Appunti di Programmazione Concorrente e Distribuita in Java

Riccardo A.

19 febbraio 2018

Premesse

Questi appunti sono stati scritti per uso personale, quindi non puntano alla formalità, né alla completezza e non sono in nessun modo da considerare una sostituzione allo studio personale del libro e delle lezioni. La teoria e gli esempi proposti si basano sul libro "Oggetti, Concorrenza, Distribuzione" con alcune integrazioni personali dalle slide del corso e dalla documentazione Java.

Riferimenti e fonti

- Libro "Oggetti, Concorrenza, Distribuzione" di S.Crafa
  http://www.libreriaprogetto.it/web/libro/9788874888030.php
- Slides PCD A.A. 2017/2018 - Parte 1
  http://www.math.unipd.it/~abujari/pcd.html
- Slides PCD A.A. 2017/2018 - Parte 2
  https://bitbucket.org/pcd1718/slidelezionimodulo2/src
- Java tutorials
  https://docs.oracle.com/javase/tutorial/java/index.html
- Documentazione Java 7
  https://docs.oracle.com/javase/7/docs/api/
- Documentazione Java 8
  https://docs.oracle.com/javase/8/docs/api/
- Repository GitHub del documento
  https://github.com/ARDbones/PCD

Indice

1 Struttura dei programmi Java **5**
- 1.1 Java e il Main . . . 5
- 1.2 Pacchetti - Package . . . 5
- 1.3 Operatori logici . . . 6
- 1.4 Costrutto *enhanced for* - il for migliorato . . . 6

2 Classi e oggetti **7**
- 2.1 I tipi primitivi . . . 7
- 2.2 Oggetti e riferimenti . . . 7
- 2.3 Assegnazione e riferimenti . . . 8
- 2.4 Stringhe ed Array . . . 8
  - 2.4.1 Stringhe . . . 8
  - 2.4.2 Array . . . 8
- 2.5 Passaggio dei parametri . . . 9
- 2.6 Overloading di metodi . . . 9
- 2.7 Riferimento this . . . 10
- 2.8 Controllo degli accessi a campi e metodi . . . 10
- 2.9 Costruttori . . . 10
- 2.10 Campi e metodi statici . . . 12
- 2.11 Caricamento di classi . . . 12
- 2.12 Inizializzazione dei campi statici . . . 12
- 2.13 Tempo di vita e inizializzazione delle variabili . . . 13

3 Ereditarietà **14**
- 3.1 Tipo Object . . . 14
  - 3.1.1 Operatore di uguaglianza VS equals . . . 14
- 3.2 Ereditarietà e contratti . . . 15
- 3.3 Costruttori nelle sottoclassi . . . 15
- 3.4 Polimorfismo . . . 16

3.5 Conversioni di tipo . . . 17
3.6 Identificazione dei tipi run-time . . . 17
3.6.1 Classe Class . . . 17
3.7 Keyword Final . . . 18
3.8 Suggerimenti per la progettazione di classi . . . 18
3.9 Suggerimenti per la progettazione di gerarchie . . . 18
3.10 Classi wrapper . . . 19

4 Organizzare le classi **20**
4.1 Classi astratte . . . 20
4.2 Interfacce . . . 21
4.3 Classi interne . . . 22

5 Generics e Collezioni **24**
5.1 Generics . . . 24
5.1.1 Vincoli sui tipi delle variabili e Wildcards . . . 25
5.1.2 Generics e Ereditarietà . . . 25
5.1.3 Type erasure . . . 25
5.2 Collezioni . . . 26
5.2.1 Interfaccia List . . . 26
5.2.2 Interfaccia Map . . . 27
5.2.3 Interfaccia Set . . . 28
5.2.4 Classe Collections . . . 30
5.2.5 Classe Arrays . . . 30

6 Eccezioni **32**
6.1 Tipi di eccezioni . . . 32
6.2 Sollevare un'eccezione, blocchi try-catch, il costrutto finally . . . 32
6.3 Clausole throws e overriding di metodi . . . 33
6.4 RuntimeExceptions comuni . . . 33
6.5 Try-with-resources . . . 34

7 Programmazione Concorrente **35**
7.1 Processi e Thread . . . 35
7.2 Creare Thread in Java . . . 36
7.3 Ciclo di vita di un Thread . . . 37
7.4 Thread e le eccezioni . . . 37
7.5 Correttezza di un programma concorrente . . . 37
7.6 Terminazione di un thread . . . 38
7.7 Il problema della condivisione dei dati . . . 38

7.7.1 Lock . . . 38
7.7.2 Wait e Notify . . . 39
7.7.3 Semafori . . . 40
7.8 Dati thread-safe . . . 40
7.8.1 Collezioni concorrenti . . . 41
7.9 Thread Pool e Executors . . . 41
7.10 Callables e Future . . . 42

8 Stream, File e Lambda expressions **43**
8.1 Stream e `java.io` . . . 43
8.2 Stream e file . . . 44
8.3 Serializzazione . . . 45
8.4 Espressioni Lambda . . . 45
8.5 Stream API . . . 46

9 Programmazione Distribuita **47**
9.1 Socket TCP . . . 47
9.2 Datagram . . . 48
9.3 URL . . . 49
9.4 Channel . . . 49
9.5 Modello RMI . . . 49

Capitolo 1

Struttura dei programmi Java

1.1 Java e il Main

Il linguaggio Java è orientato completamente agli oggetti, quindi non ci sono funzioni, solo metodi di classe. Anche il `main` è un metodo di classe, **statico** (non ha oggetto di invocazione), **pubblico** (accessibile da chiunque), con argomento di tipo `String[ ]` e senza valore di ritorno (**void**). Un programma deve avere una classe pubblica con il metodo main, invocato automaticamente all'inizio dell'esecuzione.

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

1.2 Pacchetti - Package

Un file *.java* contiene una classe **pubblica** che ha lo stesso nome del file. Le classi correlate tra loro stanno nello stesso pacchetto. Un pacchetto evita il conflitto di nomi e serve a partizionare i moduli di un programma.

Ogni file che appartiene ad un pacchetto deve avere la dichiarazione come prima istruzione.

```
package nomepacchetto;
```

Il nome completo di una classe è quindi **nomepacchetto.nomeclasse**. Si può creare una gerarchia di pacchetti, ma non da privilegi di accesso tra i pacchetti.

Per usare in una classe un tipo dello stesso pacchetto basta il nome della classe, mentre se si usa un tipo esterno serve il nome completo o si "include" il pacchetto che lo contiene (`import pacchetto.*;`) o si utilizza la "dichiarazione d'uso" (`import pacchetto.classe;`). A differenza di C++, i pacchetti non vengono incluse fisicamente, quindi non influenzano le prestazioni.

1.3 Operatori logici

La valutazione degli operatori logici condizionali avviene per short-circuiting, cioè evitando di controllare il secondo operando se possibile. In particolare:

- `w && z` - se w valuta a **false** allora z non viene valutato;
- `x || y` - se x valuta a **true** allora y non viene valutato.

1.4 Costrutto *enhanced for* - il for migliorato

In Java esiste il for migliorato (o anche foreach) che permette di semplificare le iterazioni di array e collezioni. Si usa ponendo tra parentesi il `tipo_contenuto : nome_contenitore`. Il costrutto itera automaticamente finché sono presenti valori nel contenitore.

```
1 int[] array = {1, 2, 3, 4, 5};
2 for(int valore : array)
3 {
4   // fai qualcosa con i valori, es:
5   System.out.println(valore);
6 }
```

Capitolo 2

Classi e oggetti

2.1 I tipi primitivi

Ogni tipo è una classe, compresi quelli base (es: Integer per int). Per comodità esistono i tipi primitivi non classe: int, long, double, float, boolean, char. I valori di questi non sono oggetti.

Java è fortemente tipato, quindi un'espressione deve avere un valore compatibile, cioè stesso tipo o sottotipo della variabile a cui viene assegnata, secondo la relazione:

```
byte < short < int < long < float < double
```

Non si può convertire implicitamente con perdita di informazioni! È necessario un typecast, ma se la conversione è tra tipi incompatibili (relazione sopra), dà comunque errore (es: int-bool). La conversione tra tipo primitivo e tipo wrapper (es: int-Integer) può avvenire in automatico.

