

Costrutti di concorrenza nei linguaggi di alto livello

Introduzione

Scopo di questa lezione è imparare a:

- ▶ ricondurre le primitive di sincronizzazione dei linguaggi di alto livello ai meccanismi di locking visti nel modello
- ▶ comprendere il flusso di esecuzione di un programma concorrente scritto in un linguaggio di alto livello

Considereremo inizialmente il multithreading in Java

Multithreading e concorrenza in Java

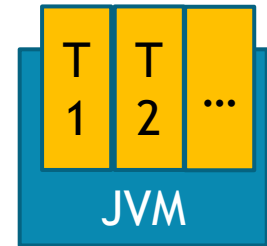
Multithreading e Concorrenza in Java

Java è stato **uno dei primi** linguaggi a fornire costrutti di multithreading abbastanza semplici da utilizzare

- ▶ Multithreading disponibile fin dalle prime versioni di Java

Il modello originale di multithreading in Java prevede che i thread abbiano

- ▶ **ognuno il proprio stack**
- ▶ **heap condiviso** (shared memory)



Nelle prime versioni, lo scheduling dei thread **era gestito dalla JVM**

Nelle versioni più recenti, la JVM sfrutta **servizi di threading del Sistema Operativo**

- ▶ Consente **esecuzione parallela** se sono disponibili più processori/core

Evoluzioni in Java 5 e in Java 8

Dalla versione 5 di Java è disponibile la **Concurrency API**, una parte della libreria standard del linguaggio dedicata alla concorrenza

- Consente di programmare usando **un'ampia gamma di modelli di concorrenza e parallelismo**, e meccanismi di sincronizzazione

Dalla versione 8 di Java la Concurrency API è stata arricchita di elementi di **programmazione funzionale**

- Ad esempio, la possibilità di usare **lambda espressioni** per definire task da eseguire concorrentemente

Primo esempio: Hello Thread!

Un thread in Java è un oggetto di tipo Thread:

- deve contenere un metodo **run()** e si avvia con **start()**

```
class Hello extends Thread{  
    public void run() {  
        System.out.println("Hello Thread!");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Thread t = new Hello();  
        t.start();  
    }  
}
```

```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```

Primo esempio: Hello Thread!

Un thread in Java è un oggetto di tipo Thread:

- ▶ deve contenere un metodo **run()** e si avvia con **start()**

```
class Hello extends Thread{  
    public void run() {  
        System.out.println("Hello Thread!");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Thread t = new Hello();  
        t.start();  
    }  
}
```

```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```

Attenzione a non fare
t.run()

Stamperebbe ugualmente
"Hello Thread!" ma non in
modo concorrente...

In modo equivalente, la classe da eseguire come thread può implementare l'interfaccia Runnable

- ▶ contiene il solo metodo **run()**
- ▶ rappresenta un **task** (o **job**) che può essere eseguito come thread

```
class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello Thread!");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Runnable r = new HelloRunnable();  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```


In modo equivalente, la classe da eseguire come thread può implementare l'interfaccia Runnable


- ▶ contiene il solo metodo **run()**
- ▶ rappresenta un **task** (o **job**) che può essere eseguito come thread

```
class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello Thread!");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Runnable r = new HelloRunnable();  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```

Crea il thread passandogli
il task (Runnable) da
eseguire



Thread o Runnable?

Facendo un uso basilare dei thread **non c'è molta differenza tra estendere Thread e implementare Runnable**


Inizializzare un thread ha un costo computazionale

- ▶ Alcune funzionalità avanzate della Concurrency API consentono di gestire **pool di thread**
- ▶ Ad esempio: inizializzo 50 thread una volta per tutte e poi passo loro diversi (es. migliaia) di oggetti Runnable da eseguire

Raffinamenti... (classe anonima)

```
class Main {  
    public static void main(String[] args) {  
        Runnable r = new Runnable() {  
            public void run() {  
                System.out.println("Hello Thread!");  
            }  
        };  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

Anonymous inner class
(monouso...)

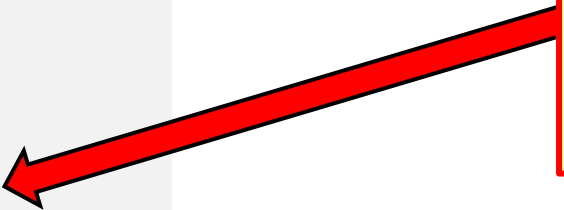


```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```

Raffinamenti... (lambda expression)

```
class Main {  
    public static void main(String[] args) {  
        Runnable r =  
            () -> { System.out.println("Hello Thread!"); };  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

Oggetto Runnable come
funzione anonima
(lambda expression)



```
> javac Main.java  
> java Main  
Hello Thread!  
> _
```

Esempio: Timer

Vediamo un esempio meno banale:

- La classe **Timer** descrive un thread che, quando avviato, descrive il passare del tempo **stampando un messaggio ogni secondo**

1 sec

2 sec

...

- Finché non viene chiamato il metodo **finish()** per terminare l'esecuzione

```
public class Timer implements Thread {  
    private boolean go=true;  
    public void run() {  
        int seconds=0;  
        try {  
            while (go) {  
                Thread.sleep(1000); seconds++;  
                if (go) System.out.println(seconds + " sec");  
            }  
        }  
        catch (InterruptedException ie) {  
            System.out.println("Errore...");  
        }  
    }  
    public void finish() { go=false; }  
}
```

Esempio: Timer

Indica lo stato del timer
(avviato/fermo)

