Segnala che si è verificato qualche tipo di errore a livello di I/O, quindi quando il flusso dati viene interrotto, o il file non esiste.
- `ClassNotFoundException`. Quando la JVM tenta di caricare una classe tramite il suo nome, ma non riesce a trovare corrispondenze.
- `CloneNotSupportedException`. Caso molto raro, cioè quando il metodo `clone()` di `Object` viene chiamato per clonare un oggetto, ma non viene implementata

l'interfaccia `Cloneable`.

- `RuntimeException`. Quando ci troviamo in presenza di eccezioni che possono essere lanciate durante l'esecuzione e, quindi, sono eccezioni non controllate.

Quando parliamo di un'eccezione controllata possiamo scegliere di gestirla, quindi lanciarla tramite la parola chiave throw, oppure non gestirla e dichiarare di poterla lanciare nella dichiarazione del metodo tramite throws. Di solito, è buona norma gestire le eccezioni, per evitare esecuzioni anomale di un programma e, quindi, installare il gestore di eccezioni. Per farlo si usa l'enunciato `try`, seguito da tante clausole `catch` quante sono le eccezioni da gestire. Così facendo, noi cattureremo l'eccezione. All'interno del blocco try vengono eseguite determinate istruzioni. Se nessuna eccezione viene lanciata, allora il blocco catch viene ignorato. Una volta lanciata l'eccezione, invece, si esegue ciò che è in catch. Il lancio di un'eccezione arresta il metodo che la sta chiamando. Prima dell'arresto, però, a volte è necessario eseguire altre istruzioni. Per questo, al blocco try-catch, possiamo allegare la clausola `finally`, che indica un'istruzione che viene eseguita sempre. A volte, possono non bastare le eccezioni standard di `Throwable` e, quindi, possiamo progettarne di nuove. Per farlo sarà necessario creare una nuova classe Java ed estendere `RuntimeException`, se sono eccezioni non controllate, o `Exception`, se sono controllate. Da qui, poi, la classe potrà sovrascrivere un metodo toString(), che ci permette di scrivere il messaggio di errore che sarà stampato all'interno del catch. All'interno del try, invece, si stabilirà la condizione necessaria per lanciare l'eccezione, usando un semplice throw che abbiamo già definito prima.

I/O E FLUSSI

In Java, input e output sono definiti in termini di flussi (***stream***). I flussi possono essere di due tipi, **binari** o di **testo**. E, ogni tipo di flusso, è gestito da determinate classi, tutte all'interno di `java.io`. Vediamole:

- `InputStream`. Dichiara i metodi che servono a leggere flussi binari da una sorgente specifica, come `read()`, che legge un byte alla volta, e `close()`, che chiude il flusso. Inoltre, mette a disposizione diverse sottoclassi, come `FileInputStream`, che permette di creare oggetti di questa classe.
- `OutputStream`. Dichiara i metodi che servono a scrivere flussi binari in una destinazione specifica, come `write()`, che scrive un byte alla volta, e `close()`, che chiude il flusso di output. Inoltre, mette a disposizione diverse sottoclassi, come `PrintStream`, che aggiunge tutti i metodi per stampare vari tipi di dati, oppure `FileOutputStream`, che serve a creare oggetti di questa classe.
- `Reader`. Dichiara i metodi che servono a leggere flussi di caratteri da una sorgente specifica, come `read()`, che legge un carattere alla volta, e `close()`, che chiude il flusso. Inoltre, mette a disposizione diverse sottoclassi, come `InputStreamReader`, che converte il flusso in input binario in un flusso di caratteri. A sua volta, quest'ultima mette a disposizione un'altra sottoclasse, detta `FileReader`, che serve per creare oggetti di questa classe.
- `Writer`. Dichiara i metodi che servono a scrivere flussi di caratteri in una destinazione specifica, come `write()`, che scrive un carattere alla volta, e `close()`, che chiude il flusso. Inoltre, mette a disposizione diverse sottoclassi, come `PrintWriter`, che contiene tutti i metodi `print` e `println` di `PrintStream`.