2.2 Oggetti e riferimenti

Un oggetto viene costruito in due passaggi:

1. creazione del valore
2. memorizzazione in una variabile

La variabile oggetto a cui si assegna il valore NON alloca memoria di un oggetto ed è chiamato riferimento. L'oggetto viene creato e allocato con la new + costruttore. Il riferimento è un puntatore all'oggetto e una dichiarazione alloca il puntatore e basta. L'accesso all'oggetto puntato non avviene attraverso la dereferenziazione ma tramite il riferimento.

Se viene usato un riferimento non inizializzato si ottiene l'eccezione `NullPointerException`.

2.3 Assegnazione e riferimenti

L'assegnazione tra riferimenti copia i riferimenti, "spostando il puntatore", e non copia l'oggetto. Se si perde l'accesso ad un oggetto, viene gestito automaticamente dal garbage collector.

2.4 Stringhe ed Array

Stringhe ed array sono tipi classe, non primitivi.

2.4.1 Stringhe

String appartiene a `java.lang` ed è sempre incluso. Le stringhe si possono indicare tra doppi apici: "stringa". In questo caso si dice che è un letterale stringa.

Le stringhe occupano un certo spazio di memoria e si possono vedere come riferimenti a quell'area di memoria; quando viene utilizzata una stringa ci si riferisce al relativo spazio. Con `new String("prova")` si crea un oggetto che rappresenta la stringa. Quindi `String a = "prova";` è diverso da `String b = new String("prova");`.

Due stringhe si possono concatenare con l'operatore +. È sempre possibile convertire un tipo T ad uno String; grazie a questo si può concatenare String+T qualsiasi.

2.4.2 Array

Sono usati per raggruppare valori dello stesso tipo:

```
1 // riferimento = oggetto
2 char[] s = new char[20];
3 // costruisce 20 variabili di tipo char
4 classe[] c = new classe[200];
5 // costruisce 200 riferimenti ad oggetti di tipo
     classe
```

Gli oggetti effettivi vanno creati con le istruzioni `c[i]=new class();`.

Gli array vengono inizializzati in automatico in base al tipo (numeri=0, char='`\u0000`', bool=false, class=null). Si può inizializzare esplicitamente:

```
String[] nomi = {"gigi", "pino"};

classe[] c = {new classe(), new classe()};
```

Limiti di un array: da 0 a `length` (campo implicito dell'array). Java fa un controllo run-time per vedere se un indice su un array appartiene all'intervallo del relativo array. La dimensione dell'array una volta creato è fissa; per cambiare la dimensione si deve creare un nuovo array e "spostare il puntatore", ma si perde il contenuto. È disponibile `System.arraycopy()` per copiare il contenuto degli array, ma solo i riferimenti.

Gli array possono avere più dimensioni:

```
// crea un array di 3 elementi inizializzati a null
int[][] mat = new int[3][]
// inizializza i 3 elementi con un array di 5
   elementi
mat[0] = new int[5];
mat[1] = new int[5];
mat[2] = new int[5];
// risulta un array 3 righe per 5 colonne
```

2.5 Passaggio dei parametri

L'unico modo per passare i parametri ai metodi è per valore, quindi i parametri formali sono inizializzati con copie degli attuali. Se è un tipo primitivo, non c'è side effect, mentre se è di tipo classe si (sull'oggetto, non sul riferimento).

2.6 Overloading di metodi

Più metodi possono avere lo stesso nome, ma deve cambiare il numero e il tipo di parametri. Se cambia solo il tipo di ritorno non funziona.

Il compilatore sceglie prima il metodo con i parametri dello stesso tipo, poi quello dove i parametri formali hanno il minimo supertipo (cioè deve fare meno conversioni) rispetto ai parametri attuali. Se c'è ambiguità, non compila.

```
void fun(double d){}
void fun(float f){}
```

```
3
4 fun(5); // invoca float (int < float < double)
```

2.7 Riferimento this

This è un riferimento all'oggetto d'invocazione e si può usare solo sui metodi non statici. Di solito si usa per passare l'oggetto di invocazione come un argomento di un metodo.

2.8 Controllo degli accessi a campi e metodi

Ci sono 4 modificatori di accessibilità:

- campo/metodo marcato **private** è accessibile solo all'interno della classe
- campo/metodo senza marcatura ottiene accessibilità **package**, cioè è accessibile all'interno del pacchetto in cui è definito (pacchetto!=sottopacchetto)
- campo/metodo marcato **protected** è accessibile dal pacchetto e dalle sottoclassi
- campo/metodo marcato **public** è accessibile da chiunque

Anche una classe può avere diverse accessibilità:

- public
- senza marcature → package
- protected e private solo se sono classi interne

2.9 Costruttori

Comportamento del costruttore:

1. viene allocato l'oggetto e i campi sono inizializzati a zero
2. i campi sono inizializzati secondo i valori delle inizializzazioni esplicite
3. viene eseguito il corpo del costruttore

Il costruttore ha il nome della classe, nessun tipo di ritorno, per default la stessa accessibilità della classe.

Ogni classe ha il costruttore di default senza parametri ma la definizione di un costruttore disabilita lo standard di default. Non esiste il costruttore di copia, ma si usa un apposito metodo clone. Non esiste la lista di inizializzazione, né i valori di default per i parametri.

È utile avere costruttori a più parametri; in Java sono costruttori che si invocano tra loro tramite il `this(n_parametri)`, che deve essere il primo statement del costruttore.

```
public class Dipendente{

    private String nome;
    private int stipendio;

    public Dipendente(String n, int s) {
        nome = n;
        stipendio = s;
    }

    public Dipendente(String n) {
        this(n,0);
    }

    public Dipendente() {
        this(" ");
    }
}
```

L'inizializzazione dei campi può quindi avvenire:

- con i costruttori
- con l'inizializzazione automatica a zero
- con l'inizializzazione esplicita, che può avvenire nel momento della dichiarazione del campo o in un apposito *blocco di inizializzazione*, un blocco dentro la classe ma esterno da metodi

```
// blocco di inizializzazione
{
    nome="esempio";
```

```
    stipendio=0;
}
```

Il blocco di inizializzazione, detto anche d'istanza, viene eseguito ogni volta che viene creato un oggetto della classe. Se ci sono più blocchi viene seguito l'ordine in cui sono dichiarati. I blocchi d'istanza vengono eseguiti prima del costruttore.

2.10 Campi e metodi statici

I campi statici sono condivisi tra gli oggetti della classe; c'è una sola copia in memoria. Sono utili per la comunicazione tra oggetti, per esempio per contare le istanze create. Si usa con il nome della classe: `C.s`.

I metodi statici non hanno oggetto di invocazione e si usano con il nome della classe: `C.m()`.

All'interno della classe non serve richiamare il nome della classe. Metodi e campi statici vengono creati e inizializzati al caricamento della classe.

2.11 Caricamento di classi

Java carica il bytecode di una classe al primo statement che usa la classe. Se viene solo dichiarato un riferimento, non viene caricata la classe.

2.12 Inizializzazione dei campi statici

I campi statici si possono inizializzare tramite il blocco di inizializzazione con lo static davanti e viene eseguito quando la classe viene caricata (quindi una sola volta!). Se una classe ha più blocchi statici, vengono eseguiti sequenzialmente.

```
public static int n = 3;
static {
  // codice che fa qualcosa; potrebbe anche non
   modificare la variabile statica
  n++;
}
```

2.13 Tempo di vita e inizializzazione delle variabili

- **Variabili locali di un metodo**: la variabile deve essere inizializzata, altrimenti da errore; sono allocate nello stack quando dichiarate e deallocate a fine metodo; se è un riferimento, non viene deallocato l'oggetto puntato, a meno che non abbia più riferimenti che lo puntano.
- **Parametri formale**: allocati/deallocati in memoria all'inizio/fine dell'esecuzione del metodo; inizializzati con i valori dei parametri attuali.
- **Variabili di istanza (campi dati)**: vengono creati quando un oggetto viene costruito e esistono finché esiste un riferimento a quell'oggetto; inizializzate come indicato sopra.
- **Variabili di classe (statiche)**: le variabili statiche sono create al caricamento della classe e vengono deallocate a fine dell'esecuzione; inizializzate a zero, poi con il blocco di inizializzazione (se presente).

Capitolo 3

Ereditarietà

Una classe può avere solo una superclasse. Una derivata è indicata da `Class C extends nomeClasseBase`.