```
public class Timer implements Thread {
```

```
    private boolean go=true;
```

```
    public void run() {
```

```
        int seconds=0;
```

```
        try {
```

```
            while (go) {
```

```
                Thread.sleep(1000); seconds++;
```

```
                if (go) System.out.println(seconds + " sec");
```

```
            }
```

```
        }
```

```
        catch (InterruptedException ie) {
```

```
            System.out.println("Errore...");
```

```
        }
```

```
    }
```

```
    public void finish() { go=false; }
```

```
}
```

Fintanto che vale go
stampa un messaggio
ogni 1000 millisecondi

Thread.sleep() potrebbe
sollevare un'eccezione
che va gestita...

Il metodo da chiamare
per fermare il timer

Esempio Timer: cronometrare un ordinamento

Supponiamo di voler usare il timer per cronometrare il tempo di esecuzione di una parte di programma che esegue un ordinamento

- ▶ Effetto "progress bar": feedback continuo sull'avanzamento

Servono due thread concorrenti:

- ▶ Quello che esegue il codice da cronometrare
- ▶ Il timer

Se non servisse l'effetto "progress bar" il multithreading non sarebbe necessario:

- ▶ Basterebbe leggere l'orologio di sistema (sequenzialmente) prima e dopo l'esecuzione del codice e fare la differenza tra i due tempi

```
import java.util.*;

public class Main {

    public static void main(String[] args) {

        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("FINITO");
    }
}
```

Il codice da
cronometrare

```
public class Timer implements Thread {

    private boolean go=true;
    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException ie) {
            System.out.println("Errore...");
        }
    }
    public void finish() { go=false; }
}
```

Il Timer di prima


```
import java.util.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Timer timer = new Timer();
```

```
        ArrayList<Integer> data = new ArrayList<Integer>();
```

```
        Random rng = new Random();
```

```
        for (int i=0; i<5000000; i++)
```

```
            data.add(rng.nextInt(1000));
```

```
        System.out.println("ORDINAMENTO IN CORSO...");
```

```
        timer.start();
```

```
        Collections.sort(data);
```

```
        timer.finish();
```

```
        System.out.println("FINITO");
```

```
    }
```

```
}
```

Crea il Timer

Popola un ArrayList con
5Mln di numeri casuali

Avvia il timer

Ordina l'ArrayList

Ferma il timer

```

import java.util.*;

public class Main {

    public static void main(String[] args) {

        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("FINITO");
    }
}

```

```

> javac *.java
> java Main
ORDINAMENTO IN CORSO...
1 sec
2 sec
3 sec
4 sec
5 sec
FINITO

```

```

public class Timer implements Thread {

    private boolean go=true;

    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException ie) {
            System.out.println("Errore...");
        }
    }

    public void stop() { go=false; }
}

```

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("FINITO");
    }
}
```

```
public class Timer implements Thread {
    private boolean go=true;
    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException ie) {
            System.out.println("Errore...");
        }
    }
}
```

```
> javac *.java
> java Main
ORDINAMENTO IN CORSO...
1 sec
2 sec
3 sec
4 sec
5 sec
FINITO
```

Il thread Timer sta stampando i tempi intanto che l'ArrayList viene ordinato

Domande...

Nell'esempio appena illustrato:

1. Quanti thread sono stati eseguiti?
2. Il numero dei thread attivi è stato sempre costante o è variato durante l'esecuzione?
3. Qual è stato il flusso di esecuzione di ogni thread (che parti del codice ha eseguito)?
4. Quale thread ha settato la variabile go a false?
5. Sono stati usati dei meccanismi di sincronizzazione?
6. E' stato usato qualcosa di simile ai lock?
7. Sarebbe opportuno aggiungere dei lock?

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("FINITO");
    }
}
```

Avvio del main thread

Avvio del Timer thread

```
public class Timer implements Thread {
    private boolean go=true;
    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException ie) {
            System.out.println("Errore...");
        }
    }

    public void finish() { go=false; }
}
```

Risposte:

1. Quanti thread sono stati eseguiti?

Due! Tutti i programmi partono con un main thread a cui si possono aggiungere altri thread facendo start() su un oggetto di tipo Thread

```
import java.util.*;


public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("ORDINAMENTO COMPLETO");
    }
}
```



Avvio del Timer thread

```
public class Timer implements Thread {
    private boolean go=true;
    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException e) {
            System.out.println("Timer thread interrupted");
        }
    }
    public void finish() { go=false; }
}
```



Terminazione del
Timer thread, alla fine
del metodo run()

Risposte:

2. Il numero dei thread attivi è stato sempre costante o è variato durante l'esecuzione?
E' variato... prima 1, poi 2 (avvio Timer), poi di nuovo 1 (terminazione Timer)

Numero di thread nel modello di concorrenza

Nel modello di concorrenza che abbiamo studiato, il numero dei thread era costante e determinato dalla struttura dell'espressione

► Esempio: $\ell_1 := 10; \ell_2 := 3 \mid \ell_3 = !\ell_3 + 1$ (due thread!)

La creazione di nuovi thread di Java potrebbe essere descritta più o meno così:

$e ::= \dots \mid \text{new Thread}(e)$

$\text{new Thread}(e_1); e_2 \rightarrow e_1 \mid e_2$

