Programmazione Java

Terza Parte

Alfonso Domenici

Argomenti

Metodi clone e getClass della classe Object

Gestione I/O

Gestione delle date

Threads

Interfacce grafiche: Swing, JavaFX

Patterns

Metodi clone e getClass della classe Object

Il metodo **clone** fornisce la copia di un oggetto in maniera efficiente. La classe i cui oggetti possono essere clonati deve implementare l'interfaccia Cloneable (che è vuota) e ridefinire clone come pubblic (da protected).

La clonazione può essere superficiale o profonda.

Il metodo getClass fornisce un oggetto di tipo Class mediante il quale si possono ottenere informazioni sulle proprietà (attributi, metodi e costruttori) della classe con il supporto del package java.lang.reflect .

Superficiale: vengono clonati solo i valori. Se l'oggetto contiene riferimenti ad altri oggetti questi ultimi non vengono clonati.

Profonda: vengono clonati anche i riferimenti ad altri oggetti ricorsivamente. Va fatto a mano

Clonazione superficiale di Point

```
public Point clone(){
    try {return (Point)super.clone();}
    catch (CloneNotSupportedException e) {return null;}
}
```

Clonazione profonda di Rectangle

```
public Rectangle clone(){
    try {Rectangle r = (Rectangle) super.clone();
    r.origin = origin.clone();return r;}
    catch (CloneNotSupportedException e) {return null;}
}
```

Approccio alternativo

NB. utilizzo di Clonable e clone() abbastanza critico e fragile..

paradigma complicato (implementare e riscrivere protected..)

problemi con i campi final, utilizzo cast..,checked exception ecc..

Altri approcci più convenienti

```
// Copy constructor
public Point(Point p) { ... };
// Copy factory
public static Point newInstance(Point p) { ... };
```

Esempi di uso di Class

```
Point p1 = new Point(10,20);
Rectangle r1 = new Rectangle(p1,100,200);
Class<? extends Rectangle> c1 = r1.getClass();
// r1 potrebbe puntare ad un oggetto di una classe derivata da Rectangle
System.out.println(c1. getName ());
// nome del package.Rectangle
```

Nota: il metodo getSimpleName () dà soltanto il nome della classe.

Metodo newInstance

Class<T> offre il metodo

public T newInstance() throws InstantiationException, IllegalAccessException

mediante il quale si può istanziare un oggetto con il costruttore privo di parametri.

Rectangle r2 = c1.newInstance();

System.out.println(r2); x = 0 y = 0 w = 20 h = 10

Metodo forName

Il metodo statico forName

static Class<?> forName(String className)

dà l'oggetto Class in base al nome della classe preceduto dal nome del package.

Se Rectangle si trova nel package p

String className = "p.Rectangle";

@SuppressWarnings("unchecked")

Class<Rectangle> c2 = (Class<Rectangle>) Class. forName (className);

Nota: si può generare un oggetto dato il nome della classe!

// il cast serve perché non si sa a priori di quale classe si tratti.

Rectangle r3 = c2.newInstance();

x = 0 y = 0 w = 20 h = 10

Gestione I/O

Le librerie per I/O usano spesso l'astrazione di stream , che rappresenta una sorgente o una destinazione di dati come un oggetto capace di produrre o ricevere dati in forma di flusso.

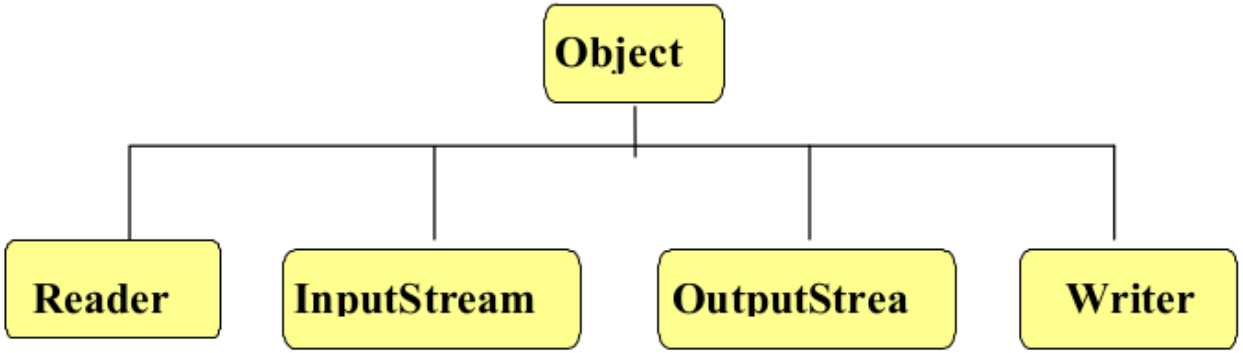
Algoritmo generico per leggere/scrivere:

Per leggere	Per scrivere
open uno stream	**open** a stream
while ci sono dati	**while** ci sono dati
read dati	**write** dati
close lo stream	**close** lo stream

Gli Stream sono suddivisi in due categorie: stream di byte e di caratteri

NB. E' necessario trattare separatamente i byte dai caratteri, in quanto i caratteri Java non sono byte.

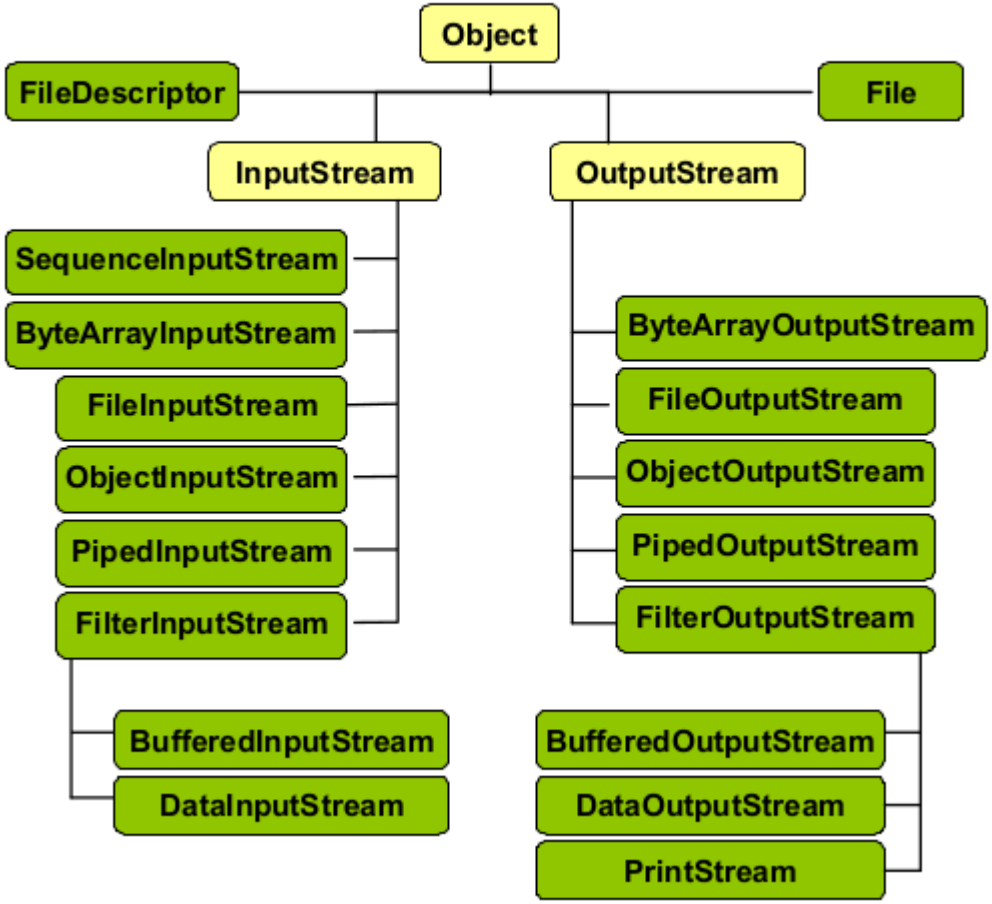
4 classi astratte di base



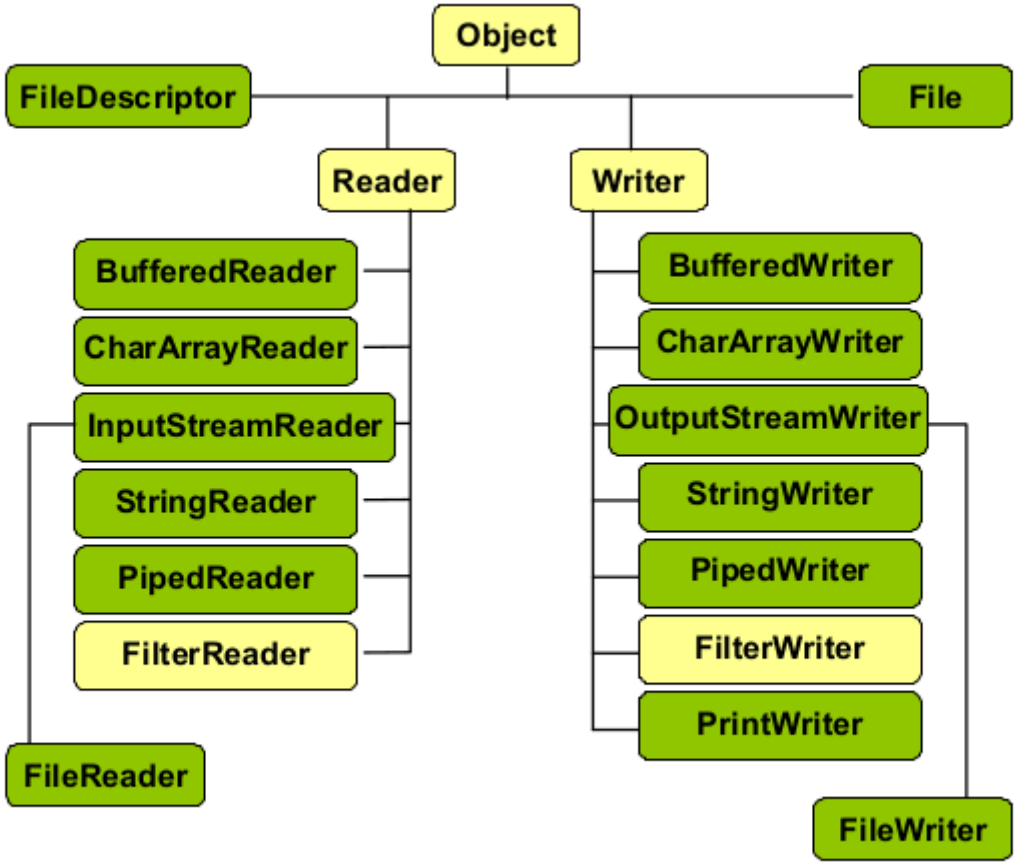
Le classi derivate si dividono in due categorie, specializzate in due sensi:

- classi (dette **sorgenti**) che, senza aggiungere funzionalità, specializzano le classi astratte rispetto alla sorgente/destinazione destinazione dei flussi
- classi (dette di **filtraggio**) che, di nuovo non preoccupandosi della sorgente/destinazione dei flussi, effettuano un trattamento dei dati

LA GERARCHIA DEGLI STREAM DI BYTE



La Gerarchia degli Stream di caratteri



File

I file sono visti come stream (sequenze) di caratteri o di dati in formato binario.

Tipi di stream:

byte stream: letti e scritti a byte

character stream: letti e scritti a caratteri (codifica dei caratteri in Unicode a 16 bit)

testuali (line-oriented): letti e scritti a linee ; sono usati buffer per migliorare l'efficienza

data stream: contengono valori e stringhe in formato binario; la struttura deve essere nota a priori .

Le operazioni principali sono: apertura di file in lettura oppure in scrittura, chiusura di file, lettura di elemento, scrittura di elemento.

Le operazioni possono sollevare eccezioni di tipo IOException (sottoclasse di Exception); IOException è checked.

Le classi si trovano nel package java.io.

Try con risorse

```
try ( // risorse che saranno chiuse automaticamente all'uscita dalla try
)
{
} catch ...
```

Senza risorse, nell'esempio precedente serve una clausola finally

```
finally {
	if (in != null) in.close();
	if (out != null) out.close();
}
```