3.1 Tipo Object

`Object` è la classe base implicita di ogni classe (non serve l'extends). Le classi ereditano i suoi metodi, tipo:

- `boolean equals(Object obj)` che verifica se due riferimenti puntano allo stesso oggetto; un riferimento è l'oggetto di invocazione, l'altro è parametro passato.
- `String toString()` ritorna l'oggetto di invocazione in formato stringa; la versione di default restituisce il nome della classe e l'indirizzo del riferimento (quindi meglio ridefinirlo per classe).
- `void finalize()` è una sorta di "distruttore" invocato dal garbage collector su un oggetto senza riferimenti; può aver senso ridefinirlo se si vuole rilasciare risorse o per eseguire azioni di clean-up.

3.1.1 Operatore di uguaglianza VS equals

L'operatore == esegue un confronto tra il contenuto di variabili qualsiasi. Se sono riferimenti, ritorna true solo se entrambi si riferiscono allo stesso oggetto. Se entrambi sono null, l'uguaglianza è vera.

L'`equals()` in versione non ridefinita, usato con `ref1.equals(ref2)`, ritorna true se si riferiscono allo stesso oggetto, esattamente come ==. L'`equals()` però in genere si ridefinisce per confrontare gli oggetti e non

i riferimenti in modo da poter dire se due oggetti diversi possano essere considerati "uguali" secondo determinate condizioni.

3.2 Ereditarietà e contratti

Una classe ha tre contratti:

- **pubblico**, per gli utenti della classe, che definisce le funzionalità primarie del tipo
- **protetto**, per chi estende e specializza la classe, che definisce funzionalità disponibili ai sottotipi
- **package**, che definisce funzionalità fornite all'intero pacchetto e permette la cooperazione tra tipi dello stesso pacchetto.

Controllo degli accessi a campi e metodi:

Modificatore	**Stessa classe**	**Stesso package**	**Sottoclasse**	**Universo**
public	Si	Si	Si	Si
protected	Si	Si	Si	No
nessuno (package)	Si	Si	No	No
private	Si	No	No	No

3.3 Costruttori nelle sottoclassi

Le derivate ereditano i campi della superclasse, ma non possono inizializzarli. Nei costruttori devono richiamare il costruttore della base tramite la keyword `super()`. Comportamento del costruttore di una derivata:

1. viene allocata la memoria per tutti i campi, sia propri che ereditati
2. tutti i campi vengono inizializzati a zero
3. viene richiamato il costruttore della superclasse diretta; è un passo ricorsivo e si risale fino a Object
4. vengono eseguite le inizializzazioni esplicite dei campi della derivata
5. esegue il corpo del costruttore

Se `super` non è invocato, il compilatore inserisce una chiamata implicita al costruttore di default della base con `super()`; se quello di default non è disponibile, dà errore.

3.4 Polimorfismo

Principio di sostituzione: sia T1 sottotipo di T2, un oggetto di tipo T1 può essere usato in ogni contesto al posto di un oggetto di tipo T2.
Relazione di sottotipo: T1≤T2 se:

- T1 e T2 sono dello stesso tipo
- T1 è definito come class T1 extends T2 (derivata)
- T1 e T2 sono tipi array, T1=A[ ], T2=B[ ] e A≤B
- Esiste un tipo T3 tale che T1≤T3 e T3≤T2
- T2 è Object

Il tipo statico di un riferimento TS(ref) è il tipo con cui è stato dichiarato; il tipo dinamico del riferimento TD(ref) è il tipo effettivo dell'oggetto a cui si riferisce. Vale sempre TD≤TS.

In Java non c'è distinzione tra ridefinizione e overriding. Tutti i metodi sono implicitamente virtuali se presenti in più classi. Ma un metodo ridefinito:

- deve avere la stessa segnatura (nome, numero e tipo di argomenti, tipo di ritorno uguale o più piccolo)
- non può diventare meno accessibile del metodo originale
- non può sollevare più eccezioni del metodo originale

Binding (o dispatching): staticamente il compilatore controlla se esiste una definizione del metodo usato nella classe del tipo statico, dinamicamente viene scelta la ridefinizione più specifica rispetto al tipo dinamico (quindi con meno conversioni).

Per i metodi statici e i campi dati il dispatching è sempre statico. Anche per i metodi privati (perché non possono essere ridefiniti) e quando viene invocato un metodo della superclasse con la keyword `super`. `Super` in modo simile al `this` si riferisce all'oggetto corrente ma con il tipo della superclasse. Quindi un `super.m()` invoca sempre il metodo della superclasse.

L'overloading dei metodi **non** nasconde tutti i metodi della base, ma solo quelli con la stessa segnatura e tipo di ritorno.

3.5 Conversioni di tipo

Le conversioni sono possibili solo se un tipo è minore dell'altro. Se quello di partenza è più piccolo si chiama upcast, viceversa downcast. L'upcasting è sempre sicuro perché rispecchia il principio di sostituzione.

Il downcasting si usa quando si ha la necessità di usare metodi specifici del tipo dinamico (il *dynamic_cast* di C++); viene fatto esplicitamente con `(tipoTarget)nomeVariabile`. Questa operazione ritorna un riferimento con lo stesso valore ma tipo statico diverso (cioè del tipo target), ma il riferimento originale ha ancora lo stesso tipo statico.

```
1 T1 ref=new T2();
2 T2 nuovoRef=(T2)ref;
3 // TS(ref)=T1, TS(nuovoRef)=T2, TD(entrambi)=T2.
```

La conversione T(ref):

- staticamente compila se TS(ref) e T sono nella stessa gerarchia
- dinamicamente è corretta se TD(ref)≤T, altrimenti lancia l'eccezione `ClassCastException`

3.6 Identificazione dei tipi run-time

Esiste l'operatore booleano `ref instanceof T` che permette di vedere se è possibile effettuare il downcasting; ritorna true se ref è non nullo e TD(ref)≤T.

3.6.1 Classe Class

Ogni classe è rappresentata da un oggetto della classe `Class`. Con `Class forName(String s)` è possibile ottenere l'oggetto che rappresenta la classe di nome `s`.
Il metodo `ref.getClass()` ritorna un oggetto di tipo `Class` che rappresenta il tipo dinamico del riferimento `ref`. Altri metodi:

- `String getName()`
- `Class getSuperclass()`
- `boolean isInterface()`
- `Class[ ] getInterfaces()`

3.7 Keyword Final

`Final` viene usato per indicare qualcosa di costante. Il riferimento `this` è implicitamente `final` (NB: non esistono oggetti costanti).

Un campo dati di tipo primitivo marcato `final` è costante: non si può modificare il valore. Se è statico e `final` deve essere inizializzato esplicitamente, altrimenti è errore; se è solo `final`, deve essere inizializzato esplicitamente o con il costruttore (l'inizializzazione automatica a zero non basta).

Un riferimento marcato `final` si riferisce solo ad uno oggetto fisso, ma l'oggetto può essere modificato. Di solito non si usano campi dati riferimenti `final`.

Gli argomenti di un metodo possono essere marcati `final`: non si può modificare l'argomento; se è un riferimento, si può solo modificare l'oggetto.

Un metodo marcato `final` non può essere ridefinito nelle sottoclassi (usato per motivi di sicurezza o efficienza). Un metodo privato è già implicitamente `final` (ma se viene ridefinito, non è segnato errore).

Una classe marcata `final` non può avere derivate.

3.8 Suggerimenti per la progettazione di classi

- progettare al meglio i contratti della classe (public, protected, package)
- i campi dati è meglio che siano privati, se possibile
- inizializzare i dati esplicitamente, non fidandosi dell'inizializzazione automatica
- raggruppare i campi
- una singola classe non dovrebbe svolgere troppi compiti
- usare identificatori significativi

3.9 Suggerimenti per la progettazione di gerarchie

- le operazioni comuni vanno in una superclasse
- evitare i campi protected
- usare l'ereditarietà solo quando un oggetto del sottotipo è intuitivamente anche oggetto del supertipo

- usare l'ereditarietà solo se tutti i metodi hanno senso nelle classi che li ereditano
- l'overriding di un metodo non deve modificarne il contratto
- limitare il downcast

3.10 Classi wrapper

Sono classi che racchiudono tipi primitivi per poterli trattare come oggetti.
Es: Integer $\rightarrow$ int

Capitolo 4

Organizzare le classi

La relazione tra classi si può distinguere tra "è un" (*is-a*) ed "ha un" (*has-a*), ovvero ereditarietà di tipo e ereditarietà di implementazione. Quella di tipo si ha quando si estende effettivamente la classe e si sfrutta il polimorfismo, mentre quella di implementazione si ha quando è comodo riutilizzare il codice di un'altra classe.