```
import java.util.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        ArrayList<Integer> data = new ArrayList<Integer>();  
        Random rng = new Random();  
        for (int i=0; i<5000000; i++)
```

Codice eseguito dal
main thread

```
            System.out.println("IN CORSO...");
```

```
        timer.start();  
        Collections.sort(data);  
        timer.finish();  
        System.out.println("FINITO");  
    }
```

```
}
```

```
public class Timer implements Thread {
```

```
    private boolean go=true;
```

```
    public void run() {
```

```
        int seconds=0;
```

```
        try {
```

```
            while (go) {
```

```
                Thread.sleep(1000); seconds++;
```

```
                if (go) System.out.println(seconds + " sec");
```

```
            }
```

```
        }
```

```
        catch (InterruptedException ie) {
```

```
            System.out.println("Errore...");
```

```
        }
```

```
    }
```

```
    public void finish() { go=false; }
```

```
}
```

Codice eseguito dal
Timer thread

Risposte:

3. Qual è stato il flusso di esecuzione di ogni thread (che parti del codice ha eseguito)?

Le aree rosse il main thread, l'area verde il Timer thread

Attenzione al flusso di esecuzione!

- ▶ I thread si possono **muovere liberamente** da una classe all'altra (rispettando le regole di visibilità dei metodi)

```
import java.util.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        ArrayList<Integer> data = new ArrayList<Integer>();  
        Random rng = new Random();  
        for (int i=0; i<5000000; i++)
```

Codice eseguito dal
main thread

```
            System.out.println("IN CORSO...");
```

```
        timer.start();
```

```
        Collections.sort(data);
```

```
        timer.finish();
```

```
        System.out.println("FINITO");
```

```
    }
```

```
}
```

```
public class Timer implements Thread {
```

```
    private boolean go=true;
```

```
    public void run() {
```

```
        int seconds=0;
```

```
        try {
```

```
            while (go) {
```

```
                Thread.sleep(1000); seconds++;
```

```
                if (go) System.out.println(seconds + " sec");
```

```
            }
```

```
        }
```

```
        catch (InterruptedException ie) {
```

```
            System.out.println("Errore...");
```

```
        }
```

```
    }
```

```
    public void finish() { go=false; }
```

```
}
```

Codice eseguito dal
Timer thread

Risposte:

4. Quale thread ha settato la variabile go a false?

Il main thread, con la chiamata a finish()

```
import java.util.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        ArrayList<Integer> data = new ArrayList<Integer>();  
        Random rng = new Random();  
        for (int i=0; i<5000000; i++)
```

Codice eseguito dal
main thread

```
            System.out.println("IN CORSO...");
```

```
        timer.start();  
        Collections.sort(data);  
        timer.finish();  
        System.out.println("FINITO");  
    }
```

```
}
```

```
public class Timer implements Thread {
```

```
    private boolean go=true;
```

```
    public void run() {
```

```
        int seconds=0;
```

```
        try {
```

```
            while (go) {
```

```
                Thread.sleep(1000); seconds++;
```

```
                if (go) System.out.println(seconds + " sec");
```

```
            }
```

```
        }
```

```
        catch (InterruptedException ie) {
```

```
            System.out.println("Errore...");
```

```
        }
```

```
    }
```

```
    public void finish() { go=false; }
```

```
}
```

Codice eseguito dal
Timer thread

Risposte:

5. Sono stati usati dei meccanismi di sincronizzazione?

La variabile go... non è un costrutto linguistico specifico, ma è stata usata per sincronizzare i due thread (segnalare al Timer che l'ordinamento è terminato)

```
import java.util.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        ArrayList<Integer> data = new ArrayList<Integer>();  
        Random rng = new Random();  
        for (int i=0; i<5000000; i++)
```

Codice eseguito dal
main thread

```
            System.out.println("IN CORSO...");
```

```
        timer.start();  
        Collections.sort(data);  
        timer.finish();  
        System.out.println("FINITO");  
    }
```

```
}
```

```
public class Timer implements Thread {
```

```
    private boolean go=true;
```

```
    public void run() {
```

```
        int seconds=0;
```

```
        try {
```

```
            while (go) {
```

```
                Thread.sleep(1000); seconds++;
```

```
                if (go) System.out.println(seconds + " sec");
```

```
            }
```

```
        }
```

```
        catch (InterruptedException ie) {
```

```
            System.out.println("Errore...");
```

```
        }
```

```
    }
```

```
    public void finish() { go=false; }
```

```
}
```

Codice eseguito dal
Timer thread

Risposte:

6. E' stato usato qualcosa di simile ai lock?

No... nessuna delle operazioni svolte è (neanche potenzialmente) bloccante.

La sleep(1000) è solo un meccanismo di pausa temporanea interno al thread.