In Java è possibile anche creare una rappresentazione astratta di un file, usufruendo della classe `File`, che mette a disposizione nuovi metodi come `delete()`, cancella il file, `renameTo()`, rinomina il file, `lenght()`, restituisce la lunghezza in byte, `exists()`, verifica se il file esiste. Per memorizzare oggetti in un flusso, dobbiamo effettuarne la **serializzazione**. Ogni oggetto riceve un numero di serie nel flusso. L'oggetto da inserire nel flusso deve essere serializzabile e,

quindi, appartenere ad una classe che implementa l'interfaccia `Serializable`. Poiché operazioni di serializzazione e deserializzazione potrebbero essere effettuate da classi diverse, è opportuno creare una costante `serialVersionUID`, che viene definito all'inizio automaticamente dall'IDE. Tutto questo per evitare problemi di incompatibilità o collisioni, cioè creare due numeri di serie uguali e, quindi, due riferimenti allo stesso oggetto.

PROGRAMMAZIONE GRAFICA

In Java, quando ci riferiamo alla programmazione grafica, parliamo di visualizzare informazioni all'interno di una **finestra**, dotata di un **titolo** e una **cornice** (***frame***). La JVM esegue ogni frame su un thread separato, cioè un flusso di esecuzione. Per creare le finestre, usiamo metodi della classe `JFrame`. Per visualizzare qualcosa all'interno della finestra, bisogna creare dei componenti e si fa usando la classe `JComponent`. Da questo, poi, derivano le due classi `Graphics` e `Graphics2D`, che servono a manipolare forme grafiche. Metodi importanti sono `paintComponent()`, che disegna una componente e `draw()` che fa il cast tra `Graphics` e `Graphics2D`. Una volta costruito il frame e il componente, dobbiamo aggiungere quest'ultimo al frame tramite `add()`. Al posto dei componenti si possono usare anche gli **applet**. Gli applet sono programmi che vengono eseguiti in un web browser. Per implementarlo dobbiamo scrivere un file HTML col tag `applet`. Una volta terminata questa procedura, è possibile definire **forme grafiche**, che possono essere rettangoli, ellissi o linee. A seconda della forma, cambiano i parametri che serviranno poi a istanziarla. Tramite la classe `Color` si possono anche definire nuovi colori, usando il formato RGB e il metodo `setColor()`. Possiamo anche disegnare figure più complesse, definite separatamente in ogni classe. Un'applicazione grafica può anche prendere del testo in input. Lo si fa lanciando un oggetto `JOptionPane` che utilizza il metodo `showInputDialog()` per attendere l'input utente e per restituire la stringa digitata.

GESTIONE DEGLI EVENTI

Ogni volta che l'utente esegue un'azione (clic mouse, premere un tasto), viene generato un evento. Alla base degli eventi ci sono:

- **ricevitore** (***listener***). Riceve una notifica quando un evento accade e i suoi metodi descrivono le azioni da eseguire quando si verificano gli eventi.
- **sorgente** (***source***). La componente che ha generato l'evento. Quando accade un evento, la sorgente notifica a tutti i ricevitori dell'evento.

Usando gli eventi, rendiamo l'applicazione **reattiva**, e il programma si limiterà soltanto a inizializzare l'applicazione, istanziando gli osservatori e associandovi degli handler. Il package `javat.awt.Event` contiene tutte le classi necessarie, tra cui le interfacce `Listener` e `Adapter`. Possiamo distinguere diversi tipi di eventi:

- `MouseEvent`. Eventi del mouse come click o spostamenti. Esso può implementare `mousePressed()`, in caso di un pulsante qualsiasi premuto, `mouseReleased()`, pulsante rilasciato, `mouseClicked()`, se il pulsante è stato premuto e rilasciato, `mouseEntered()`, se il mouse passa sopra una componente, `mouseExited()`, se il mouse esce. Tutti questi metodi devono essere implementati e, se non usati, lasciati vuoti.
- `ActionEvent`. Eventi di azione su componenti, come il click su un bottone. Si crea un bottone creando un oggetto `JButton`.
- `AdjustmentEvent`. Eventi di modifica, che hanno un valore numerico modificabile.
- `ItemEvent`. Selezione o deselezione di un elemento.
- `KeyEvent`. Eventi da tastiera, come la pressione su un tasto.
- `WindowEvent`. Eventi relativi a finestre

Come abbiamo già anticipato, le interfacce Listener sono i ricevitori di eventi e definiscono i metodi che devono essere implementati da ogni oggetto che desidera essere informato dell'accadere di un particolare evento. A seconda degli eventi definiti sopra, abbiamo il corrispettivo Listener.