4.1 Classi astratte

Si può creare una classe astratta con la keyword `abstract`. Questo permette di dichiarare metodi astratti (con `abstract`) che posso non definire. In compenso, non posso creare oggetti di quel tipo, ma si possono creare dei riferimenti che saranno sicuramente polimorfi.

Se una classe contiene un metodo astratto, la classe va marcata astratta o si ha un errore. La classe astratta può avere dei campi dati; in questi casi viene definito un costruttore astratto che verrà richiamato dalle derivate. Dei metodi concreti possono invocare metodi astratti che a runtime verranno invocati virtualmente nel modo corretto. Una classe astratta non può avere metodi statici/final astratti perché richiedono un dispatching statico.
Una sottoclasse rimane astratta finché non definisce tutti i metodi astratti della base.

Di solito le classi astratte si usano per l'**ereditarietà di implementazione** perché permettono di scrivere una struttura riutilizzabile dalle derivate.

4.2 Interfacce

Le interfacce si usano per l'**ereditarietà di tipo**. Le interfacce definiscono funzionalità su un tipo senza indicare come è implementato quel tipo. Si dichiara con la keyword `interface Nome{...}` e ha regole specifiche:

- si può dichiarare solo campi marcati public static e final, e metodi astratti pubblici (`abstract` è implicito)
- non può avere metodi statici (da Java 8 è possibile)
- non sono istanziabili, neanche come sottooggetti, quindi non hanno il costruttore
- può avere accessibilità public o package

Un'interfaccia non può essere istanziata, quindi l'unico modo per usare il tipo dell'interfaccia è implementandola con una classe che ne definisce **tutti** i metodi. La classe diventa sottotipo, quindi è possibile usare un riferimento dell'interfaccia in maniera polimorfa. La classe implementa con la keyword `implements`.

Le interfacce permettono di implementare una sorta di "ereditarietà multipla" di tipo; infatti una classe può implementare più interfacce, ma estendere solo una classe. Può farlo contemporaneamente, prima `extends`, poi `implements` (`extends A implements B, C`).

Le interfacce possono estendersi tra loro, formando gerarchie; è possibile fare dei cast e instanceof. Non hanno un'interfaccia base comune, ma vale la relazione Interface≤Object.

Relazione di sottotipo aggiornata: T1≤T2 se vale

- T1 e T2 sono lo stesso tipo
- T1 è definito come T1 extends T2, T1 implements T2
- T1 e T2 sono tipi array, T1=A[ ], T2=B[ ] e A≤B
- esiste un T3 tale che T1≤T3 e T3≤T2
- T2 è di tipo Object, anche se T1 è interfaccia

Le conversioni cambiano leggermente. Se un riferimento ref ha come TS un'interfaccia, è sempre possibile convertire esplicitamente ref a qualsiasi classe, senza dare errori a compile time. Può però dare ClassCastException dinamicamente se TD(ref) non è ≤T.

Le interfacce hanno un grosso problema: non è possibile aggiungere metodi e funzionalità successivamente perché invalidano tutte le classi che hanno implementato l'interfaccia. Da Java 8 esistono i metodi di default (**defender methods** [1]) che servono a risolvere questo problema. Questi metodi hanno un'implementazione di default della funzione così da permettere di far evolvere un'interfaccia senza invalidare le classi che l'hanno implementata. Se una classe eredita due implemetazioni del metodo di default, verrà scelta quella della classe concreta più specifica.

Da Java 8 è possibile creare metodi statici in un'interfaccia, ma non possono essere ridefiniti dai sottotipi (vanno definiti nella prima dichiarazione).

4.3 Classi interne

Le classi interne sono classi definite dentro ad altre classi, come membro (classi interne di istanza o statiche) o dentro un blocco di codice (classe interna locale). Un tipo innestato è parte del tipo che lo racchiude ed entrambe possono accedere a tutti i membri dell'altra classe (anche privati!).

Le classi interne possono essere private in modo da non essere accessibili dalle altre classi del package. La classe interna non è sottotipo della classe contenitore.
Nota per gli esempi: Inner = tipo interno, Outer = tipo esterno.

Classe interna di istanza È considerata un membro, quindi ha un qualsiasi marcatore di accesso e può essere static. Se la classe non è statica, un oggetto della classe interna ha un riferimento "`outerThis`[2]" all'oggetto della classe esterna che lo ha creato.

La classe interna può accedere a tutti i membri della classe esterna tramite l'`outerThis`, ovvero `Outer.this.membroEsterno`. La classe contenitore può accedere a tutti i membri della classe interna tramite oggetti della classe interna.

Per costruire un oggetto di tipo Inner è necessario usare la classe esterna. Se la classe non è statica, si usa `this.new Inner()`, mentre se è statica `ref.new Inner()`, dove ref è di tipo Esterno. Se la classe interna è accessibile dall'esterno, si può usare il tipo Inner anche al di fuori del contenitore tramite `nomeClasseEsterna.Inner`. Non contiene membri statici.

```
1 Outer ref = new Outer();
```

[1] https://dzone.com/articles/introduction-default-methods
[2] Nota: non esiste letteralmente la keyword outerThis!

```
// se non e' statica
Outer.Inner InnerObj = this.new Inner();
// se e' statica
Outer.Inner InnerObj = ref.new Inner();
```

La classe interna può implementare un'interfaccia; in particolare se fatto privato viene nascosta dall'esterno e l'implementazione delle classi esterne non dipende da quella dell'interfaccia. Allo stesso modo è possibile fare classi interne che estendono una classe, creando una sorta di ereditarietà multipla: la classe esterna estende una classe D1, mentre la classe interna estende D2; in questo modo la classe esterna può accedere anche ai campi di D2.

Classe interna statica A volte non serve che la classe interna faccia riferimento ad un oggetto della esterna, quindi torna comodo dichiararla statica. In questo caso, non esiste l'`outerThis` e non serve un oggetto della classe esterna per crearne uno di quella interna. Però la classe interna non può accedere ai campi non statici della classe esterna.

Classe interna anonima Sono classi interne senza nome, di solito usate per implementare interfacce. Si possono creare oggetti di quella classe, ma non possono essere usati come tipi statici; non ha costruttori (a parte quello di default).

```
public interface Interfaccia {...}

class C {
    // metodo che ritorna un oggetto di un sottotipo anonimo a Interfaccia
    public Interfaccia ritornaInterfaccia()
    {
        return new Interfaccia() {...}; // serve il punto e virgola!
    }
}
```

Classi innestate in interfacce Anche le interfacce possono contenere classi e sono tutte statiche.

Capitolo 5

Generics e Collezioni

5.1 Generics

In Java sono presenti i template di funzione e di classe, e si chiamano funzioni/classi generiche. La definizione utilizza un tipo parametrico T che deve essere specificato quando si vuole usare un'istanza di quel template[1] di classe/interfaccia, mentre per i metodi viene dedotto dal compilatore. Il tipo T non può essere un primitivo; vengono accettati solo tipi classe e interfaccia. Per definire un generic si usano le parentesi angolari:

```
1 public class Classe<T,Z> { }
2
3 public interface Interfaccia<T> { }
4
5 public <T> void f(T[] array) { }
```

Un metodo statico di una classe generica che utilizza un tipo parametrico deve essere un metodo generico o si ha un errore di compilazione.

```
1 public class Classe<T>{
2   public static <T> void m(T[] arg) { }
3 }
```

Se un generic viene esteso o implementato, deve essere indicato il tipo esplicitamente nella segnatura.

[1] Ndr: ognivolta che verrà usato "template" si intende un qualsiasi Generic.

5.1.1 Vincoli sui tipi delle variabili e Wildcards

È possibile porre dei vincoli sui tipi accettabili dal template per evitare errori logici (es: String in un template che confronta numeri). I vincoli sono espressi all'interno delle parentesi angolari indicando quali classi estende:

```
public static <T extends C> min(T[ ] arg){ }
// dove C e' la classe/interfaccia base
```

Nell'esempio è sensato usare l'interfaccia `Comparable` al posto di C.
Se si vuole che T sia sottotipo di più tipi, si usa `<T extends A & ... & C>`. Nell'elenco devono andare prima la classe, poi le interfacce.

Un altro modo per porre dei vincoli e l'utilizzo di **wildcards**, cioè dei caratteri jolly che hanno un significato implicito.