```

import java.util.*;

public class Main {

    public static void main(String[] args) {

        Timer timer = new Timer();
        ArrayList<Integer> data = new ArrayList<Integer>();
        Random rng = new Random();
        for (int i=0; i<5000000; i++)
            data.add(rng.nextInt(1000));

        System.out.println("ORDINAMENTO IN CORSO...");
        timer.start();
        Collections.sort(data);
        timer.finish();
        System.out.println("FINITO");
    }
}

```

```

public class Timer implements Thread {

    private boolean go=true;

    public void run() {
        int seconds=0;
        try {
            while (go) {
                Thread.sleep(1000); seconds++;
                if (go) System.out.println(seconds + " sec");
            }
        } catch (InterruptedException ie) {
            System.out.println("Errore...");
        }
    }

    public void finish() { go=false; }
}

```

Risposte:

7. Sarebbe opportuno aggiungere dei lock?

No... sull'unica variabile condivisa (go) si fanno solo operazioni atomiche

Inoltre i due thread non competono su go: uno scrive solo e l'altro legge solo

Altro esempio: contatore

Prendiamo un contatore:

- ▶ incrementiamo 500000 volte e
- ▶ decrementiamolo 500000 volte

In due modi:

- ▶ prima sequenzialmente
- ▶ poi concorrentemente

```
class Main { // sequenziale
    public static void main(String[] args) {
        Counter c = new Counter();

        for (int i=0; i<500000; i++) { c.incr(); }
        System.out.println("DONE1: " + c.getVal());

        for (int i=0; i<500000; i++) { c.decr(); }
        System.out.println("DONE2: " + c.getVal());
    }
}
```

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

```
> javac *.java
```

```
> java Main
```

```
DONE1: 500000
```

```
DONE2: 0
```

```
> _
```

Valore del contatore
quando finisce di
incrementare

Valore del contatore
quando finisce di
decrementare

```
class Main { // concorrente
    public static void main(String[] args) {
        Counter c = new Counter();
        Runnable r1 = () -> {
            for (int i=0; i<500000; i++) {c.incr(); }
            System.out.println("DONE R1: " + c.getVal());
        };
        Runnable r2 = () -> {
            for (int i=0; i<500000; i++) {c.decr(); }
            System.out.println("DONE R2: " + c.getVal());
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

>


```
class Main { // concorrente
    public static void main(String[] args) {
        Counter c = new Counter();

        Runnable r1 = () -> {
            for (int i=0; i<500000; i++) {c.incr(); }
            System.out.println("DONE R1: " + c.getVal());
        };

        Runnable r2 = () -> {
            for (int i=0; i<500000; i++) {c.decr(); }
            System.out.println("DONE R2: " + c.getVal());
        };

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

>



Inserisce ogni ciclo
in un Runnable

```
class Main { // concorrente
    public static void main(String[] args) {
        Counter c = new Counter();
        Runnable r1 = () -> {
            for (int i=0; i<500000; i++) {c.incr(); }
            System.out.println("DONE R1: " + c.getVal());
        };
        Runnable r2 = () -> {
            for (int i=0; i<500000; i++) {c.decr(); }
            System.out.println("DONE R2: " + c.getVal());
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

Inizializza due
Thread con i due
Runnable e li avvia

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

>

```
class Main { // concorrente
    public static void main(String[] args) {
        Counter c = new Counter();
        Runnable r1 = () -> {
            for (int i=0; i<500000; i++) {c.incr(); }
            System.out.println("DONE R1: " + c.getVal());
        };
        Runnable r2 = () -> {
            for (int i=0; i<500000; i++) {c.decr(); }
            System.out.println("DONE R2: " + c.getVal());
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

```
> javac *.java
```

```
> java Main
```

```
DONE R1: 48150
```

```
DONE R2: -159230
```

```
> _
```



Che cosa è andato storto?

```
class Main { // concorrente
    public static void main(String[] args) {
        Counter c = new Counter();
        Runnable r1 = () -> {
            for (int i=0; i<500000; i++) {c.incr(); }
            System.out.println("DONE R1: " + c.getVal());
        };
        Runnable r2 = () -> {
            for (int i=0; i<500000; i++) {c.decr(); }
            System.out.println("DONE R2: " + c.getVal());
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

```
public class Counter { // contatore
    private int n;
    public Counter() {n=0;}
    public void incr() {n=n+1;}
    public void decr() {n=n-1;}
    public int getVal() {return n;}
}
```

```
> javac *.java
```

```
> java Main
```

```
DONE R1: 48150
```

```
DONE R2: -159230
```

```
> _
```

Queste operazioni
non sono atomiche
e sono svolte
concorrentemente
dai due thread...

Lock e sincronizzazioni

In Java ogni oggetto dispone di un lock (mutex)

- ▶ E' possibile eseguire un metodo in mutua esclusione acquisendo il lock sull'oggetto tramite il modificatore **synchronized**
- ▶ L'esecuzione di un metodo "sincronizzato" è **bloccante** rispetto ad altre chiamate dello stesso metodo o di altri metodi sincronizzati sullo stesso oggetto.
- ▶ Chiamate allo stesso metodo (d'istanza) su **oggetti diversi** non si bloccano tra loro

```
class Main {  
    public static void main(String[] args) {  
        Counter c = new Counter();  
        // ...  
    }  
}
```

incr() e decr() ora sono sincronizzati!