Character stream

Esempi: file in input Anagrafica.txt

lettura e scrittura di un byte alla volta

lettura e scrittura di un carattere alla volta

lettura e scrittura di una linea alla volta

lettura in un’unica operazione di tutte le linee

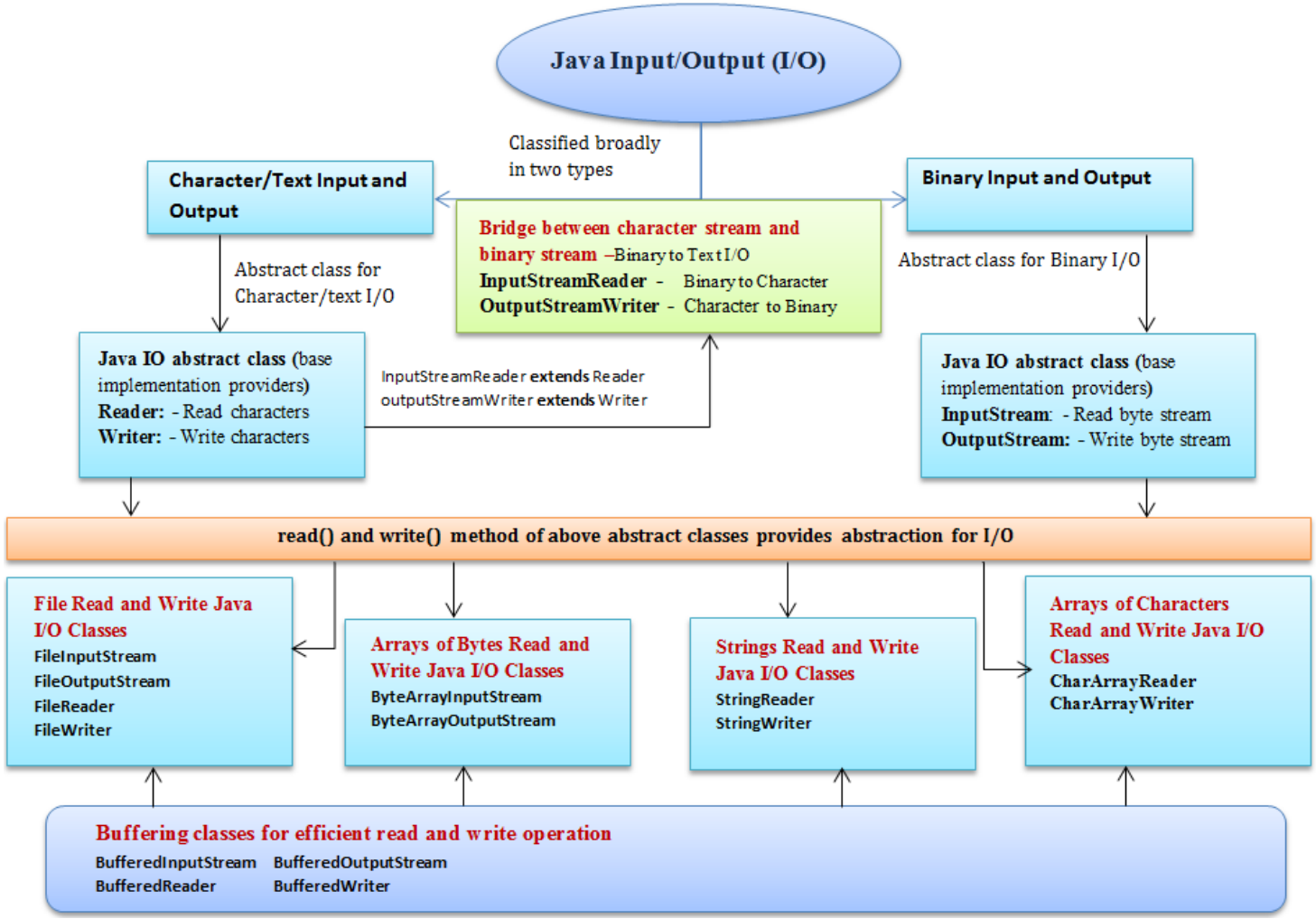
copia da file testuale a file binario (struttura nota a priori)

lettura e stampa di un file binario (struttura nota a priori)

leggere una stringa da tastiera e la convertirla in maiuscolo finché non si legge “end”

copiare un file di testo numerando le righe

Riepilogo java.io



Nuove classi e interfacce per l'accesso al file system

Si trovano nel package **java.nio.file**.

Interfaccia **Path**

Classe **Paths**

Classe **Files**

Path e Paths

L'interfaccia **Path** rappresenta il percorso di un file o cartella.