Nome	Sintassi	Significato
upper bounded	? extends B	qualsiasi sottotipo di B
lower bounded	? super B	qualsiasi supertipo di B
unbounded	?	qualsiasi tipo

5.1.2 Generics e Ereditarietà

Se un tipo è l'estensione di un altro, non comporta che lo siano anche le classi generiche istanziate.

```
class Dirigente extends Dipendente {...}
class Agente extends Dipendente {...}
ArrayList<Dirigente> a = new ArrayList<Dirigente>();
ArrayList<Dipendente> b = a; // NON FUNZIONA
Dipendente dip = new Agente();
b.add(dip); // OK
```

Ad esempio non è possibile usare un `ArrayList<Double>` al posto di un `ArrayList<Object>`, anche se Double$\leq$Object.

5.1.3 Type erasure

La JVM attua il *type erasure*, ovvero per ogni generic sostituisce il tipo parametrico con il primo bound o con Object se è unbounded. In questo modo le verifiche avvengono a compile time. A causa del type erasure di Java non è possibile:

- costruire oggetti che sono istanze del tipo parametro - `new T()`
- costruire un array del tipo parametro - `new T[k]`
- usare il tipo di una classe generica per definire campi dati statici, metodi statici o classi
- fare overloading di metodi dove cambia solo il tipo parametrico

5.2 Collezioni

Le collezioni in Java corrispondono ai contenitori, dove un oggetto rappresenta un insieme di oggetti. Mentre gli array in Java hanno una lunghezza fissa, un tipo `Collection` è un array dinamico di `Object`.

Esistono delle apposite collezioni per la concorrenza introdotte da Java 7 che sono considerate *thread-safe*.

5.2.1 Interfaccia List

Rappresenta una sequenza di elementi.

ArrayList Rappresenta un array ridimensionabile. Non è thread-safe. Alcune operazioni:

- `size`
- `isEmpty`
- `get`
- `set`
- `itarator`
- `listIterator`
- `add`
- `contains`
- `indexOf` - simile a contains, ma ritorna l'indice dell'elemento (se presente)
- `remove`

Costruttori:

- `ArrayList()` - lista con capacità iniziale a 10;
- `ArrayList(Collection c)` - costruisce una lista contenente gli elementi della collezione passata;
- `ArrayList(int capacità)` - costruisce una lista vuota di con una certa capacità iniziale.

Nota bene: la capacità non implica che venga creato un oggetto nullo! Quindi se cerco di ottenere un elemento con il `get(i)` viene lanciata l'eccezione `IndexOutOfBoundsException` se **`i`** non corrisponde ad un elemento inizializzato.

LinkedList Rappresenta una lista doppiamente linkata. Non è thread-safe. Presenta le stesse operazioni di `ArrayList` e alcune operazioni in più:

- `offer` - aggiunge un elemento in coda;
- `peek` - ritorna il primo elemento della lista;
- `pop` - ritorna e rimuove il primo elemento della lista;
- `push` - inserisce un elemento in fronte.

Costruttori:

- `LinkedList()` - lista con capacità iniziale a 10;
- `LinkedList(Collection c)` - costruisce una lista contenente gli elementi della collezione passata.

5.2.2 Interfaccia Map

Rappresenta un array associativo chiave-valore.

HashMap Insieme di coppie chiave-valore non ordinati. Non è thread-safe. Alcune operazioni:

- `clear`
- `clone`

- `containsKey`
- `containsValue`
- `entrySet`
- `keySet`
- `get`
- `put`
- `size`
- `remove`

Costruttori:

- `HashMap()` - costruisce una mappa con capacità iniziale a 16 e fattore di carico a 0.75;
- `HashMap(int capacità)` - costruisce una mappa vuota di con una certa capacità iniziale e fattore di carico 0.75;
- `HashMap(int capacità, float carico)` - costruisce una mappa vuota di con una certa capacità iniziale e un certo carico;
- `HashMap(Map m)` - costruisce una mappa contenente gli elementi della collezione passata;

Nota: a volte può essere essere utile ridefinire `hashCode()` nel caso di utilizzo di chiavi custom.

TreeMap Insieme di coppie chiave-valore ordinati secondo la chiave tramite un albero rosso-nero. Non è thread-safe.

5.2.3 Interfaccia Set

Collezione di elementi che non può contenere duplicati.

HashSet Implementa un Set sfruttando le caratteristiche di un HashMap. Non è thread-safe. Alcune operazioni:

- `add`
- `clear`
- `clone`
- `contains`
- `isEmpty`
- `size`
- `remove`

Costruttori:

- `HashSet()` - costruisce un insieme vuoto con capacità iniziale a 16 e fattore di carico a 0.75;
- `HashSet(int capacità)` - costruisce un insieme vuoto di con una certa capacità iniziale e fattore di carico 0.75;
- `HashSet(int capacità, float carico)` - costruisce un insieme vuoto di con una certa capacità iniziale e un certo carico;
- `HashSet(Collection c)` - costruisce un insieme contenente gli elementi della collezione passata;

TreeSet Implementa un Set ordinato secondo l'ordine naturale degli elementi. Sfrutta l'albero rosso-nero. Non è thread-safe. Alcune operazioni:

- `add`
- `remove`
- `ceiling`
- `first`
- `higher`
- `last`

- `subSet`

Costruttori:

- `TreeSet()` - costruisce un insieme vuoto ad albero;
- `TreeSet(Comparator comp)` - costruisce un insieme vuoto di con una certa capacità iniziale e fattore di carico 0.75;
- `TreeSet(SortedSet s)` - costruisce un insieme vuoto con ordine ed elementi dell'insieme passato;
- `TreeSet(Collection c)` - costruisce un insieme ad albero contenente gli elementi della collezione passata;

5.2.4 Classe Collections

`Collections` è una classe che ha solo metodi statici e generici per agire sulle collezioni. Alcuni metodi[2]:

- `boolean addAll(Collection c, T... elements)` - aggiunge alla collezione gli elementi (più di uno);
- `int binarySearch(List L, T key)` - cerca nella lista l'argomento key;
- `void fill(List L, T obj)` - rimpiazza tutti gli elementi della lista con l'elemento obj;
- `void sort(List L)` - ordina gli elementi in ordine ascendente;
- `void copy(List Out, List In)` - copia gli elementi dalla lista **In** alla lista **Out**.

5.2.5 Classe Arrays

La classe `Arrays` offre metodi per operare su strutture lineari (array e liste). Alcuni metodi[3]:

- `List asList(T... a)` - ritorna una lista dall'array passato;
- `int binarySearch(...)` - ricerca l'oggetto key nel range specificato;

[2]Ndr: la segnatura dei seguenti metodi è stata semplificata; leggere la documentazione Java per sapere la segnatura completa.

[3]Ndr: vedi nota sopra.

- `T[] copyOf(T[] originale, int newLength)` - per ridimensionare un array;
- `String deepToString(Object[] a)` - un `toString` sui contenuti;
- `boolean deepEquals(Object[] a, Object[] b)` - un `equals` sui contenuti;
- `void sort(Object[] a)` - ordina gli elementi.

Capitolo 6

Eccezioni

6.1 Tipi di eccezioni

In Java, errori ed eccezioni sono oggetti, appartenenti a classi derivate da **Throwable**, la superclasse base. Il tutto si può distinguere tra eccezioni controllate e non controllate; le prime sono casi anomali gestibili, le seconde sono errori logici irrimediabili. In particolare:

- **Error** è la sottoclasse degli errori severi/fatali, di solito **non** gestibili; terminano il programma (es: out of memory).
- **RuntimeException** racchiude gli errori logici che avvengono runtime, **non** gestibili perché sono effettivamente errori logici che richiedono una modifica del codice (es: divisione per zero, nullpointer, ecc).
- le rimanenti **Exception** vanno sempre gestite, o viene segnalato un errore a compile time.

Si può usare un'eccezione definita dall'utente creando una classe che estende `Exception`.

6.2 Sollevare un'eccezione, blocchi try-catch, il costrutto finally

Un'eccezione viene lanciata dal `throw` creando un oggetto del tipo eccezione che si vuole lanciare. La `throw` blocca l'esecuzione e cerca il codice che deve gestirla.