Nel modello equivale a:

$incr() = \text{lock } m_c; n := !n + 1; \text{unlock } m_c$
 $decr() = \text{lock } m_c; n := !n - 1; \text{unlock } m_c$

dove m_c è il mutex dell'oggetto c creato (fresco) dal costruttore di Counter

```
Thread t2 = new Thread(r2);  
t1.start();  
t2.start();  
}  
}
```

```
public class Counter {  
    private int n;  
    public Counter() {n=0;}  
    public synchronized void incr() {n=n+1;}  
    public synchronized void decr() {n=n-1;}  
    public int getVal() {return n;}  
}
```

```
> javac *.java
```

```
> java Main
```

```
DONE R1: 466271
```

```
DONE R2: 0
```

```
> _
```

OK, non è 50000
perché t2 ha iniziato
a decrementare
mentre t1 stava
ancora incrementando

OK, il contatore è
tornato a 0

```
class Main {  
    public static void main(String[] args) {  
        Counter c = new Counter();  
        Runnable r1 = () -> {  
            for (int i=0; i<500000; i++) {c.incr(); }  
            System.out.println("DONE R1: " + c.getVal());  
        };  
        Runnable r2 = () -> {  
            for (int i=0; i<500000; i++) {c.decr(); }  
            System.out.println("DONE R2: " + c.getVal());  
        };  
        Thread t1 = new Thread(r1);  
        Thread t2 = new Thread(r2);  
        t1.start();  
        t2.start();  
    }  
}
```

```
public class Counter {  
    private int n;  
    public Counter() {n=0;}  
    public synchronized void incr() {n=n+1;}  
    public synchronized void decr() {n=n-1;}  
    public int getVal() {return n;}  
}
```

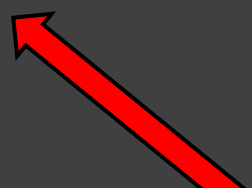
```
> javac *.java
```

```
> java Main
```

```
DONE R1: 0
```

```
DONE R2: -43181
```

```
> _
```



Altra esecuzione
possibile. Comunque
OK... t2 è stato più
veloce di t1...

Coarse-grained VS fine-grained locking

Il modificatore **synchronized** fa acquisire il lock per **tutta la durata del metodo**

- ▶ Un metodo può essere molto lungo...
- ▶ E' possibile che in realtà la parte critica di un metodo siano solo pochi comandi (o uno solo...)

Il modificatore **synchronized** applica una strategia di locking **coarse-grained**

- ▶ Per scoprire i costrutti di locking **fine-grained** vediamo un altro esempio ancora...

Un altro esempio ancora...

generatore di numeri (pseudo)casuali

La classe **Generator**

- ▶ ha un metodo **generate()** che genera numeri naturali minori di 100
- ▶ **tiene conto** di quanti numeri **pari** e **dispari** ha generato

```
import java.util.*;
class Generator {
    private Random rng;
    public int pari;
    public int dispari;

    public Generator() { rng=new Random(); pari=0; dispari=0; }

    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}
```

```
class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 10000;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++)
                        g.generate();
                    };
            Thread t = new Thread(r);
            t.start();
        }

        try { Thread.sleep(10000); }
        catch (InterruptedException ie) {}

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}
```

```
import java.util.*;
class Generator {
    private Random rng;
    public int pari;
    public int dispari;
    public Generator() { rng=new Random(); pari=0; dispari=0; }
    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}
```

Generator
(appena vista)

Main
(usa generator)

```
class Main {  
    public static void main(String[] args) {  
        final int THREADS = 10;  
        final int VALUES = 10000;  
  
        Generator g = new Generator();  
        for (int i=0; i<THREADS; i++) {  
            Runnable r =  
                () -> { for (int j=0; j<VALUES; j++)  
                        g.generate();  
                };  
            Thread t = new Thread(r);  
            t.start();  
        }  
  
        try { Thread.sleep(10000); }  
        catch (InterruptedException ie) {}  
  
        System.out.println("PARI: " + g.pari +  
                           " DISPARI: " + g.dispari);  
        if ((g.pari+g.dispari) != (THREADS*VALUES))  
            System.out.println("ERRORE");  
    }  
}
```

Si prepara a creare 10 thread, ognuno dei quali genererà (usando Generator) 10000 valori

Crea i thread con un ciclo, e ogni thread fa a sua volta un ciclo in cui chiama generate()

Intanto che i thread fanno il loro dovere, il main thread lascia passare 10 secondi e poi stampa un resoconto

```

class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 10000;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++)
                        g.generate();
                    };
            Thread t = new Thread(r);
            t.start();
        }

        try { Thread.sleep(10000); }
        catch (InterruptedException ie) {}

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}

```

```

import java.util.*;
class Generator {
    private Random rng;
    public int pari;
    public int dispari;
    public Generator() { rng=new Random(); pari=0; dispari=0; }
    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}

```

```

> javac Main.java
> java Main
PARI: 50072 DISPARI: 49928
> _

```

Ok, ha
funzionato!

```

class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 10000;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++)
                        g.generate();
                    };
            Thread t = new Thread(r);
            t.start();
        }

        try { Thread.sleep(10000); }
        catch (InterruptedException ie) {}

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}

```

```

import java.util.*;
class Generator {
    private Random rng;
    public int pari;
    public int dispari;
    public Generator() { rng=new Random(); pari=0; dispari=0; }
    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}

```

```

> javac Main.java
> java Main
PARI: 49749 DISPARI: 49917
ERRORE
> _

```

Eseguendo di nuovo ha dato errore! La somma non fa 100000...

```

class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 10000;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++)
                        g.generate();
                    };
            Thread t = new Thread(r);
            t.start();
        }

        try { Thread.sleep(10000); }
        catch (InterruptedException ie) {}

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}