Per elaborare testo in input si usano componenti del tipo `JTextField`, e usiamo `JLabel` per definire un'etichetta relativa all'input. Si legge il testo tramite `getText()`, mentre si può mettere un testo di default tramite `setText()`.

INTERFACCE GRAFICHE

È possibile utilizzare l'ereditarietà anche per i frame, scomponendo frames complessi per unità facilmente comprensibili. È, quindi, necessario progettare sottoclassi di `JFrame`, memorizzando le componendi nelle variabili d'istanza, inizializzandole nel nostro costruttore. Le componenti di un'interfaccia vengono messe all'interno di un contenitore, tipo `JPanel`. Ogni contenitore ha un **layout manager**, che si occupa del posizionamento. Esistono vari tipi di LM:

- `BorderLayout`, **layout a bordi**. Il contenitore viene diviso in cinque aree, che sono `center`, `north`, `west`, `south` e `east`. Per aggiungere un componente se ne specifica la posizione.
- `GridLayout`, **layout a griglia**. Posiziona le componenti in una griglia con un numero fissato di righe e colonne. Ogni componente viene aggiunta riga per riga, da sinistra a destra. Questo layout supporta anche `GridBagLayout`, le cui componenti sono disposte in tabella.

Per aggiungere **componenti** ad un'interfaccia grafica, basta utilizzare alcuni strumenti messi a disposizione da `JComponent`. Ne vediamo alcune:

- `JButton`. Permette di implementare diversi tipi di **pulsanti**, come quelli radio, che consente di effettuare una scelta singola tra diversi altri pulsanti, ma anche check-box e elementi di un ipotetico menù. Un pulsante può visualizzare al suo interno un testo, una foto ed è possibile cambiare colore o font, usando opportuni metodi ereditati. Si specifica un'azione al pulsante utilizzando l'interfaccia `ActionListener` che abbiamo visto prima.
- `JComboBox`. Un semplice menù a comparsa, che mostra un elenco e che l'utente può selezionarne uno di questi. Per aggiungere un elemento al menù usiamo `addItem()`, mentre per visualizzare l'elemento selezionato usiamo il `getSelectedItem()`.
- `JList`. Simile al combo-box, con la differenza che crea semplicemente un insieme di oggetti che è possibile selezionare tramite il proprio indice, usando

il metodo `getSelectedIndex()`.

- `JMenuBar`. Si tratta di un'implementazione che contiene più oggetti `JMenu` che, a loro volta, mostrano diversi `JMenuItems`. Tutte queste interfacce hanno in comune il metodo `add()`, che permette di aggiungere un elemento al menù e `getItem(int index)`, che prende un indice e restituisce un elemento.
- `JSlider`. Implementa un slider, cioè permette di selezionare un valore facendo scorrere una manopola all'interno di esso. Si possono usare tanti metodi utili, tra cui `setValue(int v)`, che imposta la dimensione dell'intervallo.
- `JTextField`. Implementa un'area di testo in cui è possibile aggiungere righe, `rows`, o colonne, `columns`.

LAMBDA EXPRESSIONS

Le espressioni lambda sono una nuova ed importante funzionalità inclusa a partire da Java 8. Esse permettono di descrivere un metodo nel punto in cui viene utilizzato.. Per capire al meglio le lambda expressions dobbiamo soffermarci su altri due concetti di Java:

- **classi anonime interne**. Forniscono un modo per implementare classi che vengono utilizzate una sola volta in un'applicazione. Per esempio le classi in cui è richiesto un elevato numero di eventi, come quelle che implementano interfacce grafiche. Abbiamo già ampiamente visto come non è necessario creare classi separate per ogni evento, ma basta aggiungere un ascoltatore ad un componente.
- **interfacce funzionali**. Si tratta di interfacce caratterizzate dalla presenza di un solo metodo. Anche queste le abbiamo già viste, come la `Comparator` delle `Collections`, oppure la classe `Runnable` dei `Thread`.