Se un metodo non gestisce l'eccezione al suo interno, è necessario indicare nella segnatura i tipi di eccezione che lancia.

```
void m() throws Exc {
    try {throw new Exc("errore");}
}
```

Il codice che può sollevare eccezioni va in un blocco `try` e l'eccezione viene gestita dal primo `catch` di tipo compatibile. L'ordine delle clausole è da sottoclasse a superclasse, altrimenti la superclasse prende tutte le eccezioni. Se non si trova la `catch` adatta, l'eccezione viene rilanciata al metodo che ha invocato il metodo che ha lanciato l'eccezione; se non viene gestita, viene terminato il programma. Una `catch` non può catturare più eccezioni in relazione di tipo tra loro.

```
catch(EccezioneBase B | EccezioneDerivata D){}
// Il compilatore segna un errore
```

Al termine del blocco try-catch è possibile inserire il blocco `finally`, un blocco di codice che viene sempre eseguito, anche se nel `try` è stato eseguito un return. È un blocco di codice protetto, che esegue sempre, ed è utile per rilasciare le risorse.

6.3 Clausole throws e overriding di metodi

Un metodo può non gestire l'eccezione al suo interno; in questo caso è necessario indicarlo nella segnatura con `throws Eccezione`. Quando si hanno metodi ridefiniti con `throws` nella segnatura, il metodo ridefinito deve lanciare eccezioni che sono sottotipo del precedente e non deve avere più eccezioni.

6.4 RuntimeExceptions comuni

- `ArithmeticException`: divisione per zero;
- `NullPointerException`: riferimento nullo;
- `ClassCastException`: cast dinamico fallito;
- `ArrayIndexOutOfBoundsException`: tentativo di accedere ad un elemento oltre i limiti di un array.

6.5 Try-with-resources

Il `try-with-resources` è stato introdotto da Java 7 e serve per semplificare il normale flusso try-catch-finally quando si usano degli stream.

```
try(InputStream in = new FileInputStream("Input");
OutputStream out = new FileOutputStream("Changed-
    Input")){
// codice che lavora con lo stream
}
```

Con questa struttura le risorse vengono automaticamente chiuse quando si esce dal blocco try. L'eventuale eccezione lanciata verrà passata al chiamante.

Capitolo 7

Programmazione Concorrente

7.1 Processi e Thread

Ci sono due tipi di multitasking:

- cooperativo, in cui i programmi in esecuzione vengono interrotti solo quando sono disposti a cedere la CPU;
- preemptive, dove il sistema operativo interrompe i processi senza consultarli, garantendo l'esecuzione a tutti (preferibile, ma più complessa da gestire).

Processo: programma in esecuzione, rappresentato dal proprio codice + proprio program counter + proprio spazio di indirizzamento.
Thread: singolo flusso di controllo all'interno di un processo. Un processo può avere più thread. Un programma è multithreaded quando consiste di più thread concorrenti.

Condizioni di Coffman Sono quattro condizioni che determinano un *deadlock* se sono tutte e quattro presenti:

- mutua esclusione
- attesa circolare
- accumulo di risorse
- impossibile rilasciare le risorse (no preemptive)

7.2 Creare Thread in Java

Anche i thread sono oggetti della classe `Thread`. Ci sono due modi per creare un thread: estendere `Thread` o implementare `Runnable`.

```
1
2  class C extends Thread{
3      public void run(){ ..codice thread.. }
4      // override
5  }
6
7  C c=new C();
8  c.start(); // solo con questa invocazione viene
       effettivamente attivato il thread dalla JVM
9
10 class C implements Runnable{
11     public void run(){ ..codice thread.. }
12     // override
13 }
14
15 C c=new C();
16 Thread t=new Thread(c);
17 t.start(); // solo con questa invocazione viene
       attivato il thread
```

Differenza tra i due Un oggetto `Thread` rappresenta il meccanismo che esegue l'attività logica del thread, mentre un oggetto `Runnable` rappresenta l'attività logica in sé. Implementare `Runnable` è utile a causa del vincolo di ereditarietà singola, mentre estendere `Thread` è utile quando si usano altri metodi di `Thread`. Inoltre `Thread` permette di usare direttamente `this` per riferirsi al thread, mentre `Runnable` necessita il metodo statico `currentThread()`.

I thread possono essere dichiarati senza riferimento se usati direttamente come oggetto di invocazione. Questi oggetti non sono preda del garbage collector perché esiste sempre un riferimento implicito ad oggetti di tipo `Thread`.

`Thread` possiede un costruttore ad un parametro di tipo stringa che permette di dare un nome ad un thread. Un nuovo thread è effettivamente attivato solo dopo l'invocazione di `start()`.

NB: anche il main ha un suo thread logico, detto main thread. La terminazione del main thread non implica la terminazione dell'esecuzione del programma, che continua finché ci sono thread attivi.

7.3 Ciclo di vita di un Thread

All'interno del ciclo di vita di un thread è possibile distinguere 4 stati:

- **new**, in cui il thread viene creato con l'istanziazione dell'oggetto
- **runnable**, quando il thread è virtualmente in esecuzione
- **not runnable**, quando il thread è bloccato in attesa di qualcosa (I/O, sleep, lock, ecc)
- **terminated**, quando il thread termina l'esecuzione con l'uscita dal metodo run

Il metodo `sleep(int millisecondi)` mette un thread in stato not runnable per un tempo indicato in millisecondi.

7.4 Thread e le eccezioni

Le eccezioni nel metodo run se sono *controllate* vanno gestite all'interno del metodo, mentre se è *non controllata* possono esserci dei problemi: se non viene gestita, viene sollevata l'eccezione, interrotto il thread, ma l'esecuzione degli altri thread prosegue, quindi potrebbe venire ignorata.

Si può installare nel thread un gestore delle eccezioni *non controllate* di tipo `Thread.UncaughtExceptionHandler`, passandolo al metodo `setUncaughtExceptionHandler()`.

7.5 Correttezza di un programma concorrente

La correttezza in un programma concorrente non è facile da assicurare. Deve essere corretto l'output, si deve garantire che tutti i thread abbiano la possibilità di essere eseguiti e assicurare che il programma non deteriori le prestazioni impegnando inutilmente la CPU. La gestione dell'ordine di esecuzione dei thread dipende dal sistema operativo.

È possibile influenzare lo scheduling tramite la priorità dei thread e il metodo `yield()`. Un thread ottiene la stessa priorità del thread che l'ha creato, ma `setPriority()` permette di assegnare una diversa priorità. Se

due thread sono nello stato runnable, viene eseguito quello con priorità maggiore. Il metodo statico `yield()` fa abbandonare la CPU dal thread che la sta utilizzando per permettere ad altri thread con la stessa priorità di usarla. Non è assicurato che il SO segua questi parametri; non bisogna basare la correttezza di un programma sulle priorità o su `yield`, ma usare piuttosto `sleep` o `wait`.

7.6 Terminazione di un thread

Il metodo `join()` permette di far attendere un thread la terminazione di un altro thread.
Il metodo `interrupt()` permette di interrompere un thread, assegnandogli lo stato di interruzione tramite un flag booleano.
Tramite `Thread.currentThread().isInterrupted()` è possibile sapere il valore del flag e decidere come reagire all'interruzione, se concludere il `run()` o ignorare la richiesta. Se il thread era nello stato not runnable, genera l'eccezione `InterruptedException`.

Ci sono dei thread che lavorano in background, detti demoni. Questi hanno un metodo run con un loop infinito che attende richieste da altri thread. È possibile creare un thread demone con `t.setDaemon(true)` prima dello `start()`. I thread demoni non mantengono attivo il programma: se sono rimasti solo demoni il programma viene terminato.

7.7 Il problema della condivisione dei dati

Due thread concorrenti possono accedere agli stessi dati; c'è bisogno di gestire gli accessi per non danneggiare i dati condivisi. Per ogni oggetto condiviso, vanno individuate le sezioni critiche e sincronizzate le entrate dei thread in queste sezioni in modo che in ogni istante un solo thread starà eseguendo le istruzioni della propria sezione critica.

7.7.1 Lock

Un metodo è quello dei **lock**, che indicano che l'oggetto sta venendo usato nella sezione critica. Le sezioni critiche vanno racchiuse nei blocchi `synchronized (exp) {...}`, dove **exp** è l'oggetto per cui il codice nel blocco è una sezione critica. Un thread che arriva al blocco richiede il lock:

- se lo ottiene, blocca l'accesso finché non termina il codice, poi rilascia il lock; viene rilasciato anche se interrotto da un'eccezione

- se non lo ottiene, viene sospeso e messo in coda finché non si libera

Se un thread viene interrotto e perde la CPU (NB: è diverso dall'interruzione per un'eccezione), mantiene il lock finché non termina l'esecuzione, bloccando tutti i thread che vorrebbero accedere al blocco synchronized dell'oggetto. Quando un thread rilascia un lock dopo un blocco synchronized, si instaura una relazione *happens-before* tra il rilascio del lock e il tentativo di riacquisirlo successivamente.