```

```

import java.util.*;
class Generator {
    private Random rng;
    public int pari;
    public int dispari;
    public Generator() { rng=new Random(); pari=0; dispari=0; }
    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}

```

Solito problema:
aggiornamenti non
atomici
concorrenti

```

> javac Main.java
> java Main

PARI: 49749 DISPARI: 49917

ERRORE

> _

```

```
class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 100;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++) {
                    g.generate();
                }
            };
            try {
                cat
            } catch (Exception e) {}
        }

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}
```

Possiamo risolvere
come prima,
sincronizzando
l'intero metodo
(coarse-grained)

Questa soluzione però fa
eseguire in mutua esclusione
anche le nextInt (che sono
computazionalmente costose e
sarebbe meglio eseguire
concorrentemente)

```
import java.util.*;

class Generator {
    private Random rng;
    public int pari;
    public int dispari;

    public Generator() { rng=new Random(); pari=0; dispari=0; }

    public void synchronized generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { pari = pari+1; }
        else { dispari = dispari+1; }
    }
}
```

```
> javac Main.java
> java Main

PARI: 50093 DISPARI: 49907
> _
```

Ok, ora
funziona!

```

class Main {
    public static void main(String[] args) {
        final int THREADS = 10;
        final int VALUES = 100;

        Generator g = new Generator();
        for (int i=0; i<THREADS; i++) {
            Runnable r =
                () -> { for (int j=0; j<VALUES; j++) {
                        g.generate();
                    } };
            Thread t = new Thread(r);
            t.start();
        }

        try { Thread.sleep(10000); }
        catch (InterruptedException ie) {}

        System.out.println("PARI: " + g.pari +
                           " DISPARI: " + g.dispari);
        if ((g.pari+g.dispari) != (THREADS*VALUES))
            System.out.println("ERRORE");
    }
}

```

**Soluzione alternativa:
blocco sincronizzato
(fine-grained)**

**Acquisisce il lock
dell'oggetto this solo
per la durata del
blocco che segue**

```

import java.util.*;

class Generator {
    private Random rng;
    public int pari;
    public int dispari;

    public Generator() { rng=new Random(); pari=0; dispari=0; }

    public void generate() {
        int val = rng.nextInt(100);
        if (val%2==0) { synchronized (this) { pari = pari+1; } }
        else { synchronized (this) { dispari = dispari+1; } }
    }
}

```

```

> javac Main.java
> java Main

PARI: 50073 DISPARI: 49927
> _

```

**Ok, funziona
ancora!**

Synchronized block

Sintassi:

```
synchronized (obj) { cmd_block }
```

Nel modello corrisponde a:

lock m_{obj} ; *cmdblock* ; **unlock** m_{obj}

obj può essere un qualunque oggetto attivo... (non necessariamente this)

Attenzione: fine-grained locking...

Ma... a che cosa bisogna stare attenti quando si adotta una strategia di locking fine-grained?

```

class Main {
    static ArrayList<Integer> al1 = new ArrayList<Integer>();
    static ArrayList<Integer> al2 = new ArrayList<Integer>();
    public static void main(String[] args) {
        init(al1,al2); // inizializza al1 e al2
        Thread t1 = new Thread() {
            public void run() {
                synchronized(al1) {
                    System.out.println("T1: lock al1");
                }
                synchronized(al2) {
                    System.out.println("T1: lock al2");
                    // qui puo' operare su al1 e al2
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                synchronized(al2) {
                    System.out.println("T2: lock al2");
                }
                synchronized(al1) {
                    System.out.println("T2: lock al1");
                    // qui puo' operare su al2 e al1
                }
            }
        };
        t1.start();
        t2.start();
    }
}

```

La classe **Main** (fittizia)

- crea **due thread** che lavorano su **due ArrayList contemporaneamente**
- ogni thread acquisisce **un lock per ogni ArrayList**

```
class Main {
```

```
    static ArrayList<Integer> al1 = new ArrayList<Integer>();  
    static ArrayList<Integer> al2 = new ArrayList<Integer>();
```

```
    public static void main(String[] args) {  
        init(al1,al2); // inizializza al1 e al2, non mostrato
```

```
        Thread t1 = new Thread() {  
            public void run() {  
                synchronized(al1) {  
                    System.out.println("T1: lock al1");  
                    synchronized(al2) {  
                        System.out.println("T1: lock al2");  
                        // qui puo' operare su al1 e al2  
                    }  
                }  
            }  
        };
```

```
        Thread t2 = new Thread() {  
            public void run() {  
                synchronized(al2) {  
                    System.out.println("T2: lock al2");  
                    synchronized(al1) {  
                        System.out.println("T2: lock al1");  
                        // qui puo' operare su al2 e al1  
                    }  
                }  
            }  
        };
```

```
        t1.start();
```

```
        t2.start();
```

```
    }
```