La classe **Paths** offre il metodo statico **get** che fornisce il **path** data una stringa contenente il percorso del file o cartella.

Esempio

Path fileP = Paths.get("/home/user/documenti/note.txt");

Il metodo **getFileName** di Path dà come path il nome del file senza il percorso, ad es. note.txt.

Classe Files

Questa classe offre metodi statici per operazioni su file e cartelle.

Per leggere tutto un file testuale si possono usare i 2 metodi seguenti:

List<String> linee = Files.readAllLines(pathFile); // pathFile è un path

Stream<String> lines = Files.lines(pathFile);

Per scrivere tutto un file testuale si può usare il metodo write:

Files.write(pathFile, linee); // pathFile è un path.

Per ottenere lo stream dei path dei file di una cartella si usa il metodo list:

Stream<Path> cartelle = Files.list(folderPath); // folderPath è un path

I metodi precedenti possono lanciare IOException.

Classe Files

Esercizi

1. leggere un file e riscriverlo con il numero di riga
2. convertire in maiuscolo le linee di un file
3. stampare i nomi dei file testuali (.txt) presenti in una cartella.

Gestione delle Date

java.time – è il package che contiene le classi base

java.time.chrono – permette l'accesso a differenti tipi di calendari

java.time.format – contiene le classi per la formattazione e il parsing di date e orari

java.time.temporal – estende il package base mettendo a disposizione classi per la manipolazione più a basso livello di date e orari.

java.time.zone – contiene le classi per la gestione delle time zones

Senza TimeZone: **LocalDate, LocalDateTime, LocalTime**

Con TimeZone: **ZonedDateTime**

Period ideale per rappresentare il periodo tra due date

Duration ideale per rappresentare l'intervallo tra due time

Instant rappresenta semplicemente il numero di secondi dalla mezzanotte del 1 Gennaio 1970 UTC.

enum: **ChronoUnit**, **Month**, **DayOfWeek**

Gestire le date LocalDate

Data e data ora dove il timezone non è richiesto

Questa classe rappresenta una descrizione di una data. come ad esempio “1 settembre 2014” che avrà inizio in momenti diversi nella timeline in base alla nostra posizione sulla terra; quindi, per dire, a Roma la stessa data locale inizierà 6 ore prima che a New York e 9 ore prima che a San Francisco

La classe *LocalDate*, così come la maggior parte delle classi della *java.time API* usano un unico calendario standard ed in particolare lo standardISO-8601 (calendario Gregoriano)

L’interfaccia *Chronology* è la classe base su cui è costruito il sistema di gestione dei calendari e 4 sono gli altri sistemi di calendario forniti in Java SE 8: il buddista thailandese, il Minguo, il giapponese, e la Hira.

Gestire orari

LocalTime rappresenta un valore senza alcuna data associata ne fuso orario ed è relativo ad una determinata zona del pianeta

```
LocalTime time = LocalTime.of(20, 30);
int hour = date.getHour(); // 20
int minute = date.getMinute(); // 30
time = time.withSecond(6); // 20:30:06
time = time.plusMinutes(3); // 20:33:06
```

Gestione congiunta di Data e Ora

```
LocalDateTime adesso = LocalDateTime.now();
System.out.println(adesso);
LocalDate d1 = oggi.plusDays(20);
LocalDate d2 = today.plus(1, ChronoUnit.WEEKS);
System.out.println(d1);
Period p1 = Period.between(oggi, d1);
System.out.println(p1);
System.out. format ("il periodo comprende %d giorni%n", p1.getDays());
System.out.println(Arrays.toString(Month.values()));
System.out.println(Arrays.toString(DayOfWeek.values()));
```

Di seguito una tabella che riassume i vari prefissi usati nei nomi dei metodi e il loro significato:

Prefisso	Descrizione
of	Metodo factory statico per la creazione di un oggetto dalle sue componenti
from	Metodo factory statico che prova ad estrarre un'istanza da un oggetto simile
now	Metodo factory statico che crea un istanza con orario impostato all'ora corrente
parse	Metodo factory statico che crea un istanza parsando una stringa passata in input
get	Restituisce una parte costituente (giorno, mese,….) di un oggetto date-time
is	Verifica se una qualche proprietà di un oggetto date-time è vera o falsa.
with	Restituisce una copia di un oggetto date-time con qualche proprietà modificata.
plus	Restituisce una copia di un oggetto date-time con un valore di una qualche proprietà aumentata di un certo tot di tempo
minus	Restituisce una copia di un oggetto date-time con un valore di una qualche proprietà diminuita di un certo tot di tempo
to	Converte un oggetto date-time in un altro
at	Combina un oggetto date-time con altri oggetti per creare un'altro oggetto date-time più complesso
format	Formatta un oggetto date-time come stringa nel formato desiderato.