Se un intero metodo rappresenta una sezione critica, si può usare synchronized nella dichiarazione del metodo, ma il marcatore non fa parte della segnatura e non viene ereditato. Le richieste di sincronizzazione fanno parte dell'implementazione, non dell'interfaccia di una classe, quindi una interfaccia non può avere metodi synchronized.

I lock di Java sono rientranti, cioè un thread può acquisire e rilasciare più volte il lock su uno stesso oggetto.

Con Java 5 è stata introdotta l'interfaccia `Lock` e alcune classi che la implementano, tra cui `ReentrantLock`, che permette di controllare in maniera esplicita e manualmente le condizioni di blocco e sblocco della sezione critica. Alcuni metodi:

- `void lock()`
- `void unlock()`
- `void tryLock()`

L'attesa su un lock può essere organizzata in base a delle condizioni, in modo da creare code diverse. Si usa l'interfaccia `Condition`. Alcuni metodi:

- `void await()`
- `void signal()`
- `void signalAll()`

Il controllo manuale è molto pericoloso e può portare facilmente a deadlock.

7.7.2 Wait e Notify

Un'alternativa al synchronized è l'utilizzo di `wait()` che mette in attesa un thread finché un altro thread non lo riattiva tramite un `notify()`.

7.7.3 Semafori

Si può controllare un insieme omogeneo di risorse tramite i semafori; sono simili ai lock, ma tengono un conteggio e non un semplice occupato/libero. Un semaforo è un oggetto della classe `Semaphore`. Il costruttore `Semaphore(int permits, boolean fair)` permette di indicare il numero di permessi e la politica di gestione: se è fair (quindi true), l'ordinamento dei thread è garantito secondo FIFO. Conviene usare un semaforo fair quando regola l'accesso ad un insieme di risorse; in casi diversi è più efficiente non fair. Metodi:

- `void acquire()` - acquisisce un permesso e viene decrementato il numero totale
- `void release()` - rilascia un permesso, incrementandone il valore
- `void tryAcquire()` - prova ad ottenere un permesso; ritorna subito false se non ha ottenuto il permesso ed è in grado di violare la fairness del semaforo

Un semaforo può essere rilasciato da un thread diverso da quello che lo ha acquisito (nei lock non si può).

Eccezioni I metodi di `Semaphore` possono lanciare:

- `InterruptedException` - se il thread viene interrotto durante l'attesa
- `IllegalArgumentException` - se il parametro è negativo

7.8 Dati thread-safe

Per condividere dati tra più thread sono necessari dati thread-safe. Se servono delle semplici variabili, per esempio per un contatore, si possono usare le **variabili atomiche**:

- `AtomicInteger`
- `AtomicLong`
- `AtomicReference`
- `AtomicIntegerArray`
- `AtomicLongArray`

- `AtomicReferenceArray`

In queste classi la modifica ai valori sono atomiche e thread-safe.

Esiste la keyword `volatile` per indicare che un dato dovrà essere sempre letto dalla memoria e mai dalla cache. Il suo uso va fatto con molta attenzione.

C'è la possibilità in opposizione alle variabili atomiche di definire variabili indipendenti per ciascun thread. Queste sono oggetti della classe `ThreadLocal`, per i quali esiste una copia differente e indipendente per ogni thread che attraversa la sua dichiarazione.

7.8.1 Collezioni concorrenti

Oltre alle variabili atomiche, esistono le collezioni ottimizzate per la concorrenza. Alcuni esempi:

- `ConcurrentMap`
- `ConcurrentHashMap`
- `BlockingQueue`

7.9 Thread Pool e Executors

Come già detto, c'è una differenza tra l'attività logica da eseguire concorrentemente (Task → oggetto Runnable) e il meccanismo che esegue l'attività concorrentemente (Thread). Di solito, prima di suddivide il lavoro in task, poi si pensa alla loro politica di esecuzione.

Creare un nuovo thread può essere costoso, oltre ad essere difficile da gestire bene. La soluzione è cedere una parte del controllo al sistema tramite i Thread pool e gli `Executors`.

I Thread pool contengono un insieme di thread pronti, ognuno con una propria coda di task, cioè di oggetti `Runnable`. Ogni thread guarda la propria coda ed invoca il metodo run per l'oggetto presente e alla fine dell'esecuzione invece di terminare verifica se ci sono altri oggetti in coda. Se la coda è vuota, il thread "ruba" (*work stealing*) una task dalle altre code o si mette in attesa.

Tramite gli oggetti della classe `Executors` è possibile istanziare dei Thread pool. Alcuni tipi:

- `FixedThreadPool` - insieme di thread di dimensione fissa

- `CachedThreadPool` - insieme di dimensione variabile in base a quanti task sono da eseguire
- `ScheduledThreadPool` - esegue le task con una temporizzazione
- `SingleThreadExecutor` - usa un solo thread
- `WorkStealingPool` - usa tutti i processori disponibili

Tramite il metodo `execute(Runnable r)` viene gestita l'esecuzione delle task da parte dell'Executor.

7.10 Callables e Future

L'interfaccia `Callable<T>` permette di definire delle task che producono un risultato di tipo T. Si deve implementare l'interfaccia contenente il solo metodo `T call()`.

L'uso delle Callables è strettamente legato a quello dell'interfaccia `Future`. Infatti per eseguire una Callable è necessario utilizzare un ExecutorService, il quale ha un metodo `submit()` per gestire la Callable e ritorna un oggetto Future. Future è un contenitore che viene riempito con un valore T il cui calcolo sta avvenendo in parallelo. Una `get()` sull'oggetto Future ritorna il contenuto se è stato calcolato, altrimenti rimane in attesa fino al termine del calcolo.
Il metodo `invokeAny()` ritorna il risultato di una task che è terminata con successo; `invokeAll()` li ritorna tutti, `invokeFirst()` solo il primo.

Un ExecutorService rimane sempre in attesa di task, quindi una volta finito va terminato con il metodo `shutdown()`.

Capitolo 8

Stream, File e Lambda expressions

8.1 Stream e `java.io`

In Java le operazioni di input e output sono trattate considerando i dati come flusso (stream). Lo stream é un'astrazione dei dispositivi di input/output: si apre lo stream, si leggono/scrivono i dati, si chiude lo stream.

In Java gli stream sono istanze di classi raccolte nel package `java.io`. Il pacchetto contiene due gerarchie di classi in base al tipo di flusso:

- flusso di caratteri, usati per la lettura/scrittura di testo
- flusso di byte, usati per la trasmissione di dati in codice binario

Alla base delle gerarchie si trovano delle classi astratte con le operazioni di base:

```
1  class Reader{
2    int read()
3    int read(char[] buff)
4    abstract int read(char[] buff, int off, int cnt)
5    int skip(long count)
6    abstract void close()
7    boolean ready()
8  }
9
10 class Writer{
11   void write(char b)
```

```
12    void write(String r)
13    abstract void write(char[] buff, int off, int cnt)
14    abstract void close()
15    abstract void flush()
16  }
17
18  class InputStream{
19    abstract int read()
20    int read(byte[] buff)
21    int read(byte[] buff, int off, inf cnf)
22    int skip(long count)
23    void close()
24    boolean available()
25  }
26
27  class OutputStream{
28    abstract void write(byte b)
29    void write(byte[] buff)
30    void write(byte[] buff, int off, int cnt)
31    void close()
32    void flush()
33  }
```

Alcuni gruppi di classi che estendono le precedenti:

- flussi `Filter`, che hanno un qualche filtro
- flussi `Buffered`, che aggiungono la presenza di un buffer per ridurre gli accessi ai dispositivi di input/output
- flussi `Piped`, che sono coppie di flussi in cui si legge in uno e si scrive nell'altro

I flussi standard (tastiera e video), sono flussi di byte.

8.2 Stream e file

Quando si trasmette con un file, serve la classe adeguata. Per i caratteri si usa `FileReader` e `FileWriter`, mentre per i byte si usa `FileInputStream` e `FileOutputStream`. L'accesso al file può essere sequenziale o diretto.