Crea due ArrayList

Ognuno dei due thread
acquisisce i lock di
entrambe le array list
nel momento in cui
devono usarle (con
blocchi sincronizzati
annidati)

```
class Main {  
    static ArrayList<Integer> al1 = new ArrayList<Integer>();  
    static ArrayList<Integer> al2 = new ArrayList<Integer>();  
    public static void main(String[] args) {  
        init(al1,al2); // inizializza al1 e al2, non mostrato  
        Thread t1 = new Thread() {  
            public void run() {  
                synchronized(al1) {  
                    System.out.println("T1: lock al1");  
                    synchronized(al2) {  
                        System.out.println("T1: lock al2");  
                        // qui puo' operare su al1 e al2  
                    }  
                }  
            }  
        };  
        Thread t2 = new Thread() {  
            public void run() {  
                synchronized(al2) {  
                    System.out.println("T2: lock al2");  
                    synchronized(al1) {  
                        System.out.println("T2: lock al1");  
                        // qui puo' operare su al2 e al1  
                    }  
                }  
            }  
        };  
        t1.start();  
        t2.start();  
    }  
}
```

```
> javac Main.java
```

```
> java Main
```


```
T1: lock al1
```

```
T1: lock al2
```

```
T2: lock al2
```

```
T2: lock al1
```

```
> _
```



Ok, lock
acquisiti da
entrambi i
thread


```
class Main {  
    static ArrayList<Integer> al1 = new ArrayList<Integer>();  
    static ArrayList<Integer> al2 = new ArrayList<Integer>();  
    public static void main(String[] args) {  
        init(al1,al2); // inizializza al1 e al2, non mostrato  
        Thread t1 = new Thread() {  
            public void run() {  
                synchronized(al1) {  
                    System.out.println("T1: lock al1");  
                    synchronized(al2) {  
                        System.out.println("T1: lock al2");  
                        // qui puo' operare su al1 e al2  
                    }  
                }  
            }  
        };  
        Thread t2 = new Thread() {  
            public void run() {  
                synchronized(al2) {  
                    System.out.println("T2: lock al2");  
                    synchronized(al1) {  
                        System.out.println("T2: lock al1");  
                        // qui puo' operare su al2 e al1  
                    }  
                }  
            }  
        };  
        t1.start();  
        t2.start();  
    }  
}
```

```
> javac Main.java
```

```
> java Main
```

```
T1: lock al1
```

```
T2: lock al2
```



Altra esecuzione...
DEADLOCK!!
il programma si è
piantato...

```
class Main {
    static ArrayList<Integer> al1 = new ArrayList<Integer>();
    static ArrayList<Integer> al2 = new ArrayList<Integer>();
    public static void main(String[] args) {
        init(al1, al2); // inizializza al1 e al2, non mostro
        Thread t1 = new Thread() {
            public void run() {
                synchronized(al1) {
                    System.out.println("T1: lock al1");
                }
                synchronized(al2) {
                    System.out.println("T1: lock al2");
                    // qui puo' operare su al1 e al2
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                synchronized(al2) {
                    System.out.println("T2: lock al2");
                }
                synchronized(al1) {
                    System.out.println("T2: lock al1");
                    // qui puo' operare su al2 e al1
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

Sulla base di quanto studiato sul modello formale, come possiamo modificare questo programma per prevenire il deadlock?

main.java

```
> java Main
T1: lock al1
T2: lock al2
```

Altra esecuzione...
DEADLOCK!!
il programma si è piantato...

```
class Main {  
    static ArrayList<Integer> al1 = new ArrayList<Integer>();  
    static ArrayList<Integer> al2 = new ArrayList<Integer>();  
    public static void main(String[] args) {  
        init(al1,al2); // inizializza al1 e al2, non mostr  
        Thread t1 = new Thread() {  
            public void run() {  
                synchronized(al1) {  
                    System.out.println("T1: lock al1");  
                    synchronized(al2) {  
                        System.out.println("T1: lock al2");  
                        // qui puo' operare su al1 e al2  
                    }  
                }  
            }  
        };  
        Thread t2 = new Thread() {  
            public void run() {  
                synchronized(al1) {  
                    System.out.println("T2: lock al1");  
                    synchronized(al2) {  
                        System.out.println("T2: lock al2");  
                        // qui puo' operare su al2 e al1  
                    }  
                }  
            }  
        };  
        t1.start();  
        t2.start();  
    }  
}
```

**Acquisendo i lock nello
stesso ordine!**

(e rilasciandoli in ordine
inverso, ma questo
segue dall'annidamento
dei blocchi)