La classe *Instant*

Quando si tratta di date e orari, di solito pensiamo in termini di anni, mesi, giorni, ore, minuti, e secondi. Tuttavia, questo è solo un modello di tempo, quello che chiamano “umano”.

Il secondo modello di uso comune è il tempo “macchina” o tempo “continuo”.

Concettualmente, rappresenta semplicemente il numero di secondi dalla mezzanotte del 1 Gennaio 1970 UTC. Dal momento che l’API è sulla base nanosecondi, la classe *Instant* fornisce una precisione nell’ordine dei nanosecondi.

```
Instant start = Instant.now();
// qualche calcolo
Instant end = Instant.now();
assert end.isAfter(start);
```

Time Zones

Un fuso orario è un insieme di regole, che corrisponde ad una zona della terra in cui il tempo standard è la stesso. Ci sono circa 40 fusi orari e sono definiti dal loro offset dal Coordinated Universal Time (UTC).

Esiste un *ZoneId* per ogni regione del pianeta. Ogni *ZoneId* corrisponde ad alcune regole che definiscono il fuso orario per quella località.

Comunemente quando si parla di fuso orario si parla di un offset fisso rispetto a UTC / Greenwich che è considerato lo zero nella divisone settoriale di ogni zona della terra.
Ad esempio diciamo che New York è cinque ore indietro (quindi offset -5) rispetto a Londra che sta nello zero.

Tempo come quantità

```
// un oggetto duration che rappresenta 3 secondi e 5 nanosecondi
Duration duration = Duration.ofSeconds(3, 5);
Duration oneDay = Duration.between(today, yesterday);

// un oggetto period che rappresenta 6 mesi
Period sixMonths = Period.ofMonths(6);

LocalDate date = LocalDate.now();
LocalDate future = date.plus(sixMonths);
```

Formattazione e parsing

In *java.time* abbiamo un intero package dedicato alla formattazione e stampa di date e orari: il *java.time.format*. Le classi cardine di questo package sono *DateTimeFormatter* e il suo relativo builder *DateTimeFormatterBuilder*. E' possibile creare un formattatore in tre modi:

1. Usando i metodi statici e le costanti predefinite di *DateTimeFormatter*, come può essere ISO_LOCAL_DATE
2. Usando i pattern del tipo dd/MM/yyyy
3. Usando gli stili locali che posso essere in formato completo, lungo, medio o corto.

Retro compatibilità

è stato aggiunto il metodo toInstant() a Date e Calendar

Esempi..

Threads

Implementano elaborazioni concorrenti (almeno logicamente).

Occorre gestire la mutua esclusione e le sincronizzazioni.

Quando si pone in esecuzione un programma, si attiva un thread di default per il main.

Scrittura dei Threads

2 possibilità:

1. Si definisce una sottoclasse di **Thread** e si scrive la logica nel metodo run (che è chiamato allo start del thread).

2. Se una classe implementa l'interfaccia **Runnable** (ossia ha un metodo run) si può associare ad un thread un suo oggetto.

Class X implements Runnable { ... }

Runnable r = new X(...);

Thread t = new Thread(r);

t.start();

public interface Runnable {

void run(); }

Classe Thread (java.lang)

Alcuni metodi:

public Thread(String name)

public Thread(Runnable target, String name)

public Thread(Runnable target) // nome generato automaticamente

public String getName()

public void interrupt ()

public static void sleep (long millis) throws InterruptedException

public void start()

L'eccezione InterruptedException è lanciata quando il thread è interrotto mentre è in attesa o è sleeping.

Esempi

1. Due thread stampano 10 stringhe, ciascuno ad intervalli casuali <= 1 secondo.

2. Un thread continua a stampare stringhe e un altro (il main) lo interrompe dopo 10 sec.

3. Un produttore e due consumatori interagiscono tramite una coda di stringhe.

4. Uso di ArrayBlockingQueue ; il package java.util.concurrent offre classi utili nella programmazione concorrente.

5. Esempio dei 5 filosofi.

6. Timer in java

Produttori e consumatori: requisiti

Esempio di produttori e consumatori che interagiscono tramite una coda di stringhe.