Un'espressione lambda è come un metodo, dato che fornisce parametri e ha un corpo. Esse servono a ridurre la dimensione del codice delle classi. Facciamo un esempio:

```
int x;
int y;

System.out.println(x+y);
```

Questo snippet di codice dichiara due interi e poi stampa a video la somma. Possiamo ridurre notevolmente il codice:

```
(int x, int y) - > x+y
```

Con questo altro snippet introduciamo il simbolo `- >`, detto **operatore lambda**. In generale, le lambda expressions utilizzano un costrutto di tipo: ***lista parametri - > istruzioni***.
Il tipo di una lambda expression è il tipo dell'interfaccia che implementa, detto **target type**. Il compilatore determina il tipo di una lambda expression dal contesto in cui viene utilizzata, per esempio in assegnamenti, dichiarazioni, valori di ritorno o argomenti. Possiamo serializzare una lambda expression solo se sono

serializzabili i suoi argomenti e il suo target type. Ovviamente, però, per le lambda expressions, la serializzazione è fortemente sconsigliata.

THREADS E PROGRAMMAZIONE MULTITHREADING

Un thread è un flusso di controllo all'interno di un programma. Si tratta, fondamentalmente, della **versione leggera di un processo**, dato che non alloca molte risorse e non ha uno spazio di indirizzamento privato. I thread sono molto utilizzati, perché migliorano l'interazione con l'utente, possono simulare attività simultanee e sfruttano a pieno i sistemi multiprocessore.
Java permette di manipolare i thread. Essi fanno parte del pacchetto `java.lang.Thread`. Creando oggetti istanza di questa classe non facciamo altro che implementare strumenti con cui comunichiamo alla JVM di creare nuovi thread e di fare determinate operazioni. In Java, ogni programma in esecuzione è un thread e il metodo `main` è associato al main thread, cioè il primo thread creato. I thread possono essere creati in diversi modi:

- il primo metodo consiste nel dichiarare una classe che estende `Thread` e che sovrascrive il metodo `run()`.
- il secondo metodo consiste nell'utilizzare l'interfaccia `Runnable`, che implementa il metodo `run()`.

Oltre a `run()`, Java mette a disposizione anche altri metodi per gestire i thread:

- `sleep()`, che mette in attesa il thread prendendo come parametro un numero in millisecondi.
- `currentThread()`, che restituisce l'oggetto Thread corrispondente al thread di esecuzione che l'ha invocato.
- `join()`, che aspetta la terminazione di un thread.
- `interrupt()`, che interrompe l'esecuzione del thread.
- `setPriority(int p)`, che imposta la priorità di esecuzione ad un determinato thread. La scala è garantita solo all'interno del linguaggio, quindi dovrà comunque passare per i filtraggi del sistema operativo.

Questo tipo di programmazione, detta **concorrente**, può far incorrere in alcuni problemi. Creando applicazioni multithread, ci troveremo di fronte molti oggetti che accedono contemporaneamente ai dati comuni, bloccandosi a vicenda. Per questo, è opportuno usare metodi `synchronized`. In alternativa ai metodi sincronizzati, è possibile utilizzare interfacce di tipo `Lock`, che servono a manipolare risorse condivise. Di solito un oggetto Lock viene aggiunto ad una classe i cui metodi condividono risorse. Il codice che manipola le risorse condivise è compreso tra i metodi `lock()` e `unlock()`. I thread possono anche comunicare tra loro:

- `wait()`, rilascia il blocco sull'oggetto e arresta il thread, mettendolo in attesa.
- `notify()`, sveglia il thread in attesa.
- `notifyAll()`, sveglia tutti i thread.