```
> javac Main.java  
> java Main  
T1: lock al1  
T1: lock al2  
T2: lock al1  
T2: lock al2  
> _
```

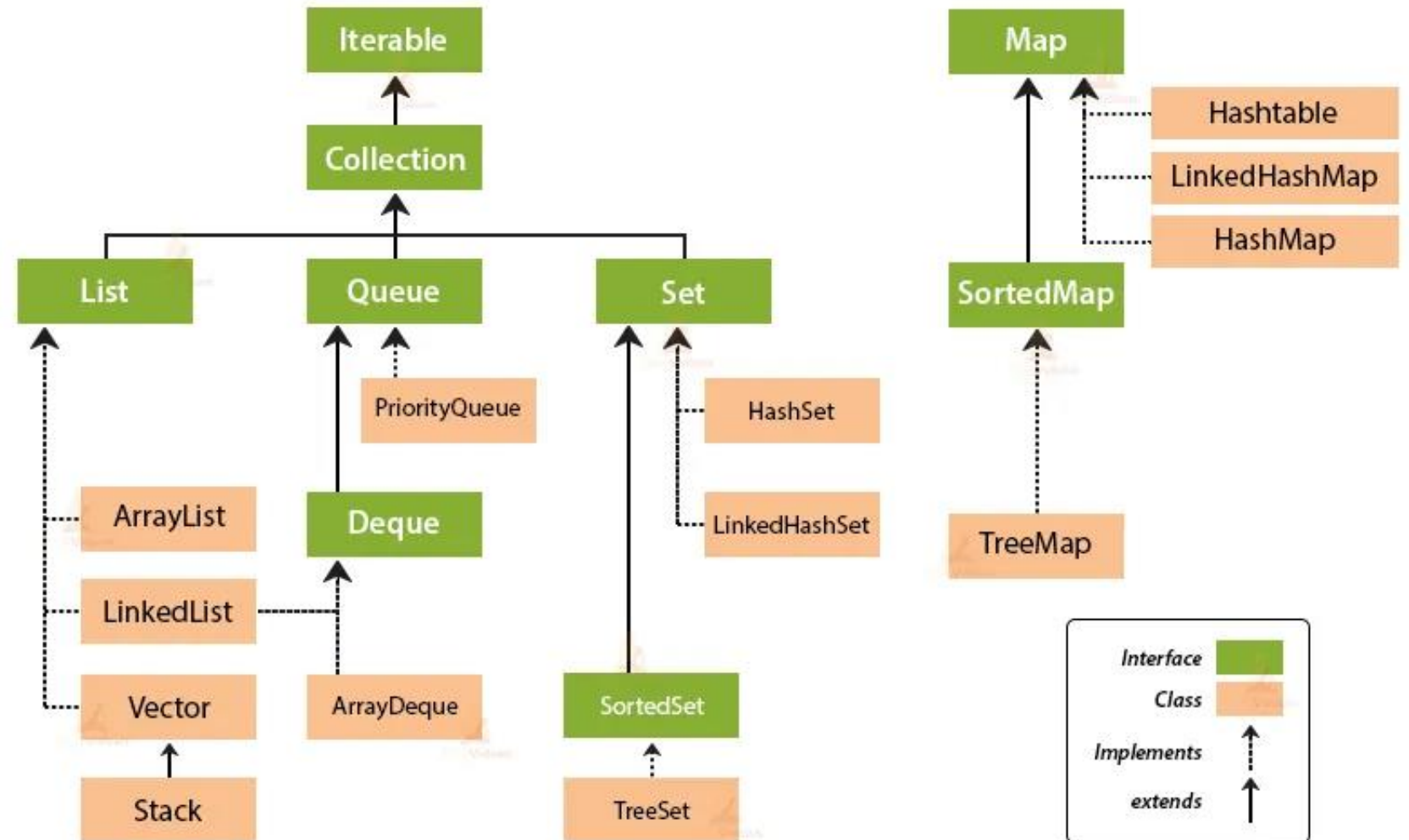
**Ok, ora funziona
sempre!**

Concorrenza nelle Java Collections

The background of the slide features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

Déjà vu...

Collection Framework Hierarchy in Java



Déjà vu

| | ArrayList | Vector | LinkedList | HashMap | LinkedHashMap | HashTable | TreeMap | HashSet | LinkedHashSet | TreeSet |
|------------------------------------|-----------|--------|------------|--|--|-----------|---|---------|---------------|---------|
| Allows Null? | Yes | Yes | Yes | Yes (But One Key & Multiple Values) | Yes (But One Key & Multiple Values) | No | Yes (But Zero Key & Multiple Values) | Yes | Yes | No |
| Allows Duplicates? | Yes | Yes | Yes | No | No | No | No | No | No | No |
| Retrieves Sorted Results? | No | No | No | No | No | No | Yes | No | No | Yes |
| Retrieves Same as Insertion Order? | Yes | Yes | Yes | No | Yes | No | No | No | Yes | No |
| Synchronized? | No | Yes | No | No | No | Yes | No | No | No | No |

Ora sappiamo cosa significhi

Locking strategies nelle collezioni

Le strutture dati synchronized sono **thread-safe**

- ▶ possono essere condivise e usate concorrentemente tra thread diversi **senza rischi** di interferenze
- ▶ **tutti i metodi** nelle implementazioni di tali classi hanno il **modificatore synchronized**
- ▶ si usa quindi un **unico lock** per l'intera struttura dati

Ad esempio, **Vector** e **ArrayList** sono **identiche**, ma

- ▶ **Vector** ha i metodi sincronizzati (thread safe)
- ▶ **ArrayList** ha metodi non sincronizzati (thread unsafe)

Locking strategies nelle collezioni

In un **contesto non concorrente**, meglio avere strutture dati **non synchronized**

- ▶ **più efficienti**... non perdono tempo con i lock

Le strutture dati **non synchronized** possono essere **trasformate in sincronized** tramite un metodo di Collections (che crea una classe **wrapper**):

- ▶ `List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());`

Le strutture dati synchronized usano **coarse-grained** locking (singolo lock). L'API prevede alcune strutture dati thread-safe con locking **fine-grained**:

- ▶ Ad es. **ConcurrentHashMap** è un'implementazione di Map basata su una tabella hash con **un lock diverso per ogni bucket** (maggiore concorrenza)