8.3 Serializzazione

I dati trasmessi in input/output a volte possono essere proprio degli oggetti. Con `ObjectInputStream e ObjectOutputStream` è possibile rappresentare flussi di oggetti.

Non tutti gli oggetti vanno bene: solo quelli che implementano l'interfaccia `Serializable`. L'interfaccia non ha metodi e serve solo a marcare che gli oggetti possono essere trasmessi tramite stream. Viene lanciata l'eccezione `NotSerializableException` se si usa un oggetto che non implementa Serializable.

La serializzazione è il processo che trasforma un oggetto in un'opportuna sequenza di byte (`writeObject(obj)`) e viceversa (`readObject()` - deserializzazione). La deserializzazione ritorna un oggetto di tipo `Object`, quindi sta al programmatore fare un cast corretto in base al tipo adatto.

Non tutti i tipi sono serializzabili. Tuttavia quando si serializza un oggetto, devono essere serializzati tutti i campi, soprattutto se sono riferimenti ad altri oggetti che però possono essere non serializzabili. In questi casi si può implementare la serializzazione custom:

- la classe deve avere il costruttore senza argomenti
- la classe deve implementare i due metodi di lettura/scrittura
- i campi non serializzabili vanno indicati con la keyword `transient`

8.4 Espressioni Lambda

Un'espressione Lambda in Java è un metodo senza dichiarazione esplicita con una forma `(parametri) -> {corpo}`. Regole generiche:

- i parametri possono essere più di uno, separati da virgola, ma anche nessuno
- le parentesi dei parametri si possono omettere se è un solo parametro
- il tipo dei parametri può essere implicito o esplicito
- le parentesi del corpo possono venire omesse se è un solo statement

Le espressioni Lambda vengono tradotte dal compilatore in funzioni di **interfacce funzionali**. Le interfacce funzionali contengono un singolo metodo astratto *il quale è la traduzione dell'espressione* (incerto). Esistono delle interfacce già definite come:

- `Function` - accetta un argomento di tipo T e ritorna un tipo R
- `Supplier` - nessun argomento e ritorna un tipo R
- `Consumer`
- `Predicate` - un solo argomento e ritorna boolean

8.5 Stream API

L'interfaccia `Stream` è stata introdotta da Java 8 in supporto alle operazioni sequenziali e parallele su un flusso. Uno stream di solito riassume una sequenza di passi su un insieme di oggetti (es: array o collezione). Non produce side-effect. Gli elementi dello stream vengono visitati una sola volta. Operazioni su steam, che possono essere intermedie (ritornano uno stream) o terminali (ritornano un valore):

- `filter()` - ritorna uno stream con gli elementi filtrati secondo il predicato passato [Intermedia]
- `map()` - ritorna uno stream con gli elementi ai quali ho applicato la funzione passata [Intermedia]
- `distinct()` - ritorna uno steam con elementi distinti [Intermedia]
- `sorted()` - ritorna uno stream con gli elementi ordinati [Intermedia]
- `parallel()` - ritorna uno stream parallelo [Intermedia]
- `sequential()` - ritorna uno stream sequenziale [Intermedia]
- `limit()` - ritorna uno stream limitato nella lunghezza in base al valore passato [Intermedia]
- `forEach()` - compie un azione per ogni elemento dello stream [Terminale]
- `count()` - ritorna il numero di elementi dello stream [Terminale]
- `anyMatch()` - ritorna true se un qualsiasi elemento rende true il predicato; può non valutarli tutti [Terminale]
- `allMatch()` - ritorna true se tutti gli elementi rendono true il predicato [Terminale]

Capitolo 9

Programmazione Distribuita

La programmazione distribuita entra in gioco quando si devono gestire processi su più macchine diverse. Le caratteristiche di un algoritmo distribuito sono:

- Concorrenza dei componenti
- Mancanza di un *Global Clock*, cioè l'ordine temporale non è strettamente condiviso
- Fallimenti indipendenti tra loro

L'unica risorsa che condividono le macchine è la rete, attraverso la quale si scambiano messaggi e dati.

9.1 Socket TCP

Due programmi possono comunicare attraverso gli stream associati ad una connessione di rete. Un tipo di comunicazione possibile è tramite i **socket TCP**.

Ogni servizio fornito da un server è identificato da una porta logica; per gestire la comunicazione tra più client verso una stessa porta, viene usato un socket per ogni client.

Il protocollo di comunicazione client/server è il seguente:

1. il server si mette in ascolto in attesa di richieste da parte di un client
2. un client richiede un servizio

- il client crea un socket per comunicare con il server utilizzando l'indirizzo dell'host del server e la porta del servizio richiesto
- il server crea un socket per la comunicazione con il client richiedente

3. i due socket alle estremità vengono associati a degli stream per la comunicazione
4. una volta terminata la comunicazione, vengono chiusi i socket e la connessione

In particolare, la classe base è `Socket`, da cui deriva `ServerSocket` per la creazione dei socket del server. Il `ServerSocket` viene costruito passando come parametro il numero della porta.

La classe `InetServer` ha gli oggetti che rappresentano gli indirizzi IP. Questa classe è necessaria per la creazione dei socket da parte del client tramite il costruttore[1]:

```
1 public Socket(InetAddress address, int port) throws
    IOException
```

Con il metodo `accept()`, il ServerSocket accetta una richiesta di connessione e ritorna il relativo socket del client. Una volta connessi, è possibile ottenere gli stream collegati con i metodi `getInputStream()` e `getOutputStream()`. Questi stream sono thread-safe, ma un solo thread alla volta può scriverci e/o leggerci. Infine con il metodo `close()` viene chiuso il socket (di solito solo quello del client dato che il ServerSocket deve rimanere in attesa di altre richieste).

La programmazione distribuita tramite socket è di basso livello.

9.2 Datagram

Un'alternativa alla comunicazione tramite socket è quella dei Datagram. Questo protocollo manda pacchetti di informazioni verso una o più destinazioni, senza garanzia di ricezione o di ordinamento in arrivo.

La classe da utilizzare per ricevere o inviare i pacchetti è `DatagramSocket`. Non c'è distinzione di classe per la ricezione o l'invio. Un DatagramSocket si può connettere con il metodo `connect()` ad un indirizzo specifico, ma in

[1] Ndr: ovviamente ci sono anche costruttori con più parametri, sia per Socket, che per ServerSocket.

questo modo i pacchetti vengono spediti o ricevuti solo verso/da quell'indirizzo. Con il metodo `send()` è possibile inviare un pacchetto rappresentato da un oggetto di `DatagramPacket`, mentre con `receive()` si riceve un pacchetto. La classe `DatagramPacket` è `final`, quindi non si può estendere. Infine il DatagramSocket va chiuso con il metodo `close()`.

9.3 URL

Esiste inoltre la possibilità di interagire direttamente con risorse web tramite la classe `final URL e URLConnection`. [...][2]

9.4 Channel

L'interfaccia `Channel` rappresenta un canale di I/O aperto o chiuso. L'interfaccia `NetworkChannel` invece è un Channel per un socket, cioè rappresenta una comunicazione su una rete. La classe `AsynchronousServerSocketChannel` è un canale asincrono basato su un server socket. [...][3]

9.5 Modello RMI

Il RPC (Remote Procedure Call) è un sistema per chiamare procedure di un'altra macchina rendendo la chiamata più simile possibile ad una chiamata locale. L'adattamento locale/remoto viene implementato da un componente detto **stub**. Questo meccanismo viene ripreso nei linguaggi orientati agli oggetti e prende il nome di RMI (Remote Method Invocation). Una chiamata RMI è composta dai seguenti passi:

1. il client chiama lo stub locale
2. lo stub prepara i parametri in un messaggio
3. lo stub invia il messaggio e il client viene bloccato
4. il server riceve il messaggio
5. il server controlla il messaggio e cerca l'oggetto chiamato
6. il server recupera i parametri e chiama il metodo destinatario

[2]Ndr: non verrà approfondito.
[3]Ndr: non verrà approfondito.

7. l'oggetto esegue il metodo e ritorna il risultato
8. il server prepara il risultato in un messaggio
9. il server invia il messaggio allo stub
10. lo stub riceve il messaggio di ritorno
11. lo stub recupera il risultato dal messaggio
12. lo stub ritorna al client il risultato

Con l'evolvere della tecnologia, la strategia RMI viene usato poco e principalmente in ambienti piccoli e controllati.