I produttori aggiungono stringhe lette da un array ricevuto come parametro; i consumatori tolgono stringhe.

Produttori e consumatori hanno un nome e un intervallo massimo di azione; l'intervallo effettivo è un valore casuale tra 0 e l'intervallo massimo.

La coda ha una capacità : se è piena, il produttore aspetta; se è vuota, il consumatore aspetta. La coda registra il numero di attese dei produttori e quello dei consumatori.

I thread scrivono in un log ciò che hanno letto (scritto) dalla (nella) coda.

Produttori e consumatori

Problemi:

• accesso mutuamente esclusivo ad un metodo; metodo synchronized

• sospensione se una condizione risulta falsa; wait

• risveglio dei thread sospesi; notify o notifyAll

Il metodo wait può lanciare un'eccezione di tipo InterruptedException.

I metodi wait, notify e notifyAll sono definiti nella classe Object e sono final.

Metodi sincronizzati e guarded blocks

Mentre un thread esegue un metodo **synchronized** di un oggetto, gli altri thread che chiamano un metodo synchronized dello stesso oggetto sono bloccati.

Ogni oggetto ha un lock, chiamato monitor. Quando un thread chiama un metodo synchronized di un oggetto il cui lock è libero, il thread acquisisce il lock (che diventa impegnato) ed esegue il metodo. Se invece il lock è impegnato si accoda.

Quando un metodo synchronized termina, il lock diventa libero e può essere acquisito da un eventuale thread in coda.

Guarded blocks

Se occorre che una certa condizione sia vera e il thread la trova falsa, allora può eseguire una wait: come effetto il thread rilascia il lock e si sospende. Verrà poi risvegliato da una notify o notifyAll eseguita da un altro thread. Il thread risvegliato deve accertarsi che la condizione di prosecuzione sia verificata: in caso contrario chiamerà ancora la wait (di solito si usa un loop).

Uso di ArrayBlockingQueue

ArrayBlockingQueue implementa BlockingQueue.

Si trovano in java.util.concurrent.

Interface BlockingQueue<E>

void **put**(E e); inserisce l'elemento e aspetta se la coda è piena.

E **take**(); rimuove il primo elemento e aspetta se la coda è vuota.

Costruttore di ArrayBlockingQueue

public ArrayBlockingQueue(int capacity)

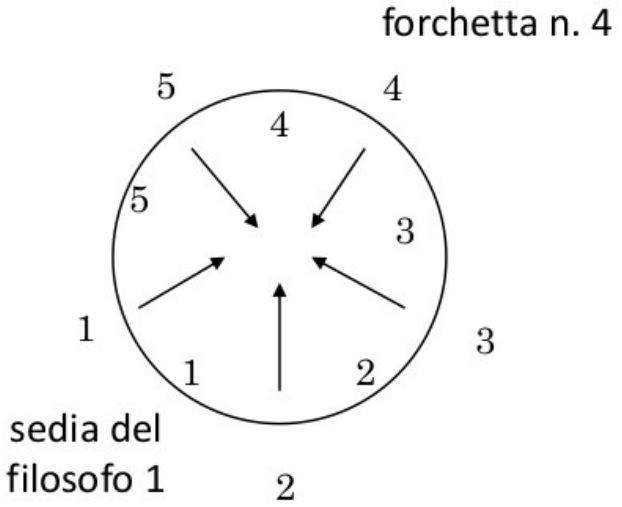
Esempio 5 filosofi

I filosofi ripetutamente lavorano e poi si recano in sala da pranzo per mangiare. La tavola ha 5 forchette e ciascuno ne usa 2.

La tavola registra il n. di attese (nWait) e il n. di successi (n) per filosofo.

Il periodo in cui un filosofo pensa (mangia) dura fino a 2 (3) secondi.

Per evitare un deadlock ciascun filosofo deve prendere simultaneamente le 2 forchette e non una per volta.



Nota:
nell'implementazione filosofi e forchette sono numerati da 0 a 4.

Classi usate:

1. Tavola (condivisa dai filosofi, ThreadSafe…)

2. Filosofo

3.App